Sliding-Window Filtering: An Efficient Algorithm for Incremental Mining

Chang-Hung Lee, Cheng-Ru Lin, and Ming-Syan Chen
Department of Electrical Engineering
National Taiwan University
Taipei, Taiwan, ROC
E-mail: mschen@cc.ee.ntu.edu.tw, {chlee, owenlin}@arbor.ee.ntu.edu.tw

ABSTRACT

We explore in this paper an effective sliding-window filtering (abbreviatedly as SWF) algorithm for incremental mining of association rules. In essence, by partitioning a transaction database into several partitions, algorithm SWF employs a filtering threshold in each partition to deal with the candidate itemset generation. Under SWF, the cumulative information of mining previous partitions is selectively carried over toward the generation of candidate itemsets for the subsequent partitions. Algorithm SWF not only significantly reduces I/O and CPU cost by the concepts of cumulative filtering and scan reduction techniques but also effectively controls memory utilization by the technique of sliding-window partition. Algorithm SWF is particularly powerful for efficient incremental mining for an ongoing time-variant transaction database. By utilizing proper scan reduction techniques, only one scan of the incremented dataset is needed by algorithm SWF. The I/O cost of SWF is, in orders of magnitude, smaller than those required by prior methods, thus resolving the performance bottleneck. Experimental studies are performed to evaluate performance of algorithm SWF. It is noted that the improvement achieved by algorithm SWF is even more prominent as the incremented portion of the dataset increases and also as the size of the database increases.

Keywords

Data mining, association rules, time-variant database, incremental mining

1. INTRODUCTION

The importance of data mining is growing at rapid pace recently. Analysis of past transaction data can provide very valuable information on customer buying behavior, and business decisions. In essence, it is necessary to collect and analyze a sufficient amount of sales data before any meaningful conclusion can be drawn therefrom. Since the amount of these processed data tends to be huge, it is important to devise efficient algorithms to conduct mining on these data. Mining association rules was first introduced in [2], where it was shown that the problem of mining association rules is composed of the following two subproblems: (1) discovering the frequent itemsets, i.e., all sets of itemsets that have transaction support above a pre-determined minimum support s, and (2) using the frequent itemsets to generate the association rules for the database. The overall performance of mining association rules is in fact determined by the first subproblem. After the frequent itemsets are identified, the corresponding association rules can be derived in a straightforward manner [2]. Among others, Apriori [2], DHP [16], and partition-based ones [15, 19] are proposed to solve the first subproblem efficiently. In addition, several novel mining techniques, including TreeProjection [1], FP-tree [11, 12, 18], and constraint-based ones [10, 14, 17, 23, 24] also received a significant amount of research attention.

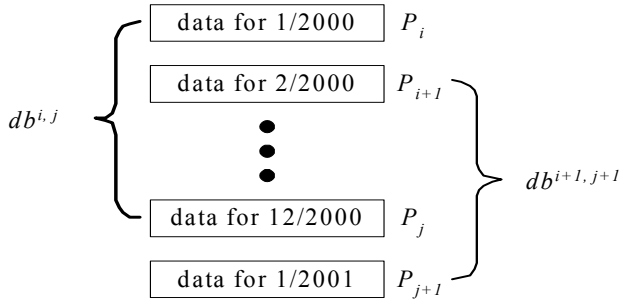

Figure 1: Incremental mining for an ongoing time-variant transaction database.

Recent important applications have called for the need of incremental mining. This is due to the increasing use of the record-based databases whose data are being continuously added. Examples of such applications include Web log records, stock market data, grocery sales data, transactions in electronic commerce, and daily weather/traffic records, to name a few. In many applications, we would like to mine the transaction database for a fixed amount of most recent data (say, data in the last 12 months). That is, in the incremental mining, one has to not only include new data (i.e., data in the new month) into, but also remove the old data (i.e., data in the most obsolete month) from the mining process.

Consider the example transaction database in Figure 1. Note that $db^{i,j}$ is the part of the transaction database formed by a continuous region from partition P_i to partition P_j. Suppose we have conducted the mining for the transaction database $db^{i,j}$. As time advances, we are given the new data of January of 2001, and are interested in conducting an incremental mining against the new data. Instead of taking all the past data into consideration, our interest is limited to mining the data in the last 12 months. As a result, the mining of the transaction database $db^{i+1,j+1}$ is called for. Note that since the underlying transaction database has been changed as time advances, some algorithms, such as Apriori, may have to resort to the regeneration of candidate itemsets for the determination of new frequent itemsets, which is, however, very costly even if the incremental data subset is small. On the other hand, while FP-tree-based methods [11, 18] are shown to be efficient for small databases, it is expected that their deficiency of memory overhead due to the need of keeping a portion of database in memory, as indicated in [13], could become more severe in the presence of a large database upon which an incremental mining process is usually performed.

To the best of our knowledge, there is little progress made thus far to explicitly address the problem of incremental mining except noted below. In [8], the FUP algorithm updates the association rules in a database when new transactions are *added* to the database. Algorithm FUP is based on the framework of Apriori and is designed to discover the new frequent itemsets iteratively. The idea is to store the counts of all the frequent itemsets found in a previous mining operation. Using these stored counts and examining the newly added transactions, the overall count of these candidate itemsets are then obtained by scanning the original database. An extension to the work in [8] was reported in [9] where the authors propose an algorithm FUP_2 for updating the existing association rules when transactions are *added* to and *deleted* from the database. In essence, FUP_2 is equivalent to FUP for the case of insertion, and is, however, a complementary algorithm of FUP for the case of deletion. It is shown in [9] that FUP_2 outperforms Apriori algorithm which, without any provision for incremental mining, has to re-run the association rule mining algorithm on the whole updated database. Another FUP-based algorithm, call $\text{FUP}_2\mathcal{H}$, was also devised in [9] to utilize the hash technique for performance improvement. Furthermore, the concept of *negative borders* in [21] and that of UWEP, i.e., update with early pruning, in [4] are utilized to enhance the efficiency of FUP-based algorithms.

However, as will be shown by our experimental results, the above mentioned FUP-based 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 database. First, consider the problem of a potentially huge set of candidate itemsets. Note that the FUP-based algorithms deal with the combination of two sets of candidate itemsets which are independently generated, i.e., from the original data set and the incremental data subset. Since the set of candidate itemsets includes all the possible permutations of the elements, FUP-based algorithms may suffer from a very large set of candidate itemsets, especially from candidate 2-itemsets. More importantly, in many applications, one may encounter new itemsets in the incremented dataset. While adding some new products in the transaction database, FUP-based algorithms will need to resort to multiple scans of database. Specifically, in the presence of a new frequent itemset L_k generated in the data subset, k scans of the database are needed by FUP-based algorithms in the worst case. That is, the case of $k = 8$ means that the database has to be scanned 8 times, which is very costly, especially in terms of I/O cost. The problem of a large set of candidate itemsets will hinder an effective use of the scan reduction technique [16] by an FUP-based algorithm.

To remedy these problems, we shall devise in this paper an algorithm based on sliding-window filtering (abbreviatedly as SWF) for incremental mining of association rules. In essence, by partitioning a transaction database into several partitions, algorithm SWF employs a filtering threshold in each partition to deal with the candidate itemset generation. For ease of exposition, the processing of a partition is termed a *phase* of processing. Under SWF, the cumulative information in the prior phases is selectively carried over toward the generation of candidate itemsets in the subsequent phases. After the processing of a phase, algorithm SWF outputs a *cumulative filter*, denoted by CF, which consists of a progressive candidate set of itemsets, their occurrence counts and the corresponding partial support required. As will be seen, the cumulative filter produced in each processing phase constitutes the key component to realize the incremental mining. An illustrative example for the operations of SWF is presented in Section 3.1 and a detailed description of algorithm SWF is given in Section 3.2. It will be seen that algorithm SWF proposed has several important advantages. First, with employing the prior knowledge in the previous phase, SWF is able to reduce the amount of candidate itemsets efficiently which in turn reduces the CPU and memory overhead. The second advantage of SWF is that owing to the small number of candidate sets generated, the scan reduction technique [16] can be applied efficiently. As a result, only one scan of the ongoing time-variant database is required. As will be validated by our experimental results, this very advantage of SWF enables SWF to significantly outperform FUP-based algorithms. The third advantage of SWF is the capability of SWF to avoid the data skew in nature. As mentioned in [15], such instances as severe whether conditions may cause the sales of some items to increase rapidly within a short period of time. The performance of SWF will be less affected by the data skew since SWF employs the cumulative information for pruning false candidate itemsets in the early stage.

Extensive experiments are performed to assess the performance of SWF. As shown in the experimental results, SWF produces a significantly smaller amount of candidate 2-itemsets than prior algorithms. In fact, the number of the candidate itemsets C_ks generated by SWF approaches to its theoretical minimum, i.e., the number of frequent k-itemsets, as the value of the minimal support increases. It is shown by our experiments that SWF in general significantly outperforms FUP-based algorithms. Explicitly, the execution time of SWF is, in orders of magnitude, smaller than those required by prior algorithms. Sensitivity analysis on various parameters of the database is also conducted to provide many insights into algorithm SWF. The advantage of SWF becomes even more prominent not only as the amount of incremented dataset increases but also as the size of the database increases.

The rest of this paper is organized as follows. Preliminaries and related works are given in Section 2. Algorithm SWF is described in Section 3. Performance studies on various schemes are conducted in Section 4. This paper concludes with Section 5.

2. PRELIMINARIES

Let $\mathcal{I}$=$\{i_1, i_2, ..., i_m\}$ 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 $T \subseteq \mathcal{I}$. 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 $X \Longrightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$ and $X \bigcap Y = \phi$. The rule $X \Longrightarrow Y$ holds in the transaction set D with *confidence* c if c% of transactions in D that contain X also contain Y. The rule $X \Longrightarrow Y$ has *support* s in the transaction set D if s% of transactions in D contain $X \bigcup Y$. For a given pair of confidence and support thresholds, the problem of mining association rules 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].

Most of the previous studies, including those is [2, 5, 8, 16, 20, 22], belong to Apriori-like approaches. Basically, an Apriori-like approach is based on an anti-monotone Apriori heuristic [2], i.e., if any itemset of length k is not frequent in the database, its length $(k+1)$ super-itemset will never be frequent. The essential idea is to iteratively generate the set of candidate itemsets of length $(k+1)$ from the set of frequent itemsets of length k (for $k \geq 1$), and to check their corresponding occurrence frequencies in the database. As a result, if the largest *frequent* itemset is a j-itemset, then an Apriori-like algorithm may need to scan the database up to $(j+1)$ times.

In Apriori-like algorithms, C_3 is generated from $L_2 \star L_2$. In fact, a C_2 can be used to generate the candidate 3-itemsets. This technique is referred to as scan reduction in [6]. Clearly, a C'_3 generated from $C_2 \star C_2$, instead of from $L_2 \star L_2$, will have a size greater than $|C_3|$ where C_3 is generated from $L_2 \star L_2$. However, if $|C'_3|$ is not much larger than $|C_3|$, and both C_2 and C_3 can be stored in main memory, we can find L_2 and L_3 together when the next scan of the database is performed, thereby saving one round of database scan. It can be seen that using this concept, one can determine all L_ks by as few as two scans of the database (i.e., one initial scan to determine L_1 and a final scan to determine all other frequent itemsets), assuming that C'_k for $k \geq 3$ is generated from C'_{k-1} and all C'_k for $k > 2$ can be kept in the memory. In [7], the technique of scan-reduction was utilized and shown to result in prominent performance improvement.

3. SWF: INCREMENTAL MINING WITH SLIDING-WINDOW FILTERING

In essence, by partitioning a transaction database into several partitions, algorithm SWF employs a filtering threshold in each partition to deal with the candidate itemset generation. As described earlier, under SWF, the cumulative information in the prior phases is selectively carried over toward the generation of candidate itemsets in the subsequent phases. In the processing of a partition, a progressive candidate set of itemsets is generated by SWF. Explicitly, a progressive candidate set of itemsets is composed of the following two types of candidate itemsets, i.e., (1) the candidate itemsets that were carried over from the previous progressive candidate set in the previous phase and remain as candidate itemsets after the current partition is taken into consideration (Such candidate itemsets are called type α candidate itemsets); and (2) the candidate itemsets that were not in the progressive candidate set in the previous phase but are newly selected after only taking the current data partition into account (Such candidate itemsets are called type β candidate itemsets). As such, after the processing of a phase, algorithm SWF outputs a *cumulative filter*, denoted by CF, which consists of a progressive candidate set of itemsets, their occurrence counts and the corresponding partial support required. With these design considerations, algorithm SWF is shown to have very good performance for incremental mining. In Section 3.1, an illustrative example of SWF is presented. A detailed description of algorithm SWF is given in Section 3.2.

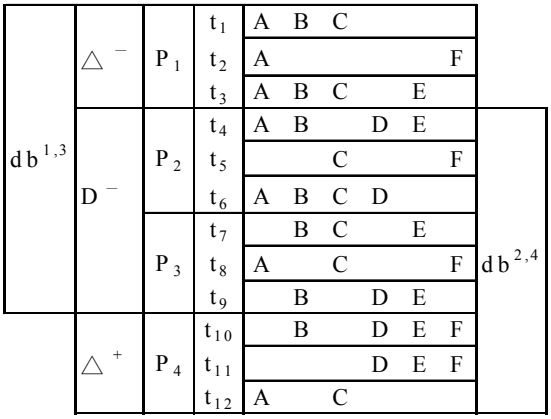

Figure 2: An illustrative transaction database

3.1 An example of incremental mining by SWF

Algorithm SWF proposed can be best understood by the illustrative transaction database in Figure 2 and Figure 3 where a scenario of generating frequent itemsets from a transaction database for the incremental mining is given. The minimum transaction support is assumed to be $s = 40\%$. Without loss of generality, the incremental mining problem can be decomposed into two procedures:

1. Preprocessing procedure: This procedure deals with mining on the original transaction database.

2. Incremental procedure: The procedure deals with the update of the frequent itemsets for an ongoing time-variant transaction database.

The preprocessing procedure is only utilized for the initial mining of association rules in the original database, e.g., $db^{1,n}$. For the generation of mining association rules in $db^{2,n+1}$, $db^{3,n+2}$, $db^{i,j}$, and so on, the incremental procedure is employed. Consider the database in Figure 2. Assume that the original transaction database $db^{1,3}$ is segmented into three partitions, i.e., $\{P_1, P_2, P_3\}$, in the pre-

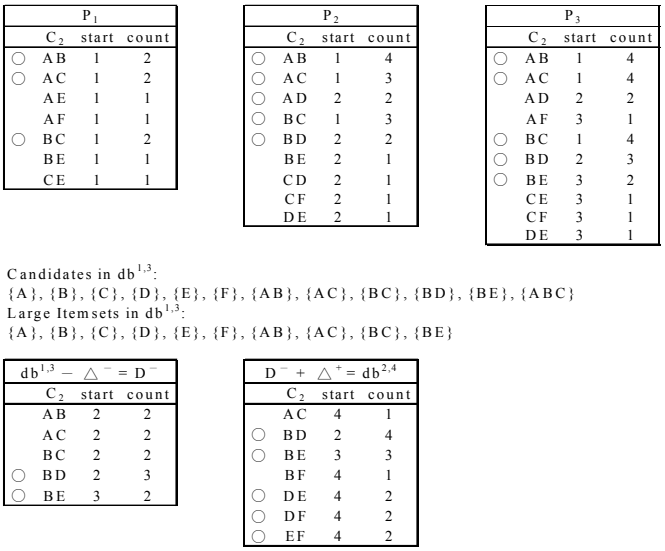

Figure 3: Large itemsets generation for the incremental mining with SWF

processing procedure. Each partition is scanned sequentially for the generation of candidate 2-itemsets in the first scan of the database $\mathrm{db}^{1,3}$. After scanning the first segment of 3 transactions, i.e., partition P_1, 2-itemsets {AB, AC, AE, AF, BC, BE, CE} are generated as shown in Figure 3. In addition, each potential candidate itemset $c \in C_2$ has two attributes: (1) *c.start* which contains the identity of the starting partition when c was added to C_2, and (2) *c.count* which contains the number of occurrences of c since c was added to C_2. Since there are three transactions in P_1, the partial minimal support is $\lceil 3 * 0.4 \rceil = 2$. Such a partial minimal support is called the *filtering threshold* in this paper. Itemsets whose occurrence counts are below the filtering threshold are removed. Then, as shown in Figure 3, only {AB, AC, BC}, marked by "◯", remain as candidate itemsets (of type β in this phase since they are newly generated) whose information is then carried over to the next phase of processing.

Similarly, after scanning partition P_2, the occurrence counts of potential candidate 2-itemsets are recorded (of type α and type β). From Figure 3, it is noted that since there are also 3 transactions in P_2, the filtering threshold of those itemsets carried out from the previous phase (that become type α candidate itemsets in this phase) is $\lceil (3+3) * 0.4 \rceil = 3$ and that of newly identified candidate itemsets (i.e., type β candidate itemsets) is $\lceil 3 * 0.4 \rceil = 2$. It can be seen from Figure 3 that we have 5 candidate itemsets in C_2 after the processing of partition P_2, and 3 of them are type α and 2 of them are type β.

Finally, partition P_3 is processed by algorithm SWF. The resulting candidate 2-itemsets are $C_2 = \{AB, AC, BC, BD, BE\}$ as shown in Figure 3. Note that though appearing in the previous phase P_2, itemset $\{AD\}$ is removed from C_2 once P_3 is taken into account since its occurrence count does not meet the filtering threshold then, i.e., $2 < 3$. However, we do have one new itemset, i.e., BE, which joins the C_2 as a type β candidate itemset. Consequently, we have 5 candidate 2-itemsets generated by SWF, and 4 of them are of type α and one of them is of type β. Note that instead of 15 candidate itemsets that would be generated if Apriori were used[1], only 5 candidate 2-itemsets are generated by SWF.

After generating C_2 from the first scan of database $\mathrm{db}^{1,3}$, we employ the scan reduction technique and use C_2 to generate C_k $(k = 2, 3, ..., n)$, where C_n is the candidate *last*-itemsets. It can be verified that a C_2 generated by SWF can be used to generate the candidate 3-itemsets and its sequential C'_{k-1} can be utilized to generate C'_k. Clearly, a C'_3 generated from $C_2 \star C_2$, instead of from $L_2 \star L_2$, will have a size greater than $|C_3|$ where C_3 is generated from $L_2 \star L_2$. However, since the $|C_2|$ generated by SWF is very close to the theoretical minimum, i.e., $|L_2|$, the $|C'_3|$ is not much larger than $|C_3|$. Similarly, the $|C'_k|$ is close to $|C_k|$. All C'_k can be stored in main memory, and we can find L_k $(k = 1, 2, ..., n)$ together when the second scan of the database $\mathrm{db}^{1,3}$ is performed. Thus, only two scans of the original database $\mathrm{db}^{1,3}$ are required in the preprocessing step. In addition, instead of recording all L_ks in main memory, we only have to keep C_2 in main memory for the subsequent incremental mining of an ongoing time variant transaction database.

$db^{i,j}$	Partition_database ($\mathcal{D}$) from P_i to P_j
s	Minimum support required
$\|P_k\|$	Number of transactions in partition P_k
$N_{p_k}(I)$	Trans. No. in P_k that contain itemset I
$\|db^{1,n}(I)\|$	Trans. No. in $db^{1,n}$ that contain itemset I
$C^{i,j}$	The progressive candidate sets of $db^{i,j}$
Δ^-	The deleted portion of an ongoing database
D^-	The unchanged portion of an ongoing database
Δ^+	The added portion of an ongoing database

Table 1: Meanings of symbols used

The merit of SWF mainly lies in its incremental procedure. As depicted in Figure 3, the mining database will be moved from $\mathrm{db}^{1,3}$ to $\mathrm{db}^{2,4}$. Thus, some transactions, i.e., t_1, t_2, and t_3, are *deleted* from the mining database and other transactions, i.e., t_{10}, t_{11}, and t_{12}, are *added*. For ease of exposition, this incremental step can also be divided into three sub-steps: (1) generating C_2 in $D^- = db^{1,3} - \Delta^-$, (2) generating C_2 in $db^{2,4} = D^- + \Delta^+$ and (3) scanning the database $\mathrm{db}^{2,4}$ only once for the generation of all frequent itemsets L_k. In the first sub-step, $db^{1,3} - \Delta^- = D^-$, we check out the pruned partition P_1, and reduce the value of c.count and set $c.start = 2$ for those candidate itemsets c where $c.start = 1$. It can be seen that itemsets {AB, AC, BC} were removed. Next, in the second sub-step, we scan the incremental transactions in P_4. The process in $D^- + \Delta^+ = db^{2,4}$ is similar to the operation of scanning partitions, e.g., P_2, in the preprocessing step. Three new itemsets, i.e., DE, DF, EF, join the C_2 after the scan of P_4 as type β candidate itemsets. Finally, in the third sub-step, we use C_2 to generate C'_k as mentioned above. With scanning $db^{2,4}$ only once, SWF ob-

[1]The details of the execution procedure by Apriori are omitted here. Interested readers are referred to [3].

tains frequent itemsets {A, B, C, D, E, F, BD, BE, DE} in $db^{2,4}$.

3.2 Algorithm of SWF

For ease exposition, the meanings of various symbols used are given in Table 1. The preprocessing procedure and the incremental procedure of algorithm SWF are described in Section 3.2.1 and Section 3.2.2, respectively.

3.2.1 Preprocessing procedure of SWF

The preprocessing procedure of Algorithm SWF is outlined below. Initially, the database $db^{1,n}$ is partitioned into n partitions by executing the preprocessing procedure (in Step 2), and CF, i.e., cumulative filter, is empty (in Step 3). Let $C_2^{i,j}$ be the set of progressive candidate 2-itemsets generated by database $db^{i,j}$. It is noted that instead of keeping L_ks in the main memory, algorithm SWF only records $C_2^{1,n}$ which is generated by the preprocessing procedure to be used by the incremental procedure.

Preprocessing procedure of **Algorithm SWF**

1. n = Number of partitions;
2. $|db^{1,n}| = \sum_{k=1,n} |P_k|$;
3. $CF = \emptyset$;
4. begin for $k = 1$ to n // 1st scan of $db^{1,n}$
5. begin for each 2-itemset $I \in P_k$
6. if ($I \notin CF$)
7. $I.count = N_{p_k}(I)$;
8. $I.start = k$;
9. if ($I.count \geq s * |P_k|$)
10. $CF = CF \cup I$;
11. if ($I \in CF$)
12. $I.count = I.count + N_{p_k}(I)$;
13. if ($I.count < \lceil s * \sum_{m=I.start,k} |P_m| \rceil$)
14. $CF = CF - I$;
15. end
16. end
17. select $C_2^{1,n}$ from I where $I \in CF$;
18. keep $C_2^{1,n}$ in main memory;
19. $h = 2$; //C_1 is given
20. begin while ($C_h^{1,n} \neq \emptyset$) //Database scan reduction
21. $C_{h+1}^{1,n} = C_h^{1,n} \star C_h^{1,n}$;
22. $h = h + 1$;
23. end
24. refresh $I.count = 0$ where $I \in C_h^{1,n}$;
25. begin for $k = 1$ to n //2nd scan of $db^{1,n}$
26. for each itemset $I \in C_h^{1,n}$
27. $I.count = I.count + N_{p_k}(I)$;
28. end
29. for each itemset $I \in C_h^{1,n}$
30. if ($I.count \geq \lceil s * |db^{1,n}| \rceil$)
31. $L_h = L_h \cup I$;
32. end
33. return L_h;

From Step 4 to Step 16, the algorithm processes one partition at a time for all partitions. When partition P_i is processed, each potential candidate 2-itemset is read and saved to CF. The number of occurrences of an itemset I and its starting partition are recorded in $I.count$ and $I.start$, respectively. An itemset, whose $I.count \geq \lceil s * \sum_{m=I.start,k} |P_m| \rceil$, will be kept in CF. Next, we select $C_2^{1,n}$ from I where $I \in CF$ and keep $C_2^{1,n}$ in main memory for the subsequent incremental procedure. With employing the scan reduction technique from Step 19 to Step 23, $C_h^{1,n}$s ($h \geq 3$) are generated in main memory. After refreshing $I.count = 0$ where $I \in C_h^{1,n}$, we begin the last scan of database for the preprocessing procedure from Step 25 to Step 28. Finally, those itemsets whose $I.count \geq \lceil s * |db^{1,n}| \rceil$ are the frequent itemsets.

3.2.2 Incremental procedure of SWF

As shown in Table 1, D^- indicates the unchanged portion of an ongoing transaction database. The deleted and added portions of an ongoing transaction database are denoted by $\triangle^-$ and $\triangle^+$, respectively. It is worth mentioning that the sizes of $\triangle^+$ and $\triangle^-$, i.e., $|\triangle^+|$ and $|\triangle^-|$ respectively, are not required to be the same. The incremental procedure of SWF is devised to maintain frequent itemsets efficiently and effectively. This procedure is outlined below.

Incremental procedure of **Algorithm SWF**

1. Original database = $db^{m,n}$;
2. New database = $db^{i,j}$;
3. Database removed $\triangle^- = \sum_{k=m,i-1} P_k$;
4. Database database $\triangle^+ = \sum_{k=n+1,j} P_k$;
5. $D^- = \sum_{k=i,n} P_k$;
6. $db^{i,j} = db^{m,n} - \triangle^- + \triangle^+$;
7. loading $C_2^{m,n}$ of $db^{m,n}$ into CF where $I \in C_2^{m,n}$;
8. begin for $k = m$ to $i - 1$ // one scan of $\triangle^-$
9. begin for each 2-itemset $I \in P_k$
10. if ($I \in CF$ and $I.start \leq k$)
11. $I.count = I.count - N_{p_k}(I)$;
12. $I.start = k + 1$;
13. if ($I.count < \lceil s * \sum_{m=I.start,n} |P_m| \rceil$)
14. $CF = CF - I$;
15. end
16. end
17. begin for $k = n + 1$ to j // one scan of $\triangle^+$
18. begin for each 2-itemset $I \in P_k$
19. if ($I \notin CF$)
20. $I.count = N_{p_k}(I)$;
21. $I.start = k$;
22. if ($I.count \geq s * |P_k|$)
23. $CF = CF \cup I$;
24. if ($I \in CF$)
25. $I.count = I.count + N_{p_k}(I)$;
26. if ($I.count < \lceil s * \sum_{m=I.start,k} |P_m| \rceil$)
27. $CF = CF - I$;
28. end
29. end
30. select $C_2^{i,j}$ from I where $I \in CF$;
31. keep $C_2^{i,j}$ in main memory
32. $h = 2$ //C_1 is well known.
33. begin while ($C_h^{i,j} \neq \emptyset$) //Database scan reduction
34. $C_{h+1}^{i,j} = C_h^{i,j} \star C_h^{i,j}$;
35. $h = h + 1$;
36. end
37. refresh $I.count = 0$ where $I \in C_h^{i,j}$;
38. begin for $k = i$ to j //only one scan of $db^{i,j}$
39. for each itemset $I \in C_h^{i,j}$
40. $I.count = I.count + N_{p_k}(I)$;
41. end
42. for each itemset $I \in C_h^{i,j}$
43. if ($I.count \geq \lceil s * |db^{i,j}| \rceil$)
44. $L_h = L_h \cup I$;
45. end
46. return L_h;

As mentioned before, this incremental step can also be divided into three sub-steps: (1) generating C_2 in $D^- = db^{1,3} - \triangle^-$, (2) generating C_2 in $db^{2,4} = D^- + \triangle^+$ and (3) scanning the database $\mathrm{db}^{2,4}$ only once for the generation of all frequent itemsets L_k. Initially, after some update activities, old transactions $\triangle^-$ are removed from the database $db^{m,n}$ and new transactions $\triangle^+$ are added (in Step 6). Note that $\triangle^- \subset db^{m,n}$. Denote the updated database as $db^{i,j}$. Note that $db^{i,j} = db^{m,n} - \triangle^- + \triangle^+$. We denote the unchanged transactions by $D^- = db^{m,n} - \Delta^- = db^{i,j} - \Delta^+$. After loading $C_2^{m,n}$ of $db^{m,n}$ into CF where $I \in C_2^{m,n}$, we start the first sub-step, i.e., generating C_2 in $D^- = db^{m,n} - \Delta^-$. This sub-step tries to reverse the cumulative processing which is described in the preprocessing procedure. From Step 8 to Step 16, we prune the occurrences of an itemset I, which appeared before partition P_i, by deleting the value $I.count$ where $I \in CF$ and $I.start < i$. Next, from Step 17 to Step 36, similarly to the cumulative processing in Section 3.2.1, the second sub-step generates new potential $C_2^{i,j}$ in $db^{i,j} = D^- + \triangle^+$ and employs the scan reduction technique to generate $C_h^{i,j}$s from $C_2^{i,j}$. Finally, to generate new L_ks in the updated database, we scan $db^{i,j}$ for only once in the incremental procedure to maintain frequent itemsets. Note that $C_2^{i,j}$ is kept in main memory for the next generation of incremental mining.

Note that SWF is able to filter out false candidate itemsets in P_i with a hash table. Same as in [16], using a hash table to prune candidate 2-itemsets, i.e., C_2, in each accumulative ongoing partition set P_i of transaction database, the CPU and memory overhead of SWF can be further reduced.

$\|\mathcal{D}\|$	Transaction No. in the database
$\|\Delta^+\|$	The added transaction No.
$\|\Delta^-\|$	The deleted transaction No.
$\|d\|$	The incremental transaction No.
$\|T\|$	Average size of the transactions
$\|I\|$	Average No. of frequent itemsets
$\|L\|$	No. of frequent itemsets
N	Number of items

Table 2: Meanings of various parameters

4. EXPERIMENTAL STUDIES

To assess the performance of algorithm SWF, we performed several experiments on a computer with a CPU clock rate of 450 MHz and 512 MB of main memory. The transaction data resides in the NTFS file system and is stored on a 30GB IDE 3.5" drive with a measured sequential throughput of 10MB/second. The simulation program was coded in C++. The methods used to generate synthetic data are described in Section 4.1. The performance comparison of SWF, FUP_2 and Apriori is presented in Section 4.2. Section 4.3 shows the I/O and CPU overhead among SWF, $\mathrm{FUP}_2\mathcal{H}$ and Apriori. Results on scaleup experiments are presented in Section 4.4.

4.1 Generation of synthetic workload

For obtaining reliable experimental results, the method to generate synthetic transactions we employed in this study is similar to the ones used in prior works [4, 8, 16, 21]. Explicitly, we generated several different transaction databases from a set of potentially frequent itemsets to evaluate the performance of SWF. These transactions mimic the transactions in the retailing environment. Note that the efficiency of algorithm SWF has been evaluated by some real databases, such as Web log records and grocery sales data. However, we show the experimental results from synthetic transaction data so that the work relevant to data cleaning, which is in fact application-dependent and also orthogonal to the incremental technique proposed, is hence omitted for clarity. Further, more sensitivity analysis can then be conducted by using the synthetic transaction data. Each database consists of $|D|$ transactions, and on the average, each transaction has $|T|$ items. Table 2 summarizes the meanings of various parameters used in the experiments. The mean of the correlation level is set to 0.25 for our experiments.

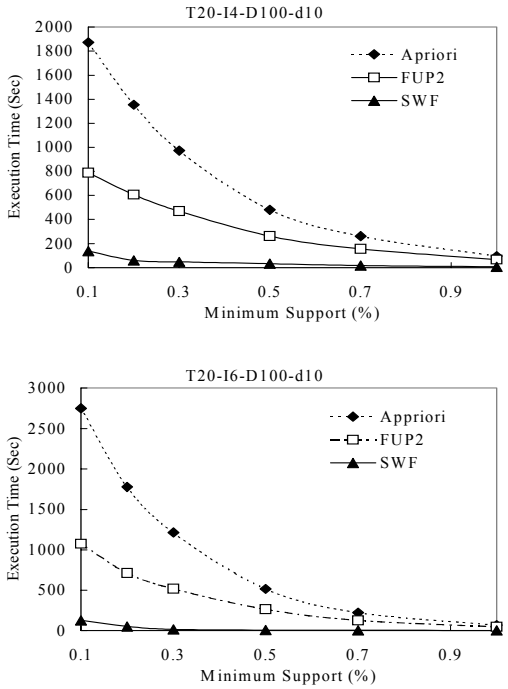

Figure 4: Relative performance

Recall that the sizes of $|\Delta^+|$ and $|\Delta^-|$ are not required to be the same for the execution of SWF. Without loss of generality, we set $|d| = |\Delta^+| = |\Delta^-|$ for simplicity. Thus, by denoting the original database as $db^{1,n}$ and the new mining database as $db^{i,j}$, we have $|db^{i,j}| = |db^{1,n} - \Delta^- + \Delta^+| = |\mathcal{D}|$, where $\Delta^- = db^{1,i-1}$ and $\Delta^+ = db^{n+1,j}$. In the following, we use the notation $Tx - Iy - Dm - dn$ to represent a database in which $D = m$ thousands, $d = n$ thousands, $|T| = x$, and $|I| = y$. We compare relative performance of three methods, i.e., Apriori, FUP-based algorithms and SWF.

As mentioned before, without any provision for incremental mining, Apriori algorithm has to re-run the association rule mining algorithm on the whole updated database. As reported in [8, 9], with reducing the candidate itemsets, FUP-based algorithms outperform Apriori. As will be shown by our experimental results, with the sliding window technique that carries cumulative information selectively, the execution time of SWF is, in orders of magnitude, smaller than those required by prior algorithms. In order to

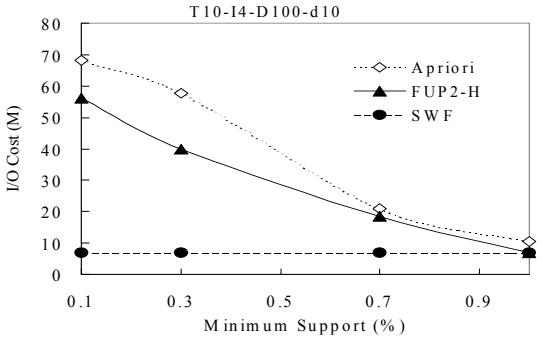

Figure 5: I/O cost performance

T10-I4-D100-d10				
Candidates	Apriori	FUP_2H	SWF	Freq. Itemsets
C_2	3399528	104145	7482	L_2=6656
C_3	8353	8353	9241	L_3=8135
C_4	7882	7882	8679	L_4=7616
C_5	6762	6382	7162	L_5=6077
C_6	5437	4709	5578	L_6=4658
C_7	3918	3417	3951	L_7=3412

Figure 6: Reduction on candidate itemsets when the dataset T10-I4-D100-d10 was used

conduct our experiments on a database of size $db^{i,j}$ with an increment of Δ^+ and a removal of Δ^-, a database of $db^{1,j}$ is first generated and then $db^{1,i-1}$, $db^{1,n}$, $db^{n+1,j}$, and $db^{i,j}$ are produced separately.

4.2 Experiment one: Relative performance

We first conducted several experiments to evaluate the relative performance of Apriori, FUP_2 and SWF. For interest of space, we only report the results on $|L| = 2000$ and $N = 10000$ in the following experiments. Figure 4 shows the relative execution times for the three algorithms as the minimum support threshold is decreased from 1% support to 0.1% support. When the support threshold is high, there are only a limited number of frequent itemsets produced. However, as the support threshold decreases, the performance difference becomes prominent in that SWF significantly outperforms both FUP_2 and Apriori. As shown in Figure 4, SWF leads to prominent performance improvement for various sizes of $|T|$, $|I|$ and $|d|$. Explicitly, SWF is in orders of magnitude faster than FUP_2, and the margin grows as the minimum support threshold decreases. Note that from our experimental results, the difference between FUP_2 and Apriori is consistent with that observed in [9]. In fact, SWF outperforms FUP_2 and Apriori in both CPU and I/O costs, which are evaluated next.

4.3 Experiment two: Evaluation of I/O and CPU overhead

To evaluate the corresponding of I/O cost, same as in [18], we assume that each sequential read of a byte of data consumes one unit of I/O cost and each random read of a byte of data consumes two units of I/O cost. Figure 5 shows the number of database scans and the I/O costs of Apriori, $FUP_2\mathcal{H}$, i.e., hash-type FUP in [9], and SWF over data sets $T10 - I4 - D100 - d10$. As shown in Figure 5, SWF outperforms Apriori and $FUP_2\mathcal{H}$ where without loss of generality a hash table of 250 thousand entries is employed for those methods. Note that the large amount of database scans is the performance bottleneck when the database size does not fit into main memory. In view of that, SWF is advantageous since only one scan of the updated database is required, which is independent of the variance in minimum supports.

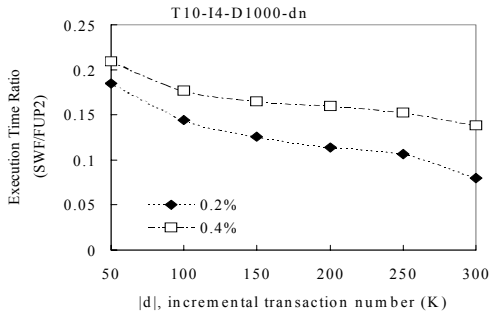

Figure 7: Scaleup performance with the execution time ratio between SWF and FUP

As explained before, SWF substantially reduces the number of candidate itemsets generated. The effect is particularly important for the candidate 2-itemsets. The experimental results in Figure 6 show the candidate itemsets generated by Apriori, $FUP_2\mathcal{H}$, and SWF across the whole processing on the datasets $T10-I4-D100-d10$ with minimum support threshold $s = 0.1\%$. As shown in Figure 6, SWF leads to a 99% candidate reduction rate in C_2 when being compared to Apriori, and leads to a 93% candidate reduction rate in C_2 when being compared to $FUP_2\mathcal{H}$. Similar phenomena were observed when other datasets were used. This feature of SWF enables it to efficiently reduce the CPU and memory overhead. Note that the number of candidate 2-itemsets produced by SWF approaches to its theoretical minimum, i.e., the number of frequent 2-itemsets. Recall that the C_3 in either Apriori or $FUP_2\mathcal{H}$ has to be obtained by L_2 due to the large size of their C_2. As shown in Figure 6, the value of $|C_k|$ $(k \geq 3)$ is only slightly larger than that of Apriori or $FUP_2\mathcal{H}$, even though SWF only employs C_2 to generate C_ks, thus fully exploiting the benefit of scan reduction.

4.4 Experiment three: Scaleup on the incremental portion

To further understand the impact of |d| to the relative performance of algorithms SWF and FUP-based algorithms, we conduct the scaleup experiments for both SWF and FUP_2 with two minimum support thresholds 0.2% and 0.4 %. The results are shown in Figure 7 where the value in y-axis corresponds to the ratio of the execution time of SWF to that of FUP_2. Figure 7 shows the execution-time-ratio for different values of |d|. It can be seen that since the size of |d| has less influence on the performance of SWF, the execution-time-

ratio becomes smaller with the growth of the incremental transaction number |d|. This also implies that the advantage of SWF over FUP_2 becomes even more prominent as the amount of incremental portion increases.

5. CONCLUSION

We explored in this paper an efficient sliding-window filtering algorithm for incremental mining of association rules. Algorithm SWF not only significantly reduces I/O and CPU cost by the concepts of cumulative filtering and scan reduction techniques but also effectively controls memory utilization by the technique of sliding-window partition. Extensive simulations have been performed to evaluate performance of algorithm SWF. With proper sampling and partitioning methods, the technique devised in this paper can be applied to progressive mining.

Acknowledgment

The authors are supported in part by the Ministry of Education Project No. 89-E-FA06-2-4-7 and the National Science Council, Project No. NSC 89-2219-E-002-028 and NSC 89-2218-E-002-028, Taiwan, Republic of China.

6. REFERENCES

[1] R. Agarwal, C. Aggarwal, and V.V.V. Prasad. A Tree Projection Algorithm for Generation of Frequent Itemsets. *Jornal 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. *Proc. of ACM SIGMOD*, pages 207–216, May 1993.

[3] R. Agrawal and R. Srikant. Fast Algorithms for Mining Association Rules in Large Databases. *Proc. of the 20th International Conference on Very Large Data Bases*, pages 478–499, September 1994.

[4] N.F. Ayan, A.U. Tansel, and E. Arkun. An Efficient Algorithm to Update Large Itemsets with Early Pruning. *Proc. of 1999 Int. Conf. on Knowledge Discovery and Data Mining*, 1999.

[5] S. Brin, R. Motwani, J. D. Ullman, and S. Tsur. Dynamic Itemset Counting and Implication Rules for Market Basket Data. *ACM SIGMOD Record*, 26(2):255–264, May 1997.

[6] M.-S. Chen, J. Han, and P. S. Yu. Data Mining: An Overview from Database Perspective. *IEEE Transactions on Knowledge and Data Engineering*, 8(6):866–883, December 1996.

[7] M.-S. Chen, J.-S. Park, and P. S. Yu. Efficient Data Mining for Path Traversal Patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, April 1998.

[8] D. Cheung, J. Han, V. Ng, and C.Y. Wong. Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique. *Proc. of 1996 Int'l Conf. on Data Engineering*, pages 106–114, February 1996.

[9] D. Cheung, S.D. Lee, and B. Kao. A General Incremental Technique for Updating Discovered Association Rules. *Proc. International Conference On Database Systems For Advanced Applications*, April 1997.

[10] J. Han, L. V. S. Lakshmanan, and R. T. Ng. Constraint-Based, Multidimensional Data Mining. *COMPUTER (special issues on Data Mining)*, pages 46–50, 1999.

[11] J. Han and J. Pei. Mining Frequent Patterns by Pattern-Growth: Methodology and Implications. *ACM SIGKDD Explorations (Special Issue on Scaleble Data Mining Algorithms)*, December 2000.

[12] J. Han, J. Pei, B. Mortazavi-Asl, Q. Chen, U. Dayal, and M.-C. Hsu. FreeSpan: Frequent pattern-projected sequential pattern mining. *Proc. of 2000 Int. Conf. on Knowledge Discovery and Data Mining*, pages 355–359, August 2000.

[13] J. Hipp, U. Güntzer, and G. Nakhaeizadeh. Algorithms for association rule mining – a general survey and comparison. *SIGKDD Explorations*, 2(1):58–64, July 2000.

[14] L. V. S. Lakshmanan, R. Ng, J. Han, and A. Pang. Optimization of Constrained Frequent Set Queries with 2-Variable Constraints. *Proc. of 1999 ACM-SIGMOD Conf. on Management of Data*, pages 157–168, June 1999.

[15] J.-L. Lin and M.H. Dunham. Mining Association Rules: Anti-Skew Algorithms. *Proc. of 1998 Int'l Conf. on Data Engineering*, pages 486–493, 1998.

[16] J.-S. Park, M.-S. Chen, and P. S. Yu. Using a Hash-Based Method with Transaction Trimming for Mining Association Rules. *IEEE Transactions on Knowledge and Data Engineering*, 9(5):813–825, October 1997.

[17] J. Pei and J. Han. Can We Push More Constraints into Frequent Pattern Mining? *Proc. of 2000 Int. Conf. on Knowledge Discovery and Data Mining*, August 2000.

[18] J. Pei, J. Han, and L.V.S. Lakshmanan. Mining Frequent Itemsets with Convertible Constraints. *Proc. of 2001 Int. Conf. on Data Engineering*, 2001.

[19] A. Savasere, E. Omiecinski, and S. Navathe. An Efficient Algorithm for Mining Association Rules in Large Databases. *Proc. of the 21th International Conference on Very Large Data Bases*, pages 432–444, September 1995.

[20] R. Srikant and R. Agrawal. Mining Generalized Association Rules. *Proc. of the 21th International Conference on Very Large Data Bases*, pages 407–419, September 1995.

[21] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka. An Efficient Algorithm for the Incremental Updation of Association Rules in Large Databases. *Proc. of 1997 Int. Conf. on Knowledge Discovery and Data Mining*, 1997.

[22] H. Toivonen. Sampling Large Databases for Association Rules. *Proc. of the 22th VLDB Conference*, pages 134–145, September 1996.

[23] A. K. H. Tung, J. Han, L. V. S. Lakshmanan, and R. T. Ng. Constraint-Based Clustering in Large Databases. *Proc. of 2001 Int. Conf. on Database Theory*, January 2001.

[24] K. Wang, Y. He, and J. Han. Mining Frequent Itemsets Using Support Constraints. *Proc. of 2000 Int. Conf. on Very Large Data Bases*, September 2000.