A Sliding-Window Approach to Mining Maximal Large Itemsets for Large Databases

Ye-In Chang

Chen-Chang Wu

Jiun-Rung Chen

Yuan-Feng Chang

Proceedings of The Seventh International Conference on Electronic Business, Taipei, Taiwan, December 2-6, 2007, pp.22-28

A SLIDING-WINDOW APPROACH TO MINING MAXIMAL LARGE ITEMSETS FOR LARGE DATABASES

Ye-In Chang, National Sun Yat-Sen University, Taiwan, changyi@cse.nsysu.edu.tw
Chen-Chang Wu, National Sun Yat-Sen University, Taiwan, wucc@db.cse.nsysu.edu.tw
Jiun-Rung Chen, National Sun Yat-Sen University, Taiwan, chenjr@db.cse.nsysu.edu.tw
Yuan-Feng Chang, National Sun Yat-Sen University, Taiwan, changyf@db.cse.nsysu.edu.tw

ABSTRACT

In this paper, we propose a Sliding-Window approach, the SWMax algorithm, which could provide good performance for both mining maximal itemsets and incremental mining. Our SWMax algorithm is a two-passes partition-based approach. For incremental mining, if an itemset with size equal to 1 is not large in the original database, it could not be found in the updated database based on the SWF algorithm. Our SWMax algorithm will support incremental mining correctly. From our simulation, the results show that our SWMax algorithm could generate fewer number of candidates and needs less time than the SWF algorithm.

Keywords: data mining, association rules, maximal large itemsets, incremental mining, partition.

INSTRUCTIONS

Mining association rules, means a process of nontrivial extraction of implicit, previously and potentially useful information from data in database, which has recently attracted tremendous amount of attention in the database research community [5]. The most important task is to discover association rules. The computer can transform the processed data into the useful information and knowledge. Therefore, *mining association rules* has become a research area with increasing importance [8] [11].

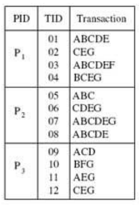
Mining Maximal Large Itemsets

Mining maximal large itemsets is a further work of mining association rules, which aims to find the set of all subsets of large (or frequent) itemsets that has representative of all large itemsets [2] [3] [9]. For example, there are three large itemsets, L_1, L_2 and L_3. L_1= {{A},{B},{C},{D}}. L_2= {{AB},{AC},{BC},{CD}}. L_3= {{ABC}}.The maximal large itemsets are {ABC} and {CD}, which can cover L_1, L_2 and L_3. The prototypical application is in *market basket analysis,* where the items represent products and records the point-of-sales data at large grocery or departmental stores.

Previous algorithms to mining maximal large itemsets can be classified into two approaches: exhausted and shortcut. For example, the Apriori and AprioriTid [1], and Partition [12], SWF [7] algorithms belong to the exhausted approach. They find out all large itemsets and compare the resulting itemsets, L_i. Then, the maximal large itemsets are found. The Max-Miner [2], MAFIA [3], Pincer-Search [9] algorithms belong to the shortcut approach. As compared with the exhausted approach, the shortcut approach could generate smaller number of candidate itemsets in every scan of the transaction database, resulting in better performance in terms of time and storage space.

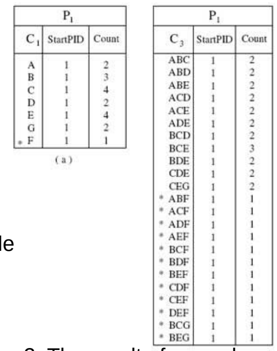
On the other hand, when updates to the transaction databases occur, one possible approach is to re-run the mining algorithm on the whole database. All the computation done at finding out the large itemsets are originally wasted and all large itemsets have to be computed again from scratch. The other approach is incremental mining, which aims for efficient maintenance of discovered association rules without re-running the mining algorithms when updates occur [11]. So, some people proposed a concept, *negative border*. The negative border is used to decide when to scan the whole database and it can be used in conjunction with any Apriori-like algorithm.

Therefore, in this paper, we focus on the design of an algorithm which could provide good performance for both mining maximal itemsets and incremental mining. Based on some observations, for example, `` *if an itemset is large, all its subsets must be large; therefore, those subsets need not to be examined further*'', we propose a Sliding-Window approach, the SWMax



PID	TID	Transaction
P_1	01	ABCDE
	02	CEG
	03	ABCDEF
	04	BCEG
P_2	05	ABC
	06	CDEG
	07	ABCDEG
	08	ABCDE
P_3	09	ACD
	10	BFG
	11	AEG
	12	CEG

Figure 1: An example transaction database.

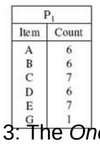


P_1		
C_1	StartPID	Count
A	1	2
B	1	3
C	1	4
D	1	2
E	1	4
G	1	2
* F	1	1

(a)

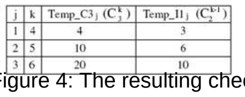
P_1		
C_3	StartPID	Count
ABC	1	2
ABD	1	2
ABE	1	2
ACD	1	2
ACE	1	2
ADE	1	2
BCD	1	2
BCE	1	3
BDE	1	2
CDE	1	2
CEG	1	2
* ABF	1	1
* ACF	1	1
* ADF	1	1
* AEF	1	1
* BCF	1	1
* BDF	1	1
* BEF	1	1
* CDF	1	1
* CEF	1	1
* DEF	1	1
* BCG	1	1
* BEG	1	1

Figure 2: The result of procedure *gen_count_itemset* after scanning P_1.



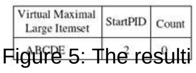
P_1	
Item	Count
A	6
B	6
C	7
D	6
E	7
G	1

Figure 3: The *OneI* Table after scanning P_1.



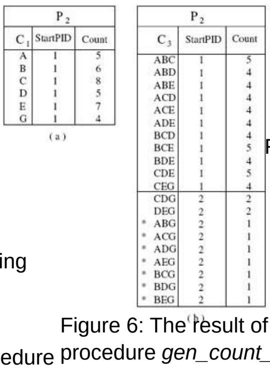
j	k	Temp_C3$_j$ (C_3^k)	Temp_I1$_j$ (C_2^{k-1})
1	4	4	3
2	5	10	6
3	6	20	10

Figure 4: The resulting checking table



Virtual Maximal Large Itemset	StartPID	Count
ABCDE	2	0

Figure 5: The resulti of procedure gen_VMLI after scanning P_1.



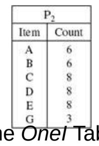
P_2		
C_1	StartPID	Count
A	1	5
B	1	6
C	1	8
D	1	5
E	1	7
G	1	4

(a)

P_2		
C_3	StartPID	Count
ABC	1	5
ABD	1	4
ABE	1	4
ACD	1	4
ACE	1	4
ADE	1	4
BCD	1	4
BCE	1	5
BDE	1	4
CDE	1	5
CEG	1	4
CDG	2	2
DEG	2	2
* ABG	2	1
* ACG	2	1
* ADG	2	1
* AEG	2	1
* BCG	2	1
* BDG	2	1
* BEG	2	1

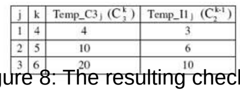
(b)

Figure 6: The result of procedure *gen_count_itemset* after scanning P_2.



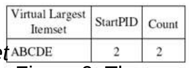
P_2	
Item	Count
A	6
B	6
C	8
D	8
E	8
G	3

Figure 7: The *OneI* Table after scanning P_2.



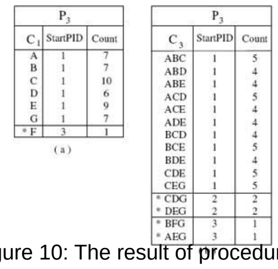
j	k	Temp_C3$_j$ (C_3^k)	Temp_I1$_j$ (C_2^{k-1})
1	4	4	3
2	5	10	6
3	6	20	10

Figure 8: The resulting checking table



Virtual Largest Itemset	StartPID	Count
ABCDE	2	2

Figure 9: The resulti of procedure gen_VMLI after scanning P_2.

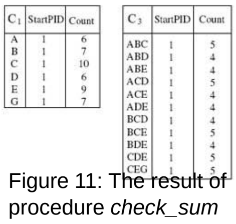


P_3		
C_1	StartPID	Count
A	1	7
B	1	7
C	1	10
D	1	6
E	1	9
G	1	7
* F	3	1

(a)

P_3		
C_3	StartPID	Count
ABC	1	5
ABD	1	4
ABE	1	4
ACD	1	5
ACE	1	4
ADE	1	4
BCD	1	4
BCE	1	5
BDE	1	4
CDE	1	5
CEG	1	5
* CDG	2	2
* DEG	2	2
* BFG	3	1
* AEG	3	1

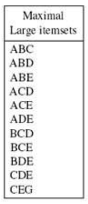
Figure 10: The result of procedure *gen_count_itemset* after scanning P_3.



C_1	StartPID	Count
A	1	6
B	1	7
C	1	10
D	1	6
E	1	9
G	1	7

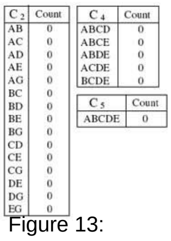
C_3	StartPID	Count
ABC	1	5
ABD	1	4
ABE	1	4
ACD	1	5
ACE	1	4
ADE	1	4
BCD	1	4
BCE	1	5
BDE	1	4
CDE	1	5
CEG	1	5

Figure 11: The result of procedure *check_sum* in P_3.



Maximal Large itemsets
ABC
ABD
ABE
ACD
ACE
ADE
BCD
BCE
BDE
CDE
CEG

Figure 12:

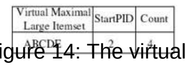


C_2	Count
AB	0
AC	0
AD	0
AE	0
AG	0
BC	0
BD	0
BE	0
BG	0
CD	0
CE	0
CG	0
DE	0
DG	0
EG	0

C_4	Count
ABCD	0
ABCE	0
ABDE	0
ACDE	0
BCDE	0

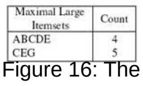
C_5	Count
ABCDE	0

Figure 13:



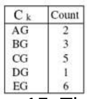
Virtual Maximal Large Itemset	StartPID	Count
ABCDE	2	4

Figure 14: The virtual maximal large itemset.



Maximal Large Itemsets	Count
ABCDE	4
CEG	5

Figure 16: The final result.



C_k	Count
AG	2
BG	3
CG	5
DG	1
EG	6

Figure 15: The result of candidate

algorithm, for efficiently mining maximal large itemsets and incremental mining.

Our SWMax algorithm is a two-passes partition-based approach. We will find all candidate 1-itemsets (C_1,) candidate 3-itemsets (C_3,) large 1-itemsets (L_1), and large 3-itemsets (L_3) in the first pass. We generate the virtual maximal large itemsets after the first pass. Then, we use L_1 to generate C_2, use L_3 to generate C_4, use C_4 to generate C_5, until there is no C_K generated. In the second pass, we use the virtual maximal large itemsets to prune C_K, and decide the maximal large itemsets. For incremental mining, we consider two cases: data insertion and data deletion. If an itemset with size equal to 1 is not large in the original database, it could not be found in the updated database based on the SWF algorithm. A missing case could occur in the incremental mining process of the SWF algorithm, while our SWMax algorithm could support incremental mining correctly.

The rest of paper is organized as follows. Section 2 presents the proposed SWMax algorithm. In Section 3, we study the performance and make a comparison of the proposed algorithms, SWMax algorithm with some other previous proposed algorithms. Finally, Section 4 gives the conclusions.

THE SWMax ALGORITHM

In this section, we first describe some interesting observations, which have motivated us to design our SWMax algorithm. Next, we present the proposed SWMax algorithm. Then, based on some other observations, we present the approach of incremental mining in the SWMax algorithm.

Interesting Observations for Mining Maximal Large Itemsets

The Pincer-Search algorithm has presented an interesting observation: If an itemset is frequent, all its subsets must be frequent; therefore, those subsets need not to be examined further. The number of the 3-itemset subsets of a *k*-itemset should be C_3^k . In other words, if the number of 3-itemset subsets is smaller than C_3^k, then such a *k*-itemset will not be frequent and should be discarded from the candidate of maximal large itemsets.

Moreover, we find another interesting observation. Let's use the following example to show this observation. For each item in these 3-itemset subsets of the frequent 5-itemset {ABCDE}, we could find that the occurrence of each item is 6 (=C_2^{5-1}). For example, item A occurs 6 times among itemsets {ABC}, {ABD}, {ABE}, {ACD}, {ACE}, and {ADE}. That is, the number of the occurrence of item A in the 6 3-itemset subsets of the frequent 5-itemset {ABCDE} is C_2^{5-1}. In general, the number of the occurrence of a certain item in these C_3^k 3-itemset subsets of a frequent *k*-itemset is C_2^{k-1}. We will make use of these two observations in our SWMax algorithm.

Sketch of Our Algorithm

From the above observations, we develop our SWMax algorithm for mining maximal large itemsets. We use an example database shown in Figure 1 to illustrate our algorithm which is a partition-based approach. Those 12 (= *N*) data records are divided into 3 (= *PN*) partition. Therefore, the number of transactions, *NT*, in each partition P_i is 12/3 = 4, $1 \le i \le 3$. The global support for the total 12 transactions is *s* = 26%. When the number of transactions containing an itemset is greater than or equal to $\lceil 12 * 26\% \rceil = 4$, this itemset is frequent.

In the first pass, we have to consider the local support number, *LocalS*, for candidates generated in partition P_i. The value of *LocalS* is equal to (number of transactions in partitions P_1, P_2, …, P_i) times *s*. For example, we have *LocalS* = $\lceil 4 * 26\% \rceil = 2$ for candidate generated in partition P_1, when we scan partition P_1. We have *LocalS* = $\lceil 8 * 26\% \rceil = 3$ in partition P_1 and *LocalS* = $\lceil 4 * 26\% \rceil = 2$ in partition P_2, when we scan partition P_2.

In pass 1, we aim to find all temporary large 1-itemsets and 3-itemsets. We scan the whole database by focusing on each partition P_i in sequence, $1 \le i \le PN$. When we scan the item list of each transaction in partition P_1, we will generate temporary candidate 1-itemsets and 3-itemsets, and record the result in variables *Temp_C1I* and *Temp_C3I*, respectively, by calling procedure *gen_count_itemset*. We add a temporary candidate *X* into *Temp_C1I /Temp_C3I* only if *X* does not occur in *Temp_C1I /Temp_C3I*. When a temporary candidate *X* is added to *Temp_C1I /Temp_C3I*, we also record its *StartPID* as PID, where *StartPID* means the starting PID in which the candidate is generated. Moreover, we increase the count of such a candidate by one.

After all transactions in partition P_1 is scanned, we call procedure *check_sum*. A temporary candidate *X* can become the formal candidate if *X*.count > *LocalS*, when *LocalS* is the local support of partition, *X.PID*. If a temporary candidate could not pass the checking step, it is removed from *Temp_C1I/ Temp_C3I*. The final results as shown in Figure 2 has removed all these itemsets which are marked with *.

Up to this point, if the scanned partition is not the last partition, we will generate virtual maximal large itemsets, *gen_VMLI*, from *Temp_C1I* and *Temp_C3I* as shown in Figure 4. In procedure *gen_VMLI*, for each 3-itemset element *X* in *Temp_C3I*, we will check whether each of elements *Y* in *X* is in the table *OneI* or not. If it occurs the first time, we will add element *Y* to table *OneI*. Moreover, we will count the occurrence of each element *Y*. The result for this example is shown in Figure 3. After we get the count of each 1-item element occurring in candidate 3-itemset C_3, we start to call procedure *gen_check_table* to create a checking table as shown in Figure 4, to help us determine the virtual maximal large itemset. In procedure *gen_check_table(ND)*, we generate the values of C_3^k and $C_2^{k-1}, 4 \le k \le ND$. The purpose of these values has been explained in our observations as described before. For this example, we have *ND* = 6, since the size of table *OneI* is 6, which implies the size of the possible virtual maximal large itemset is limited to 6. From Figure 3, the possible case is itemset {ABCDEG}. To decide whether itemset {ABCDEG} is the virtual maximal large itemset, we could make the decision based on these observations as mentioned in Section 3.1 : *If an k-itemset is large, all its subsets must be frequent*. That is , those C_3^k 3-itemsets subset of a frequent *k*-itemset should also be frequent. Moreover, all the *k* items will have C_2^{k-1} occurrences among these frequent 3-itemsets. For example, if {ABCDEG} is a large itemset, it should have C_3^6 = 20 3-itemsets subsets of this 6-itemset {ABCDEG}. Moreover, item {A} will appear in {ABC}, {ABD}, {ABE}, {ABG}, {ACD}, {ACE}, {ACG}, {ADE}, {ADG} and {AEG}. That is, item A will occur C_2^5 = 10 times. Item {B}, item {C}, item {D}, item {E} and item {G} will have the same case as item {A}. That is the reason why we dissolve the 3-itemset of *Temp_C3I* to get the individual items, and stored them and related counts in table *OneI*, as shown in Figure 3.

Since *ND* = 6, in fact, there may exist 3 possibilities of the size of the virtual maximal large itemset, 4, 5, 6. Due to that the size of passed candidate C_3, | *Temp_C3I* | = 11, the size of a virtual maximal large itemset could not be 6, since 11 < C_3^6 (= 20). Based on the same reason, the size of a virtual maximal large itemset could be 5, since $10\ (= C_3^5) \le 11 < 20\ (= C_3^6)$. Therefore, we decide the size of the virtual maximal large itemset should be 5 by calling function *CheckRange*(*W*) with *W* = 6, due to $Temp\_C3_2$ (= 10) ≤ | *Temp_C3I*|(= 11) < $Temp\_C3_3$ (= 20). Moreover, we find the corresponding occurrence of each 1-item in such a virtual maximal large itemset with size = 5 should be 6 times. That is, we choose the one, $Temp\_I1_2$ = 6, as the threshold, *threshold1*, for the times of the occurrence of 1-item in *Temp_C3I*, since $Temp\_C3_j = Temp\_C3_2 = C_3^5 = C_3^k$ = 10, where *k* = 5 and *j* = 2. In the following for loop, we check whether the count of the 1-item stored in table *OneI* is greater than or equal to such a threshold, If element *Y.count* satisfies this condition, we then concatenate *Y* with in the string variable *Z*. In our example, element *G* does not satisfy this condition. Therefore, finally, we have the virtual maximal large itemset *Z* = {ABCDE} with *StartPID* = 2. We store this result in table *Virtual_MLI*.

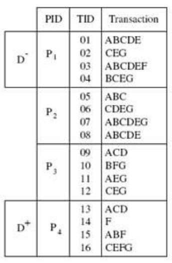
Similarly, transactions in partition P_2 are scanned and procedure *gen_count_itemset* is called to generate new 1-itemsets and candidate 3-itemsets if it is possible. Then, we call procedure *checksum* to check whether candidate 1-itemsets/3-itemsets generated in partitions P_1 and P_2 could be the passed candidates or not. After passed candidate 1-itemsets/3-itemsets are determined, we try to generate the virtual maximal large itemset from the result scanned so far. We call procedure *gen_VMLI* to generate the checking table again. Similarly, we have to construct the *OneI* table first for the result stored in *Temp_C3I*. Then, the checking table is created as shown in Figure 8. At this time, we have 13 3-itemsets generated. Therefore, the threshold is 6 and the virtual maximal large itemset is {ABCDE} again as shown in Figure 9. Because the virtual maximal large itemset {ABCDE} is already generated in partition P_1, it will not be added into *Virtual_MLI* at this time.

Similarly, we scan transactions in partition P_3, generate candidates as shown in Figure 10 and determine passed candidate 1-itemsets/3-itemsets as shown in Figure 11. However, for partition P_3, the last partition, we do not try to generate the virtual maximal large itemset. Because the virtual maximal large itemset generated in the last partition will scan the same number of partitions with candidate itemsets of C_k in the second scan. For example, if we have a virtual maximal large itemset, {ABCDE}, generated after scanning partition P_3. We will record *StartPID* = 1, and proceed the second scan. After the second scan, we have the final count of both each itemset of C_k and {ABCDE}. Then, we just check the count of each itemset of C_k to find out the final result of *MLI*, because {ABCDE} will also be one itemset of C_k. Therefore, the virtual maximal large itemset {ABCDE} generated after scanning the last partition in the first pass is useless.

After finishing the first scan of the transaction database, we will generate all candidate *k*-itemsets. Before we generate candidate *k*-itemsets, we can check each itemset in *Temp_C3I* and *TC1I*. If there is any itemset with *StartPID* = 1, it means that these itemsets have been finished a complete scan of the transaction database. These itemsets are L_1 or L_3. Therefore, they do not need to be counted anymore, and we add them into *MLI* which represents the maximal large itemsets, as shown in Figure 12. The procedure *gen_CK* uses L_1 joins L_1 to generate C_2, L_3 joins L_3 to generate C_4, and C_4 joins C_4 to generate C_5. Up to this point, since C_5 joins C_5 resulting in the empty set. We generate candidate *k*-itemsets by the *join* function. The basic idea of the *join* function is similar to that in the Apriori algorithm. That is, given itemsets $\{a_1a_2...a_xa_y\}$ and $\{a_1a_2...a_xa_z\}$, the result of joining the two itemsets is $\{a_1a_2...a_xa_ya_z\}$. For example, {ABC} and {ABE} will generate {ABCE}.

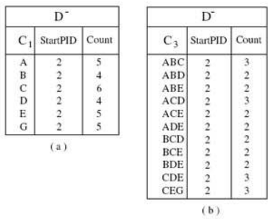
Then, the second scan of the transaction database is proceeded. We will check each itemset of *Virtual_MLI* for each partition. We can find an itemset of *Virtual_MLI* to be large as early as possible, such that some candidate itemsets can be reduced. Therefore, whether itemset {ABCDE} is the large itemset could be decided now after a complete scan of the transaction database, as shown in Figure 14. The count of the itemset {ABCDE} is 4 (> 12 * 26%), so the itemset {ABCDE} is large. Then, we call procedure *findMLI* to prune candidates by making use of the result {ABCDE} stored in *findMLI*.

In procedure *findMLI*, we check the sum of each itemset in *All_CKI*. If an item *X* is large and has not occurred in *MLI* before, we call function *remove* to remove all subsets *Y* of *X* from *MLI*. For example, *MLI* contains {{AB}, {CDE}}, and we want to add {ABC} into *MLI*. In addition to checking whether {ABC} has existed in *MLI* or not, we also will remove all the subsets of {ABC}, *e.g.*, {AB} from *MLI*. Then, we add itemset *X* into *MLI*. Moreover, we remove itemset *X* from *Virtual_MLI*. Finally, we call procedure *reduce* to prune unnecessary candidates from *All_CKI*. A candidate *X* in *All_CKI* is unnecessary if there exists large itemset *Y* in *MLI* and $X \in Y$. Moreover, if an itemset *X* in *All_CKI* is large, we call function *remove* to remove all subsets of itemset *X* from *MLI*. Then, we add itemset *X* to *MLI* and remove it from *All_CKL*. The whole process of making use of large itemsets generated from *Virtual_MLI* to prune *MLI*, making use of those large itemsets in *MLI* to prune all candidates in *All_CKI*, and making use of large itemset from *All_CKI* to prune *MLI* for transaction data in partition P_i scanned so far. Therefore, we use itemset {ABCDE} to proceed procedures *findMLI* and then *reduce*, which removes all subsets of {ABCDE} in *All_CKI*. Then, we repeat the same steps, counting and pruning, for data in partitions 2 and 3. Finally, the second scan is finished, the result is shown in Figure 15. If *All_CKI* is not an empty set, we will make the last comparison by calling procedure *reduce*. The final content of *MLI* is shown in Figure 16, which contains all maximal large itemsets.



	PID	TID	Transaction
D^-	P_1	01	ABCDE
		02	CEG
		03	ABCDEF
		04	BCEG
	P_2	05	ABC
		06	CDEG
		07	ABCDEG
		08	ABCDE
	P_3	09	ACD
		10	BFG
		11	AEG
		12	CEG
D^+	P_4	13	ACD
		14	F
		15	ABF
		16	CEFG

Figure 17: The example of the transaction database for the incremental mining process



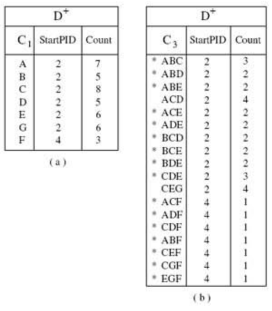
D^-		
C_1	StartPID	Count
A	2	5
B	2	4
C	2	6
D	2	4
E	2	5
G	2	5

(a)

D^-		
C_3	StartPID	Count
ABC	2	3
ABD	2	2
ABE	2	2
ACD	2	3
ACE	2	2
ADE	2	2
BCD	2	2
BCE	2	2
BDE	2	2
CDE	2	3
CEG	2	3

(b)

Figure 18: The result after scanning the deleted partition



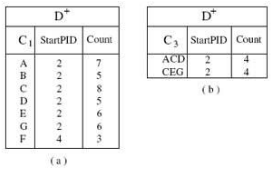
D^+		
C_1	StartPID	Count
A	2	7
B	2	5
C	2	8
D	2	5
E	2	6
G	2	6
F	4	3

(a)

D^+		
C_3	StartPID	Count
* ABC	2	3
* ABD	2	2
* ABE	2	2
ACD	2	4
* ACE	2	2
* ADE	2	2
* BCD	2	2
* BCE	2	2
* BDE	2	2
* CDE	2	3
CEG	2	4
* ACF	4	1
* ADF	4	1
* CDF	4	1
* ABF	4	1
* CEF	4	1
* CGF	4	1
* EGF	4	1

(b)

Figure 19: The result after scanning the added



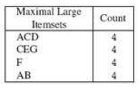
D^+		
C_1	StartPID	Count
A	2	7
B	2	5
C	2	8
D	2	5
E	2	6
G	2	6
F	4	3

(a)

D^+		
C_3	StartPID	Count
ACD	2	4
CEG	2	4

(b)

Figure 20: The result of procedure check sum.



Maximal Large Itemsets	Count
ACD	4
CEG	4
F	4
AB	4

Figure 21: The maximal large itemsets in the incremental mining process.

Interesting Observations for Incremental Mining

In this subsection, we discuss some interesting observation for incremental mining. Let's D^- be the deleted database, D^+ be the inserted database, and D' is the unchanged part of the original database.

Let's consider two cases when an updated to the database occurs: data insertion and data deletion. First, for data insertion, if an itemset with size equal to 1 is not large in the old database, it could not be found in the new database in the SWF algorithm. Second, for data deletion, if an itemset with size equal to 1 is not large in the old database, it could not be found in the new database on the SWF algorithm. Because the SWF algorithm has no way to update C_1 from D' when a data insertion occurs or a data deletion occurs. Therefore, a missing case could occur in the SWF algorithm.

Incremental Mining Process

We use the transaction database, as shown in Figure 17, to illustrate the incremental mining process. D^- means the deleted partition database, and D^+ means the added database. The SWMax algorithm reserves the candidate 3-itemsets, C_3, and the candidate 1-itemsets, C_1, for the incremental mining process. For the case of data deletion, we find out these itemsets *X*, where *X.StartPID* = 1, $Y \in D^-$, $X \in Y$, and decrease their counts. The result is shown in Figure 18.
Figure

For the case of data insertion, we scan D^+ partition to generate new candidate itemsets and increase the corresponding counts of itemsets recorded in the reserved C_1 and C_3 by procedure *gen_count_itemset*. Then, we check C_1 and C_3 with the *LocalS*, and find out the L_1 and L_3. Therefore, we use L_1 to generate C_2, use L_3 to generate C_4, use C_4 to generate C_5, until there is no C_k generated. Then, we use C_k to scan the partitions P_2, P_3, and P_4. All the iterations of the second scan are same as we have mentioned in the process of mining maximal large itemsets. Finally, we compare all candidate *k*-itemsets, C_k, and get the maximal large itemsets, as shown in Figure 21.

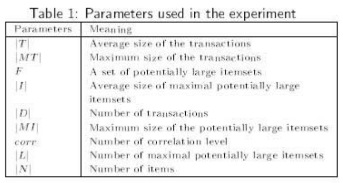
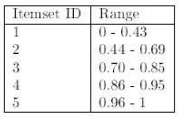


Table 1: Parameters used in the experiment

Parameters	Meaning
$\|T\|$	Average size of the transactions
$\|MT\|$	Maximum size of the transactions
F	A set of potentially large itemsets
$\|I\|$	Average size of maximal potentially large itemsets
$\|D\|$	Number of transactions
$\|MI\|$	Maximum size of the potentially large itemsets
$corr$	Number of correlation level
$\|L\|$	Number of maximal potentially large itemsets
$\|N\|$	Number of items

Table1: Parameters used in the experiment..



Itemset ID	Range
1	0 - 0.43
2	0.44 - 0.69
3	0.70 - 0.85
4	0.86 - 0.95
5	0.96 - 1

Table2: The probabilities of itemsets after

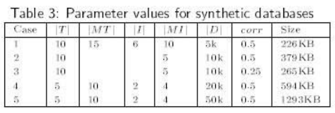


Table 3: Parameter values for synthetic databases

Case	$\|T\|$	$\|MT\|$	$\|I\|$	$\|MI\|$	$\|D\|$	$corr$	Size
1	10	15	6	10	5k	0.5	226KB
2	10			5	10k	0.5	379KB
3	10			5	10k	0.25	265KB
4	5	10	2	4	20k	0.5	594KB
5	5	10	2	4	50k	0.5	1293KB

Table3: Parameters values for synthetic databases.

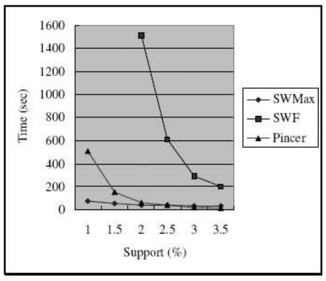


Figure 22: A comparison of the execution time (Case 1).

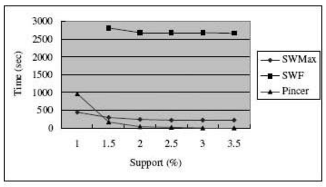


Figure 23: A comparison of the execution time (Case 2).

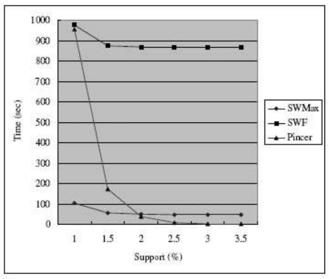


Figure 24: A comparison of the execution time (Case 3).

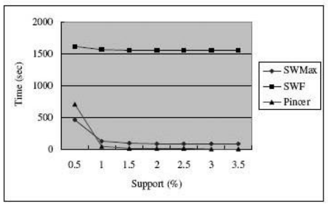


Figure 25: A comparison of the execution time (Case 5).

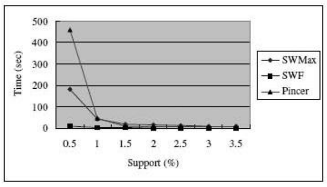


Figure 26: A comparison of the execution time for incremental mining (Case 5).

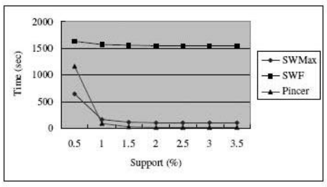


Figure 27: A comparison of the total execution time for mining maximal large itemsets and incremental mining (Case 5).

PERFORMANCE

In this section, first, we show how to generate the synthetic data which will be used in the simulation. Then, we study the performance of the Pincer-Search algorithm, the SWF algorithm and our SWMax algorithm. Finally, we make a comparison between these three algorithms.

Generation of Synthetic Data

We generated several different transaction databases from a set of large itemset to evaluate the performance of algorithms for mining maximal large itemsets. The parameters used in the generation of the synthetic data are shown in Table 1 [4].

First, the length of a transaction is determined by Poisson Distribution with a mean which is equal to | *T* |. The size of a transaction is between 1 and | *MT* |. The transaction is repeatedly assigned items from a set of potentially maximal large itemsets *F*. Then, the length of an itemset in *F* is determined according to the Poisson Distribution with a mean which is equal to | *I* |. The size of each potentially large itemset is between 1 and | *MI* |.

To model the phenomenon that large itemsets often have common items,we use an exponentially distributed random variable with a mean which is equal to the *correlation level,* to decide this fraction for each itemset. The *correlation level* was set to 0.5. The remaining items are chosen randomly. Each itemset in *F* has an associated weight that determines the probability that this itemset will be chosen. The weight is chosen from an exponential distribution with a mean equal to 1. The weights are normalized such that the sum of all weights equal to 1. These probabilities shown in Table 2 are then accumulated such that each value, which falls in these ranges, is used to select the itemsets.

For each transaction, we generate a random real number which is between 0 and 1 to determine the ID of potentially large itemset. To model the phenomenon that all the items in a large itemset are not always bought together, we assign each itemset in *F* with a *corruption level c*. When adding an itemset to a transaction, we keep dropping an item from the itemset until a uniformly distributed random number between 0 and 1 is less than *c*. The corruption level for an itemset is fixed, which is obtained from a normal distribution with mean = 0.5 and variance = 0.1. Each transaction is stored in a text file with the form of <transaction ID, item>.

Some different datasets were generated to be used in the simulation, Table 3 shows the parameters setting for each dataset. For all datasets, |*N*| was set to 1,000 and |*L*| was set to 2,000.

Experiments

Our experiments were performed on a Pentium 4 server with CPU clock rate of 1.5G MHz, 384MB of main memory, running Windows XP Service Pack 1. The transaction data is stored on a 40GB IDE 3.5" drive with a measured sequential throughput of 10MB/second. The simulation program was coded in JAVA, and compiled by JDK 1.4.2. The data was stored in a text file on a local hard disk drive.

A Comparison

For the synthetic database of Case 1, Figure 22 shows a comparison of the execution time with different values of the support between the SWF and our SWMax algorithm. The execution time of our SWMax algorithm is always less than that of the SWF algorithm. When the support decreases, the execution time of our SWMax algorithm is less than that of the Pincer-Search algorithm.

For the synthetic database of Case 2 and Case 3, these transaction databases are used to measure the influence of the *correlation level*. The results are shown in Figure 23 and Figure24. We find an interesting observation that the correlation level has no influence on the Pincer-Search algorithm, because the Pincer-Search algorithm uses L_k to generate C_{k+1}. However, our SWMax algorithm generates candidates from transactions. Therefore, in our SWMax algorithm, the *correlation level* increases, the candidates increases. However, when *support* $\leq$ 1%, the number of candidates of the Pincer-Search algorithm increases a lot, our SWMax algorithm needs shorter execution time than the Pincer-Search algorithm.

For the synthetic database of Case 5, this transaction database has a small size of large *k*-itemsets. The candidate 2-itemsets generated from the SWF algorithm could keep the useful information efficiently. However, the counting approach in our SWMax algorithm works more efficiently than the one in the SWF algorithm. For the SWF algorithm, which uses a hash array to do counting and cannot count efficiently. Our SWMax algorithm uses the hash tree to do counting. Figure 26 shows a comparison of the execution time among the SWF, Pincer-Search and our SWMax algorithms. From the result shown in Figure 25, our SWMax algorithm always requires less execution time than the SWF algorithm. However, in Case 5, the Pincer-Search algorithm has the same times of scanning the transaction database. Table6 shows the number of checks in Case 5. It is obviously to see that our SWMax needs more number of checks than the other two algorithms. Therefore, our SWMax algorithm needs longer execution time than the Pincer-Search algorithm when *support* $\geq$1%.

For the synthetic database of Case 4 and Case 5, we show the comparison of mining maximal large itemsets and incremental mining of mining maximal large itemsets among these three algorithms. Figure 26 shows the different small values of support to proceed incremental mining of mining maximal large itemsets. Figure 27 shows the total execution time for mining maximal large itemsets and incremental mining of mining maximal large itemsets. We know that if the support is small enough, the number of candidates of the Pincer-Search algorithm increases rapidly. Therefore, the time of re-running the mining approach takes longer time than the incremental mining approach. In other words, if the transaction database has large size of maximal large itemsets, *e.g.*, Case 1, our SWMax algorithm needs shorter time to do incremental mining of mining maximal large itemsets than the Pincer-Search algorithm. However, the SWF algorithm reserved all the information of C_2 in Case 5, so it needs so short time for incremental mining of mining maximal large itemsets.

Obviously, when the support is small, our SWMax algorithm needs shorter execution time than the other two algorithms in all the cases.

CONCLUSION

In this paper, we have proposed the SWMax algorithm to efficiently support both mining maximal large itemsets and incremental mining. We have presented the concept of the virtual maximal large itemset. Our SWMax algorithm is a two-passes partition-based approach, and the virtual maximal large itemsets help us to reduce the number of candidate itemsets in the second scan. The simulation results have shown that the proposed SWMax algorithm outperforms the SWF algorithm in all relational database settings. How to extend our SWMax algorithm for distributed processing is our future work.

ACKNOWLEDGEMENT

This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-93-2213-E-110-003 and National Sun Yat-Sen University. The authors also like to thank "Aim for Top University Plan" project of NSYSU and Ministry of Education, Taiwan, for partially supporting the research.

REFERENCES

[1] Agrawal, R. and Srikant, R. (1994) “Fast Algorithms for Mining Association Rules”, Proc. of the 20th Int. Conf. on Very Large Data Bases.

[2] Bayardo, R.J. (1998) “Efficiently Mining Long Patterns from Databases”, Proc. of Int. Conf. on Data Eng., pp. 85-93.

[3] Burdick, D., and Calimlim, M. and Gehrke, J. (2001) “MAFIA: A Maximal Frequent Itemset Algorithm for Transactional Databases”, Proc. of Int. Conf. on Data Eng, pp. 443-452.

[4] Chang, Y.I. and Hsieh, Y.M. (2002) “SETM*-Lmax: An Efficient SET-Based Approach to Find Maximal Large Itemsets”, Proc. of Int. Conf. on Computer Symposium: Workshop on Software Eng. and Database Systems.

[5] Chen, M.S., Han, J. and Yu, P.S. (1996) “Data Mining: An Overview from A Database Perspective”, IEEE Trans. on Knowledge and Data Eng., Vol. 8, No. 5, pp. 866-882.

[6] Cheung, D.W., Han, J., Ng, V.T. and Wong, C.Y. (1996) “Maintenance of Discovered Association Rules in Large Databases: An Incremental Updating Technique”, Proc. of the 12th IEEE Int. Conf. on Data Eng.

[7] Lee, C. H., Lin, C. R. and Chen, M. S. (2001) “Sliding-Window Filtering: An Efficient Algorithm for Incremental Mining”, Proc. of the 10th ACM Int. Conf. on Information and Knowledge Management, pp. 263-270

[8] Lian, W., Cheung D.W. and Yiu, S.M. (2007) “Maintenance of Maximal Frequent Itemsets in Large Databases”, Proc. of ACM Symp. on Applied Computing, pp. 388-392.

[9] Lin, D. I. and Kedem, Z. M. (2002) “Pincer-Search: An Efficient Algorithm for Discovering the Maximum Frequent Set”, IEEE Trans. on Knowledge and Data Eng, Vol. 14, No. 3.

[10] Pudi, V. and Haritsa, J. R. (2002) “Quantifying The Utility of The Past in Mining Large Databases”, Information Systems, Vol. 25, No. 5, pp. 323-343.

[11] Sarda, N.L. and Srinivas, N.V. (1998) “An Adaptive Algorithm for Incremental Mining of Association Rules”, Proc. of the 9th Int. Workshop on Database and Expert Systems Applications, pp. 240-246.

[12] Savasere, A., Omiecinski, E. and Navath, S. (1995) “An Efficient Algorithm for Mining Association Rules in Large Databases”, Proc. of the 21st VLDB Conf., pp. 432-444.

[13] Tsay, Y.J. and Chang-Chien, Y.W. (2004) “An Efficient Cluster and Decomposition Algorithm for Mining Association Rules”, Information Sciences, Vol. 160, pp. 161-171.

[14] Zhang, M., Kao, B., Cheung, D. and Yip, C. L. (2002) “Efficient Algorithms for Incremental Update of Frequent Sequences”, Proc. of the 6th Pacific-Asia Conf. on Knowledge Discovery and Data Mining, pp. 186-197.