ISSN : 2231-0711
Supriya S. Borhade et al | IJCSET(www.ijcset.net) | June 2015 | Vol 5, Issue 6,173-181

Time Efficient Algorithms for Mining Association Rules

Ms. Supriya S. Borhade[1], Dr. S. V. Gumaste[2]

[1]*M.E. Student, Computer Engineering Department, SPCOE , Dumberwadi, Pune Maharashtra, India*

[2] *Professor & Head, Computer Engineering Department, SPCOE , Dumberwadi, Pune Maharashtra, India*

***Abstract*— The difficulty of discovering association rules between items in a large database of corporate or retail organizations sales transactions. In this paper studied and presented two new algorithms for solving this problem that are basically different from the known algorithms. Experiments with mock as well as real-life data show that these algorithms do better than known algorithms by factors ranging from three for small problems to more than an order of magnitude for large problems. Also showing how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called Apriori Hybrid. Scale-up experiments show linear scales that with the number of transactions using Apriori Hybrid algorithm. Apriori Hybrid also has outstanding scale-up properties with respect to the transaction size and the number of items in the database.**

***Keywords*— Apriori, Apriori-Tid, AIS, SETM, Association rule.**

I. INTRODUCTION

Most large retail organizations or corporate are facing the challenge of database mining, which is motivated by the decision support [1]. Progress in bar-code technology has made it possible for retail organizations to collect and store huge amounts of sales data, referred to as the market basket data. A record in such data typically consists of the transaction date and the items brought in the transaction. Most successful organizations view such databases as vital pieces of the marketing strategies [2]. Miners are interested in formulating information-driven marketing processes, managed by database technology, that enable marketers to develop and implement customized marketing programs and strategies [3]. The problem of mining association rules over basket data was introduced in [4]. An example of such a rule might be that 98% of customers who purchase tires and auto accessories also get automotive services done. Searching all such rules is valuable for cross-marketing and attached mailing applications. Other applications include catalogue design, add-on sales, store layout, and customer segmentation based on frequent buying patterns. The databases involved in these applications are very large. It is crucial, therefore, to have quick or fast algorithms for data mining task. The following is a formal statement of the problem [5]:

Let $I = \{i_1, i_2, \ldots, i_N\}$ be a set of literals, called items. Let D be a set of transactions, where each transaction T is a set of items such that $T \subseteq I$. Associated with each transaction is a unique identifier, called its TID. Consider that a transaction T contains X, a set of some items in I, if $X \subseteq T$. An *association rule* is an implication of the form $\rightarrow Y$, where $X \subset I, Y \subset I$, and $X \cap Y = \emptyset$. The rule $X \rightarrow Y$ holds in the transaction set D with confidence c if c% of transactions in D that contain X also contain Y. The rule $X \rightarrow Y$ has support s in the transaction set D if s% of transactions in D contain $\cup Y$. Our rules are somewhat more general than in [4] in that allowed a consequent to have more than one item. Given a set of transactions D, the problem of mining association rules is to generate all association rules that have support and confidence greater than the user-specified minimum support (called $minsup$) and minimum confidence (called $minconf$) respectively. This discussion is neutral with respect to the representation of *D*. For example, *D* could be a data le, a relational table, or the result of a relational expression. An algorithm for finding all association rules, henceforth referred to as the *AIS* algorithm, was presented in [4]. Another algorithm for this task, called the *SETM* algorithm, has been proposed in [17]. In this paper, present two new algorithms, *Apriori* and *Apriori-Tid*, that differs fundamentally from these algorithms. The performance gap is shown to increase with problem size, and ranges from a factor of three for small problems to more than an order of magnitude for large problems. Then discussed how the best features of *Apriori* and *Apriori-Tid* can be combined into a hybrid algorithm, called *AprioriHybrid*. Experiments show that the *AprioriHybrid* has excellent scale-up properties, opening up the feasibility of mining association rules over very large databases. The problem of finding association rules falls within the purview of database mining [5] [6] ,also called knowledge discovery in databases [10] [13] . Related, but not directly applicable, work includes the induction of classification rules [11] [10], discovery of causal rules [12] [14], learning of logical definitions, fitting of functions to data , and clustering [9] . The closest work in the machine learning literature is the KID3 algorithm presented in [18]. If used for finding all association rules, this algorithm will make as many passes over the data as the number of combinations of items in the antecedent, which is exponentially large. Related work in the database literature is the work on inferring functional

dependencies from data [15] [16]. Functional dependencies are rules requiring strict satisfaction.

Consequently, having determined a dependency $X \rightarrow A$, the algorithms in [15] [16]consider any other dependency of the form $X + Y \rightarrow A$ redundant and do not generate it. The association rules considered here are probabilistic in nature. The presence of a rule $X \rightarrow A$ does not necessarily mean that $X + Y \rightarrow A$ also holds because the latter may not have minimum support. Similarly, the presence of rules $X \rightarrow Y$ and $Y \rightarrow Z$ does not necessarily mean that $X \rightarrow Z$ holds because the latter may not have minimum confidence. There has been work on quantifying the ``usefulness" or ``interestingness" of a rule [18]. What is useful or interesting is often application-dependent. The need for a human in the loop and providing tools to allow human guidance of the rule discovery process has been articulated. Do not discuss these issues in this paper, except to point out that these are necessary features of a rule discovery system that may use this algorithms as the engine of the discovery process.

A. Problem Decomposition and Paper Organization

The problem of discovering all association rules can be decomposed into two sub-problems [4]:

1. Find all sets of items (itemsets) that have transaction support above minimum support. The support for an itemset is the number of transactions that contain the itemset. Itemsets with minimum support are called large itemsets, and all others small itemsets. In Section II, give new algorithms, *Apriori* and *Apriori-Tid*, for solving this problem.
2. Use the large itemsets to generate the desired rules. The algorithms for this problem are in Section III. The general idea is that if, say, *ABCD* and *AB* are large itemsets, then one can determine if the rule $AB \rightarrow CD$ holds by computing the ratio $conf = \frac{support(ABCD)}{support(AB)}$ If $conf \geq minconf$, then the rule holds. (The rule will surely have minimum support because *ABCD* is large.) Unlike [4], where rules were limited to only one item in the consequent, for this allow multiple items in the consequent. An example of such a rule might be that in 58% of the cases, a person who orders a comforter also orders a flat sheet, a fitted sheet, a pillow case, and a rule. The algorithms in Section III generate such multi-consequent rules.

In Section IV, shown the relative performance of the proposed *Apriori* and *Apriori-Tid* algorithms against the AIS [4] and SETM [17] algorithms. To make the paper self-contained, also include an overview of the AIS and SETM algorithms in this section. Also describes how the *Apriori* and *Apriori-Tid* algorithms can be combined into a hybrid algorithm, *AprioriHybrid*, and demonstrate the scale-up properties of this algorithm. Finally concluded by pointing some related open problems in Section V.

II. DISCOVERING LARGE ITEMSETS

Algorithms for discovering large itemsets make multiple passes over the data. In the first pass, count the support of individual items and determine which of them are large, i.e. have minimum support. In each subsequent pass, start with a seed set of itemsets found to be large in the previous pass. To use this seed set for generating new potentially large itemsets, called candidate itemsets, and count the actual support for these candidate itemsets during the pass over the data. At the end of the pass, determine which of the candidate itemsets are actually large, and they become the seed for the next pass. This process continues until no new large itemsets are found.

The Apriori and Apriori-Tid algorithms proposed differ fundamentally from the AIS [4] and SETM [17] algorithms in terms of which candidate itemsets are counted in a pass and in the way that those candidates are generated. In both the AIS and SETM algorithms (see Sections L and M for a review), candidate itemsets are generated on-the-y during the pass as data is being read. Specifically, after reading a transaction, it is determined which of the itemsets found large in the previous pass are present in the transaction. New candidate itemsets

are generated by extending these large itemsets with other items in the transaction. However, the disadvantage is that this results in unnecessarily generating and counting too many candidate itemsets that turn out to be small. The Apriori and Apriori-Tid algorithms generate the candidate itemsets to be counted in a pass by using only the itemsets found large in the previous pass without considering the transactions in the database. The basic intuition is that any subset of a large itemset must be large. Therefore, the candidate itemsets having k items can be generated by joining large itemsets having k1 items, and deleting those that contain any subset that is not large. This procedure results in generation of a much smaller number of candidate itemsets. The Apriori-Tid algorithm has the additional property that the database is not used at all for counting the support of candidate itemsets after the first pass. Rather, an encoding of the candidate itemsets used in the previous pass is employed for this purpose. In later passes, the size of this encoding can become much smaller than the database, thus saving much reading effort. Explaining these points in more detail when describes the algorithms.

Notation , assume that items in each transaction are kept sorted in their lexicographic order. It is straightforward to adapt these algorithms to the case where the database D is kept normalized and each database record is a <*TID*, *item*> pair, where *TID* is the identifier of the corresponding transaction.

Call the number of items in an itemset its size, and call an itemset of size k a k-itemset. Items within an itemset are kept in lexicographic order. using notation $c[1], c[2], \ldots, c[k]$ to represent a k-itemset c consisting of items $c[1], c[2], \ldots, c[k]$, where $c[1] < c[2] < \ldots < c[k]$. If $c = X.Y$ and Y is an *m-itemset*, also call Y an *m-extension* of X. Associated with each itemset is a count field to store the support for this itemset. The count field is initialized to zero when the itemset is first created. Also summarized in Table 1 the notation used in the algorithms. The set $\overline{C_k}$ is used by Apriori-Tid and will be further discussed in this algorithm.

TABLE I: Notation

k-itemset	An itemset having k items.
L_k	Set of large k-itemsets (those with minimum support) Each member of this set has two fields : i) itemset and ii) support count.
Ck	Set of candidate k-itemsets (potentially large itemsets). Each member of this set has two fields : i) itemset and ii) support count.
Ck	Set of candidate k-itemsets when the TIDs of the generating transactions are kept associated withthe candidates

B. Algorithm Apriori

Figure 1 gives the Apriori algorithm. The first pass of the algorithm simply counts item occurrences to determine the large 1-itemsets. A subsequent pass, say pass *k*, consists of two phases. First, the large itemsets L_{k-1} found in the $(k-1)$th pass are used to generate the candidate itemsets C_k, using the apriori-gen function described in Section C. Next, the database is scanned and the support of candidates in C_k is counted. For fast counting, need to efficiently determine the candidates in C_k that are contained in a given transaction t. Section E describes the subset function used for this purpose. Section 2.1.3 discusses buffer management.

// **Apriori Algorithm (Figure 1)**

1. $L_1 = \{large\ 1-itemsets\}$;
2. $for\ (k = 2; L_{k-1} \neq \emptyset; k++)\ do\ begin$
3. $C_k = apriori-gen(L_{k-1})$;
4. $For\ all\ transactions\ t \in D\ do\ begin$
5. $C_t = subset(C_k, t)$;
6. $Forall\ candidates\ c \in C_t\ do$
7. $c.count++$;
8. end
9. $L_k = \{c \in Ck \mid c.count \geq minsup\}$
10. end
11. $Answer = \cup_k L_k$;

C. Apriori Candidate Generation

The apriori-gen function takes as argument L_{k-1}, the set of all large $(k-1)-itemsets$. It returns a superset of the set of all large *k-itemsets*. The function works as follows. First, in the join step, join L_{k-1} with L_{k-1}:

// **Apriori Candidate Generation Algorithm**

1. *insert into* C_k
2. *select*

$p.item_1, p.item_2, \ldots, p.item_{k-1}, q.item_{k-1}$

3. $from\ L_{k-1}\ p, L_{k-1}\ q$
4. $where\ p.item_1 = q.item_1, \ldots, p.item_{k-2} =$

D. $q.item_{k-1} < p.item_{k-1} < q.item_{k-1}$;

5. *Next, in the prune step, delete all itemsets* $c \in Ck$ *such that some* $(k-1)-subset$ *of* c *is not in* L_{k-1}*:*
6. *forall itemsets* $c \in C_k$ *do*
7. *forall* $(k-1)$*-subsets s of c do*
8. *if* $(s \in L_{k-1})$ *then*
9. *delete c from* C_k*;*

Example :Let L3 be {{123}, {124},{134}, {135},{234}}. After the join step, C4 will be {{1234},{1345}}. The prune step will delete the itemset {1345} because the itemset {1 4 5} is not in L3. Then be left with only {1234} in C4. contrast this candidate generation with the one used in the AIS and SETM algorithms. In pass k of these algorithms (see Section IV for details), a database transaction t is read and it is determined which of the large itemsets in L_{k-1} are present in t. Each of these large itemsets l is then extended with all those large items that are present in t and occur later in the lexicographic ordering than any of the items in l. Continuing with the previous example, consider a transaction {12345}. In the fourth pass, AIS and SETM will generate two candidates, {1234} and {1 235}, by extending the large itemset {123}. Similarly, an additional three candidate itemsets will be generated by extending the other large itemsets in L_3, leading to a total of 5 candidates for consideration in the fourth pass. Apriori, on the other hand, generates and counts only one itemset, {1345}, because it concludes a priori that the other combinations cannot possibly have minimum support.

Correctness –Also need to show that $C_k \supseteq L_k$. Clearly, any subset of a large itemset must also have minimum support. Hence, if extended each itemset in L_{k-1} with all possible items and then deleted all those whose *(k-1)-subsets* were not in L_{k-1}, one would left with a superset of the itemsets in L_k. The join is equivalent to extending L_{k-1} with each item in the database and then deleting those itemsets for which the (k-1)-itemset obtained by deleting the (k-1)th item is not in L_{k-1}. The condition $p.item_{k-1} < q.item_{k-1}$ simply ensures that no duplicates are generated. Thus, after the join step, $C_k \supseteq L_k$. By similar reasoning, the prune step, delete from C_k all itemsets whose (k-1)-subsets are not in L_{k-1}, also does not delete any itemset that could be in L_k.

Variation: Counting Candidates of Multiple Sizes in One Pass Rather than counting only candidates of size k in the kth pass, one can also count the candidates C'_{k+1}, where C'_{k+1} is generated from C_k, etc. Note that $C'_{k+1} \supseteq C_{k+1}$+1 since $C_{k+1}$is generated from L_k. This variation can pay in the later passes when the cost of counting and keeping in memory additional $C'_{k+1} - C_{k+1}$ candidates becomes less than the cost of scanning the database.

Membership Test :The prune step requires testing that all (k-1)-subsets of a newly generated k-candidate-itemset are present in L_{k-1}. To make this membership test fast, large itemsets are stored in a hash table.

E. Subset Function

Candidate itemsets C_k are stored in a hash-tree. A node of the hash-tree either contains a list of itemsets (a leaf node) or a hash table (an interior node). In an interior node, each bucket of the hash table points to another node. The root of the hash-tree is defined to be at depth 1. An interior node at depth d points to nodes at depth *d + 1*. Itemsets are stored in the leaves. When miner adds an itemset *c*, start from the root and go down the tree until to reach a leaf. At an interior node at depth d, decide which branch to follow by applying a hash function to the d^{th} item of the itemset. All nodes are initially created as leaf nodes. When the number of itemsets in a leaf node exceeds a specified threshold, the leaf node is converted to an interior node. Starting from the root node, the subset function finds all the candidates contained in a transaction t as follows. If one is at a leaf, find which of the itemsets in the leaf are contained in t and add references to them to the answer set. If one is at an interior node and have reached it by hashing the item i, hash on each item that comes after i in t and recursively apply this procedure to the node in the corresponding bucket. For the root node, hash on every item in t. To see why the subset function returns the desired set of references, consider what happens at the root node. For any itemset c contained in transaction t, the first item of c must be in t. At the root, by hashing on every item in t, one can ensure that he is the only ignore itemsets that start with an item not in t. Similar arguments apply at lower depths. The only additional factor is that, since the items in any itemset are ordered, if reach the current node by hashing the item i, only need to consider the items in t that occur after i. If k is the size of a candidate itemset in the hash-tree, one can find in O(k) time whether the itemset is contained in a transaction by using a temporary bitmap. Each bit of the bitmap corresponds an item. The bitmap is created once for the data structure, and reinitialized for each transaction. This initialization takes O(size(transaction)) time for each transaction.

F. Buffer Management

In the candidate generation phase of pass *k*, need storage for large itemsets L_{k-1} and the candidate itemsets C_k. In the counting phase, need storage for C_k and at least one page to buffer the database transactions. First, assume that L_{k-1} fits in memory but that the set of candidates C_k does not. The Apriori-gen function is modified to generate as many candidates of C_k as will t in the buffer and the database is scanned to count the support of these candidates. Large itemsets resulting from these candidates are written to disk, while those candidates without minimum support are deleted. This procedure is repeated until all of C_k has been counted. If L_{k-1} does not t in memory either, externally sort L_{k-1}. It was bring into memory a block of L_{k-1} in which the first (k-2) items are the same. Now generate candidates using this block. Keep reading blocks of L_{k-1} and generating candidates until the memory fills up, and then make a pass over the data. This procedure is repeated until all of C_k has been counted. Unfortunately, one can no longer prune those candidates whose subsets are not in L_{k-1}, as the whole of L_{k-1} is not available in memory.

G. Algorithm Apriori-Tid

The Apriori-Tid algorithm, shown in Figure 2, also uses the apriori-gen function (given in Section C) to determine the candidate itemsets before the pass begins. The interesting feature of this algorithm is that the database D is not used for counting support after the first pass. Rather, the set $\bar{C}_k$ is used for this purpose. Each member of the set $\bar{C}_k$ is of the form $< TID, \{X_k\} >$, where each X_k is a potentially large k-itemset present in the transaction with identifier TID. For $k = 1, \bar{C}_1$ corresponds to the database D, although conceptually each item i is replaced by the itemset {i}. For k > 1, $\bar{C}_k$ is generated by the algorithm (step 10). The member of $\bar{C}_k$ corresponding to transaction t is $<t.TID, \{c \in C_k \mid c \text{ contained in } t\}>$. If a transaction does not contain any candidate k-itemset, then $\bar{C}_k$ will not have an entry for this transaction. Thus, the number of entries in $\bar{C}_k$ may be smaller than the number of transactions in the database, especially for large values of k. In addition, for large values of k, each entry may be smaller than the corresponding transaction because very few candidates may be contained in the transaction. However, for small values for k, each entry may be larger than the corresponding transaction because an entry in C_k includes all candidate k-itemsets contained in the transaction. Further explore this trade-off in Section IV. Establish the correctness of the algorithm in Section H. In Section I , given the data structures used to implement the algorithm, and discussed buffer management in Section J.

// **Apriori-Tid Algorithm (figure 2)**

1. $L_1 = \{large\ 1-itemsets\}$;
2. $\bar{C}_1 = database\ D$;
3. $for\ (k = 2;\ L_{k-1} \neq \emptyset;\ k++)$ do begin
4. $C_k = apriori-gen(L_{k-1})$;
5. $\bar{C}_k = \emptyset$;
6. forall entries $t \in \bar{C}_{k-1}$ do begin
7. $C_t = \{c \in C_k \mid (c - c[k]) \in t.set-of-itemsets \wedge (c - c[k-1]) \in t.set-of-itemsets\}$;
8. $forall\ candidates\ c \in C_t\ do$
9. $c.count++$;
10. $if\ (C_t \neq \emptyset)\ then\ \bar{C}_k += < t.TID, C_t >$;
11. end
12. $L_k = \{c \in C_k \mid c.count \geq minsup\}$
13. end
14. $Answer = \cup_k L_k$;

Example: Consider the database in Figure 3 and assume that minimum support is 2 transactions. Calling apriori-gen with L_1 at step 4 gives the candidate itemsets C_2. In steps 6 through 10, count the support of candidates in C_2 by iterating over the entries in C_1 and generate C2. The first entry in $\bar{C}_1$ is { {1}{3} {4} }, corresponding to transaction 100. The C_t at step 7 corresponding to this entry *t* is {{1 3}}, because {1 3}g is a member of C_2 and both ({1 3} –

{1}) and ({1 3} –{3}) are members of t.set-of-itemsets. Calling apriori-gen with L_2 gives C_3. Making a pass over the data with C_2 and C_3 generates C_3. Note that there is no entry in C_3 for the transactions with TIDs 100 and 400, since they do not contain any of the itemsets in C_3. The candidate {235} in C_3 turns out to be large and is the only member of L_3. after generating C_4 using L_3, it turns out to be empty, and terminate.

H. Correctness

Rather than using the database transactions, Apriori-Tid uses the entries in $\bar{C}_k$ to count the support of candidates in C_k.

TID	Items
100	134
200	235
300	1235
400	2 5

Database

TID	Set-of-Itemsets
100	{{1},{3},{4}}
200	{{2}{3}{5}}
300	{{1}{2}{3}{5}}
400	{2}{5}

$\bar{C}_1$

Itemset	Support
{1}	2
{2}	3
{3}	3
{5}	3

L_1

Itemset
{1 2 }
{1 3}
{1 5}
{2 3}
{2 5}
{3 5}

C_2

TID	Set-of-Itemsets
100	{{1 3}}
200	{{2 3}{2 5}{3 5}}
300	{{1 2}{1 3}{1 5} {2 3}{2 5}{3 5}}
400	{{2 5}}

$\bar{C}_2$

Itemset	Support
{1 3}	2
{2 3}	2
{2 5}	3
{3 5}	2

L_2

Itemset
{2 3 5}

C_3

TID	Set-of-Itemsets
200	{2 3 5}
300	{2 3 5}

$\bar{C}_3$

Itemset	Support
{2 3 5}	2

L_3

Figure. 3

To simplify the proof, it's assumed that in step 10 of Apriori-Tid, always add $\langle t.TID, C_t \rangle$ to $\bar{C}_k$, rather than adding an entry only when C_t is non-empty. For correctness, one need to establish that the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier $t.TID$. It says that the set C_k is complete if $\forall t \in \bar{C}_k$, $t.set-of-itemsets$ includes all large k-itemsets contained in the transaction with identifier $t.TID$. It says that the set $\bar{C}_k$ is correct if $\forall t \in C_k$, $t.set-of-itemsets$ does not include any k-itemset not contained in the transaction with identifier $t.TID$. The set L_k is correct if it is the same as the set of all large k-itemsets. It says that the set C_t generated in step 7 in the kth pass is correct if it is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier $t.TID$.

Lemma 1 : $\forall t \geq 1$, if $\bar{C}_{k-1}$ is correct and complete and L_{k-1} is correct, then the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier *t.TID*. By simple rewriting, a candidate itemset $c = c[1]\ c[2]\ \ldots\ c[k]$ is present in transaction *t.TID* if and only if both $c^1 = (c - c[k])$ *and* $c^2 = (c - c[k-1])$ are in transaction *t.TID*. Since C_k was obtained by calling apriori-gen(L_{k-1}), all subsets of $c \in C_k$ must be large. So, c^1 and c^2 must be large itemsets. Thus, if $c \in C_k$ is contained in transaction *t.TID* c^1 and c^2 must be members of $t.set-of-itemsets$ since $\bar{C}_{k-1}$ is complete Hence c will be a member of C_t. Since $\bar{C}_{k-1}$ is correct, if $c^1(c^2)$ is not present in transaction *t.TID* then $c^1(c^2)$ is not contained in $t.set-of-itemsets$. Hence, if $c \in C_k$ is not contained in transaction $t.TID$, c will not be a member of C_t.

Lemma 2 : $\forall t \geq 1$, if L_{k-1} is correct and the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier t.TID, then the set $\bar{C}_k$ is correct and complete.

Since the apriori-gen function guarantees that, $C_k \supseteq L_k$ the set C_t includes all large k-itemsets contained in *t.TID*. These are added in step 10 to $\bar{C}_k$ and hence $\bar{C}_k$ is complete. Since C_t only includes itemsets contained in the transaction *t.TID*, and only itemsets in C_t are added to $\bar{C}_k$, it follows that $\bar{C}_k$ is correct.

Theorem 1 : $\forall t \geq 1$, the set C_t generated in step 7 in the kth pass is the same as the set of candidate k-itemsets in C_k contained in the transaction with identifier $t.TID$.

First prove by induction on k that the set C_k is correct and complete and L_k correct for all $(k \geq 1)$. For $k = 1$, this is trivially true since $\bar{C}_1$ corresponds to the database D. By definition, L_1 is also correct. Assume this holds for $k = n$. From Lemma 1, the set C_t generated in step 7 in the $(n+1)$th pass will consist of exactly those itemsets in C_{n+1} contained in the transaction with identifier $t.TID$.

Since the *apriori-gen* function guarantees $C_{n+1} \supseteq L_{n+1}$ and C_t is correct, L_{n+1} will be correct. >From Lemma 2, the set C_{n+1} will be correct and complete. Since $\bar{C}_k$ is correct and complete and L_k correct for all $k \geq 1$, the theorem follows directly from Lemma 1.

I. Data Structures

By assigning each candidate itemset a unique number, called its ID. Each set of candidate itemsets C_k is kept in an array indexed by the IDs of the itemsets in C_k. A member of $\bar{C}_k$ is now of the form $\langle TID; \{ID\} \rangle$.

Each C_k is stored in a sequential structure. The apriori-gen function generates a candidate k-itemset ck by joining two large (k-1)-itemsets. Maintain two additional fields for each candidate itemset: i) generators and ii) extensions. The generators field of a candidate itemset C_k stores the IDs of the two large (k-1)-itemsets whose join generated ck. The extensions field of an itemset C_k stores the IDs of all the (k+1)-candidates that are extensions of C_k. Thus, when a candidate C_k is generated by joining l^1_{k-1} and l^2_{k-1}, save the IDs of l^1_{k-1} and l^2_{k-1} in the generators field for C_k. At the same time, the ID of ck is added to the extensions field of l^1_{k-1}.

Now describing how 7 of Figure 2 Step is implemented using the above data structures. Recall that $t.set-of-itemsets$ field of an entry t in C_{k-1} gives the IDs of all (k-1)-candidates contained in transaction t.TID. For each such candidate C_{k-1} the extensions field gives T_k, the set of IDs of all the candidate k-itemsets that are extensions of C_{k-1}. For each C_k in T_k, the generators field gives the IDs of the two itemsets that generated ck. If these itemsets are present in the entry for $t.set-of-itemsets$, one can conclude that C_k is present in transaction t.TID, and add C_k to C_t.

For this actually need to store only l^2_{k-1} in the generators field, since reached C_k starting from the ID of l^1_{k-1} in t. Omitted this optimization in the above description to simplify exposition. Given an ID and the data structures above, one can find the associated candidate itemset in constant time. One can also find in constant time whether or not an ID is present in the $t.set-of-itemsets$ field by using a temporary bitmap. Each bit of the bitmap corresponds to an ID in C_k. This bitmap is created once at the beginning of the pass and is reinitialized for each entry t of C_k.

J. Buffer Management

In the kth pass, Apriori-Tid needs memory for L_{k-1} and C_k during candidate generation. During the counting phase, it needs memory for C_{k-1}, C_k, and a page each for C_{k-1} and C_k. Note that the entries in C_{k-1} are needed sequentially and that the entries in Ck can be written to disk as they are generated. At the time of candidate generation, after joining L_{k-1} with itself, it fill up roughly half the buffer with candidates. This allows us to keep the relevant portions of both Ck and C_{k-1} in memory during the counting phase. In addition, its ensure that all candidates with the same first (k-1) items are generated at the same time. The computation is now effectively partitioned because none of the candidates in memory that turn out to large at the end of the pass will join with any of the candidates not yet generated to derive potentially large itemsets. Hence one can assume that the candidates in memory are the only candidates in C_k and find all large itemsets that are extensions of candidates in C_k by running the algorithm to completion. This may cause further partitioning of the computation downstream. Having thus run the algorithm to completion, return to L_{k-1}, generate some more candidates in C_k, count them, and so on. Note that the prune step of the apriori-gen function cannot be applied after partitioning because one may do not know all the large k-itemsets. When L_k does not fit in memory, miner need to externally sort L_k as in the buffer management scheme used for Apriori.

III. Discovering Rules

The association rules that considered here are somewhat more general than in [4] in that it allow a consequent to have more than one item; rules in [4] were limited to single item consequents. first give a straightforward generalization of the algorithm in [4] and then present a faster algorithm. To generate rules, for every large itemset l, find all non-empty subsets of l. For every such subset a, output a rule of the form $a \rightarrow (l - a)$ if the ratio of support(l) to support(a) is at least $minconf$. Consider all subsets of l to generate rules with multiple consequents. Since the large itemsets are stored in hash tables, the support counts for the subset itemsets can be found efficiently. One can improve the above procedure by generating the subsets of a large itemset in a recursive depth-first fashion. For example, given an itemset ABCD, first consider the subset ABC, then AB, etc. Then if a subset a of a large itemset l does not generate a rule, the subsets of a need not be considered for generating rules using l. For example, if $ABC \rightarrow D$ does not have enough confidence, need not check whether $AB \rightarrow CD$ holds. Do not miss any rules because the support of any subset ~a of a must be as great as the support of a. Therefore, the confidence of the rule $\bar{a} \rightarrow (l - \bar{a})$ cannot be more than the confidence of $a \rightarrow (l - a)$. Hence, if a did not yield a rule involving all the items in l with a as the antecedent, neither will $\bar{a}$. The following algorithm embodies these ideas:

// Simple Algorithm

forall large itemsets l_k, $k \geq 2$ do
call genrules(l_k, l_k);
The genrules generates all valid rules
$\bar{a} \rightarrow l_k - \bar{a}$, for all $\bar{a} \subset a_m$
procedure genrules(l_k: large k − itemset, a_m: large m − itemset)
1) *$A = \{(m-1)$ − itemsets a_{m-1} | $a_{m-1} \subset a_m\}$;*
2) *forall $a_{m-1} \in A$ do begin*
3) *$conf = support(l_k)/support(a_{m-1})$;*
4) *if ($conf \geq minconf$) then begin*
7) *output the rule $a_{m-1} \rightarrow (l_k - a_{m-1})$,*
with confidence = conf and support = $support(l_k)$;
8) *if ($m - 1 > 1$) then*
9) *call genrules(l_k, a_{m-1});*
10) *end*
11) *end*

K. A Faster Algorithm

In previous section if $a \rightarrow (l - a)$ does not hold, neither does $\bar{a} \rightarrow (l - \bar{a})$ for any $\bar{a} \subset a$. By rewriting, it follows that for a rule $(l - c) \rightarrow c$ to hold, all rules of the form $(l - \bar{c}) \rightarrow \bar{c}$ must also hold, where $\bar{c}$ is a non-empty subset of c. For example, if the rule AB →CD holds, then the rules $ABC \rightarrow D$ and $ABD \rightarrow$ C must also hold. Consider the above property that for a given large itemset, if a rule with consequent c holds then so do rules with consequents that are subsets of c. This is similar to the property that if an itemset is large then so is all its subsets. From a large itemset l, therefore, first generate all rules with one item in the consequent. Then by using the consequents of these rules and the function apriori-gen in Section C to generate all possible consequents with two items that can appear in a rule generated from l, etc. An algorithm using this idea is given below. The rules having one-item consequents in step 2 of this algorithm can be found by using a modified version of the preceding general rules function in which steps 8 and 9 are deleted to avoid the recursive call.

// Faster Algorithm

1) *forall large k − itemsets* $l_k, k \geq 2$ *do begin*
2) $H_1 = \{$*consequents of rules derived from* l_k *with one item in the consequent*$\}$;
3) *call ap − genrules*(l_k, H_1);
4) *end*
procedure ap − genrules$(l_k$: *large k − itemset*, H_m: *set of* m *− item consequents*)
if $(k > m + 1)$ *then begin*
$H_{m+1} = apriori\_gen(H_m)$;
forall $h_{m+1} \in H_{m+1}$ *do begin*
$conf = support(l_k) / support(l_k - h_{m+1})$;
if $conf \geq minconf$ *then*
output the rule $(l_k - h_{m+1}) \rightarrow h_{m+1}$ *with*
con_dence $= conf$ *and support* $= support(l_k)$;
else
delete h_{m+1} *from* H_{m+1};
end
call ap − genrules(l_k, H_{m+1});
end

As an example of the advantage of this algorithm, consider a large itemset $ABCDE$. Assume that $ACDE \rightarrow B$ and $ABCE \rightarrow D$ are the only one-item consequent rules derived from this itemset that have the minimum confidence. If use the simple algorithm, the recursive call genrules $(ABCDE, ACDE)$ will test if the two-item consequent rules $ACD \rightarrow BE, ADE \rightarrow BC$, $CDE \rightarrow BA$, and $ACE \rightarrow BD$ hold. The first of these rules cannot hold, because $E \subset BE$, and ABCD → E does not have minimum con_dence. The second and third rules cannot hold for similar reasons. The call genrules $(ABCDE, ABCE)$ will test if the rules $ABC \rightarrow DE$, $ABE \rightarrow DC$, $BCE \rightarrow DA$ and $ACE \rightarrow BD$ hold, and will find that the first three of these rules do not hold. In fact, the only two-item consequent rule that can possibly hold is $ACE \rightarrow BD$, where B and D are the consequents in the valid one-item consequent rules. This is the only rule that will be tested by the faster algorithm.

IV. PERFORMANCE

To assess the relative performance of the algorithms for discovering large itemsets, performed several experiments on an IBM RS/6000 530H workstation with a CPU clock rate of 33 MHz, MB of main memory, and running AIX 3.2. The data resided in the AIX file system and was stored on a 2GB SCSI 3.5" drive, with measured sequential throughput of about 2 MB/second.

L. The AIS Algorithm

Figure 4 summarizes the essence of the AIS algorithm (see [4] for further details). Candidate itemsets are generated and counted on-the-y as the database is scanned. After reading a transaction, it is determined which of the itemsets that were found to be large in the previous pass are contained in this transaction (step 5). New candidate itemsets are generated by extending these large itemsets with other items in the transaction (step 7). A large itemset l is extended with only those items that are large and occur later in the lexicographic ordering of items than any of the items in l. The candidates generated from a transaction are added to the set of candidate
itemsets maintained for the pass, or the counts of the corresponding entries are increased if they were created by an earlier transaction (step 9).

// AIS Algorithm (figure 4)

1) $L_1 = \{large\ 1 - itemsets\}$;
2) *for* $(k = 2; L_{k-1} \neq \emptyset; k++)$ do begin
3) $C_k = \emptyset$;
4) *forall transactions* $t \in D$ *do begin*
5) $L_t = subset(L_{k-1}, t)$;
6) *forall large itemsets* $l_t \in L_t$ *do begin*
7) $C_t = 1 - extensions\ of\ l_t\ contained\ in\ t$;
8) *forall candidates* $c \in C_t$ *do*
9) *if* $(c \in C_k)$ *then*
add 1 to the count of c to corresponding in C_k;
else
add c to C_k *with a count of 1*;
10) *end*
11) $L_k = \{c \in C_k \mid c.count \geq minsup\}$
12) *end*
13) $Answer = \bigcup_k L_k$;

Data Structures: The data structures required for maintaining large and candidate itemsets were not specified in [4]. Store the large itemsets in a dynamic multi-level hash table to make the subset operation in step 5 fast, using the algorithm described in the previous section. Candidate itemsets are kept in a hash table associated with the respective large itemsets from which they originate in order to make the membership test in step 9 fast. Buffer Management When a newly generated candidate itemset causes the buffer to overflow, discard from memory the corresponding large itemset and all candidate itemsets generated from it. This reclamation procedure is executed as often as necessary during a pass. The large itemsets discarded in a pass are extended in the next pass. This technique is a simplified version of the buffer management scheme presented in [4].

M. The SETM Algorithm

The SETM algorithm [17] was motivated by the desire to use SQL to compute large itemsets. Our description of this algorithm in Figure 5 uses the same notation as used for the other algorithms, but is functionally identical to the SETM algorithm presented in [17]. $\bar{C}_k(\bar{L}_k)$ in Figure 5 represents the set of candidate (large) itemsets in which the TIDs of the generating transactions have been associated with the itemsets. Each member of these sets is of the form< TID; itemset >. Like AIS, the SETM algorithm also generates candidates on-the-y based on transactions read from the database. It thus generates and counts every candidate itemset that the AIS algorithm generates. However, to use the standard SQL join operation for candidate generation, SETM separates candidate generation from counting. It saves a copy of the candidate itemset together with the TID of the generating transaction in a sequential structure (step 9). At the end of the pass, the support count of candidate itemsets is determined by sorting (step 12) and aggregating this sequential structure (step 13).

SETM remembers the TIDs of the generating transactions with the candidate itemsets. To avoid needing a subset operation, it uses this information to determine the large itemsets contained in the transaction read (step 6). $\bar{L}_k \subseteq \bar{C}_k$ and is obtained by deleting those candidates that do not have minimum support (step 13). Assuming that the database is sorted in TID order, SETM can easily find the large itemsets contained in a transaction in the next pass by sorting $\bar{L}_k$on TID (step 15). In fact, it needs to visit every member of $\bar{L}_k$ only once in the TID order, and the candidate generation in steps 5 through 11 can be performed using the relational merge-join operation [17]. The disadvantage of this approach is mainly due to the size of candidate sets $\bar{C}_k$. For each candidate itemset, the candidate set now has as many entries as the number of transactions in which the candidate itemset is present. Moreover, ready to count the support for candidate itemsets at the end of the pass, $\bar{C}_k$ is in the wrong order and needs to be sorted on itemsets (step 12). After counting and pruning out small candidate itemsets that do not have minimum support, the resulting set $\bar{L}_k$ needs another sort on TID (step 15) before it can be used for generating candidates in the next pass.

//SETM Algorithm (figure5)

1) L_1 = {large 1− itemsets};
2) $\bar{L}_1$ = {large 1− itemsets together with the TIDs in which they appear sorted on TID};
3) for (k = 2; $\bar{L}_{k-1} \neq \emptyset$; k++) do begin
4) $\bar{C}_k = \emptyset$;
5) forall transactions $t \in D$ do begin
6) L_t = {$l \in \bar{L}_{k-1}$ | $l.TID = t.TID$};
7) forall large itemsets $l_t \in L_t$ do begin
8) C_t = 1− extensions of l_t contained in t;
9 $\bar{C}_k$+= {< $t.TID, c$ > | $c \in C_t$};
10} end
11} end
12} sort $\bar{C}_k$on itemsets;
13} delete all itemsets $c \in \bar{C}_k$ for which c.count < minsup giving $\bar{L}_k$;
14} $\bar{L}_k$ = {< l, itemset count of l in $\bar{L}_k$ > | $l \in \bar{L}_k$};
15} sort $\bar{L}_k$ on TID;
16} end
17} Answer = $\cup_k \bar{L}_k$;

Buffer Management: The performance of the SETM algorithm critically depends on the size of the set $\bar{C}_k$relative to the size of memory. If $\bar{C}_k$ fits in memory, the two sorting steps can be performed using an in-memory sort. In [17], $\bar{C}_k$ was assumed to fit in main memory and buffer management was not discussed.

If $\bar{C}_k$ is too large to fit in memory, write the entries in $\bar{C}_k$ to disk in FIFO order when the buffer allocated to the candidate itemsets fills up, as these entries are not required until the end of the pass. However, $\bar{C}_k$ now requires two external sorts.

V. Conclusions and Future Work

In this paper presented two algorithms, Apriori and Apriori-Tid, for discovering all major association rules between items in a large database of transactions. Here compared these algorithms to the previously known algorithms, the AIS [4] and SETM [17] algorithms. Also presented the proposed algorithms always do better AIS and SETM. The performance gap increased with the problem size, and ranged from a factor of three for small problems to more than an order of magnitude for large problems. Also studied the best features of the two proposed algorithms can be combined into a hybrid algorithm, called Apriori-Hybrid, which then becomes the algorithm of choice for this problem. Scale-up experiments showed that Apriori-Hybrid scales linearly with the number of transactions. In addition, the execution time decreases a little as the number of items in the database increases. As the average transaction size increases (while keeping the database size constant), the execution time increases only gradually. Feasibility study of using Apriori-Hybrid in real applications involves very large databases. The algorithms presented in this paper have been implemented on several data repositories, including the AIX file system, DB2/MVS, and DB2/6000. In the future, plan to extend this study work along the following dimensions:

1)Multiple taxonomies (is-a hierarchies) over items are often available. An example of such a hierarchy is that a dish washer is a kitchen appliance is a heavy electric appliance, etc. also would like to be able to find association rules that use such hierarchies.

2) For this study work did not considered the quantities of the items bought in a transaction, which are useful for some applications. Finding such rules needs further work.

The work reported in this paper has been done in the context of the Quest project at the IBM Almaden Research Center. Explore the various aspects of the database mining problem. Besides the problem of discovering association rules, some other problems that looked into include the improvement of the database capability with classification queries [8] and similarity queries over time sequences [7]. Believe that database mining is an important new

application area for databases, combining commercial interest with interesting research questions.

REFERENCES

[1] M. Stonebraker et al. The DBMS research at crossroads. In Proc. of the VLDB Conference, Dublin, August 1993.

[2] Direct Marketing Association. Managing database marketing technology for success, 1992.

[3] David Shepard Associates. The new direct marketing. Business One Irwin, Illinois,1990.

[4] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Mining association rules between sets of items in large databases. In Proc. of the ACM SIGMOD Conference on Management of Data, pages 207-216, Washington, D.C., May 1993.

[5] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami. Database mining: A performance perspective. IEEE Transactions on Knowledge and Data Engineering, 5(6):914-925, December 1993. Special Issue on Learning and Discovery in Knowledge-Based Databases.

[6] Tarek M. Anwar, Howard W. Beck, and Shamkant B. Navathe. Knowledge mining by imprecise querying: A classi_cation-based approach. In IEEE 8th Int'l Conf. on Data Engineering, Phoenix, Arizona, February 1992.

[7] Rakesh Agrawal, Christos Faloutsos, and Arun Swami. E_cient similarity search in sequence databases. In Proc. of the Fourth International Conference on Foundations of Data Organization and Algorithms, Chicago, October 1993. Also in Lecture Notes in Computer Science 730, Springer Verlag, 1993, 69-84.

[8] Rakesh Agrawal, Sakti Ghosh, Tomasz Imielinski, Bala Iyer, and Arun Swami, An interval classifier for database mining applications. In Proc. of the VLDB Conference,pages 560-573, Vancouver, British Columbia, Canada, August 1992.

[9] Tarek M. Anwar, Shamkant B. Navathe, and Howard W. Beck., Knowledge mining in databases: A unified approach through conceptual clustering. Technical report, GeorgiaInstitute of Technology, May 1992.

[10] Jiawei Han, Yandong Cai, and Nick Cercone, Knowledge discovery in databases: An attribute oriented approach. In Proc. of the VLDB Conference, pages 547{559, Vancouver, British Columbia, Canada, 1992.

[11] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, Classification and Regression Trees. Wadsworth, Belmont, 1984.

[12] G. Cooper and E. Herskovits, A Bayesian method for the induction of probabilistic networks from data. Machine Learning, 1992.

[13] David J. Lubinsky, Discovery from databases: A review of AI and statistical techniques.In IJCAI-89 Workshop on Knowledge Discovery in Databases, pages 204-218, Detroit,August 1989.

[14] J. Pearl, Probabilistic reasoning in intelligent systems: Networks of plausible inference,1992.

[15] D. Bitton, Bridging the gap between database theory and practice, 1992.

[16] Heikki Mannila and Kari-Jouku Raiha, Dependency inference. In Proc. of the VLDB Conference, pages 155-158, Brighton, England, 1987.

[17] Maurice Houtsma and Arun Swami. Set-oriented mining of association rules. Research Report RJ 9567, IBM Almaden Research Center, San Jose, California, October 1993.