

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 School
Baruch College, CUNY
tansel@baruch.cuny.edu

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

Abstract

We present an efficient algorithm (*UWEP*) for updating large itemsets when new transactions are added to the set of old transactions. *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. It differs from the other update 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

Association rules aim at discovering the patterns of occurrences of attributes in a database. The problem of discovering association rules was first explored in [1] on supermarket basket data. Many efficient algorithms have been proposed for finding the frequent patterns in a database [1, 2, 8, 10].

When new transactions are added to the old transaction database, discovering association rules in the updated database efficiently is an important issue. The straightforward solution is to re-run an algorithm, say *Apriori* [2], on the updated database. 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 [4, 5, 7, 9, 11]. These algorithms take the set of association rules in the old database into account, and use this knowledge 1) to re-

move itemsets that no longer exist in updated database, and 2) to add new itemsets which were not in the set of old transactions but now exist in the updated database.

In this paper, we propose an algorithm called *UWEP* (Update With Early Pruning) that follows the approaches of *FUP₂* [5] and *Partition Update* [7] algorithms, and provides an improvement over them. 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 candidate itemsets in order to determine the new set of large itemsets. Moreover, it prunes an itemset that will become small from the set of generated candidates as early as possible by a look-ahead pruning strategy. Thus, look-ahead pruning results in a much smaller number of candidates in the computation of new large itemsets.

The rest of the paper is organized as follows. In Section 2, problems of discovering and updating association rules, and related algorithms are presented. Section 3 presents the *UWEP* algorithm in detail. 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 [1, 2]. Let $I = \{I_1, \dots, I_m\}$ be a set of items, D be a set of transactions involving items where 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 . The support of an itemset X in D , $support_D(X)$, is the number of transactions in D that contain X . An itemset X is called a *large itemset* if $support_D(X)$ exceeds a minimum support threshold, and a *small itemset* otherwise. An implication of the form $X \Rightarrow Y$, where $X \subset I$, $Y \subset I$, and $X \cap Y = \emptyset$, is called an *association rule*. The problem of finding association rules can be decomposed into two parts [1, 2]:

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

KDD-99 San Diego CA USA

Copyright ACM 1999 1-58113-143-7/99/08...\$5.00

Notation	Definition
DB	Set of old transactions
db	Set of new transactions
$DB + db$	Set of transactions after update
$ A $	Number of transactions in A
$minsup$	Minimum support threshold
$support_A(X)$	Support of X in A
$tidlist_A(X)$	Transaction list of X in A
C_A^k	Candidate k -itemsets in A
L_A^k	Large k -itemsets in A
$PruneSet$	Large itemsets in DB with 0 support in db
$Unchecked$	Large k -itemsets in DB that are not counted in db

Table 1: Notations Used in the Paper

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.

The second subproblem is straightforward. However, discovering large itemsets is a non-trivial issue, where the efficiency of an algorithm strongly depends on the size of the candidate set. Therefore, we will concentrate on the update of large itemsets.

2.2 Update of Large Itemsets

Table 1 summarizes the notations used in the remainder of the paper. We also use the terminology of [1, 2].

Updating association rules was first introduced in [4]. Given $DB, db, |DB|, |db|, minsup$ and L_{DB} , the problem of updating large itemsets is to find the set L_{DB+db} of large itemsets in $DB + db$.

The *FUP* algorithm proposed by Cheung et al. [4] works iteratively and its framework is similar to *Apriori* [2] and *DHP* [8]. *FUP₂* [5] is a generalization of the *FUP* algorithm that handles insertions to and deletions from an existing set of transactions. In [9, 11], the concept of *negative border*, that was introduced in [12], is used to compute the new set of large itemsets in the updated database. A recent study [7] is based on the *Partition* algorithm of [10].

3 Update with Early Pruning

3.1 Description of the Algorithm

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 $DB + db$.

Algorithm *UWEP* has the following five steps:

```

UWEP( $DB, db, L_{DB}, |DB|, |db|, minsup$ );
1  $C_{db}^1 =$  all 1-itemsets in  $db$  with support  $> 0$ 
2  $PruneSet = L_{DB}^1 - C_{db}^1$ 
3 initial_pruning( $PruneSet$ ) % Figure 2
4  $k = 1$ 
5 while  $C_{db}^k \neq \emptyset$  and  $L_{DB}^k \neq \emptyset$  do
6    $Unchecked = L_{DB}^k$ 
7   for all  $X \in C_{db}^k$  do
8     if  $X$  is small in  $db$  and  $X$  is large in  $DB$  then
9       remove  $X$  from  $Unchecked$ 
10    if  $X$  is small in  $DB + db$  then
11      remove all supersets of  $X$  from  $L_{DB}$ 
12    else
13      add  $X$  to  $L_{DB+db}$ 
14    else if  $X$  is large both in  $db$  and  $DB$ 
15      remove  $X$  from  $Unchecked$ 
16      add  $X$  to  $L_{DB+db}$  and  $L_{db}^k$ 
17    else if  $X$  is large in  $db$  but small in  $DB$  then
18      find  $support_{DB}(X)$  using tidlists
19      if  $X$  is large in  $DB + db$  then
20        add  $X$  to  $L_{DB+db}$  and  $L_{db}^k$ 
21    for all  $X \in Unchecked$  do
22      find  $support_{db}(X)$  using tidlists
23      if  $X$  is small in  $DB + db$  then
24        remove all supersets of  $X$  from  $L_{DB}$ 
25      if  $X$  is large in  $DB + db$  then
26        add  $X$  to  $L_{DB+db}$ 
27     $k = k + 1$ 
28  $C_{db}^k = generate\_candidate(L_{db}^{k-1})$ 

```

Figure 1: Algorithm *UWEP*: Update of Large Itemsets

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 in 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 , as in [10]. A *tidlist* for an itemset X is an ordered list 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*.

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 previ-

```

initial_pruning(PruneSet);
1 while PruneSet  $\neq \emptyset$  do
2   X = first element of PruneSet
3   if X is small in DB + db then
4     remove X and its supersets from LDB
5     remove X and its supersets from PruneSet
6   else
7     add supersets of X in LDB to PruneSet
8     add X to LDB+db and remove X from LDB
9     remove X from PruneSet

```

Figure 2: Initial Pruning Algorithm

ously small in *DB*, then it is also small in *DB* + *db*. 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.

In the following, we will introduce three lemmas that are useful in pruning the candidate itemsets. Their proofs can be found in [2, 4, 5, 11].

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 [4, 5] 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 1 *Let X be a k-itemset which contains items I_1, \dots, 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 examined before checking the itemsets in *C_{db}¹*. Then, for each element in the *PruneSet*, we check whether its support exceeds the minimum support threshold. 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 data skewness is present in the set of transactions.

Lines 4–28 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 4–28 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.*

Let *X* be a candidate *k*-itemset in *db*. If *X* is small in *db*, then we have to check whether *X* is in *L_{DB}* or not. If $X \notin L_{DB}$, *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}*.

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 put *X* into *L_{DB+db}* with the total support. If *X* is small in *DB*, we have to check whether it is large in *DB* + *db* or not. However, we do not know $support_{DB}(X)$. We can obtain it by scanning *DB*. In this scan, for each 1-itemset in *DB*, we find its *tidlist*, and then use these *tidlists* in order to find the *tidlists* and supports of longer itemsets. After counting the support of *X* in *DB*, we place *X* into *L_{DB+db}* if $support_{DB+db}(X) \geq minsup \times |DB + db|$.

An important issue here is to decide which candidates go to the set of large *k*-itemsets in *db*. *FUP₂* [5] algorithm places all itemsets that are large in the whole database into *L_{db}^k* in the *k*th iteration. Others [7, 11, 9] 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. The second **for** loop is used to verify them against *db*.

The candidate generation procedure is similar to that in [10], so we will not repeat it here. For an example execution of the algorithm *UWEP* and its performance comparison with the existing algorithms, please see [3].

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.

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

For the proofs, please refer to [3].

4 Experimental Results

In order to measure the performance of *UWEP*, we conducted several experiments using the synthetic data introduced in [2]. 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. 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 [4] to denote databases in which $|DB| = m$ thousands, $|db| = n$ thousands, $|T| = x$ and $|I| = y$.

For the first experiment, we measured the speedup gained by *UWEP* over rerunning *Partition* [10]. We have chosen *Partition* since the same data structures and methodology for finding large itemsets are used in both algorithms. Figure 3a shows the results for $T10.I4.D100.d10$. The *y*-axis in the graph represents $\frac{\text{Execution Time of Partition}}{\text{Execution Time of UWEP}}$, and *x*-axis represents different support levels. As it can be seen from Figure 3a, *UWEP* performs better than re-running *Partition*, a speedup between 1.5 to 6.

In the second experiment, we measured the effect of the size of the incremental database on the execution time of the algorithms. Figure 3b 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*. Despite adding 100% transactions, *UWEP* still performs better than re-running *Partition*. One interesting feature of *UWEP*

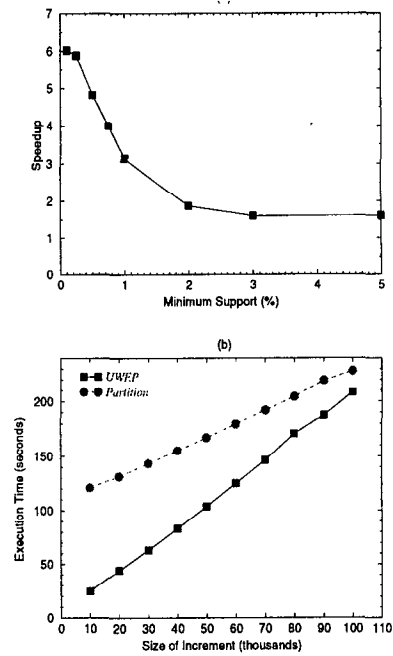


Figure 3: **a)** Speedup by *UWEP* over *Partition* algorithm **b)** Execution times of *UWEP* vs. *Partition* algorithms (*minsup* = 0.5%)

is that its execution time is linear to the size of incremental database under a specified minimum support.

The third experiment investigates the number of candidates generated and counted for the three update algorithms, *Partition Update*, *FUP₂*, and *UWEP*. For this experiment, we generated an increment database containing a smaller number of items than that in the original database. Table 2 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 to see the effects of data skewness in the update of large itemsets. As Table 2 shows, *UWEP* generates between 32%–53% fewer number of candidates than those generated by *FUP₂* and *Partition Update*. The *Partition Update* algorithm counts more candidates than *UWEP* or *FUP₂* counts, up to 69%.

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. The major advantage of *UWEP* over the previously proposed update algorithms is 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 candidate set C_{db}^k from the set of itemsets that are large both in

	<i>minsup</i>	(1)	(2)	(3)	Imprv. on (1)	Imprv. on (2)
		<i>PU</i>	<i>FUP₂</i>	<i>UWEP</i>		
Candidates Generated in <i>db</i>	0.75%	100177	99797	53759	46%	46%
	0.5%	146431	161746	90884	38%	44%
	0.1%	351652	511717	239662	32%	53%
Candidates Counted in <i>db</i>	0.75%	100341	53762	53762	46%	–
	0.5%	147740	91417	91417	38%	–
	0.1%	379352	251963	251963	34%	–
Candidates Counted in <i>DB</i>	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 2: Number of candidates generated and counted on synthetic data

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 conducted experiments on synthetic data and found that *UWEP* achieves a better performance than re-running *Partition* [10] over the whole set of transactions. Naturally, this is true for re-running other algorithms like *Apriori* [2] since the previous work is discarded and the entire database is scanned again. Moreover, experiments on the number of candidates generated and counted show that *UWEP* outperforms *Partition Update* and *FUP₂* algorithms.

UWEP may also be used in other knowledge discovery techniques that compute large itemsets, such as variations of association rules [6]. We will investigate this avenue in a separate study. We also plan to incorporate updates and deletions to *UWEP*.

References

- [1] R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. ACM SIGMOD'93*, pages 207–216, May 1993.
- [2] R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proc. VLDB'94*, pages 487–499, 1994.
- [3] N. F. Ayan, A. U. Tansel and E. Arkun. An efficient algorithm to update large itemsets with early pruning. Technical Report BU-CEIS-9908, Dept. of CEIS, Bilkent University, June 1999.
- [4] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong. Maintenance of discovered association rules in large databases: An incremental update technique. In *Proc. ICDE'96*, pages 106–114, February 1996.
- [5] D. W. Cheung, S. D. Lee, and B. Kao. A general incremental technique for maintaining discovered association rules. In *Proc. DASFAA '97*, pages 185–194, April 1997.
- [6] R. T. Ng, L. V. Lakshmanan, J. Han, and A. Pang. Exploratory mining and pruning optimizations of constrained association rules. In *Proc. ACM SIGMOD'98*, pages 13–24, June 1998.
- [7] E. Omiecinski and A. Savasere. Efficient mining of association rules in large dynamic databases. In *Proc. BNCOD'98*, pages 49–63, 1998.
- [8] J. S. Park, M. S. Chen, and P. S. Yu. An effective hash based algorithm for mining association rules. In *Proc. ACM SIGMOD'95*, pages 175–186, May 1995.
- [9] N. L. Sarda and N. V. Srinivas. An adaptive algorithm for incremental mining of association rules. In *Proc. DEXA Workshop'98*, pages 240–245, 1998.
- [10] A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. VLDB'95*, pages 432–444, September 1995.
- [11] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An efficient algorithm for the incremental updation of association rules in large databases. In *Proc. KDD'97*, pages 263–266, 1997.
- [12] H. Toivonen. Sampling large databases for association rules. In *Proc. VLDB'96*, pages 134–145, September 1996.