An Efficient Algorithm To Update Large Itemsets With Early Pruning

Necip Fazıl Ayan
Dept. of CEIS, Bilkent University, 06533, Ankara, Turkey
fayan@cs.bilkent.edu.tr

Abdullah Uz Tansel
PhD Program in CS, Graduate Center
Baruch College, The City University of New York
tansel@baruch.cuny.edu

Erol Arkun
Dept. of CEIS, Bilkent University, 06533, Ankara, Turkey
arkun@bilkent.edu.tr

Abstract

Although many efficient algorithms have been proposed for the discovery of association rules, the process of updating large itemsets is still a complicated issue for dynamic databases that involve frequent additions. We present an efficient algorithm for updating large itemsets ($UWEP$) when new transactions are added to the set of old transactions in a transaction database. $UWEP$ employs a dynamic look-ahead strategy in updating the existing large itemsets by detecting and removing those that will no longer remain large after the contribution of the new set of transactions. $UWEP$ executes iteratively, but it differs from the other proposed algorithms by scanning the existing database at most once and the new database exactly once. Moreover, it generates and counts the minimum number of candidates in the new database. The experiments on synthetic data show that $UWEP$ outperforms the existing algorithms in terms of the candidates generated and counted.
Keywords. Maintenance of association rules, dynamic pruning, large itemsets.

1 Introduction

With the recent developments in computer storage technology, many organizations have collected and stored massive amounts of data. Even though very useful information is buried within this data, this information is not readily available for the users. Obviously, there is a need for developing techniques and tools that assist users to analyze and automatically extract hidden knowledge. Knowledge discovery in databases includes techniques and tools to address this need.

Association rules are one of the promising aspects of data mining as a knowledge discovery tool, and have been widely explored to date. An association rule, $X \Rightarrow Y$, is a statement of the form "for a specified fraction of transactions, a particular value of an attribute set X determines the value of attribute set Y as another particular value". Thus, association rules aim at discovering the patterns of co-occurrences of attributes in a database. The problem of discovering association rules was first explored in [2] on supermarket basket data, that is the set of transactions that include items purchased by the customers. In this pioneering work, mining of association rules was decomposed into two subproblems: discovering all frequent patterns (represented by large itemsets defined below), and generating the association rules from those frequent itemsets. The second subproblem is straightforward, and can be done efficiently in a reasonable time. However, the first subproblem is very tedious and computationally expensive for very large databases and this is the case for many real life applications. Many efficient algorithms have been proposed for finding the frequent patterns in a database [1, 2, 4, 5, 6, 9, 10, 11, 13, 15, 17].

Maintenance of association rules is an important problem. When new transactions are added to the set of old transaction database, how can we update the association rules discovered in the set of old transactions efficiently? Naturally, when new transactions are added to a database, some of the existing frequent patterns may disappear whereas new frequent patterns that do not exist before may also emerge. The straightforward solution is to re-run an algorithm, say *Apriori* [4], on the set of whole transactions, i.e., old transactions plus new transactions. However, this process is not efficient since it ignores the previously discovered rules, and repeats all the work done previously. Therefore, algorithms for efficiently updating the association rules were proposed in [7, 8, 12, 14, 16]. These algorithms take the set of association rules in the old database into account, and use this knowledge 1) to remove itemsets that do not exist in the updated database, and 2) to add new rules which were not in the set of old transactions but implied in the updated database. Particularly, when the size of old transactions is large, these algorithms discover the new set of association rules much faster than by re-running an algorithm over the whole database.

In this paper, we propose an algorithm called *UWEP* (**U**pdate **W**ith **E**arly **P**runing) that follows the approaches of FUP_2 [8] and *Partition Update* [12] algorithms. It works iteratively on the new set of transactions, like the previous algorithms. The advantages of $UWEP$ are that it scans the existing database at most once and new database exactly once, and it generates and counts the minimum number of candidates in order to determine the new set of association rules. Similar to [15], in one scan of the database, it creates a *tidlist* for each item in the database, and uses these structures in order to compute the support of supersets of that item. Moreover, it prunes an itemset that will become small from the set of generated candidates as early as possible by a look-ahead pruning. In other words, it does not wait for the k^{th} iteration for pruning a small k-itemset. This look-ahead pruning results in a much smaller number of candidates in the set of new transactions. Another reason for generation of a smaller candidate set is the fact that $UWEP$ promotes a candidate itemset to the set of large itemsets only if it is large both in the new set of transactions and in the whole database. This feature yields a much smaller candidate set when some of the old large itemsets are eliminated due to their absence in the new set of transactions, and this can be done without scanning the old database.

The rest of the paper is organized as follows. In Section 2, formal descriptions of discovering and updating association rules, and related algorithms are presented. Section 3 presents the $UWEP$ algorithm as well as an example to demonstrate it. In this section, we also prove the correctness of $UWEP$ algorithm, and that it generates and counts a minimum number of candidates. Details of the experiments and performance results on synthetic data are provided in Section 4. The paper concludes with a discussion of the results in Section 5.

2 Formal Problem Description

2.1 Discovery of Association Rules

Agrawal et al. define the problem of discovering association rules in databases in [2, 4]. Let $I = \{I_1, \ldots, I_m\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$, and each transaction is associated with a unique identifier called TID. Let X, called an *itemset*, be a set of items in I. An itemset X is called a k-itemset if it contains k items from I. We say that a transaction T satisfies X if $X \subseteq T$. The support of an itemset X in D, $support_D(X)$, is defined as the number of transactions in D that satisfy X. An itemset X is called a *large itemset* if the support of X in D exceeds a minimum support threshold explicitly declared by the user, and a *small itemset* otherwise.

By an association rule, we mean an implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$. We call X the *antecedent* of the rule, and Y

the *consequent* of the rule. The rule $X \Rightarrow Y$ holds in D with confidence c where $c = \frac{support_D(X \cup Y)}{support_D(X)}$. The rule $X \Rightarrow Y$ has support s in D if the fraction s of the transactions in D contain $X \cup Y$.

Given a set of transactions D, the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified *minsup* and *minconf*, respectively. Formally, the problem is generating all association rules $X \Rightarrow Y$, where $support_D(X \cup Y) \geq minsup \times |D|$ and $\frac{support_D(X \cup Y)}{support_D(X)} \geq minconf$.

The problem of finding association rules can be decomposed into two parts [2, 4]:

Step 1: Generate all combinations of items with fractional transaction support (i.e., $\frac{support_D(X)}{|D|}$) above a certain threshold, called *minsup*.

Step 2: Use the large itemsets to generate association rules. For every large itemset l, find all non-empty subsets of l. For every such subset a, output a rule of the form $a \Rightarrow (l - a)$ if the ratio of $support(l)$ to $support(a)$ is at least *minconf*. If an itemset is found to be large in the first step, the support of that itemset should be maintained in order to compute the confidence of the rule in the second step.

The second subproblem is straightforward, and an efficient algorithm for extracting association rules from the set of large itemsets is presented in [3]. On the other hand, discovering large itemsets is a non-trivial issue. The efficiency of an algorithm strongly depends on the size of the candidate set. The smaller the number of candidate itemsets is, the faster the algorithm will be. As the minimum support threshold decreases, the execution times of these algorithms increase because the algorithm needs to examine a larger number of candidates and larger number of itemsets. Association rule algorithms generally differ on a) the generation of the candidates, b) counting of the support of a candidate itemset c) number of scans over the database, and d) the data structures employed. Readers are referred to [1, 2, 4, 5, 6, 9, 10, 11, 13, 15, 17] for some algorithms for discovering large itemsets.

2.2 Update of Association Rules

Table 1 summarizes the notations used in the remainder of the paper. Updating association rules was first introduced in [7]. Given $DB, db, |DB|, |db|, minsup$ and L_{DB}, the problem of updating association rules is to find the set L_{DB+db} of large itemsets in $DB + db$.

The *FUP* algorithm proposed by Cheung et al. [7] works iteratively and its framework is similar to *Apriori* [4] and *DHP* [13]. Initially, the candidate set of 1-itemsets of *db* is the set of items which exist in at least one transaction in *db*. At the end of the k^{th} iteration, the new set of candidates are computed from the set of large k-itemsets in the updated database. There are three optimizations employed

Notation	Definition
DB	The set of old transactions
db	The set of new transactions
$DB + db$	The total set of transactions
$\|A\|$	The number of transactions in the transaction database A
$minsup$	Minimum support threshold
$support_A(X)$	Support of X in the set of transactions A
$tidlist_A(X)$	Transaction list of X in the set of transactions A
C_A^k	Set of candidate k-itemsets in a set of transactions A
L_A^k	Set of large k-itemsets in a set of transactions A
$PruneSet$	Set of large itemsets in DB that have 0 support in db
$Unchecked$	Set of large k-itemsets in DB that are not counted in db

Table 1: Notations Used in the Paper

in FUP, two of which are based on the reduction of transactions (i.e., if X is a small itemset in D, remove X from the transactions in D). The other is the computation of an upper bound value for the support of an itemset, and deciding whether the itemset is small without scanning the database.

FUP_2 [8] is a generalization of the FUP algorithm that handles insertions to and deletions from an existing set of transactions. The algorithms FUP and FUP_2 scan DB and db as many times as the length of the maximal large itemset in the updated database, and generates a large number of candidates in db since it generates C_{db}^k from L_{DB+db}^{k-1}.

In [14, 16], the concept of *negative border*, that was introduced in [17], is used to compute the new set of large itemsets in the updated database. The negative border consists of all itemsets that were candidates but did not have enough support while computing large itemsets in DB, i.e., $NBD(L_k) = C_k - L_k$. It is assumed that the negative border of the set of large itemsets in DB and their counts in DB are available. In [16], the set of large itemsets in db is first computed by a scan over db. In the same scan, the supports of all itemsets in L_{DB} and $NBD(L_{DB})$ over db are also counted. Then, all itemsets that are large both in DB and db are promoted to the set of large itemsets in $DB + db$. If an itemset X is large in db but small in DB, X and its supersets are checked against DB using the negative border of L_{DB}. If such an itemset is promoted to the set of large itemsets in $DB + db$, the negative border is computed again, and this process is repeated until there is no change in the negative border. This algorithm scans DB at most once and db as many times as the length of the maximal large itemset in db. However, recomputing the negative border again and again reduces its performance. The approach in [14] is very similar to the one in [16]. It first counts the supports of itemsets in L_{DB} and

$NBD(L_{DB})$ over db. If any of the itemsets in the negative border is found to be large in db, then it computes L_{db} and validates those against DB by scanning DB once. Its major advantage is that it does not scan DB if there is no new itemset in db.

A recent study [12] uses the framework in [15], and assumes that the set of large itemsets in the old database is available. Then, it computes the large itemsets in db by using *Partition* [15]. Its final step involves computing the support of large itemsets in DB against db, and vice versa. This requires one additional scan of DB and db.

3 Update with Early Pruning ($UWEP$)

3.1 Description of the Algorithm

In this section, we will explain how our algorithm works, and the optimizations it employs. The algorithm $UWEP$ is presented in Figure 1. Inputs to the algorithm are DB, db, L_{DB} (along with their supports in DB), $|DB|$, $|db|$, and *minsup*. The output of the algorithm is L_{DB+db}, the set of large itemsets in $DB + db$.

We can break down the algorithm $UWEP$ into five steps as identified below.

1. Counting 1-itemsets in db and creating a *tidlist* for each item in db

2. Checking the large itemsets in DB whose items are absent in db and their supersets for largeness in $DB + db$

3. Checking the large itemsets in db for largeness in $DB + db$

4. Checking the large itemsets in DB that are not counted over db for largeness in $DB + db$

5. Generating the candidate set from the set of large itemsets obtained at the previous step.

In the first step of the $UWEP$ algorithm(line 1 in Figure 1), we count the support of 1-itemsets and create a *tidlist* for each 1-itemset in db. The idea of using *tidlists* was first discussed in [15] in order to count the support of candidate k-itemsets. A *tidlist* for an itemset X is an ordered list (ascending or descending) of the transaction identifiers (TID) of the transactions in which the items are present. The support of an itemset X is the length of the corresponding *tidlist*. It is assumed that the transactions are sorted according to TIDs and thus the created *tidlists* are also sorted in the same order of TIDs.

The second part of the algorithm (procedure *initial_pruning* in Figure 2) deals with the 1-itemsets whose support is 0 in db but large in DB. In this case, for an itemset X, it is by definition true that $support_{DB+db}(X) = support_{DB}(X)$. If X was previously small in DB, then it is also small in $DB + db$ since its support has not

UWEP$(DB, db, L_{DB}, |DB|, |db|, minsup)$**;**

C^1_{db} = all 1-itemsets in db whose support is greater than 0
$PruneSet = L^1_{DB} - C^1_{db}$
$initial\_pruning(PruneSet)$ %See Figure 2
$k = 1$
while $C^k_{db} \neq \emptyset$ and $L^k_{DB} \neq \emptyset$ **do begin**
  $Unchecked = L^k_{DB}$
  for all $X \in C^k_{db}$ **do**
    if X is small in db and X is large in DB **then**
      remove X from $Unchecked$
      if X is small in $DB + db$ **then**
        remove all supersets of X from L_{DB}
      else
        add X to L_{DB+db}
    end
    else if X is large both in db and DB **then begin**
      remove X from $Unchecked$
      add X to L_{DB+db} and L^k_{db}
    end
    else if X is large in db but small in DB **then begin**
      find $support_{DB}(X)$ using $tidlists$
      if X is large in $DB + db$ **then**
        add X to L_{DB+db} and L^k_{db}
    end
  for all $X \in Unchecked$ **do begin**
    find $support_{db}(X)$ using tidlists
    if X is small in $DB + db$ **then**
      remove all supersets of X from L_{DB}
    if X is large in $DB + db$ **then**
      add X to L_{DB+db}
  end
  $k = k + 1$
  $C^k_{db} = generate\_candidate(L^{k-1}_{db})$ %See Figure 3
end

Figure 1: Update of Frequent Itemsets

```
initial_pruning(PruneSet);
while PruneSet ≠ ∅ do begin
    X = first element of PruneSet
    if X is small in DB + db then
        remove X and all its supersets from L_DB and PruneSet
    else
    begin
        add the supersets of X in L_DB to the PruneSet
        add X to L_DB+db and remove X from L_DB
    end
    remove X from PruneSet
end
```

Figure 2: Initial Pruning Algorithm

changed and the number of total transactions has increased. On the other hand, if X is large in DB, we have to check whether $support_{DB}(X) \geq minsup \times |DB + db|$ or not. The itemset X could be large or small in the updated database, and we examine each case below.

In the following, we will introduce three lemmas that are useful in pruning the candidate itemsets. Their proofs can be found in [4, 7, 8, 16].

Lemma 1 *All supersets of a small itemset X in a database D are also small in D.*

Now suppose that X is small in the updated database. Then, by Lemma 1, any superset of X must also be small in the updated database. $UWEP$ differs from the previous algorithms [7, 8] at this point, by pruning all supersets of an itemset from the set of large itemsets in DB as soon as it is established to be small. In the previous algorithms, a k-itemset is only checked in the k^{th} iteration, but $UWEP$ does not wait until the k^{th} iteration in order to prune the supersets of an itemset in L_{DB} that are small in L_{DB+db}.

Definition 3.1 *Let X be a k-itemset which contains items $I_1, \ldots, I_k$. An immediate superset of X is a $(k+1)$-itemset which contains the k items in X and an additional item I_{k+1}.*

Now, suppose that X is large in the updated database. Then, we add all immediate supersets of X in L_{DB} to the $PruneSet$, which holds the itemsets that must be checked before checking the itemsets in C^1_{db}. Then, for each element in the $PruneSet$, we check whether its support exceeds the minimum support threshold. The operations of pruning and adding immediate supersets are repeated for each itemset in the $PruneSet$. So, all itemsets in L_{DB} that contain a non-existing item in db are removed from L_{DB}, and the ones that are large are added to L_{DB+db} before advancing to the first iteration. This pre-pruning step is particularly useful when the data skewness is present in the set of transactions. For example, in a supermarket, soup is probably large in winter transactions while it may be small in summer transactions.

Lines 4–33 in Figure 1 are used 1) to check whether any candidate itemset in db qualifies to be large in the whole database and to adjust their supports in L_{DB+db} and 2) to check whether any of the large itemsets in DB which are small in db qualifies to be in the set of L_{DB+db}. The two **for** loops between lines 4–33 perform these two operations. Let us investigate the first case: checking the candidates in db in the k^{th} iteration.

Lemma 2 *Let X be an itemset. If $X \notin L_{DB}$, then $X \in L_{DB+db}$ only if $X \in L_{db}$.*

Corollary 1 *Let X be an itemset. If X is small both in DB and db, then X can not be large in $DB + db$.*

Now suppose that X is a candidate k-itemset in db. If it is small in db, then we have to check whether X is in L_{DB} or not. If it is also small in DB (i.e., $X \notin L_{DB}$), X can not be a large itemset in $DB + db$ by Corollary 1. Otherwise, we have to check the support of X in $DB + db$. Since we have the support of X in DB and db in hand, we can quickly determine whether it is large or not. If $(support_{DB}(X) + support_{db}(X)) < minsup \times |DB+db|$, then X is small in $DB+db$. By Lemma 1, all supersets of X must also be small, thus they are eliminated from L_{DB}. Otherwise, X is large and we add X to L_{DB+db}. Another advantage of our algorithm occurs here by not adding X to the set of L^k_{db} to keep the candidate set smaller, which we will explain later in detail.

Now assume that a candidate k-itemset X is large in db. There are two possibilities: X is either large or small in DB.

Lemma 3 *Let X be an itemset. If $X \in L_{DB}$ and $X \in L_{db}$, then $X \in L_{DB+db}$.*

If X is large in DB, then X is also large in $DB + db$ by Lemma 3. In this case, we add the corresponding supports of X in db and DB, and put X into L_{DB+db} with the new support. If X is small in DB, we have to check whether it is large in $DB + db$ or not. However, we do not have the support of X in DB since it is not large. We can obtain it by scanning DB. In this scan, for each 1-itemset in DB, we determine its support and its *tidlist*, as explained before in this section. We will then use these *tidlists* in order to find the support of longer itemsets whenever

generate_candidate(L_{db}^{k-1});

$C_{db}^{k} = \emptyset$
for all itemsets $X \in L_{db}^{k-1}$ and $Y \in L_{db}^{k-1}$ **do**
    if $X_1 = Y_1 \wedge \cdots \wedge X_{k-2} = Y_{k-2} \wedge X_{k-1} < Y_{k-1}$ **then begin**
        $C = X_1 X_2 \ldots X_{k-1} Y_{k-1}$
        if all subsets S of C is an element of L_{db}^{k-1} **then begin**
            $tidlist_{db}(C) = tidlist_{db}(X) \cap tidlist_{db}(Y)$
            $support_{db}(C) = |tidlist_{db}(C)|$
        end
    end

Figure 3: Candidate generation procedure

they are needed. After counting the support of X in DB, we place X into L_{DB+db} if its support in $DB + db$ is larger than $minsup \times |DB + db|$.

An important issue here is to decide which candidates go to the set of large k-itemsets in db. FUP_2 [8] algorithm places all itemsets that are large in the whole database into L_{db}^{k} in the k^{th} iteration. Others [12, 16, 14] place those candidates that are large in db regardless of whether they are small or large in DB. We choose another strategy and put only those candidates into C_{db}^{k} that are large in db and $DB + db$. In other words, if a k-itemset X is large in db but small in $DB + db$, we do not place it into L_{db}^{k}. This is the most important advantage of $UWEP$ since this significantly reduces the number of candidates in db.

In $UWEP$, there is a possibility that a large k-itemset in DB may not be generated in C_{db}^{k}, since we include those candidates that are large both in db and $DB + db$. The solution is to keep the set of itemsets that must be verified against db, namely *Unchecked*, which contains the large k-itemsets in DB that are not generated in db. In the beginning of the k^{th} iteration, we place all large k-itemsets in DB to the set of *Unchecked* (line 6 in Figure 1). Whenever we check a candidate k-itemset in C_{db}^{k}, we will remove it from the set *Unchecked*. When we complete the first **for** loop between lines 7–23 in Figure 1, *Unchecked* contains the large itemsets in DB that are not verified against db. The second **for** loop is used to verify them against db. Since we do not generate them from L_{db}^{k-1}, we do not have their supports in db, therefore we have to compute their support from the *tidlists* of the individual items contained in that itemset. If the total support of any element in *Unchecked* exceeds the minimum support threshold, it is added to L_{DB+db}. Otherwise, the supersets of that itemset are removed from L_{DB} again by Lemma 1.

Figure 3 gives the candidate generation procedure that is adopted from [15].

DB			*db*	
TID	**Items**		**TID**	**Items**
1	A,C,D,E,F			
2	B,D,F			
3	A,D,E		1	A,F
4	A,B,D,E,F		2	B,C,F
5	A,B,C,E,F		3	A,C
6	B,F		4	B,F
7	A,D,E,F		5	A,B,C
8	A,B,D,F		6	A,C,D
9	A,D,F			

Table 2: Set of Transactions DB and db

For two $(k-1)$-itemsets in L_{db}^{k-1}, if the first $(k-2)$ items are the same, then a candidate k-itemset is generated from those $(k-1)$-itemsets by concatenating the last item in the second itemset to the end of the first itemset, assuming that the last item of the second itemset is greater than the last item in the first itemset. However, a candidate generated in this process is pruned from the set of candidates if any of its $(k-1)$-subsets is not large.

3.2 An Example Execution of the Algorithm

We now introduce an example that illustrates the benefits of our algorithm and compare the number of candidates generated and counted with *Apriori* and FUP_2 algorithms. We will write an itemset $\{A_1, \ldots, A_n\}$ as $A_1, \ldots, A_n$, and a pair $(itemset, support)$ refers to an itemset and its support in the corresponding set of transactions.

In Table 2, the set of transactions in DB and db are provided. $|DB| = 9, |db| = 6, |DB + db| = 15$. The minimum support threshold *minsup* is set to 0.3. Thus, an itemset X must be present in at least 3 transactions in DB, in at least 2 transactions in db, and in at least 5 transactions in $DB + db$ in order to be a large itemset.

Initially, we assume that the set of large itemsets in DB are given. In the example database DB, the sets of large k-itemsets along with their counts are as follows.

$$L_{DB}^1 = \{(A,7),(B,5),(D,7),(E,5),(F,8)\}$$
$$L_{DB}^2 = \{(AB,3),(AD,6),(AE,5),(AF,6),(BD,3),(BF,5),(DE,4),(DF,6),(EF,4)\}$$
$$L_{DB}^3 = \{(ABF,3),(ADE,4),(ADF,5),(AEF,4),(BDF,3),(DEF,3)\}$$

$L_{DB}^4 = \{(ADEF, 3)\}$

In the first step of the algorithm, db is scanned in order to find the support of 1-itemsets in db. In this scan, we generate the *tidlist* for each 1-itemset. In the example, the candidate 1-itemsets in db, along with their supports, are:

$C_{db}^1 = \{(A, 4), (B, 3), (C, 4), (D, 1), (F, 3)\}$.

Note that we do not include E in C_{db}^1 since its support is zero in db. On the other hand, E is added to the $PruneSet$ in order to check itemsets including E in L_{DB}. Since the support of E is 5 and is thus large in $DB + db$, we remove it from L_{DB} and include it in L_{DB+db} and add its supersets in L_{db}^2 to the $PruneSet$, namely AE, DE, EF. Then for each element of the $PruneSet$, we repeat the same operation. We add AE to L_{DB+db} since its support is also 5. However, the supports of DE and EF are 4, and they fail to qualify to go into L_{DB+db}. In this step, we remove DE and EF and all their supersets from L_{DB}, namely $ADE, DEF, AEF, ADEF$ (By Lemma 1). After these pruning operations, the new sets of large itemsets in DB and set of large itemsets in $DB + db$ are as follows.

$L_{DB}^1 = \{(A, 7), (B, 5), (D, 7), (F, 8)\}$

$L_{DB}^2 = \{(AB, 3), (AD, 6), (AF, 6), (BD, 3), (BF, 5), (DF, 6)\}$

$L_{DB}^3 = \{(ABF, 3), (ADF, 5), (BDF, 3)\}$

$L_{DB}^4 = \emptyset$

$L_{DB+db} = \{(E, 5), (AE, 5)\}$

In the first iteration, A, B, C, D, F are added to L_{DB+db}. We add all large 1-itemsets in db to L_{db}^1, namely A, B, C, F. We do not include D in L_{db} since it does not qualify to be large in db. After the first iteration,

$L_{db}^1 = \{(A, 4), (B, 3), (C, 4), (F, 3)\}$, and

$L_{DB+db}^1 = \{(A, 11), (B, 8), (C, 6), (D, 8), (E, 5), (F, 11)\}$

In the second iteration, we begin with the set of candidates in db,

$C_{db}^2 = \{(AB, 1), (AC, 3), (AF, 1), (BC, 2), (BF, 2), (CF, 1)\}$, and

$Unchecked = \{AB, AD, AF, BD, BF, DF\}$.

AB is found to be small in db, but large in DB. AB fails to be large in $DB + db$ since $support_{DB+db}(AB) = 4$. By Lemma 1, we remove ABF from L_{DB}. The itemset AC is large in db but small in DB. Since we do not have support of AC in DB in hand, we find AC's support in DB by intersecting the *tidlists* of A and C in DB, which is 2. ($tidlist_{DB}(A) = \{1, 3, 4, 5, 7, 8, 9\}$, $tidlist_{DB}(C) = \{1, 5\}$, their intersection is $\{1, 5\}$) Since the total support of AC is 3+2=5, AC is added to L_{DB+db} (Application of Lemma 2). AF is small in db, with a total support of 7. Therefore, AF is added to L_{DB+db}, but we do not include it in L_{db}^2. BC is large in db but small in DB. So, we compute the support of BC in DB, which is 1. The total support of BC is 3, so we do not include it in L_{DB+db} nor in L_{db}^2. BF is large both in DB and db. So it is large in $DB + db$ with a support of 7. Since CF is small both in DB and db, it is small in $DB + db$ by Corollary 1. Up to this point, we checked each element of C_{db}^2, but not all elements of L_{DB}^2. At this moment,

$Unchecked = \{AD, BD, DF\}$.

We did not compute the supports of these itemsets in db since we did not include D in L_{db}^1, so for each of them we have to compute its support in db using *tidlists* of the items contained in the itemset. Supports of AD, BD, DF in db are 1, 0, 0, respectively. We find the total support of these itemsets by adding their supports in DB and db. In our case, the supports of AD, BD, DF in $DB + db$ are 7,3,6, respectively. AD and DF are found to be large in the whole database, so we add them to L_{DB+db}. Since BD is small in the whole database, we have to remove its supersets from L_{DB}, namely BDF.

At the end of the second iteration, we find that

$L_{db}^2 = \{(AC, 3), (BF, 2)\}$, and

$L_{DB+db}^2 = \{(AC, 5), (AD, 7), (AE, 5), (AF, 7), (BF, 7), (DF, 6)\}$

Before proceeding to third iteration, we compute

$C_{db}^3 = \emptyset$

$Unchecked = \{ADF\}$

Since, $C_{db}^3 = \emptyset$, we proceed with checking the elements of $Unchecked$. The support of ADF is 0 in db and its support in $DB + db$ is 5. Thus, we add ADF into L_{DB+db} and finish the update operation. The final set of large itemsets in $DB + db$ are:

$L_{DB+db}^1 = \{(A, 11), (B, 8), (C, 6), (D, 8), (E, 5), (F, 11)\}$

$L_{DB+db}^2 = \{(AC, 5), (AD, 7), (AE, 5), (AF, 7), (BF, 7), (DF, 6)\}$

$L_{DB+db}^3 = \{ADF, 5\}$

3.3 Completeness and Efficiency of the Algorithm

The algorithm $UWEP$ presented in Figure 1 correctly and completely computes the set of large itemsets in the updated database.

Lemma 4 *Given a set of old transactions (DB), a set of new transactions (db), and a set of itemsets L_{DB} which are large over DB, the algorithm in Figure 1 discovers all the large itemsets over $DB + db$ correctly.*

Proof. Let X be a k-itemset. By Corollary 1, X must be large in either DB or db, or both. Thus, in order to compute large itemsets in $DB + db$, we have to check large itemsets in DB against db, and large itemsets in db against DB. Let us investigate these two cases:

Case 1: Checking for all $X \in L_{DB}$ against db

In the initial pruning step (algorithm in Figure 2), all itemsets X in L_{DB} such that $support_{db}(X) = 0$ are checked. If X is small in $DB + db$, all of its supersets are removed from consideration since they can not also be large in $DB+db$ by Lemma 1. If X is large in $DB + db$, we put it into L_{DB+db}, and its immediate supersets into the $PruneSet$. This process is repeated until the $PruneSet$ is empty. In the end, any large itemset in DB whose support in db is zero is checked against db. Thus, before the **while** loop on line 5 in Figure 1, L_{DB} contains the large itemsets in DB whose support in db is greater than zero, and L_{DB+db} contains all large itemsets containing the items whose support is zero in db.

In the k^{th} iteration, $Unchecked$ is initialized to the set of large k-itemsets in DB. Any element of $Unchecked$ that is present in C^k_{db} is checked on lines 9 and 16. If an itemset in $Unchecked$ does not exist in C^k_{db}, then the second **for** loop counts their support in db, and decides which of them are large in the updated database. Therefore, all elements of L^k_{DB} are checked against db, and the ones that are large in $DB + db$ are determined.

Case 2: Checking for all $X \in L_{db}$ against DB

In the $UWEP$ algorithm, C^k_{db} contains possibly large itemsets over $DB + db$, instead of possibly large itemsets in db. In the first **for** loop, only those in C^k_{db} that are large over $DB + db$ are put into L^k_{db} (lines 17 and 22). If a k-itemset X is large in db but not in $DB + db$, then it is a waste of effort to put it into L^k_{db} because it is not possible that a superset of X is large in $DB + db$ by Lemma 1. Since any superset of X is certainly small in $DB + db$, we do not need to check whether any superset of X is large in db or not. Because, even if a superset of X is large in db, it will be certainly small in the updated database. Since our purpose is to generate the large itemsets in $DB + db$, putting X into L^k_{db} is a waste of effort, and reduces the performance of the algorithm.

Thus, the first **for** loop checks for all the itemsets in C^k_{db} against DB. If any large itemset in C^k_{db} is also large in DB, then we simply put it into L^k_{DB+db} on line 17 by Lemma 3. If it is small in DB, then we count its support in DB using $tidlists$, and decide to put it into L^k_{DB+db} and L^k_{db} on line 22. Therefore, all elements large in db are checked against DB.

As a consequence of Case 1 and Case 2, the algorithm $UWEP$ computes the large itemsets in $DB + db$ correctly and completely. □

Lemma 5 *The number of candidates generated and counted by the algorithm $UWEP$ in Figure 1 is minimum.*

Proof. The only candidate generation operation is over db. Therefore, to prove that the number of candidates generated is minimum, we only deal with the set

		Apriori	FUP_2	*UWEP*
Iteration 1	Candidates generated in db	6	6	5
	Candidates counted in DB	–	1	1
	Candidates counted in db	–	6	6
	Total # of candidates counted	6	7	7
Iteration 2	Candidates generated in db	15	15	6
	Candidates counted in DB	–	2	2
	Candidates counted in db	–	9	9
	Total # of candidates counted	15	11	11
Iteration 3	Candidates generated in db	1	1	0
	Candidates counted in DB	–	0	0
	Candidates counted in db	–	1	1
	Total # of candidates counted	1	1	1

Table 3: Number of candidates generated and counted in the example database

of candidates in *db*. C^1_{db} contains only the itemsets whose support is greater than zero. This is the minimum bound because to decide which of the itemsets is large in $DB+db$, we have to know at least the support of each item in *db*. Therefore, C^1_{db} contains the minimum number of candidates. In the k^{th} iteration, we put only the itemsets that are large over $DB + db$ into L^k_{db}. The completeness of this operation is shown in the proof of Lemma 4. We have to put those itemsets that are large over $DB + db$ into L^k_{db} because, their supersets are possibly large over $DB + db$, and we have to check them in order to complete the update operation. Since, we do not include any other itemset in L^k_{db}, this is the minimum bound for a level-wise algorithm. As explained in Figure 3, the candidate set C^{k+1}_{db} is computed from L^k_{db}, so the number of candidates generated in *db* is also minimum.

Since the candidates generated in *db* is minimum, the number of candidates counted in *db* is also minimum. The only remaining issue is the number of candidates counted in DB. Since, we only scan DB in order to find the support of an itemset that is not large in DB, this is also the lower bound. Hence, the number of candidates counted is minimum. □

3.4 Comparison with the Existing Algorithms

Table 3 shows the number of candidates generated and counted by the *Apriori*, FUP_2, and *UWEP* algorithms over the example database given in Table 2. It is worth noting that the *Apriori* algorithm re-runs over the whole set of transactions, and therefore counting candidates over DB and *db* is irrelevant.

As Table 3 shows, our algorithm generates a much smaller number of candidate sets than *Apriori* or FUP_2. Especially for the second iteration, $UWEP$ achieves $\frac{15-6}{15} = 60\%$ improvement over the two algorithms. Overall, $UWEP$ has a performance improvement of $\frac{22-11}{22} = 50\%$ over the two algorithms. Note that, the candidates counted by $UWEP$ is the same as FUP_2, but the number of candidates generated by FUP_2 is larger than the ones generated by $UWEP$.

In case of running the Partition Update algorithm (PU) of [12], the number of candidates counted is much greater than that of $UWEP$. In db there are four large 1-itemsets and three large 2-itemsets. In order to find them, 11 candidates are generated and counted in db. Since we know the support of four of them in DB, PU has to count only 3 candidates on DB. However, it has to count 17 large itemsets of DB over db since their supports in db are not available. Therefore, a total of 3 itemsets are counted in DB and $11 + 17 = 28$ itemsets are counted in db. On the other hand, $UWEP$ counts 3 candidates in DB and $6 + 9 + 1 = 16$ candidates in db. Even only one scan of db and DB is enough for counting itemsets, the number of candidates counted is very high in comparison to the $UWEP$ algorithm, where $UWEP$ achieves a $\frac{28-16}{28} = 43\%$ improvement over *Partition Update* algorithm in the number of candidates counted in db.

$UWEP$ also yields a smaller candidate set in comparison to other update algorithms. FUP_2 [8], which is a generalization of FUP [7], examines a large k-itemset only in the k^{th} iteration and generates the candidate set C_{db}^k from the set of large $(k-1)$-itemsets in the updated database. Then, by means a few optimizations, it prunes some of the candidates and counts the remaining over DB and db. $PartitionUpdate(PU)$ finds the set of large itemsets in db and then checks large itemsets in DB against db and vice versa. In this sense, it generates the candidate set C_{db}^k from the set of large $(k-1)$-itemsets in the incremental database. The algorithms in [16, 14] generate the candidate set C_{db}^k from the set of large itemsets L_{db}^{k-1}, with the same number of candidates in PU. On the other hand, $UWEP$ generates the set of candidate set C_{db}^k from the set of itemsets that are large both in db and in the updated database. This results in a much smaller candidate set in comparison to the mentioned algorithms.

4 Experimental Results

In order to measure the performance of $UWEP$, we conducted several experiments using the synthetic data introduced in [4]. Before proceeding to the details of the experiments, we would like to present the parameters used in the data generation procedure.

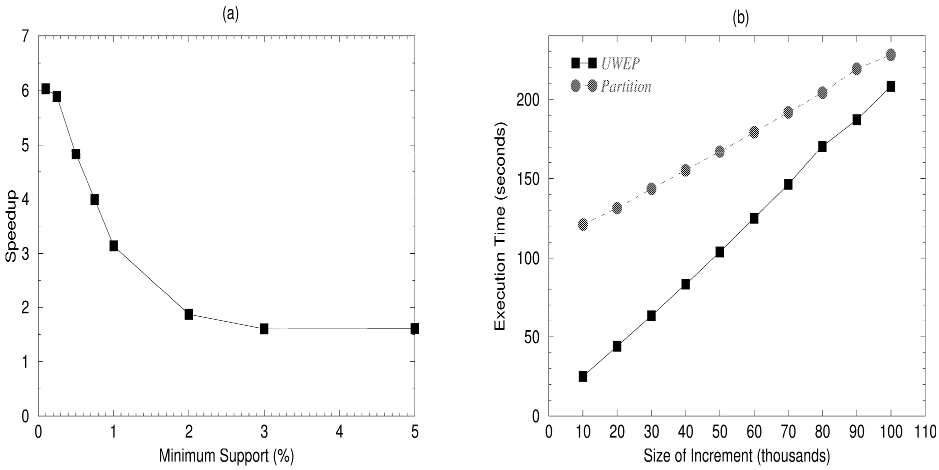

Figure 4: a) Speedup by $UWEP$ over $Partition$ algorithm b) Execution times of $UWEP$ vs. $Partition$ algorithms

The synthetic data generated in [4] mimics the transactions in the retailing environment. Our synthetic data generation procedure is a simple extension of the method used in [4]. We generated a transaction database of size $2 \times |DB|$, where the first $|DB|$ transactions were placed into the set of old transactions. From the remaining transactions, we took the first $\frac{|DB|}{10}$ transactions for the first incremental database, took the first $\frac{2 \times |DB|}{10}$ transactions for the second incremental database, and so on. Since all transactions are generated using the same statistical pattern, the transactions in the incremental database exhibit the same regularities in the original database. In the experiments, we used the following parameters. Number of maximal potentially large itemsets=$|L|$=2000, number of transactions=$|D|$=200,000, average size of the transactions=$|T|$=10, number of items=N=1000 and average size of the maximal potentially large itemset=$|I|$=4. We follow the notation $Tx.Iy.Dm.dn$ used in [7] to denote databases in which $|DB| = m$ thousands, $|db| = n$ thousands, $|T| = x$ and $|I| = y$. Readers not familiar with these parameters are referred to [4].

For the first experiment, we measured the speedup gained by $UWEP$ over rerunning Partition algorithm [15]. We have chosen $Partition$ since the same data structures and methodology for finding large itemsets are used in both algorithms. Figure 4a shows the results for $T10.I4.D100.d10$. The y-axis in the graph represents $\frac{Execution\ Time\ of\ Partition}{Execution\ Time\ of\ UWEP}$, and x-axis represents different support levels. As it can be seen from Figure 4a, $UWEP$ performs much better than re-running $Partition$. Figure 4a shows that at lower support levels, the speedup gain of $UWEP$ increases from 1.5 to 6 as the minimum support decreases from 3% to 0.1%. For support

	minsup	(1) *PU*	(2) FUP_2	(3) *UWEP*	Imprv. on (1)	Imprv. on (2)
Candidates Generated in *db*	0.75%	100177	99797	53759	46%	46%
	0.5%	146431	161746	90884	38%	44%
	0.1%	351652	511717	239662	32%	53%
Candidates Counted in *db*	0.75%	100341	53762	53762	46%	–
	0.5%	147740	91417	91417	38%	–
	0.1%	379352	251963	251963	34%	–
Candidates Counted in *DB*	0.75%	206	187	187	9%	–
	0.5%	1612	571	571	65%	–
	0.1%	28040	8675	8675	69%	–
Candidates Counted Totally	0.75%	100547	53949	53949	46%	–
	0.5%	149352	91988	91988	38%	–
	0.1%	407392	260638	260638	36%	–

Table 4: Number of candidates generated and counted on synthetic data

levels higher than 3%, the speedup seems to converge to 1.5.

In the second experiment, we measured the effect of the size of the incremental database on the execution time of the algorithms. Figure 4b shows the execution times for $UWEP$ and $Partition$ algorithms for $T10.I4.D100.dn$, where n varies from 10 to 100, with the minimum support set to 0.5%. For smaller sizes of the incremental database, $UWEP$ achieves a much better performance than $Partition$. As the size of the new transactions increases, the execution time of $UWEP$ gets closer to that of the $Partition$. On the other hand, despite adding 100% transactions, $UWEP$ still performs better than re-running $Partition$. One interesting feature of $UWEP$ is that its execution time is linear to the size of incremental database under a specified minimum support. In this sense, $UWEP$ can scale up linearly to the size of incremental database whatever the minimum support is.

The third experiment investigates the number of candidates generated and counted for the three update algorithms, *Partition Update*, FUP_2, and $UWEP$. For this experiment, we generated an increment database containing a smaller number of items than that in the original database. Table 4 shows the number of generated and counted candidates for three algorithms for $T10.I4.D100.d10$ with 900 items in the new set of transactions. The reason behind smaller number of items in the incremental database is to see the effects of data skewness in the update of large itemsets. As Table 4 shows, $UWEP$ generates a much smaller number of candidates in comparison to the other two algorithms, between 32%–53% of those generated by FUP_2 and *Partition Update*. The number of candidates counted by $UWEP$ is exactly the same as that by FUP_2. However, the *Partition Update* algorithm counts more candidates than $UWEP$ counts, up to 69%. The results indicate that $UWEP$ performs much better than the other two algorithms when some of the large itemsets in DB are absent in db, thus in $DB + db$, as well.

5 Conclusion

We presented an efficient algorithm, $UWEP$, for updating large itemsets when a set of new transactions are added to the database of transactions. We proved that $UWEP$ generates and counts the possible minimum number of candidates for a level-wise algorithm. The major advantages of $UWEP$ over the previously proposed update algorithms are the facts that it prunes the supersets of a large itemset in DB as soon as it is known to be small in the updated database, without waiting until the k^{th} iteration. Moreover, $UWEP$ generates the set of candidate set C_{db}^k from the set of itemsets that are large both in db and in the updated database. As shown in Section 4, this methodology yields a much smaller candidate set especially when the set of new transactions does not contain some of the old large itemsets.

We have conducted experiments on synthetic data and found that $UWEP$ achieves a better performance than re-running *Partition* [15] algorithm over the whole set of transactions. Naturally, this is true for re-running other algorithms like *Apriori* [4] since the previous work is discarded and the entire database is scanned again. Especially for the smaller support levels, the speedup obtained by $UWEP$ is very large. Moreover, experiments on the number of candidates generated and counted show that $UWEP$ outperforms *Partition Update* and FUP_2 algorithms.

There are several directions for future research. We are in the process of extending our performance experiments and compare speedup of $UWEP$ against the other update algorithms [14, 16] to gain a better insight about the performance of $UWEP$. We also plan to extend our algorithm to handle deleted or modified transactions as well.

References

[1] Charu C. Aggarwal and Philip S. Yu. Online generation of association rules. In *Proceedings of ICDE'98*, pages 402–411, February 1998.

[2] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD'93*, pages 207–216, May 1993.

[3] Rakesh Agrawal, Heikki Mannila, Ramakrishnan Srikant, Hannu Toivonen, and A. Inkeri Verkamo. Fast discovery of association rules. In Usama Fayyad, Gregory Piatetsky-Shapiro, Padhraic Smyth, and Ramaswamy Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

[4] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules. In *Proceedings of VLDB'94*, pages 487–499, 1994.

[5] Roberto J. Bayardo. Efficiently mining long patterns from databases. In *Proceedings of ACM SIGMOD'98*, pages 85–93, 1998.

[6] Sergey Brin, Rajeev Motwani, Jeffrey D. Ullman, and Sergey Tsur. Dynamic itemset counting and implication rules for market basket data. In *Proceedings of ACM SIGMOD'97*, pages 255–264, June 1997.

[7] David Wai-Lok Cheung, Jiawei Han, Vincent T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental update technique. In *Proceedings of ICDE'96*, pages 106–114, February 1996.

[8] David Wai-Lok Cheung, Sau Dan Lee, and Benjamin Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of DASFAA'97*, pages 185–194, April 1997.

[9] Christian Hidber. Online association rule mining. Technical Report TR-98-033, International Computer Science Institute, Berkeley, September 1998.

[10] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules in relational databases. In *Proceedings of ICDE'95*, pages 25–33, 1995.

[11] Heikki Mannila, Hannu Toivonen, and A. I. Verkamo. Efficient algorithms for discovering association rules. In *Proceedings of KDD'94*, pages 181–192, July 1994.

[12] Edward Omiecinski and Ashok Savasere. Efficient mining of association rules in large dynamic databases. In *Proceedings of BNCOD'98*, pages 49–63, 1998.

[13] Jong Soo Park, Ming-Syan Chen, and Philip S. Yu. An effective hash based algorithm for mining association rules. In *Proceedings of ACM SIGMOD'95*, pages 175–186, May 1995.

[14] N. L. Sarda and N. V. Srinivas. An adaptive algorithm for incremental mining of association rules. In *Proceedings of DEXA Workshop'98*, pages 240–245, 1998.

[15] A. Savasere, Edward Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proceedings of VLDB'95*, pages 432–444, September 1995.

[16] Shiby Thomas, Sreenath Bodagala, Khaled Alsabti, and Sanjay Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proceedings of KDD'97*, pages 263–266, 1997.

[17] Hannu Toivonen. Sampling large databases for association rules. In *Proceedings of VLDB'96*, pages 134–145, September 1996.