Efficient Mining of Association Rules by Reducing the Number of Passes over the Database

LI Qingzhong (李庆忠), WANG Haiyang (王海洋), YAN Zhongmin (闫中敏) and MA Shaohan (马绍汉)

Institute of Computing Technology, The Chinese Academy of Sciences, Beijing 100080, P.R. China
Department of Computer Science, Shandong University, Jinan 250100, P.R. China

E-mail: lqz@cs.sdu.edu.cn

Received March 26, 1999; revised February 17, 2000.

Abstract This paper introduces a new algorithm of mining association rules. The algorithm RP counts the itemsets with different sizes in the same pass of scanning over the database by dividing the database into m partitions. The total number of passes over the database is only $(k+2m-2)/m$, where k is the longest size in the itemsets. It is much less than k.

Keywords data mining, association rule, itemset, large itemset

1 Introduction

Mining for association rules is a form of data mining introduced in [1]. The prototypical example is based on a list of purchases in a store. An association rule for this list is a rule such as "85% of all customers who buy products A and B also buy products C and D". Discovering such customer buying patterns is useful for customer segmentation, cross-marketing, catalog design and product placement.

We give a problem description which follows [2]. The *support* of an itemset (a set of items) in a transaction sequence is the fraction of all transactions containing the itemset. An itemset is called *large* if its support is greater than or equal to a user-specified support threshold, otherwise it is called *small.* An *association rule* is an expression $X \Rightarrow Y$ where X and Y are disjoint itemsets. The support of this rule is the support of $X \Rightarrow Y$. The *confidence* of this rule is the fraction of all transactions containing X, that also contain Y, i.e., the support of $X \Rightarrow Y$ divided by the support of X. In the example above, "85%"is the confidence of the rule $\{A, B\} \Rightarrow \{C, D\}$. For an association rule to hold, it must have a support $\geq$ a user-specified confidence threshold.

For an itemset X, its support is defined similarly as the percentage of transactions in DB which contains X. We also use $X.\mathrm{sup}$ to denote its support count, which is the number of transactions in DB containing X. Given a minimum support threshold *minsup*, an itemset X is large if its support is no less than *minsup*. Moreover, for presentation purpose, we will call an itemset of size-k k-itemset.

Existing algorithms proceed in two steps to compute association rules:

1) Find all large itemsets.

2) Construct rules which exceed the confidence threshold from the large itemsets in Step 1). For example, if $\{A, B, C\}$ is a large itemset, we might check the confidence of $\{A, B\} \Rightarrow \{C\}$, $\{A, C\} \Rightarrow \{B\}$ and $\{B, C\} \Rightarrow \{A\}$.

We will address the first step, since the second step can be easily handled. Existing large itemset computation algorithms have an offline or batch behavior: given the user-specified support threshold, the transaction sequence is scanned and rescanned, often several times, and eventually all large itemsets are produced. However, the user does not know, in general, an appropriate support threshold in advance. An inappropriate choice yields, after a long

wait, either too many or too few large itemsets which often results in useless or misleading association rules.

We introduce an algorithm RP to count the itemsets with different sizes in the same pass of scanning over the database by dividing the database into m partitions. The total number of passes over the database is only $(k+2m-2)/m$, where k is the longest size in the itemsets. It is much less than k. As the result it lowers the cost of I/O operations greatly. Although the algorithm adopted the idea of *tidlist* in the algorithm *Partition*[5], the idea is only used within the first pass and it does not cause overload of memory.

2 Related Work

Most large itemset computation algorithms are related to the Apriori algorithm due to Agrawal and Srikant[3]. See [4] for a survey of large itemset computation algorithms. Apriori exploits the observation that all subsets of a large itemset are large themselves. It is a multi-pass algorithm, where in the k-th pass all large itemsets of cardinality k are computed. Hence Apriori needs up to $c+1$ scans of the database where c is the maximal cardinality of a large itemset. In [5] a 2-pass algorithm called Partition is introduced. The general idea is to partition the database into blocks such that each block fits into main-memory. In the first pass, each block is loaded into memory and all large itemsets, with respect to that block, are computed using Apriori. Merging all resulted sets of large itemsets then yields a superset of all large itemsets. In the second pass, the actual support of each set in the superset is computed. After removing all small itemsets, Partition produces the set of all large itemsets.

In contrast to Apriori, the DIC (Dynamic Itemset Counting) algorithm counts itemsets of different cardinalities simultaneously[2]. The transaction sequence is partitioned into blocks. The itemsets are stored in a lattice which is initialized by all singleton sets. While a block is scanned, the count (number of occurrences) of each itemset in the lattice is adjusted. After a block is processed, an itemset is added to the lattice if and only if all its subsets are potentially large, i.e., large with respect to the part of the transaction sequence for which its count was maintained. At the end of the sequence, the algorithm rewinds to the beginning. It terminates when the count of each itemset in the lattice is determined. Thus after a finite number of scans, the lattice contains a superset of all large itemsets and their counts. For suitable block sizes, DIC requires fewer scans than Apriori.

Random sampling algorithms have been suggested in [6, 7]. The general idea is to take a random sample of suitable size 4 from the transaction sequence and compute the large itemsets using Apriori or Partition with respect to that sample. For each itemset, an interval is computed such that the support lies within the interval with probability greater than or equal to some threshold.

Several algorithms based on Apriori were proposed to update a previously computed set of large itemsets after insertion or deletion of transactions[8-10]. These algorithms require a rescan of the full transaction sequence whenever an itemset becomes large due to an insertion.

In [11] an Online Analytical Processing (OLAP)-style algorithm is proposed to compute association rules. The general idea is to precompute all large itemsets relative to some support threshold using a traditional algorithm. The association rules are then generated online relative to an interactively specified confidence threshold and support threshold greater than or equal to s.

1) The support threshold s must be specified before the precomputation of the large itemsets;
2) The large itemset computation remains off-line; and
3) Only rules with support greater than or equal to s can be generated.

3 Algorithm RP

RP is an algorithm of reducing passes over database. The algorithm uses the function Apriori-gen of algorithm Apriori. RP introduces a function gen_2_itemsets by modifying the function gen_large_itemsets of algorithm Partition. The algorithm introduces a new idea, that is, counting the itemsets with different lengths in the same scanning. In this way the passes of scanning can be reduced enormously.

3.1 The Notations of Algorithm RP

DB is a partitioned database. A partition P of the database refers to any subset of the transactions contained in the database DB. Any two different partitions are non-overlapping, i.e., $P^i \cap P^j = \emptyset$, $i \neq j$. So $DB = \{P^1, P^2, \ldots, P^n\}$. Let the size of DB (the number of transactions in DB) be D ($|DB| = D$). The size of P^i is D^i ($|P^i| = D^i$).

We use C_k^i to denote the k-itemset generated in P^i, that is, the k-candidates in P^i. The itemsets are generated differently according to the size k:

1) We use C_1^i to denote the locally large 1-itemset. C_1^i is generated in the first pass of scanning P^i. Every itemset of C_1^i has a set of "tidlist". The "tidlist" records all the identifiers of transactions which contain the itemset.

2) We use C_2^i to denote the locally large 2-itemset. It is generated by gen_2_itemset(C_1^i) at the end of the first scanning of P^i. Refer to Subsection 3.2 about this function.

3) All the C_k^i's ($k > 2$) are generated by Apriori_gen(C_{k-1}^i). C_k^i is k-candidates in partition P^i.

Let A be a counting cache which stores all the items appearing in P^i and the corresponding item I's tidlist (I.tidlist) which records the transaction identifier including item I in P^i. As a counting cache A is only used for generating C_1^i in the first scanning of the database. The tidlist in A is cleared to empty before scanning P^i. When scanning P^i the transaction identifiers including item I are put into corresponding I.tidlist. The item I satisfying count (I.tidlist) $> minsup * |P^i|$ is C_1^i when the scanning of P^i finishes.

M_k has two attributes: one stores itemset, the other stores the counter of the itemset.

M_1 stores all the items appearing in the database (item is added to M_1 successively when going through from P^1 to P^m). I.count records the counting of I in the scanning of database for each item I. Before scanning P^i, I.count is the sum of numbers of the occurrences of I in $P^1, \ldots, P^{i-1}$. After scanning of P^i, I.count is changed and it becomes the sum of original value and count (I.tidlist) in A.

M_k ($k \geq 2$) stores $U_{i=1}^m C_k^i$. It is used for counting when scanning database. When each C_k^i is generated, it is stored in M_k using union operation. This method can avoid duplicate counting of the common elements in C_k^i and C_k^j ($i \neq j$). When C_k^i is counted through one pass over the database, it is pruned away from M_k using difference operation. This guarantees that the candidate itemsets are counted in one pass over the database, and duplicate counting is avoided.

L_k stores globally large k-itemsets.

3.2 Generating Fewer 2-Candidates

The selection of 2-candidates can influence the efficiency of finding large itemsets. Each C_k^i is generated from the previous C_{k-1}^i. If C_2^i is derived from C_1^i directly, the number of candidates in the result is very huge and the RP algorithm counts M_{k+1} before L_k is generated. Then there exists a chain reflection to influence the numbers of items in candidates generated afterwards. So to generate fewer 2-candidates is very important in our algorithm.

We use the notation tid from the algorithm Partition[5]. Tid stands for transaction identifier in transaction database. TidList is a list of tid. We assign the tid of the transactions containing locally large 1-itemset to tidlist. In this way we can compute the counts of all the combinations of the items in C_1^i. Then we get the locally large 2-itemsets in P^i. So the 2-candidates are reduced. The function is gen_2_itemset(). The input parameter is C_1^i, and the output is C_2^i. The program is as follows:

1) C_2^i=Apriori_gen(C_1^i);
2) **for** all $c \in C_2^i$
3) {c.tidlist=c[1].tidlist $\cap$ c[2].tidlist; //getting transactions containing both of the two items in c;
4) **if** |c.tidlist|/$D^i \leq$ *minsup* **then**
5) remove c from C_2^i;
6) **else** c.count=|c.tidlist|}
7) **return** C_2^i (including c.count).

From the above program, c.count, i.e., the occurrences of every itemset c in C_2^i in P^i, is computed using tidlist. C_2^i is computed by calling the function at the end of scanning P^i. If c.count of the itemset is not computed, we must compute it again in the next pass over P^i. Using tidlist can reduce the operation on M_2, and one only needs to store tidlist when generating C_1^i in the scanning of P^i. After the function ends, the memory occupied is released.

Example 1. We divide the database into four partitions. $DB = \{P^1, P^2, P^3, P^4\}$. Suppose that there are no 4-itemsets. We can get the following 1-itemsets by scanning the database. $C_1^1 = \{A, B, C, D\}$, $C_1^2 = \{B, C, D, E\}$, $C_1^3 = \{A, C, E, F\}$, $C_1^4 = \{B, D, F, G\}$. We can get the following 2-itemsets using the function gen_2_itemset(). $C_2^1 = \{AB, BC, AC, BD, CD\}$, $C_2^2 = \{BC, CD, BD, CE\}$, $C_2^3 = \{AC, CE, EF, CF\}$, $C_2^4 = \{BD, DF, GB, DG, FG\}$. The RP only generates twelve 2-itemsets, while Apriori will generate eighteen 2-itemsets. This comparison indicates that the gen_2_itemset function is effective.

3.3 Description of the Algorithm RP

The following is the description of the algorithm RP.

The first pass over the database (i is from 1 up to m, m is the number of partitions).

1) Count non-empty M_k $(1 < k \leq i)$ and generate C_1^i and then modify M_1 when scanning P^i. After the scanning of P^i, C_2^i is generated by gen_2_itemset(C_1^i) and then is added to M_2 by union operation. C_{k+1}^{i-k+1} is generated by Apriori_gen(C_k^{i-k+1}) and is added to M_{k+1} $(1 < k \leq i)$.

2) When i increases from m to $m+1$, the first pass ends. The itemset which satisfies the condition in M_1 is L_1. At the moment all the C_2^i's $(1 \leq k \leq m)$ are generated. C_2^1 has passed the whole database. Add the generated large 2-itemset to L_2. Remove C_2^1 from M_2.

The second pass over the database (i is from 1 up to m).

1) Scan P^i and count non-empty M_k $(k > 1)$. C_2^{i+1} has been counted completely. Add large 2-itemsets evaluated to L_2. Remove C_2^{i+1} from M_2. For $i \geq 2$, C_3^{i-1}, $C_4^{i-2}, \ldots, C_{i+1}^1$ have been counted completely and add large 3-, $\ldots, i+1$-itemsets evaluated to $L_3, \ldots, L_{i+1}$ respectively and then remove C_3^{i-1}, $C_4^{i-2}, \ldots, C_{i+1}^1$ from $M_3, \ldots, M_{i+1}$ respectively.

2) $C_{k+1}^{i-k+m+1}$ $(k = i+1, k = i+2, \ldots, k = i+m)$ is generated by Apriori_gen($C_k^{i-k+m+1}$) before scanning P^i and it is added to M_{k+1}.

3) Assign the partition number m to j to prepare for the rest passes over the database.

The rest passes over the database.

1) j increases by 1 before the scanning of each partition. Scan each partition and count non-empty M_k $(k > 1)$. For n and t satisfying $n + t - 2 = j$ and $n + t - 2 > m$ and $n > 2$,

each C_n^t has been counted completely. Add large n-itemsets evaluated to L_n. Remove C_n^t from M_n.

2) $C_{k+1}^{j-k+m+1}$ $(k = j+1, k = j+2, \ldots, k = j+m)$ is generated by Apriori_gen $(C_k^{j-k+m+1})$ before scanning each partition and it is added to M_{k+1}.

Every pass repeats the operations above until $L_k = \emptyset$ and $M_{k-1} = \emptyset$. Then the algorithm stops.

Example 2. Suppose that the database in Example 1 has 200 transactions. Every partition has 50 transactions. The *minsup* $s = 10\%$. First we show C_1^i, C_2^i and their support in Tables 1 and 2.

Table 1

C_1^1		C_1^2		C_1^3		C_1^4	
X	$X.\text{sup}^1$	X	$X.\text{sup}^2$	X	$X.\text{sup}^3$	X	$X.\text{sup}^4$
A	7	B	6	A	7	B	6
B	8	C	7	C	6	D	7
C	8	D	7	E	8	F	8
D	9	E	8	F	7	G	10

Table 2

C_2^1		C_2^2		C_2^3		C_2^4	
X	$X.\text{sup}^1$	X	$X.\text{sup}^2$	X	$X.\text{sup}^3$	X	$X.\text{sup}^4$
AB	7	BC	6	AC	6	BD	6
BC	8	CD	7	CE	6	DF	7
AC	5	BD	6	EF	7	BG	6
BD	7	CE	7	CF	6	DG	7
CD	7					FG	8

The algorithm runs as follows:

In the first pass over the database, C_1^1 is generated by scanning P^1. Then C_2^1 is generated by function gen_2_itemset(C_1^1). Scan P^2 to generate C_1^2. In the mean time M_2 is counted. Then C_2^2 is generated. From C_2^1 we can generate $C_3^1 = \{ABC, BCD\}$. Scan P^3 to generate C_1^3. In the mean time M_2 and M_3 are counted. Then C_2^3 is generated. From C_2^2 we can generate $C_3^2 = \{BCD\}$. Scan P^4 to generate C_1^4. In the mean time M_2 and M_3 and M_4 are counted. Then C_2^4 is generated by gen_2_itemset(C_1^4). By Apriori_gen(C_2^3), $C_3^3 = \{CEF\}$ is generated. L_1 is $\{A, B, C, D, F, G\}$. And the globally large 2-itemset in C_2^1 is $\{AB, BC, CD, BD\}$. Do $M_2 = M_2 - C_2^1$.

In the second pass, scan P^1 and count M_2 and M_3. The global large itemset in C_2^2 is $\emptyset$ (because $\{BC, CD, BD\}$ has appeared in C_2^1, and CE here is not a large itemset). Let $M_2 = M_2 - C_2^2$. Generate $C_3^4 = \{DFG, BDG\}$. The large itemset we get in C_3^2 is $\{CF\}$ by scanning P^2. The large itemset we get in C_3^2 is $\{BCD\}$. Then let $M_2 = M_2 - C_2^3$ and $M_3 = M_3 - C_3^1$. By scanning P^3 we know that the large itemset in C_2^4 is $\{DF, DG, FG\}$ and the large itemset in C_3^2 is $\emptyset$ (because $\{BCD\}$ is already in L_3). Then let $M_2 = M_2 - C_2^4$ and there is no large itemset in C_3^3.

In the third pass, after scan P^1, the large itemset we get in C_3^4 is $\{DFG\}$. Here $L_4 = \emptyset$, $M_3 = \emptyset$ and the algorithm stops.

From the above we can see that the database was scanned 2.25 passes.

The following will prove that the algorithm RP can reduce the number of passes of database scanning efficiently.

Proof. First we consider C_n^i ($n > 2$ and n is a constant). When P^i is scanned in the first pass over the database, C_1^i is generated, and then another $n-2$ partitions are scanned before C_n^i is generated from C_{n-1}^i. From now on C_n^i is counted until another m partitions are scanned (m is the number of partitions). From the beginning of the algorithm until then there are $(m + n + i - 2)$ partitions to be scanned. To generate L_n, count all the candidates in C_n^i $(i = 1, 2, \ldots, m)$ by scanning the database. C_n^m, the n-candidates in P^m, must go through a complete pass over the database, then L_n is computed. So $(2m + n - 2)$

partitions must be scanned. From the above we know if the largest size of an itemset is k, the algorithm has to scan the database $(k + 2m - 2)/m$ passes. Obviously $(k + 2m - 2)/m$ is much less than k. So the algorithm RP reduces the number of passes efficiently.

3.4 The Pruning of Candidates

For the candidate C_n^i which has gone through the whole database, $M_n = M_n - C_n^i$ must be calculated. We can use some already proved theorem to prune the candidates which are not counted completely to optimize the algorithm.

From [12] we know any large itemset must be a locally large itemset at least in one of the partitions, we call such an itemset *heavy itemset*. If an itemset is large, all the subsets of it are also large. If an itemset is heavy in partition P^i, all the subsets of it are also heavy in partition P^i.

L_1 is generated at the end of the first pass over the database. Now we can prune M_i $(i > 1)$ by removing all the itemsets that contain some item not in L_1. All the heavy k-itemsets in P^i are generated at the end of counting C_k^i $(k > 1)$. Then we can prune the candidates by removing C_j^i $(j > k)$. Notice that the generation of C_k^i $(k > 1)$ is delayed in succession in order to prune the candidates based on the theorem above.

Example 3. Following Examples 1 and 2 we illustrate the problem. At the end of the first pass, L_1 is generated. So $\{CE\}$ in C_2^2, $\{CE, EF\}$ in C_2^3, $\{CEF\}$ in C_3^3, must be removed, after some large itemsets are generated from C_2^1. We know that $\{AC\}$ is not a large itemset. So $\{ABC\}$ in C_3^1 can be removed. After scanning P^3 in the second pass over the database we know $\{BG\}$ is not large. So $\{BDG\}$ in C_3^4 can be removed.

Using this method only in the first pass, the redundant candidates are counted. We postpone the generating of C_k^i $(k > 1)$ to reduce unnecessary operations. If we don't do so, that is, generating C_k^i at the same time, before the end of the first pass there exists unnecessary count of the candidates which are pruned in the example and this costs a lot of memory. The algorithm prunes candidates by postponing the generation of C_k^i.

4 Conclusion

The algorithm RP reduces the number of passes of scanning the database. As a result it lowers the cost of I/O operations greatly. Although the algorithm adopted the idea of tidlist in the algorithm Partition[5], the idea is only used within the first pass and it does not cause overload of memory. Because the number of candidates in algorithm RP is too large, the result of pruning candidates in Subsection 3.4 is not very good.

We did a simple test about this algorithm. The tested transaction database has 100,000 transactions in which there are about 200 items. We implemented the algorithm in power-builder.

We tested the number of partitions. For a transaction database the number of partitions is determined by the size of the partitions (that is the number of transactions in one partition). We tried the sizes of 100, 500, 1000 and 10,000. The test shows that the middle sizes of 500 and 1000 work the best. The size of 100 shows slower operation. The size of 10,000 was the worst due to more passes over the databases and much heavier overhead.

We also implemented the algorithm Apriori in power-builder. The result shows that if the support threshold is very high (for example 0.02), Apriori is 10% faster than RP; and if the support threshold is not very high (we tried 0.002), RP runs about 20% faster than Apriori. Because the possiblity of too high support level is very low, so the RP algorithm is practical.

References

[1] Agrawal R, Imielinski T, Swami A. Mining association rules between sets of items in large databases. In *Proc. the ACM SIGMOD Conference on Management of Data,* Washington D.C., May 1993, pp.207–216.

[2] Sergey Brin, Rajeev Motwani, Jerry D Ullman *et al.* Dynamic itemset counting and implication rules for market basket data. In *Proceeding of the ACM SIGMOD International Conference on Management of Data,* Volume 26, 2 of SIGMOD 28 Record, New York, May 13–15 1997, ACM Press, pp.255-264..

[3] Agrawal R, Srikant R. Fast algorithms for mining association rules. In *Proc. the 20th Int. Conf. Very Large Databases,* Santiago, Chile, Sept. 1994, pp.487–499.

[4] Charu C Agrawal, Philip S Yu. Mining large itemsets for association rules. In *Bulletin of the IEEE Computer Society Technical Committee on Data Engineering,* March 1998, pp.23–31.

[5] Savasere A, Omiecinski E, Navathe S. An effcient algorithm for mining association rules in large database. In *Proceedings of the Very Large Data Base Conference,* Zurich, September 1995, pp.432–444.

[6] Hannu Toivonen. Sampling large databases for association rules. In *Proceedings of 22nd International Conference on Very Large Data Bases,* Mumbai, India, Sept. 1996, Morgan Kaufmann, pp.134–145.

[7] Mohammed Javeed Zaki, Srinivasan Parthasarathy, Wei Li *et al.* Evaluation of sampling for data mining of association rules. Technical Report 617, Computer Science Dept., U. Rochester, May 1996.

[8] Cheung D, Han J, Ng V *et al.* Maintenance of discovered association rules in large databases: An incremental updating technique. In *Proc. 1996 Int. Conf. Data Engineering,* New Orleans, Lousiana, USA, Feb. 1996, http://www.cs.hku.hk/~dcheungs/publication/icde96.ps

[9] David W L Cheung, Lee S D, Benjamin Kao. A general incremental technique for maintaining discovered association rules. In *Proceedings of the Fifth International Conference on Database Systems for Advanced Applications,* Melbourne, Australia, March 1997, pp.185–194.

[10] Shiby Thomas, Sreenath Bodagala, Khaled Alsabti *et al.* An efficient algorithm for the incremental updating of association rules in large database. In *Proceedings of the 3rd International Conference on Knowledge Discovery and Data Mining,* 1997, pp.134–145.

[11] Charu C Agrawal, Philip S Yu. Online generation of association rules. Technical Report RC 20899 (926090), IBM Research Division, T.J. Watson Research Center, Yorktown Heights, NY. June 1997.

[12] Cheung D, Ng V, Fu A *et al.* Efficient mining of association rules in distributed database. *IEEE Trans. Knowledge and Data Eng.,* 1996, 8(6): 911–922.

LI Qingzhong received his B.S. degree in computer software form Shandong University in 1989 and his Ph.D. degree in computer science and technology from Institute of Computing Technology, The Chinese Academy of Sciences in 2000. He is now an associate professor of Shandong University. His research interests include database systems, data mining.

WANG Haiyang received his B.S. degree in computer software from Shandong University in 1988 and his Ph.D. degree in computer science and technology from Institute of Computing Technology, The Chinese Academy of Sciences in 1999. He is now a professor of Shandong University. His research interests include database systems, data flow system.

YAN Zhongmin is now a B.S. candidate of Department of Computer Science of Shandong University. Her research interests include database systems, data mining.

MA Shaohan is now a professor of Shandong University. He is also a Supervisor of Ph.D. candidates. His research interests include algorithm analysis, artificial intelligence.