Enhancing SWF for Incremental Association Mining by Itemset Maintenance

Chia-Hui Chang and Shi-Hsan Yang

Department of Computer Science and Information Engineering,
National Central University, Chung-Li, Taiwan 320
chia@csie.ncu.edu.tw, carry@db.csie.ncu.edu.tw

Abstract. Incremental association mining refers to the maintenance and utilization of the knowledge discovered in the previous mining operations for later association mining. Sliding window filtering (SWF) is a technique proposed to filter false candidate 2-itemsets by segmenting a transaction database into several partitions. In this paper, we extend SWF by incorporating previously discovered information and propose two algorithms to boost the performance for incremental mining. The first algorithm FI_SWF (SWF with Frequent Itemset) reuses the frequent itemsets of previous mining task as FUP2 to reduce the number of new candidate itemsets that have to be checked. The second algorithm CI_SWF (SWF with Candidate Itemset) reuses the candidate itemsets from the previous mining task. Experiments show that the new proposed algorithms are significantly faster than SWF.

1 Introduction

Association rule mining from transaction database is an important task in data mining. The problem, first proposed by [2], is to discover associations between items from a (static) set of transactions. Many interesting works have been published to address this problem [9, 8, 11, 1, 6, 7, 10]. However, in real life applications, the transaction databases are changed with time. For example, Web log records, stock market data, and grocery sales data, etc. In these dynamic databases, the knowledge previously acquired can facilitate successive discovery processes, or become obsolete due to behavior change. How to maintain and reuse such knowledge is called incremental mining.

Incremental mining was first introduced in [4] where the problem is to update the association rules in a database when new transactions are added to the database. The proposed algorithm, FUP, stores the counts of all the frequent itemsets found in a previous mining operation and iteratively discover $(k+1)$-itemsets as the Apriori algorithm [2]. An extension to the work is addressed in [5] where the authors propose the FUP2 algorithm for updating the existing association rules when transactions can be deleted from the database. In essence, FUP2 is equivalent to FUP for the case of insertion, and is, however, a complementary algorithm of FUP for the case of deletion.

In the recent years, several algorithms, including the MAAP algorithm [13] and the PELICAN algorithm [12], are proposed to solve the problem incremental mining. Both MAAP and PELICAN are similar to FUP2 algorithm, but they only focus on how to maintain maximum frequent itemsets when the database are updated. In other words, they do not care non-maximum frequent itemsets, therefore, the counts of non-maximum frequent itemsets can not be calculated. The difference of these two algorithms is that MAAP calculates maximum frequent itemsets by Apriori-based framework while PELICAN calculates maximum frequents itemsets based on vertical database format and lattice decomposition.

As we know, Apriori-like algorithms tend to suffer from two inherent problems, namely (1) the occurrence of a potentially huge set of candidate itemsets, and (2) the need of multiple scans of databases. To remedy these problems, Lee et al. proposed a sliding window filtering technique for incremental association mining [3]. By partitioning a transaction database into several partitions, SWF employs the minimum support threshold in each partition to filter unnecessary candidate itemsets such that the information in the prior partition is selectively carried over toward the generation of candidate itemsets in the subsequent partitions. It is shown that SWF maintains a set of candidate itemsets that is very close to the set of frequent itemsets and outperforms FUP2 in a great deal. However, SWF does not utilize previously discovered frequent itemsets which are used to improve incremental mining in FUP2. In this paper, we show two simple extensions of SWF which significantly improve the performance by incorporate previously discovered information such as the frequent itemsets or candidate 2-itemsets from the old transaction database.

The rest of this paper is organized as follows. Preliminaries on association mining are illustrated in Section 2. The idea of extending SWF with previously discovered information is introduced in Section 3. Experiments are presented in Section 4. Finally, the paper concludes with Section 5.

2 PRELIMINARIES

Let $I=\{i_1, i_2, ..., i_m\}$ 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 $T \subseteq I$. Note that the quantities of items bought in a transaction are not considered, meaning that each item is a binary variable representing if an item was bought. Each transaction is associated with an identifier, called TID. Let X be a set of items. A transaction T is said to contain X if and only if $X \subseteq T$. An association rule is an implication of the form $X \rightarrow Y$, where $X \subset I$, $Y \subset I$ and $X \cap Y = \emptyset$. The rule $X \rightarrow Y$ holds in the transaction set D with confidence c if $c\%$ of the transactions in D that contain X also contain Y. The rule $X \rightarrow Y$ has support s in the transaction set D if $s\%$ of transactions in D contain $X \cup Y$. For a given pair of confidence and support thresholds, the problem of association rule mining is to find out all the association rules that have confidence and support greater than the corresponding thresholds. This problem can be reduced to the problem of finding all frequent itemsets for the same support threshold [2].

For a dynamic database, old transactions ($\triangle^-$) are deleted from the database

D and new transactions ($\triangle^+$) are added. Naturally, $\triangle^- \subseteq D$. Denote the updated database by D', therefore $D' = (D - \triangle^-) \cup \triangle^+$. We also denote the set of unchanged transactions by $D^- = D - \triangle^-$.

2.1 The SWF algorithm

In [3], Lee, et al., proposed a new algorithm, called SWF algorithm for incremental mining. The key idea of SWF algorithm is to compute a set of candidate 2-itemsets as close to L_2 as possible. The concept of algorithm is described as follows. Suppose the database is divided into n partitions $P_1, P_2, \ldots, P_n$, and processed one by one. For each frequent itemset I, there must exist some partition P_k such that I is frequent from partition P_k to P_n. If we know a candidate 2-itemset is not frequent from the first partition where it presents frequent to current partition P_k, we can delete this itemset even though we don't know if it is frequent from P_{k+1} to P_n. If somehow this itemset is indeed frequent, it must be frequent in some partition P'_k, where we can add it to the candidate set again.

The SWF algorithm maintains a list of 2-itemsets CF that is frequent for the lasted partitions as described below. For each partition P_i, SWF adds frequent 2-itemsets (together with its starting partition P_i and supports) that is not in CF and checks if the present 2-itemsets are continually frequent from its' start partition to the current partitions. If a 2-itemset is no longer frequent, SWF deletes this itemset from CF. If an itemset becomes frequent at partition P_j ($j > i$), the algorithm put this itemset into CF and check it's frequency from P_j to each successive partition P'_j ($j' > j$). It was shown that the number of the reserved candidate 2-itemsets will be close to the number of the frequent 2-itemsets.

For a moderate number of candidate 2-itemsets, scan reduction technique [9] can be applied to generate all candidate k-itemsetsss. Therefore, one database scan is enough to calculate all candidate itemsets' supports and determine the frequent ones. In summary, the total number of database scans is kept as small as 2.

However, SWF does not utilize previously discovered frequent itemsets as applied in FUP2. In fact, for most frequent itemsets, the supports can be updated by simply scan the changed partitions of the database. It doesn't need to scan the whole database for these old frequent itemsets. Only those newly generated candidate itemsets have to be checked by scanning the total database. In such cases, we can speed up not only candidate generation but also database scanning. The algorithm will be described in the next section.

2.2 An Example

In the following, we use an example to introduce how to improve the SWF algorithm. The example is the same as [3]. Figure 1(a) shows the original transaction database where it contains 9 transactions, tagged $t_1, \ldots, t_9$, and the database is segmented into three partition, P_1,P_2,P_3 (denoted by $db^{1,3}$). After the database

$db^{1,3}$	$\triangle^-$	P_1	t_1	A	B	C			
			t_2	A					F
			t_3	A	B	C		E	
	D^-	P_2	t_4	A	B		D	E	
			t_5			C			F
			t_6	A	B	C	D		
		P_3	t_7		B	C		E	
			t_8	A		C			F
			t_9		B		D	E	

(a) The 1st mining

D^-	P_2	t_4	A	B		D	E		$db^{2,4}$
		t_5			C			F	
		t_6	A	B	C	D			
	P_3	t_7		B	C		E		
		t_8	A		C			F	
		t_9		B		D	E		
$\triangle^-$	P_4	t_{10}		B		D	E	F	
		t_{11}				D	E	F	
		t_{12}	A		C				

(b) The 2nd mining

Fig. 1. Example

is updated (delete P_1 and add P_4), the database contains partition P_2,P_3,P_4 (denoted by $db^{2,4}$) as shown in Figure 1(b). The minimum support is assumed to be $s = 40\%$. The incremental mining problem is decomposed into two procedures:

1. **Preprocessing procedure** which deals with mining in the original transaction database.
2. **Incremental procedure** which deals with the update of the frequent itemsets for an ongoing time-variant transaction database.

Preprocessing procedure In the preprocessing procedure, each partition is scanned sequentially for the generation of candidate 2-itemsets. Consider this example in Figure 1(a). After scanning the first segment of 3 transactions, candidate 2-itemsets {AB, AC, AE, AF, BC, BE, CE} are generated, and frequent itemsets {AB, AC, BC} are added to CF_2 with two attributes *start* partition P_1 and *count*. Note that SWF begins from 2-itemset instead of 1-itemset.

Similarly, after scanning partition P_2, the occurrence counts of candidate 2-itemsets in the present partition are recorded. For itemsets that are already in CF_2, add the occurrence counts to the corresponding itemsets in CF_2. For itemsets that are not in CF_2, record the counts and its *start* partition from P_2. Then, check if the itemsets are frequent from its start partition to current partition and delete those non-frequent itemsets from CF_2. In this example. The counts of the three itemsets {AB, AC, BC} in P_2, which start from partition P_1, are added to their corresponding counts in CF_2. The local support (or called the *filtering threshold*) of these 2-itemsets is given by $\lceil (3+3)*0.4 \rceil = 3$. The other candidate 2-itemsets of P_2, {AD, BD, BE, CD, CF, DE}, has filtering threshold $\lceil 3*0.4 \rceil = 2$. Only itemsets with counts greater than its filtering threshold are kept in CF_2, others are deleted. Finally, partition P_3 is processed by algorithm SWF. The resulting CF_2={AB, AC, BC, BD, BE}.

After all partitions are processed, the scan reduction technique is employed to generate C'_3 from CF_2. Iteratively, C'_{k+1} can be generated from C'_k for $k =$

$3, \ldots, n$, as long as all candidate itemsets can been stored in memory. Finally, we can find $\cup_k L_k$ from $\cup_k C_k$ with only one database scan.

Incremental procedure Consider Figure 1(b), when the database is updated form $db^{1,3}$ to $db^{2,4}$ where transaction, t_1 t_2 t_3, are deleted from the database and t_{10}, t_{11}, t_{12} are added to the database. The incremental step can be divided into three sub-steps: (1) generating C_2 in $D^- = db^{1,3}$- $\triangle^-$ (2) generating C_2 in $db^{2,4} = D^- + \triangle^+$ (3) scanning the database $db^{2,4}$ for the generation of all frequent itemsets L_k. In the first sub-step, we load CF_2 and check if the counts of itemsets in CF_2 will be influenced by deleting P_1. We decrease the influenced itemstes' counts that are suppored by P_1 and set the *start* position of influenced itemsets to 2. Then, we check the itemsets of CF_2 from the start position of the itemsets to partition P_3, and delete non frequent itemsets of CF_2. It can be seen that itemsets {AB, AC, BC} were removed. Now we have CF_2 corresponding to $db^{2,3}$.

The second sub-step is similar to the operation in the preprocessing step. We add the counts of the 2-itemset in P_4 to the counts of the same itemsets of CF_2, and reserve the frequent itemsets of CF_2. New frequent 2-itemsets that are not in CF_2 will be added with its *start* position P_4 and counts. Finally, at the third sub-step, SWF uses scan reduction technique to generate all candidate itemsets from $CF_2 = \{BD, BE, DE, EF, DF\}$ and scan database $db^{2,4}$ once to find frequent itemsets.

3 The Proposed Algorithm

The proposed algorithm combines the idea of FUP_2 with SWF. Consider the candidate itemsets generated in the preprocessing procedure ({A, B, C, D, E, F, AB, AC, BC, BD, BE, ABC}) and the incremental procedure ({A, B, C, D, E, F, BD, BE, DE, EF, BFE, DEF}) in SWF, we find that there are eight candidate itemsets in common and seven of them are frequent itemsets. If we incorporate previous mining result, i.e. the counts of the frequent itemsets, we can get new counts by only scanning the changed portions of database instead of scanning the whole database in the incremental procedure.

The proposed algorithm divides the incremental procedure into six sub-steps: the first two sub-steps update CF_2 as those for the incremental procedure in SWF. Let us denote the set of previous mining result as KI (known itemsets). Consider the example where KI = FI={A(6), B(6), C(6), D(3), E(4), F(3), AB(4), AC(4), BC(4), BE(4)}. In the third and fourth sub-step, we scan the changed portion of the database, P_1 and P_4 to update the counts of itemsets in KI. Denote the set of verified frequent itemsets from KI as FI (frequent itemsets). In this example, FI={A(4), B(5), C(5), D(5), E(5), F(4), BE(4)}.

In the fifth sub-step, we compare CF_2 of $db^{2,4}$ with KI and collect those 2-itemsets from CF_2 that are not in KI to a set, named NC_2 (new candidate 2-itemsets). The set NC_2 contains itemsets that have no information at all and really need to scan the whole database to get the counts. Then we use NC_2 to generate NC_3 by three JOIN operations The first JOIN operation is $NC_2 \star NC_2$. The the second JOIN operation is $NC_2 \star FI_2$ (2-itemsets of FI). The third JOIN operation is $FI_2 \star FI_2$. However, result from the third JOIN can be deleted if the itemsets are present in KI. The reason for the third JOIN operation is that NC_2 may generate itemsets which are not frequent in previous mining task because one of the 2-itemset are not frequent. While new itemsets from NC_2 sometimes makes such 3-itemsets satisfy the anti-monotone Apriori heuristic in present mining task. From the union of the three JOIN operations, we get NC_3. Similarly, we can get NC_4 by three JOIN operations from $NC_3 \star NC_3$, $NC_3 \star FI_3$ (3-itemsets of FI), and $FI_3 \star FI_3$. Again, result from the third JOIN can be deleted if the itemsets are present in KI. The union of the three JOINs results in NC_4. The process goes until no itemsets can be generated. Finally, at the sixth sub-step, we scan the database to find frequent itemsets from $\cup_k NC_k$ and add these frequent itemsets to FI.

In this example. NC_2 is {BD, DE, DF, EF} ($CF_2 - FI$). The first JOIN operation, $NC_2 \star NC_2$, generates {DEF}. The second JOIN operation, $NC_2 \star FI_2$, generates {BDE}. The third JOIN operation, $FI_2 \star FI_2$, generates nothing. So NC_3 is {BDE, DEF}. Again we use NC_3 to generate NC_4 by the three JOIN operations. Because the three JOIN operations generate nothing for NC_4, we stop the generation. At last, we scan the database and find new frequent itemsets ($\{BE, DE\}$) from the union of NC_2 and NC_3. Compare the original SWF algorithm with the extension version, we can find that the original SWF algorithm needs to scan database for thirteen itemsets, {A, B, C, D, E, F, BD, BE, DE, DF, EF, BDE, DEF}, but the proposed algorithm only needs to scan database for six itemsets, {BD, DE, DF, EF, BDE, DEF}.

In addition to reuse previous large itemsets $\cup_k F_k$ as KI, when the size of candidate itemsets $\cup_k C'_k$ from $db^{1,3}$ is close to the size of frequent itemsets $\cup_k F_k$ from $db^{1,3}$, KI can be set to the set of candidate itemsets instead of frequent itemsets. This way, it can further speed up the runtime of the algorithm.

3.1 SWF with Known Itemsets

The proposed algorithm, KI_SWF (SWF with known itemsets), employs the preprocessing procedure of SWF and extend the incremental procedure as Figure **??**. The change portion of the algorithm is presented in bold face. The preprocessing procedure is the same as SWF and is omitted here. For ease exposition, the meanings of various symbols used are given in Table 1.

At beginning, we load the cumulative filter CF_2 generated in preprocessing procedure (Step 7). From $\triangle^-$, we decrease the counts of itemsets in CF_2 and delete non frequent itemsets from CF_2 with the filtering threshold $\lceil s * D^- \rceil$ (Step 8 to 16). Similarly, we increase the counts of itemsets in CF_2 by scanning $\triangle^+$ and delete non frequent itemsets with filtering threshold $\lceil s * (D^- + \triangle^+) \rceil$ (Step 17 to 29). After these steps, we get CF_2 corresponding to database $db^{i,j}$.

Incremental Procedure of **KI_SWF**

Original databases = $db^{m,n}$;
New database= $db^{i,j}$;
Database removed $\triangle^{-}=\sum_{k=m,i-1} P_k$;
Database added $\triangle^{+}=\sum_{k=n+1,j} P_k$;
$D^{-}=\sum_{k=i,n} P_k$;
$db^{i,j} = db^{m,n} - \triangle^{-} + \triangle^{+}$;
loading $C_2^{m,n}$ of $db^{m,n}$ into CF_2;
begin for $k = m$ to $i-1$
    begin for each 2-itemset $I \in P_k$;
        if($I \in CF_2$ and $I.start \leq k$)
            $I.count = I.count - N_{p_k}(I)$;
            $I.start = k+1$;
            if $(I.count < \lceil s * \sum_{t=i}^{n} |P_t| \rceil)$
                $CF_2 = CF_2 - I$;
    end
end
begin for $k = n+1$ to j
    begin for each 2-itemset $I \in P_k$
        if($I \notin CF_2$)
            $I.count = N_{p_k}(I)$;
            $I.start = k$;
            if $(I.count \geq \lceil s * |P_k| \rceil)$
                $CF_2 = CF_2 \cup I$;
        if $(I \in CF_2)$
            $I.count = I.count + N_{p_k}(I)$;
            if $(I.count \geq \lceil s * \sum_{t=i}^{k} |P_t| \rceil)$
            $CF = CF - I$;
    end
end
loading $L_{2,h}^{m,n} of db^{m,n}$ **into** KI**;**
begin for $k=m$ **to** $i-1$
    begin for each itemset $I \in P_k$
        if $(I \in KI)$
            $I.count$**=**$I.count$**-**$N_{p_k}(I)$**;**
    end
end
begin for k=n+1 to j
    begin for each itemset $I \in P_k$
    if $(I \in KI)$
        $I.count$**=**$I.count$**+**$N_{p_k}(I)$**;**
    end
end;
set FI=$\emptyset$**;**
begin for each itemset $I \in KI$
    if $(I.count \geq \lceil s * |db^{i,j}| \rceil)$
        $FI = FI \cup I$**;**
end
set $NC_2 = CF_2 - KI$**;**
$h = 2$**;**
begin while$(NC_h \neq \emptyset)$
    NC_{h+1}**=New_Candidate**
        (NC_h, FI_h, KI_h);
    $h = h+1$**;**
end
refresh $I.count = 0$ **where** $I \in NC_{2,h}$**;**
begin for $k = i$ to j;
    for each itemset $I \in NC_{2,h}$;
        $I.count = I.count + N_{p_k}(I)$;
end

begin for each itemset $I \in NC_{2,h}$;
    if $(I.count \geq \lceil s * |db^{i,j}| \rceil)$
        $FI = FI \cup I$;
end
return FI;

Procedure of **New_Candidate**(P_h, Q_h, KI_h)
setR_h=JOIN$(P_k, Q_h, P_h \cup Q_h)$;
$R_h = R_h \cup$JOIN$(P_h, P_h, P_h \cup Q_h)$;
$R_h = R_h \cup$(JOIN$(Q_h, Q_h, P_h \cup Q_h)$-KI_h);
return R_h;

Procedure of **JOIN**(P_h, Q_h, C_h)
set $R_h = \emptyset$;
for any two itemsets I_1 of P_h and I_2 of Q_h,
    where $I_1=\{a_1, a_2, \ldots, a_h\}$ $I_2=\{b_1, b_2, \ldots, b_h\}$
if $(a_1=b_1 \wedge a_2=b_2 \wedge \ldots \wedge a_{h-1}=b_{h-1} \wedge a_h > b_h)$
    set $I_3=\{a_1, a_2, \ldots, a_h, b_h\}$;
    if (all h-itemsets of I_3 are in C_h)
        $R_h = R_h \cup I_3$;
endfor
return R_h;

Figure 2. Incremental procedure of Algorithm KI_SWF

$db^{i,j}$	Database D from partition P_i to P_j
s	Minimum support threshold
$\|P_k\|$	Number of transactions in partition P_k
$N_{p_k}(I)$	Number of transactions in P_k that cantain itemset I
$C_k^{i,j}$	The candidate k-itemsets corresponding to $db^{i,j}$
$C_{2,h}^{i,j}$	The union of candidate k-itemset $C_k^{i,j}$ $(k = 2, \ldots, h)$ from $db^{i,j}$
$L_{2,h}^{i,j}$	The union of frequent k-itemset $L_k^{i,j}$ $(k = 2, \ldots, h)$ from $db^{i,j}$
FI_j	The j-itemsets of FI
KI_j	The j-itemsets of KI
NC_i	The i-itemsets of NC

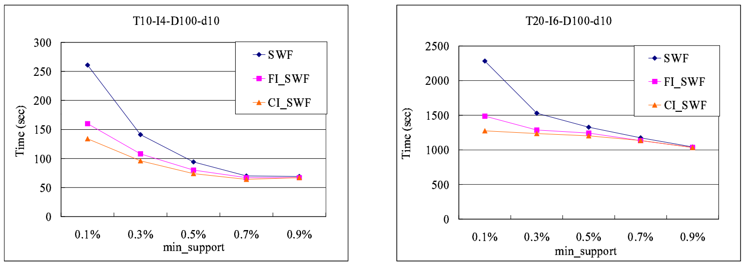
Table 1. Meanings of symbols used

Continually, we load the frequent itemsets that was generated in previous mining task into KI. We check the frequency of every itemset from KI by scanning $\triangle^-$ and $\triangle^+$, and put those remaining frequent into FI (Step 30 to 47). Let NC_2 denotes the itemsets from $CF_2 - KI$, i.e. new 2-itemsets that are not frequent in the previous mining procedure and need to be checked by scanning $db^{i,j}$. From NC_2, we can generate new candidate 3-itemsets (NC_3) by JOIN 2-itemsets from NC_2, FI_2, and KI_2. The function New_Candidate performs three JOIN operations: $NC_2 \star NC_2$, $NC_2 \star FI_2$, and $FI_2 \star FI_2$. However, if the itemsets are present in KI, they can be deleted since they have already been checked. The third operation is necessary since new candidates may contain supporting $(k-1)$-itemsets for a k-itemset to become candidate. We unite the results of the three JOIN operations and get NC_3. Iteratively (Step 48 to 53), NC_4 can be obtained from NC_3, FI_3, and KI_3. Finally, we scan the database $db^{i,j}$ and count supports for itemsets in $\cup_{i=2,h} NC_i$ and put the frequent ones to FI (Step 54 to 63).

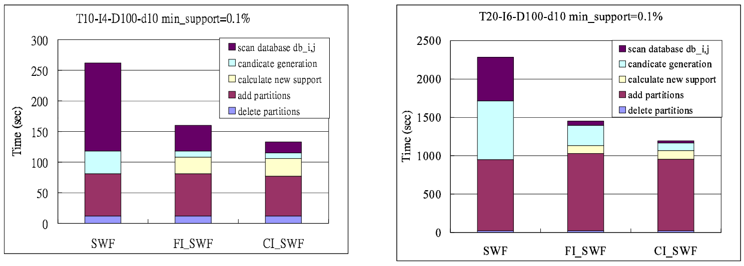
Sometimes, the size of candidate itemsets in the previous mining procedure is very close to the size of frequent itemsets. If we kept the support counts for all candidate itemsets, and load them to KI at Step 30 in the incremental procedure:

30. **Loading $C_{2,h}^{m,n}$ of $db^{m,n}$ named KI;**

The performance, including candidate generation and scanning, can further be speed up. In the next section, we will conduct experiment to compare the original SWF algorithm with the KI_SWF algorithms. If the itemsets of KI are the candidate itemsets of the previous mining task, we call this extension CI_SWF. If the itemsets of KI are the frequent itemsets of previous mining task, we call this extension FI_SWF.



(a)



(b)

Fig. 3. Execution time vs. minimum support

4 Experiments

To evaluate the performance of FI_SWF algorithm and CI_SWF algorithm, we preformed several experiments on a computer with a CPU clock rate of 800 MHz and 768 MB of main memory. The transaction data resides in the FAT32 system and is stored on a 30GB IDE 3.5 drive. The simulation program was coded in C++. The method to generate synthetic transactions we employed in this paper is similar to the ones used in [4, 9, 3]. The performance comparison of SWF, FI_SWF, and CI_SWF is presented in Section 4.1. Section 4.2 shows the memory required and I/O cost for the three algorithms. Results on scaleup experiments are presented in Section 4.3.

4.1 Running Time

First, we test the speed of the three algorithms, FI_SWF, CI_SWF, and SWF. We use two different datasets, $T10-I4-D100-d10$ and $T20-I6-D100-d10$, and set N=1000 and $|L|$=2000. Figure 3(a) shows the running time of the three algorithms as the minimum support threshold decreased from 1% to 0.1%. When the support threshold is high, the running times of three algorithms are similar. Because there are only a limited number of frequent itemsts produced. When the support threshold is low, the difference in the running time becomes large. This is because when the support threshold is low (0.1%), there are a lot number of candidate itemsets produced. SWF scanned the total databases $db^{i,j}$ for all candidate itemsets. FI_SWF scanned $db^{i,j}$ for the candidate itemsets that are not frequent in previous mining tasks or not present in previous mining tasks. CI_SWF scanned $db^{i,j}$ only for the candidate itemsets that are not present in previous mining tasks. The details of the running time of three algorithms are shown in Figure 3(b). For SWF, the incremental procedure includes four phase: delete partitions, add partitions, generate candidates and scan database $db^{i,j}$. For FI_SWF and CI_SWF, they take one more phase to count new supports for known itemsets. The time for deleting or add partitions is about the same for three algorithms, while SWF takes more time to generate candidates and scan database.

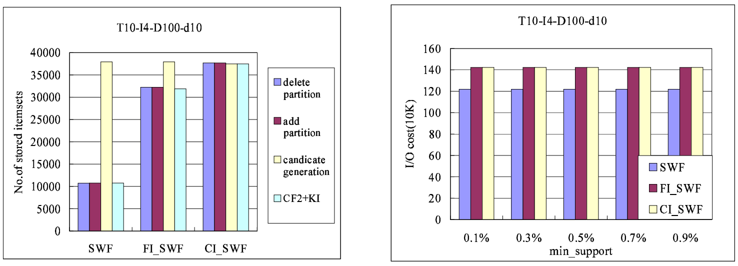
4.2 The Memory Required and I/O Cost

Intuitively, FI_SWF and CI_SWF require more memory space than SWF because they need to load the itemsets of the previous mining task and CI_SWF needs more memory space than FI_SWF. However, the maximum memory required for these three algorithms are in fact the same. We record the number of itemsets kept in memory at running time as shown in Figure 4(a). The dataset used was $T10-I4-D100-d10$ and the minimum support was set to 0.1%. At beginning, the memory space used by SWF algorithm was less than the others, but the memory space grew significantly at the candidate generation phase. The largest memory space required at candidate generation is the same as that used by CI_SWF and FI_SWF. In other words, SWF still needs memory space as large as CI_SWF and FI_SWF. The disk space required is also shown in Figure 4(a), where "$CF_2 + KI$" denotes the number of itemsets saved for later mining task. From this figure, CI_SWF require triple disk space of SWF. However, frequent itemsets are indeed by-product of previous mining task.

To evaluate the corresponding cost of disk I/O, we assume that each read of a transaction consumes one unit of I/O cost. We use $T10 - I4 - D100 - d10$ as the dataset in this experiment. In Figure 4(b), we can see that the I/O costs of FI_SWF and CI_SWF were a little higher than SWF. This is because that FI_SWF and CI_SWF must scan the changed portions of the database to update the known itemsets' supports.

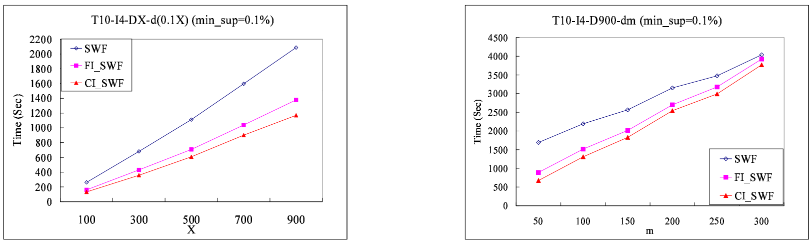
4.3 Scaleup Test

Continuously, we test the scaleup ability of FI_SWF and CI_SWF with large databases. We conduct two experiments. In the first experiment, we use $T10-I4-DX-d(X*0.1)$ as the dataset, where X represents the size of the databases and the changed portions are 10 percents of the original databases. We set $X = 300, 600, 900$ with min_support=0.1%. The result of the second experiment is shown in Figure 5(a). We find the the response time of these two algorithms were linear to the size of the databases. In the second experiment, we use $T10-$



(a) Memory required at each phase (b) I/O cost for three algorithms

Fig. 4. Memory required and I/O cost



(a) Execution time vs. the database size (b) Execution time vs. the changed portions

Fig. 5. Scalable test

$I4 - D900 - dm$ as the dataset and set min_support to 0.1%. The value of m is changed from 100, 200, to 300. The result of the this experiment is shown in Figure 5(b). As $\triangle^+$ and $\triangle^-$ became larger, it requires more time to scan the changed partitions of the dataset to generate CF_2 and calculate the new supports. From this two experiments, we can say that FI_SWF and CI_SWF have the ability of dealing with large databases.

5 Conclusion

In this paper, we extend the sliding-window-filtering technique and focus on how to use the result of the previous mining task to improve the response time. Two algorithms are proposed for incremental mining of frequent itemsets from updated transaction database. The first algorithm FI_SWF (SWF with Frequent Itemset) reuse the frequent itemsets (and the counts) of previous mining task as FUP2 to reduce the number of new candidate itemsets that have to be checked.

The second algorithm CI_SWF (SWF with Candidate Itemset) reuse the candidate itemsets (and the counts) from the previously mining task. From the experiments, it shows that the new incremental algorithm is significantly faster than SWF. In addition, the need for more disk space to store the previously discovered knowledge does not increase the maximum memory required during the execution time.

References

1. R. Agarwal, C. Aggarwal, and V.V.V Prasad. A tree projecction algorithm for generation of frequent itemsets. *Journal of Parallel and Distributed Computing (Special Issue on High Performance Data Mining)*, 2000.
2. R. Agrawal, T. Imielinski, and A. Swami. Mining association rules between sets of items in large databases. In *Proc. of ACMSIGMOD*, pages 207–216, May 1993.
3. M.-S. Chen C.-H. Lee, C.-R. Lin. Sliding-window filtering: An efficient algorithm for incremental mining. In *Intl. Conf. on Information and Knowledge Management (CIKM01)*, pages 263–270, November 2001.
4. D. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of discovered association rules in large databases: An incremental updating technique. In *Data Engineering*, pages 106–114, February 1996.
5. D. Cheung, S.D. Lee, and B. Kao. A general incremental technique for updaing discovered association rules. In *Intercational Conference On Database Systems For Advanced Applications*, April 1997.
6. J. Han and J. Pei. Mining frequent patterns by pattern-growth: Methodology and implications. In *ACM SIGKDD Explorations (Special Issue on Scaleble Data Mining Algorithms)*, December 2000.
7. J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. Freespan: Frequent parrten-projected sequential pattern mining. In *Knowledge Discovery and DAta Mining*, pages 355–359, August 2000.
8. J.-L. Lin and M.H. Dumham. Mining association rules: Anti-skew algorithms. In *Proc. of 1998 Int'l Conf. on Data Engineering*, pages 486–493, 1998.
9. J.-S Park, M.-S Chen, and P.S Yu. Using a hash-based method with transaction trimming for minng association rules. In *IEEE Transactions on Knowledge and Data Engineering*, volume 9, pages 813–825, October 1997.
10. J. Pei, J. Han, and L.V.S. Lakshmanan. Mining frequent itemsets with convertible constraints. In *Proc. of 2001 Int. Conf. on Data Engineering*, 2001.
11. A. Savasere, E. Omiecinski, and S. Navathe. An efficient algorithm for mining association rules in large databases. In *Proc. of the 21th International Conference on Very Large Data Bases*, pages 432–444, September 1995.
12. A. Veloso, B. Possas, W. Meira Jr., and M. B. de Carvalho. Knowledge management in association rule mining. In *Integrating Data Mining and Knowledge Management*, 2001.
13. Z. Zhou and C.I. Ezeife. A low-scan incremental association rule maintenance method. In *Proc. of the 14th Conadian Conference on Artificial Intelligence*, June 2001.