USING REDUCED ITEMSET LATTICE TO SUPPORT ONLINE MINING FOR DYNAMIC DATABASES

Kuen-Fang J. Jea *Chi-Hau Hsu*

Institute of Computer Science, National Chung Hsing University,

250 Kuo-Kuan Road, Taichung, Taiwan, R.O.C.

Email: {kfjea, frankhsu}@cs.nchu.edu.tw

ABSTRACT

Current online techniques of mining association rules have incurred the problems of huge space requirement for the itemset lattice and of limiting the number of items in rules or producing an incomplete set of rules. In this paper, we propose an improved structure, namely RIL (Reduced Itemset Lattice), to keep just enough itemsets to support online mining for dynamic databases. Once built with the original database, the RIL incrementally adapts itself to database updates. Instead of rebuilding RIL, a complete set of association rules can be directly mined, or expanded first if necessary, from the RIL in a short time. Experimental evaluation shows that the RIL can effectively reduce the space of itemset lattice and mine the rules online for dynamic databases.

1. INTRODUCTION

Mining association rules [2,9] from databases of transactions has become an important and active research area in data mining. It discovers the association relationship of items or attributes in the database transactions and generates implication rules among them, such as "80% of customers who buy milk also buy bread" in the supermarket database. The methods of mining association rules usually follow Apriori algorithm [3]. In each iteration, the algorithm combines related items or itemsets into potential itemsets and counts their occurrences by scanning the database. They are identified as large itemsets for the next iteration of the algorithm if their occurrences exceed a user specified threshold (minimum support). The rules are finally generated from the large itemsets.

Although Apriori algorithm offers a simple and iterative way to generate association rules from databases, it incurs the inefficiency problem due to too many database scans and itemset combinations. This problem may make it difficult to mine new association rules online as the user changes the threshold or modifies the database. In order to mine new rules online and avoid re-mining the whole database, techniques like online mining [1,4] and incremental mining [5,6,10] are proposed.

Online mining [1,4] generates rules online as the user changes the threshold by pre-computing the occurrences of all itemsets and keeping them in an itemset lattice. The itemset lattice, as shown in Figure 1, is a special lattice data structure, where its nodes represent items or itemsets (i.e., combinations of items) and each link indicates a superset-subset relationship. Once a full itemset lattice is built with the occurrence of all itemsets pre-computed and stored, the mining algorithm can accept user-specified thresholds and online generates rules without any database scan. However, since the itemset lattice may take a lot of memory as the number of items is large, the idea of saving a full itemset lattice is unattractive and how to prune the lattice becomes an important issue in the online mining.

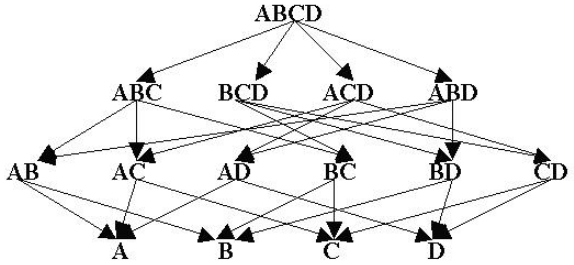


Figure 1. The full itemset lattice for 4 attributes A, B, C, D.

Incremental mining [5,6,10] is the techniques to adjust the pre-mined rules according to the database update transactions. When the database is updated, traditional mining algorithms need to re-mine the database in order to get the rules that match the database. On the other hand, incremental mining first re-computes all the strength of existing rules and erases those rules below the threshold. It then checks all small 1-itemsets, identifies those becoming large and combines them into candidate itemsets. A procedure similar to Apriori algorithm is finally applied to generate new large itemsets and new rules. Although incremental mining may reduce the re-mining cost with fewer database scans, it is still possible to combine itemsets and scan the database many times, which can result in long response time.

In dynamic databases where data may be changed frequently, existing incremental mining techniques do not guarantee a fast generation of new association rules (i.e., support mining online). If the idea of itemset lattices developed in the online mining can be integrated into incremental mining techniques for saving and incrementally updating the occurrence of itemsets, the number of database scans can be reduced or even eliminated. However, storing the full itemset lattice may take a lot of memory space, as mentioned before, it is therefore the purpose of this study to propose a new and space-saving structure, namely the reduced itemset lattice (RIL), to efficiently support online mining of association rules for dynamic databases.

This paper is organized as follows. Section 1 describes the background, motivation and goal of this study. Section 2 presents the related work on both online mining and incremental mining. Section 3 describes the RIL, how to build RIL and use RIL to mine association rules online. Section 4 presents the incremental algorithm of adjusting RIL for dynamic databases and Section 5 shows our performance experiments and compares our approach with existing techniques. Finally Section 6 gives our conclusion of this study.

2. RELATED WORK

This study aims at supporting efficient online mining of association rules for dynamic databases by employing the reduced itemset lattice (RIL). Therefore, this section will review the related research work on both areas of incremental mining and online mining.

2.1. Incremental Mining

Incremental mining [5,6,10] is the technique to adjust existing rules for dynamic databases according to the update transactions. When the database is modified, the technique first erases all itemsets that become small and re-computes all the strength of rules in order to erase those below the threshold. It then identifies all small 1-itemsets that become large and use them to combine with other large 1-itemsets for generating candidate 2-itemsets. A procedure similar to Apriori algorithm is applied to these candidate 2-itemsets, which repeatedly scans the database, computes the strength of and combines new candidate itemsets until no more new high-order candidate itemset can be generated. New rules are finally generated from the new large itemsets. As compared to re-mining using Apriori algorithm, incremental mining scans the database fewer times and handles fewer itemsets.

In order to reduce the database scanning time, two kinds of methods have been proposed. In [5,6], only the update transactions are scanned while computing the strength of candidate itemsets, and the whole database is scanned once for computing the actual strength of large itemsets at the final step of deriving new rules. On the other hand, [10] starts from finding all large itemsets of the update transactions and compares them with existing large itemsets to generate the new rules. This method also needs to scan the whole database once for the actual strength of large itemsets at the final step. Although reducing the times of scanning the whole database to once, these methods may spend much time in the strength checking at the final step. And even worse, they may miss some large itemsets and thus produce imprecise rules. For example, if some itemset X seems to be large in the update transactions but it is indeed small in the whole database, these methods would erase X and leave the subset of X unconsidered.

2.2. Online Mining

Online mining [1,4] is the technique to generate rules in a short time according to the threshold change. Without scanning databases, this technique generates rules by using the pre-computed occurrences of all itemsets saved in the itemset lattice structure. But for the consideration of memory space limit, saving the full itemset lattice may not always be possible.

[1] proposes an online mining approach for dynamic databases. By setting the maximum cardinality of itemsets (i.e., maximal number of items occurred in an itemset), it stores only those itemsets without exceeding the maximum cardinality in order to reduce the space requirement of itemset lattice. The lattice is incrementally adjusted according to the update transactions, and rules are derived online from the lattice while the user inputs a threshold of minimum support. This approach however has two drawbacks. First, the maximum cardinality is not easy to determine in advance because if the value is set too high, the itemset lattice will take a lot of memory space, and on the other way, the rules of cardinality higher than the value would lack. Second, the size of itemset lattice is still too large. For instance, if the maximum cardinality is 6 in a database of 1000 items, the lattice will have $C_6^{1000} + C_5^{1000} + C_4^{1000} + C_3^{1000} + C_2^{1000} + C_1^{1000} \approx 10^{15}$ nodes, which is still too large to save.

[4] uses the itemset lattice to offer online mining for static databases. For saving the lattice space, this approach keeps only the most possibly used itemsets in the itemset lattice. By setting a default threshold of minimum support, it erases all the itemsets not confirmed to the threshold while building the lattice. Rules are generated online by choosing itemsets from the lattice for different minimum support values. Although this approach may reduce the size of itemset lattice, it has two drawbacks. First, it cannot generate all rules when the user inputs a minimum support value lower than the default one. Second, if the database is dynamically modified, in order to make rules consistent with the database, this approach needs to rebuild the lattice very often, which is a very time-consuming task.

In summary, current incremental mining techniques have the problems of scanning the whole database and/or producing imprecise rules for dynamic databases. On the other hand, besides the huge space requirement for the itemset lattice, current online mining techniques have the problems of limiting the number of items in rules or producing an incomplete set of rules while the user needs rules for different support values. As a result, in this study we shall use the RIL (reduced itemset lattice) to reduce the space requirement of the itemset lattice, and keep just enough itemsets in the RIL to produce a complete set of precise rules. Once built, the RIL will be incrementally adjusted in response to the dynamic changes of the database and employed to produce all the rules in a short time according to the threshold of minimum support specified by the user.

3. REDUCED ITEMSET LATTICE (RIL)

In this section, we will describe the RIL, how to build RIL and use RIL to mine association rules.

The reduced itemset lattice (RIL) is a subset of the full

itemset lattice (as shown in Figure 1), which consists of nodes and directed links. Each node represents an itemset, and each directed link represents a superset-subset relationship indicating a reduction from the superset itemset (i.e., parent) to the subset one (i.e., child). Similar to [4], an RIL is built according to a default minimum support. If the support of an itemset node is no less than the default value, it is called **large itemset node**; otherwise, it is called **small itemset node**. A small node without any small child node is called **bottom small itemset node**. Unlike the itemset lattice of [4] which keeps only the large itemset nodes, the RIL keeps all large itemset nodes as well as all bottom small itemset nodes. With this extra information, instead of being rebuilt, the RIL can be incrementally adjusted in response to the dynamic changes of the database.

3.1. Building RIL

The procedure of building RIL is similar to Apriori algorithm. It is described by the following algorithm **Build_RIL**.

Algorithm **Build_RIL** (DB)

Step 1: Select an appropriate default threshold of minimum support, and let k = 1;

Step 2: Scan the database DB to count the occurrence of every k-itemset;

Step 3: Identify each k-itemset as large or small according to the default support;

Step 4: Keep all large itemset nodes and bottom small itemset nodes in RIL;

Step 5: Use large itemset nodes to form candidate (k+1)-itemsets, and let k = k+1;

Step 6: Repeat Steps 2-5, until no more candidate itemset is formed.

When building the RIL, we maintain two lists, top-level large-node list (TLL) and bottom-level small-nodes list (BSL), for efficiently generating rules and updating the RIL. The lists TLL and BSL help us easily find the top-level large nodes and bottom-level small nodes, respectively.

Example 1. Assume we have the following database DB with the default support 0.4.

TID	Items
100	ABD
200	ABCE
300	CDE
400	ABDE
500	BDE

By applying the algorithm **Build_RIL** on this DB, the procedure proceeds as follows. In the first iteration, the following support values (i.e., occurrence over the total number of transactions) for each 1-itemset are obtained after Step 2.

Itemset	Support
A	3/5
B	4/5
C	2/5
D	4/5
E	4/5

Using the default threshold 0.4, Step 3 identifies nodes A, B, C, D, and E as large itemset nodes. Step 4 then combines them into candidate 2-itemset nodes and stores them into RIL. In the second iteration, the following support values are computed.

Itemset	Support
AB	3/5
AC	1/5
AD	2/5
AE	2/5
BC	1/5
BD	3/5
BE	3/5
CD	1/5
CE	2/5
DE	3/5

Similarly, nodes AB, AD, AE, BD, BE, CE, and DE are identified as large nodes, and others as small nodes. The following candidate 3-itemset nodes are generated with their support values computed.

Itemset	Support
ABD	2/5
ABE	2/5
ADE	1/5
BDE	2/5

In the third generation, nodes ABD, ABE, and BDE are identified as large nodes, and node ADE as small node. Because no more candidate itemset can be combined, the algorithm finishes and the resulting RIL is shown in Figure 2.

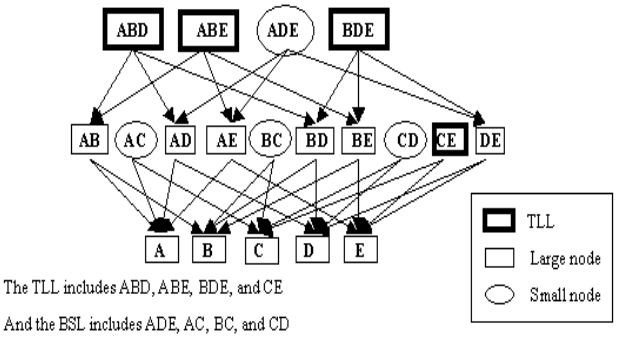


Figure 2. Building RIL before mining.

3.2. Using RIL to Mine Association Rules

To mine association rules from the RIL, we need to check whether it contains enough large itemset nodes to generate all the rules. Since the RIL is built with a default threshold and all its large nodes have the support values greater than this threshold, these large nodes may not be real large nodes with respect to the user-specified minimum support value. If the user-specified threshold is higher than the default threshold, the large nodes in the RIL are a subset of real large itemsets and we can screen

those unreal large nodes from the RIL. But if the user-specified threshold is lower than the default threshold, those nodes with support values lower than the default but higher than the user-specified threshold are not in the RIL. Therefore, we must find these nodes and expand the RIL to fill in them before rule generation. The algorithm of expanding the RIL (Algorithm **Expand_RIL**) will be described in the next subsection (Section 3.3).

The following algorithm describes how to generate association rules from the RIL. Assume df-supp denotes the default minimum support, and min_supp and min_conf denote the user-specified minimum support and confidence values respectively.

Algorithm **Mine_RIL** (Reduced_IL)
Step 1: Execute **Expand_RIL** (Reduced_IL) if min_supp ≦ df_supp;
Step 2: Perform Steps 3–6 for each node N in TLL of Reduced_IL;
Step 3: If N's support ≦ min_supp,
then perform Step 4;
otherwise, perform Steps 5-6;
Step 4: For each child X of N,
if X's support ≧ min_supp,
then let N=X and go to Step 5;
otherwise, let N=X and perform Step 4
until X has no child;
Step5: For each child C of N,
if (N's support/C's support)≧ min_conf,
then output the rule N→(N-C);
otherwise, go to Step 6;
Step 6: For every child K of N (K≠N),
let N = K, and perform Step 5
until N has no child;

The algorithm basically starts from TLL and checks if the node N in TLL and its child C can form a rule (Step 5). If they cannot form a rule, the algorithm reduces N to its subset K and further checks if K and its child can form a rule (Step 6). Since the RIL is built with df_supp, if df_supp is larger than the user-specified minimum support, the algorithm needs to expand the RIL and fill in all real large nodes (Step 1); otherwise, it needs to screen out those unreal large nodes from the RIL (Steps 3) and further consider its descendants (Step 4). The following example illustrates the algorithm **Mine_RIL**.

Example 2. Continue with Example 1, and assume the user specifies a minimum support value (min_supp) 0.6 and a minimum confidence (min_conf) 0.6. The algorithm proceeds as follows, and the resulting RIL is shown in Figure 3.

(1) Compare the support values of nodes in TLL with min_supp, and screen out those nodes ABD, ABE, BDE and CE (which become small nodes).
(2) Check their child nodes with min_supp and identify the real large nodes AB, BD, BE and DE. (note that they become top-level itemset nodes with respect to min_supp.)
(3) Use these large nodes to generate the rules A→B, B→A, B→D, D→B, B→E, E→B, D→E and E→B.

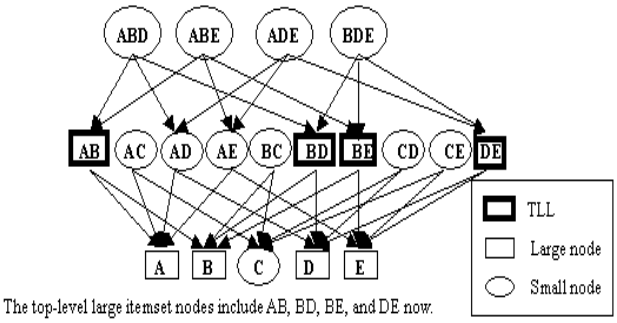


Figure 3. Mining rules from RIL.

3.3. Expanding RIL

As mentioned in the previous subsection, if the user-specified threshold is lower than the default threshold, those nodes with support values lower than the default but higher than the user-specified threshold are not in the RIL. Therefore, we must find these nodes and fill in them to expand the RIL to fill in them before rule generation. Fortunately, because the RIL keeps the bottom-level small itemset nodes, we can use this extra information to expand the RIL easily and this also makes it possible to generate all rules in a short time. As compared with the method of [4], it totally ignores these nodes and thus generates an incomplete set of rules.

Algorithm **Expand_RIL** (Reduced_IL)
Step 1: Set df_supp = min_supp, and let k = 1;
Step 2: Identify all small k-itemset nodes that become large;
Step 3: Use these nodes to combine with other large nodes (includes potential large nodes) into candidate (k+1)-itemsets, which are called potential large nodes.
Step 4: k = k + 1;
Step 5: Repeat Steps 2-4 above until no more candidate itemset node occurs;
Step 6: Scan the database once to count the occurrence of potential large nodes;
Step 7: Identify them as large nodes if their support ≧ df_supp;
Step 8: Update TLL and BSL, and erase all small nodes that are above the BSL;

Example 3. Continue with Example 1, and assume the user specifies a minimum support value (min_supp) 0.2 and a minimum confidence (min_conf) 0.6. The algorithm proceeds as follows.

(1) Because there is no small 1-itemset node in the RIL, the algorithm identifies those small 2-itemsets that become large, i.e., AC, BC, and CD.
(2) It combines them with other large itemsets into potential large itemsets, including ABC, ACD, ACE, BCD, BCE, and CDE.
(3) Then it finds the small 3-itemset ADE becomes large and combines ADE with origin large nodes into potential large itemsets, ABCE, ABDE, ABCD, and BCDE, which are further combined into ABCDE, as shown in Figure 4.

(4) The algorithm scans the database once, and counts the occurrence of these potential large itemsets.

(5) It identifies these itemsets as large or small according to df_supp, and updates the TLL to (ABCE, ABDE, CDE) and the BSL to (ACD, BCD). The resulting RIL is shown in Figure 5.

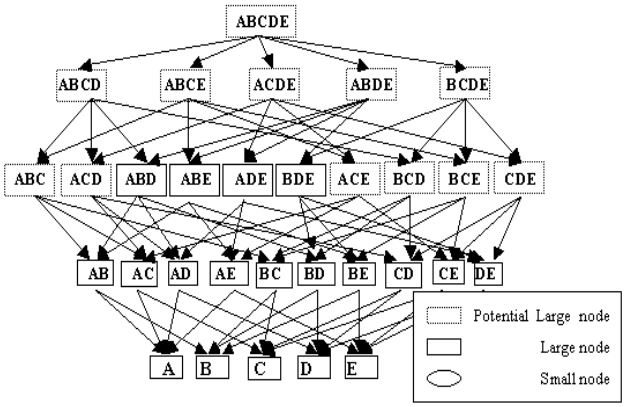


Figure 4. The expended RIL with potential large nodes.

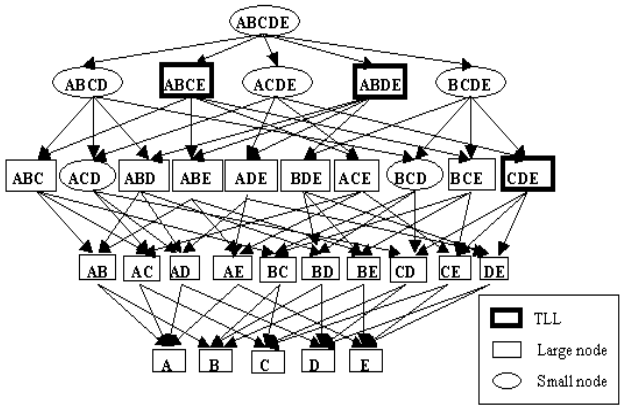


Figure 5. The RIL after database scan and node reidentication.

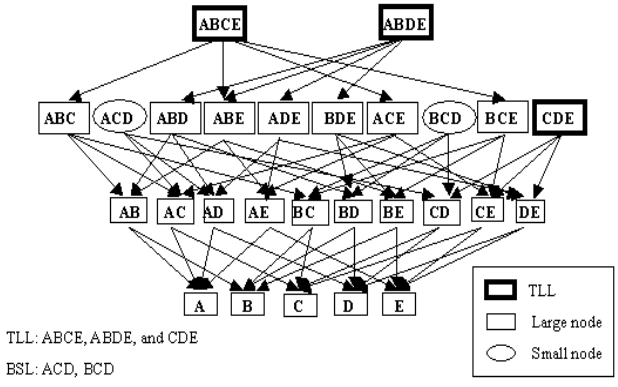


Figure 6. The final RIL after expansion.

(6) The algorithm finally erases all small itemset nodes above the BSL, including ABCDE, ABCD, ACDE, BCDE. The final RIL is shown in Figure 6.

4. INCREMENTAL UPDATE OF RIL FOR DYNAMIC DATABASES

For dynamic databases where data are changed frequently, the RIL must be incrementally adjusted so as to make it consistent with the databases. Dynamic databases may impact the RIL in two ways:

(1) Affect all itemsets. Data insertion or deletion may change the total number of transactions in the database, and thus the support value of all itemsets.

(2) Affect the occurrence of related itemsets. If an itemset is involved in some update transaction (including the insertion and delete operation), its occurrence will be increased/decreased by one.

Both impacts must be taken into account in the incremental update of the RIL.

4.1. Incremental Update Algorithm

When the database is modified, instead of rebuilding, we use algorithm **Adjust_RIL** to update the RIL. The algorithm includes two steps: it first finds all the itemsets related to the modified data for changing their occurrence, and then makes the whole RIL consistent with the default threshold. (For the details of the algorithm, please refer to [7].) Assume db is the set of update transactions and T is the itemset in each transaction.

Algorithm **Adjust_RIL** (Reduced_IL, db)

Step 1:(1) For each T in db, perform Lines (2) –(9) of step 1

(2) If T appears in Reduced_IL then increase the occurrence of T and its descendants in Reduced_IL by one, and go to (9);

(3) For each node (itemset) N in TLL

(4) If $N \subseteq T$, increase the occurrence of N and N′s descendants; go to (9);

(5) If $N \cap T \neq \varnothing$, then

(6) For each non-1-itemset child node M of N, let N = M, and go to (4);

(7) For each node N in BSL

(8) If $N \subseteq T$, increase the occurrence of N; go to (9);

(9) End of For loop in Line (1);

Step 2: (1) For each node N in TLL, perform Lines (2)-(4) of step 2;

(2) If N's support $\geqq$ df_supp then go to (1) for next N in TLL;

(3) Remove N from TLL and insert N into BSL;

(4) For each child node M of N, let N = M, and go to (2);

(5) For each node N in BSL, perform (6)-(10);

(6) If N's support < df_supp then go to (5) for next N in BSL;

(7) Combine N with other large nodes into potential large itemsets;

(8) Scan database to count these itemset's occurrences;
(9) Use df_supp to identify all potential large itemsets as large or small;
(10) Erase all the small nodes that are not bottom-level small nodes;

Example 4. Continue with Example 1, and assume two transactions ABD and ACDE are inserted into the database respectively. The algorithm of adjusting RIL proceeds as follows.

(1) For the first insertion, since the itemset ABD is in the RIL, its occurrence is increased by one, so are the occurrences of all nodes in its sublattice (i.e., AB, AD, BD, A, B, and D). After the increment, the RIL is adjusted as Figure 7 shows.

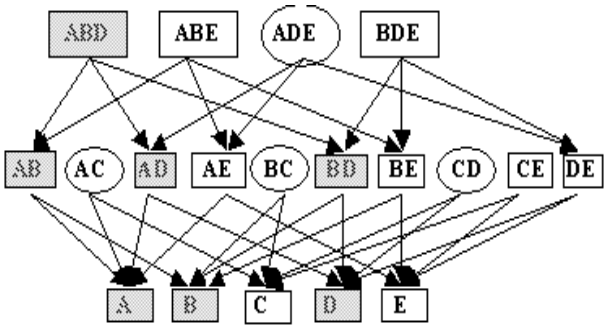
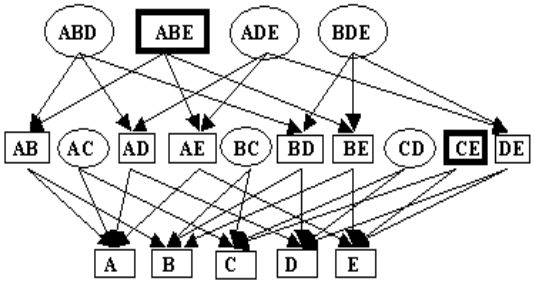


Figure 7. The affected RIL after inserting transaction ABD.

(2) For the second insertion, since the itemset ACDE is not in RIL, all the nodes in TLL and BSL must be checked:
  (a) Because the large nodes ABD, ABE and BDE are partly related to transaction ABCE, their occurrences remain unchanged and the algorithm continues to check their child nodes (described below in (3)).
  (b) Because large node CE is a subset of ACDE, the occurrences of CE and its subnodes C and E are all increased by one.
  (c) Because small nodes ADE, AC and CD are all subsets of ACDE, their occurrences are increased by one, but their subnodes are not affected.

(3) Check all subnodes of (2)(a) above, including AB, BD, AD, BE, AE and DE:
  (a) Because large nodes AD, AE and DE are subsets of ACDE, the occurrences of them and all their subnodes are increased by one.
  (b) Because itemsets AB, BD and BE are partly related to ACDE, the algorithm continues to check their subnodes A, B and E. But nodes A and E can be skipped because they have already been visited in early steps ((2)(b) and (3)(a)). And because node B is not related to ACDE, its occurrence remains unchanged.

(4) Because the large nodes ABD and BDE become small, both TLL and BSL must be updated. The resulting RIL is shown in Figure 8.



TLL includes ABE and CE.

BSL includes ABD, ADE, BDE, AC, BC and CD.

Figure 8. The RIL after inserting transaction ABCE.

4.2. Complexity Analysis

The complexity of incremental update algorithm can be analyzed in three cases:

(1) No large node becomes small and no small node becomes large. In this case, the algorithm only adjusts the occurrence of nodes in RIL. If there are K nodes in RIL, and the number of update transactions is t, the time complexity of adjusting RIL is O(Kt).

(2) Some large nodes become small. Besides the time spent in Case (1), this case needs to take into account of the time to adjust TLL and BSL, so the total time complexity is O(Kt+K) = O(Kt).

(3) Some small nodes become large. Besides the time spent in Case (1), this case needs to take into account of the time to insert large nodes into RIL and the time to scan the database. If the number of transactions in the whole database is T and the number of nodes to be inserted is K′, the time complexity of incrementally adjusting RIL is O(Kt+K′T).

5. EXPERIMENTS AND COMPARISON

5.1. Experimental Results

In order to verify the feasibility of using RIL to support online mining in dynamic databases, we implement a prototype mining system that includes those algorithms described in Sections 3 and 4. Our platform is Sun Ultra SPAC10 with 333 MHz CPU, the database system is Oracle 8i, and programs are written in C++.

Five test databases are generated by the program from IBM Data Mining Research Group [8]. Every test database has 1000 items and 100000 transactions. These test databases are named in the form of "TxIyDz" according to the parameters (x, y, and z) used by the IBM program to generate them. Parameters x, y and z are all integers, indicating the average number of items in the transaction, the average number of items in the large itemsets, and the number of transactions in the database,

respectively. For each test database, the same set of parameters is used to generate three sets of update transactions, which have 1000, 2000 and 3000 transactions respectively.

Two experiments have been performed on the RIL. The first one measures the building time and size of RIL for the five test databases. Due to different database characteristics, the first three databases are tested with three default support values 0.30%, 0.40% and 0.50%, while the last two databases are tested with the other three support values 0.80%, 0.90% and 1%. The results are shown in Table 1 and Table 2 respectively. It is observed from Table 1 that the average number of items in the transaction dominates the RIL building time, which ranges from 18 to 696 seconds. Table 2 shows that the RIL can reduce the size of itemset lattice to average 10^5 nodes, as compared to the lattice size 10^{15} of [1] mentioned in Section 2.

Default support	**0.30%**	**0.40%**	**0.50%**
T5I2D100K	29	20	18
T10I2D100K	365	136	53
T10I4D100K	150	146	52
Default support	**0.80%**	**0.90%**	**1%**
T20I2D100K	696	683	682
T20I6D100K	681	675	673

Table 1. The RIL building time (in seconds) for five test databases.

Default support	**0.30%**	**0.40%**	**0.50%**
T5I2D100K	150331	112628	76855
T10I2D100K	247114	201663	170653
T10I4D100K	255578	220456	181300
Default support	**0.80%**	**0.90%**	**1%**
T20I2D100K	201477	186950	171891
T20I6D100K	215441	199212	182535

Table 2. Number of nodes in RIL for five test databases.

The second experiment measures the rule mining time and the RIL adjusting time for the five test databases. The RIL's built in the first experiment are tested with five user-specified minimum support values (0.3%, 0.4%, 0.5%, 0.6% and 0.7% for the RIL's from first three databases, and 0.7%, 0.8%, 0.9%, 1.0%, 1.1% for those from last two databases) in order to measure their rule mining time. The results are shown in Table 3 and Table 4 respectively. It is observed from these tables that the default threshold affects the RIL mining time. If the user-specified minimum support is higher than default threshold, the mining time is less than 1 second. On the other hand, if the user-specified minimum support is lower than default threshold, the mining process takes 10 to 700 seconds due to database scan.

User support	0.30%	0.40%	0.50%	0.60%	0.70%
T5I2D100K	0.01	0.01	0.01	0.01	0.01
T10I2D100K	0.01	0.01	0.01	0.01	0.01
T10I4D100K	0.01	0.01	0.01	0.01	< 0.01

Default support = 0.3%

User support	0.30%	0.40%	0.50%	0.60%	0.70%
T5I2D100K	13	0.01	0.01	0.01	0.01
T10I2D100K	90	0.01	0.01	0.01	0.01
T10I4D100K	71	0.01	0.01	0.01	< 0.01

Default support = 0.4%

User support	0.30%	0.40%	0.50%	0.60%	0.70%
T5I2D100K	12	13	0.01	0.01	0.01
T10I2D100K	91	54	0.01	0.01	0.01
T10I4D100K	71	51	0.01	0.01	< 0.01

Default support = 0.5%

Table 3. The mining time (in seconds) of RIL for first three databases.

User support	0.70%	0.80%	0.90%	1.00%	1.10%
T20I2D100K	685	0.01	0.01	< 0.01	< 0.01
T20I6D100K	103	0.01	0.01	< 0.01	< 0.01

Default support = 0.8%

User support	0.70%	0.80%	0.90%	1.00%	1.10%
T20I2D100K	673	304	0.01	< 0.01	< 0.01
T20I6D100K	105	104	0.01	< 0.01	< 0.01

Default support = 0.9%

User support	0.70%	0.80%	0.90%	1.00%	1.10%
T20I2D100K	687	308	228	< 0.01	< 0.01
T20I6D100K	105	105	104	< 0.01	< 0.01

Default support = 1%

Table 4. The mining time (in seconds) of RIL for last two databases.

In this experiment, the RIL's built in the first experiment are also tested with three sets of update transactions in order to measure their adjusting time. Three RIL's were built from the first three databases with default support 0.3%, while the other two were from the last two databases with default support 0.8%. The test results are shown in Table 5.

# Update transactions	**1000**	**2000**	**3000**
T5I2D100K	0.58	1.09	1.6
T10I2D100K	1.17	2.16	3.16
T10I4D100K	1.12	2.03	2.95

Default support = 0.3%

# Update transactions	**1000**	**2000**	**3000**
T20I2D100K	1.33	2.47	3.66
T20I6D100K	1.26	2.36	3.42

Default support = 0.8%

Table 5. The RIL adjusting time (in seconds) for various number of update.

It is observed from Table 5 that the RIL adjusting time in every case is no more than 4 seconds. And the adjusting time is almost proportional to the number of transactions to be updated, rather than database characteristics. From Table 1 and Table 3, we observe that the RIL adjusting

time is about 0.19% to 5.5%, with an average 1.27%, of the rebuilding time.

5.2. Comparison

This subsection compares RIL with other online mining approaches [1,4] in terms of the memory cost and lattice adjusting time for dynamic databases. The full itemset lattice of [1] keeps full information for both rule generation and incremental update of the lattice. However, the full itemset lattice may contain a lot of redundant nodes that would not generate rules. As mentioned in Section 2, [1] may keep 10^{15} nodes in the itemset lattice for a database of 1000 items with maximum cardinality 6. In contrast, the RIL uses less memory (about 10^5 nodes as shown in Table 2) for the same situation and provides the same functionality. Furthermore, unlike the lattice in [1], the RIL does not set a limit on the maximum cardinality of the itemsets, which makes the RIL adaptable to different dynamics of the database.

The full itemset lattice of [4] may use less memory as it is built with respect to a default threshold and it does not keep bottom small itemset nodes (as the RIL does). But it must be rebuilt when the database is modified. Observed from Table 1, the rebuilding process is very time-consuming (in hundreds of seconds) and not suitable for dynamic databases. In contrast, the RIL needs not to be rebuilt and its incremental adjusting time is on the average about 1.27% of the rebuilding time. It is efficient enough to mine rules online for dynamic databases. Moreover, unlike [4] which may miss rules as the user-specified minimum support is lower than the default threshold, the RIL always generates a complete set of rules corresponding to the user-specified threshold.

6. CONCLUSION

In this paper, we presented a data structure RIL to support online mining for dynamic databases. By keeping fewer (than the full itemset lattice) but enough itemsets, the RIL can incrementally adjust itself in response to the database changes. As the user specifies a support threshold, the RIL can expand itself first if necessary, and mine a complete and correct set of association rules in a short time. Experimental results showed that the RIL not only effectively reduces the space of full itemset lattice but also adjusts itself in 1.27% (on the average) of lattice building time. It takes about two to ten minutes to expand the RIL. If the user specifies a support value larger than the default threshold (i.e., the expansion is not necessary), the RIL can mine association rules in less than 0.01 seconds.

7. REFERENCES

[1] A. Amir, R. Feldman, and R. Kashi, "A new and versatile method for association generation," Principles of Data Mining and Knowledge Discovery, First European Symposium, PKDD '97, pp. 221-231, 1997.

[2] R. Agrawal, T. Imielinski, and A. Swami, "Mining association rules between sets of items in large databases," Proc. of the ACM SIGMOD Conference on Management of Data, pp. 207-216, 1993.

[3] R. Agrawal and R. Srikant, "Fast algorithms for mining association rules," Proc. of the 20th VLDB Conference, pp. 487-499, 1994.

[4] C. C. Aggarwal and P. S. Yu, "Online generation of association rules," Proc. of the IEEE ICDE'98, pp. 402-411, 1998.

[5] D. W. Cheung, J. Han, V. T. Ng, and C. Y. Wong, "Maintenance of discovered association rules in large databases: An incremental updating technique," Proc. of the IEEE ICDE'96, pp. 106-114, 1996.

[6] D. W. Cheung, S.D. Lee, and B. Kao, "A general incremental technique for maintaining discovered association rules," Proc. of the Fifth International Conference On Database Systems For Advanced Applications, pp. 185-194, 1997.

[7] C.-H. Hsu, Online Generation of Association Rules in Dynamic Databases, Master Thesis, National Chung-Hsing University, Taiwan, 2000.

[8] "IBM Almaden-Quest Data Mining Synthetic Data Generation Code," http://www.almaden.ibm.com/cs/quest/syndata.html

[9] J. S. Park, M. S. Chen, and P. S. Yu, "An effective hash based algorithm for mining association rules," Proc. of the ACM SIGMOD, pp. 175-186, 1995.

[10] S. Thomas, S. Bodagala, K. Alsabti, and S. Ranka, "An efficient algorithm for the incremental updation of association rules in large databases," Proc. of the 3rd International Conference on Knowledge Discovery and Data Mining (KDD97), pp. 263-266, 1997.