12 Fast Discovery of Association Rules

Rakesh Agrawal
IBM Almaden Research Center

Heikki Mannila
University of Helsinki

Ramakrishnan Srikant
IBM Almaden Research Center

Hannu Toivonen
University of Helsinki

A. Inkeri Verkamo
University of Helsinki

Abstract

Association rules are statements of the form "98% of customers that purchase tires and automobile accessories also get automotive services." We consider the problem of discovering association rules between items in large databases. We present two new algorithms for solving this problem. Experiments with synthetic data show that these algorithms outperform previous algorithms by factors ranging from three for small problems to more than an order of magnitude for large problems. We show how the best features of the two proposed algorithms can be combined into a hybrid algorithm. Scale-up experiments show that the hybrid algorithm scales linearly with the number of transactions and that it has excellent scale-up properties with respect to the transaction size and the number of items in the database. We also give simple information-theoretic lower bounds for the problem of finding association rules, and show that sampling can be in some cases an efficient way of finding such rules.

12.1 Association Rules

Recently, Agrawal, Imielinski, and Swami (1993) introduced a class of regularities, *association rules*, and gave an algorithm for finding such rules. An association rule is an expression $X \Rightarrow Y$, where X and Y are sets of items. The intuitive meaning of such a rule is that transactions of the database which contain X tend to contain Y. An example of such a

rule might be that 98% of customers that purchase tires and automobile accessories also have automotive services carried out.

Application domains for association rules range from decision support to telecommunications alarm diagnosis and prediction. The prototypical application is in analysis of sales data. Bar-code technology has made it possible for retail organizations to collect and store massive amounts of sales data, referred to as *basket* data. A record in such data typically consists of the transaction date and the items bought in the transaction. Finding association rules from such basket data is valuable for cross-marketing and attached mailing applications. Other applications include catalog design, add-on sales, store layout, and customer segmentation based on buying patterns.

The following is a formal statement of the problem (Agrawal *et al.* 1993): Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be a set of literals, called items. Let $\mathcal{D}$ be a set of transactions, where each transaction T is an itemset such that $T \subseteq \mathcal{I}$. (In other words, $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ is a set of attributes over the binary domain $\{0, 1\}$. A tuple T of the database $\mathcal{D}$ is represented by identifying the attributes with value 1.) Associated with each transaction is a unique identifier, called its *TID*. A set of items $X \subset \mathcal{I}$ is called an *itemset*. We say that a transaction T *contains* an itemset X, if $X \subseteq T$. An *association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. The rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence* c if $c\%$ of transactions in $\mathcal{D}$ that contain X also contain Y. The rule $X \Rightarrow Y$ has *support* s in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ contain $X \cup Y$. Negatives, or missing items, are not considered of interest in this approach. Our rules are somewhat more general than in (Agrawal *et al.* 1993) in that we allow a consequent to have more than one item.

Given a set of transactions $\mathcal{D}$, the problem of mining association rules is to generate *all* association rules that have certain user-specified minimum support (called *minsup*) and confidence (called *minconf*). Note that we are interested in *discovering* all rules rather than *verifying* whether a particular rule holds. Verification has the limitation that we may miss surprising rules and changing trends. Our discussion is neutral with respect to the representation of $\mathcal{D}$. For example, $\mathcal{D}$ could be a data file, a relational table, or the result of a relational expression.

This article is the result of research undertaken independently as part of the Quest project at IBM Almaden Research Center (Agrawal and

Srikant 1994) and research at the University of Helsinki (Mannila *et al.* 1994).

12.1.1 Related Work

Related, but not directly applicable, work includes the induction of classification rules (Breiman *et al.* 1984; Quinlan 1993), discovery of causal rules (Spiegelhalter *et al.*1993), learning of logical definitions (Muggleton and Feng 1992), fitting of functions to data (Langley *et al.* 1987), and clustering (Cheeseman *et al.* 1988; Fisher 1987). The closest work in the machine learning literature is the KID3 algorithm presented in (Piatetsky-Shapiro 1991). 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 (Mannila and Räihä 1992; Mannila and Räihä 1994).

There has been work on quantifying the "usefulness" or "interestingness" of a rule (Piatetsky-Shapiro 1991; Piatetsky-Shapiro and Matheus 1994). 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, for example, in (Brachman *et al.* 1993) and (Klemettinen *et al.* 1994). We do not discuss these issues in this chapter, except to point out that these are necessary features of a rule discovery system that may use our algorithms as the engine of the discovery process.

12.1.2 Chapter Organization

The rest of this chapter is organized in the following manner. In Section 12.2, we describe new algorithms, Apriori and AprioriTid, for discovering all itemsets that have at least *minsup* support. We give an algorithm for using the itemsets to generate association rules in Section 12.3.

In Section 12.4, we provide empirical results of the performance and compare the Apriori and AprioriTid algorithms against the AIS (Agrawal *et al.* 1993) and SETM (Houtsma and Swami 1993) algorithms, the two other algorithms available in the literature. We describe how the Apriori and AprioriTid algorithms can be combined into a hybrid algorithm,

AprioriHybrid, and demonstrate the scale-up properties of this algorithm.

We study the theoretical properties of the problem of finding association rules in Section 12.5. Section 12.6 is a short conclusion.

12.2 Discovering Large Itemsets

The problem of discovering all association rules of sufficient support and confidence can be decomposed into two subproblems (Agrawal *et al.* 1993):

1. Find all combinations of items that have transaction support above minimum support. Call those combinations *large* itemsets and all other combinations *small* itemsets. We describe new algorithms, Apriori and AprioriTid, for solving this problem.
2. Use the large itemsets to generate the desired rules. We provide an algorithm for this problem in Section 12.3. The general idea is that if, say, $ABCD$ and AB are large itemsets, then we can determine if the rule $AB \Rightarrow CD$ holds by computing the ratio r = support($ABCD$)/support(AB). Only if $r \geq$ *minconf*, then the rule holds. Note that the rule will have minimum support because $ABCD$ is large.

It is easy to see that this two-step approach is in a sense optimal: the problem of finding large itemsets can be reduced to the problem of finding all association rules that hold with a given confidence. Namely, if we are given a set of transactions $\mathcal{D}$, we can find the large itemsets by adding an extra item j to each transaction in $\mathcal{D}$ and then finding the association rules that have j on the right-hand side and hold with confidence 100%.

The algorithms for discovering all large itemsets make multiple passes over the data. In each pass, we start with a *seed* set of large itemsets and use the seed set for generating new potentially large itemsets, called *candidate* itemsets. We find the support count for these candidate itemsets during the pass over the data. At the end of the pass, we 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. In the first pass, we count the support of individual

items and determine which of them are large. This could be considered breadth-first search in the space of potentially large itemsets.

In both the AIS (Agrawal *et al.* 1993) and SETM (Houtsma and Swami 1993) algorithms candidate itemsets are generated on-the-fly during the database pass. 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, as we will see, this approach results in unnecessarily generating and counting too many candidate itemsets that turn out to be small.

On the other hand, the Apriori and AprioriTid 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 combinatorial property used 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 $k-1$ 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, i.e., in effect pruning the search space.

The AprioriTid 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. (This encoding tells us what candidates were present in which transactions.) In later passes, the size of this encoding can become much smaller than the database, thus saving much reading effort.

Notation Assume for simplicity that items in transactions and itemsets are kept sorted in their lexicographic order. We call the number of items in an itemset its *size*, and call an itemset of size k a k-itemset. We use the notation $c[1] \cdot c[2] \cdot \ldots \cdot c[k]$ to represent a k-itemset c consisting of items $c[1], c[2], \ldots c[k]$, where $c[1] < c[2] < \ldots < c[k]$. Associated with each itemset is a count field to store the support for this itemset.

We summarize in Table 12.1 the notation used in the algorithms. The set $\widehat{C}_k$ is used by AprioriTid and will be further discussed when we describe this algorithm.

Table 12.1
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.
C_k	Set of candidate k-itemsets (potentially large itemsets). Each member of this set has two fields: i) itemset and ii) support count.
$\widehat{C}_k$	Set of candidate k-itemsets when the TIDs of the generating transactions are kept associated with the candidates.

```
1)  L_1 = {large 1-itemsets};
2)  for ( k = 2; L_{k-1} ≠ ∅; k++ ) do begin
3)     C_k = apriori-gen(L_{k-1}); // New candidates
4)     forall transactions t ∈ 𝒟 do begin
5)        C_t = subset(C_k, t); // Candidates contained in t
6)        forall candidates c ∈ C_t do
7)           c.count++;
8)     end
9)     L_k = {c ∈ C_k | c.count ≥ minsup}
10) end
11) Answer = ⋃_k L_k;
```

Figure 12.1
Algorithm apriori.

12.2.1 Algorithm Apriori

Figure 12.1 gives the Apriori algorithm. The first pass of the algorithm simply counts the number of occurrences of each item 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 12.2.1. Next, the database is scanned and the support of candidates in C_k is counted. The candidates in C_k that are contained in a given transaction t can be determined efficiently by using the *hash-tree*, described when we talk about the subset function momentarily.

Apriori Candidate Generation The apriori-gen function takes as an argument L_{k-1}, the set of all large $(k-1)$-itemsets. It returns a superset of the set of all large k-itemsets. First, in the *join* step, we join

L_{k-1} with L_{k-1} to obtain a superset of the final set of candidates C_k. The union $p \cup q$ of itemsets $p, q \in L_{k-1}$ is inserted in C_k if they share their $k-2$ first items:

1) **insert into** C_k
2) **select** $p[1], p[2], \ldots, p[k-1], q[k-1]$
3) **from** L_{k-1} p, L_{k-1} q
4) **where** $p[1] = q[1], \ldots, p[k-2] = q[k-2], p[k-1] < q[k-1]$;

Next, in the *prune* step, we delete all itemsets $c \in C_k$ such that some $(k-1)$-subset of c is not in L_{k-1}. To see why this generation procedure maintains completeness, note that for any itemset in L_k with minimum support, any subset of size k–1 must also have minimum support. Hence, if we 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}, we would be 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}. Thus at this stage, $C_k \supseteq L_k$. For the same reason, the pruning stage where we 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.

As an example, let L_3 be {{1 2 3}, {1 2 4}, {1 3 4}, {1 3 5}, {2 3 4}}. After the join step, C_4 will be {{1 2 3 4}, {1 3 4 5} }. The prune step will delete the itemset {1 3 4 5} because the itemset {1 4 5} is not in L_3. We will then be left with only {1 2 3 4} in C_4.

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.

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 we add an itemset c, we start from the root and go down the tree until we reach a leaf. At an interior node at depth d, we decide which branch to follow by applying a hash function to the dth item of the itemset, and following the pointer in the corresponding bucket. All nodes are initially created

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

Figure 12.2
Algorithm aprioriTid.

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 we are at a leaf, we find which of the itemsets in the leaf are contained in t and add references to them to the answer set. If we are at an interior node and we have reached it by hashing the item i, we 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, we 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, we ensure that we 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 we reach the current node by hashing the item i, we only need to consider the items in t that occur after i.

12.2.2 Algorithm AprioriTid

The AprioriTid algorithm, shown in Figure 12.2, also uses the apriori-

gen function (given in Section 12.2.1) to determine the candidate itemsets before the pass begins. The new feature of this algorithm is that the database $\mathcal{D}$ is not used for counting support after the first pass. Rather, the set $\widehat{C}_k$ is used for this purpose. Each member of the set $\widehat{C}_k$ 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 $k = 1$, $\widehat{C}_1$ corresponds to the database $\mathcal{D}$, although conceptually each item i is replaced by the itemset $\{i\}$. For $k > 1$, $\widehat{C}_k$ is generated by the algorithm (step 10). The member of $\widehat{C}_k$ corresponding to transaction t is $<t.TID, \{c \in C_k | c \text{ contained in } t\}>$. If a transaction does not contain any candidate k-itemset, then $\widehat{C}_k$ will not have an entry for this transaction. Thus, the number of entries in $\widehat{C}_k$ 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. We further explore this trade-off in Section 12.4.

Example Consider the database in Figure 12.3 and assume that the 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, we count the support of candidates in C_2 by iterating over the entries in $\widehat{C}_1$ and generate $\widehat{C}_2$. The first entry in $\widehat{C}_1$ 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} is a member of C_2 and both ({1 3}–{1}) and ({1 3}–{3}) are members of t.

Calling apriori-gen with L_2 gives C_3. Making a pass over the data with $\widehat{C}_2$ and C_3 generates $\widehat{C}_3$. Note that there is no entry in $\widehat{C}_3$ for the transactions with TIDs 100 and 400, since they do not contain any of the itemsets in C_3. The candidate {2 3 5} in C_3 turns out to be large and is the only member of L_3. When we generate C_4 using L_3, it turns out to be empty, and we terminate.

Database

TID	Items
100	1 3 4
200	2 3 5
300	1 2 3 5
400	2 5

$\widehat{C}_1$

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

L_1

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

C_2

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

$\widehat{C}_2$

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} }

L_2

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

C_3

Itemset	Support
{2 3 5}	2

$\widehat{C}_3$

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

L_3

Itemset	Support
{2 3 5}	2

Figure 12.3
Example.

12.3 Generating Rules

The association rules that we consider are somewhat more general than in (Agrawal *et al.* 1993) in that we allow a consequent to have more than one item. In this chapter we give an efficient generalization of the algorithm in (Agrawal *et al.* 1993).

For every large itemset l, we output all rules $a \Rightarrow (l - a)$, where a is a subset of l, such that the ratio support(l)/support(a) is at least *minconf*. The support of any subset $\tilde{a}$ of a must be as great as the support of a. Therefore, the confidence of the rule $\tilde{a} \Rightarrow (l - \tilde{a})$ 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 $\tilde{a}$. It follows that for a rule $(l - a) \Rightarrow a$ to hold, all rules of the form $(l - \tilde{a}) \Rightarrow \tilde{a}$ must also hold, where $\tilde{a}$ is a non-empty subset of a. For example, if the rule $AB \Rightarrow CD$ holds, then the rules $ABC \Rightarrow D$ and $ABD \Rightarrow C$ must also hold.

This characteristic is similar to the property that if an itemset is large

```
1)  forall large k-itemsets l_k, k ≥ 2 do begin
2)    H_1 = { consequents of rules from l_k with one item in the consequent };
3)    call ap-genrules(l_k, H_1);
4)  end

5)  procedure ap-genrules(l_k: large k-itemset, H_m: set of m-item consequents)
6)    if (k > m + 1) then begin
7)      H_{m+1} = apriori-gen(H_m);
8)      forall h_{m+1} ∈ H_{m+1} do begin
9)        conf = support(l_k)/support(l_k − h_{m+1});
10)       if (conf ≥ minconf) then
11)         output the rule (l_k − h_{m+1}) ⇒ h_{m+1}
                  with confidence = conf and support = support(l_k);
12)       else
13)         delete h_{m+1} from H_{m+1};
14)     end
15)     call ap-genrules(l_k, H_{m+1});
16)   end
```

Figure 12.4
Rule generation algorithm.

then so are all its subsets. From a large itemset l, therefore, we first generate all rules with one item in the consequent. We then use the consequents of these rules and the function apriori-gen in Section 12.2.1 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 in Figure 12.4.

12.4 Empirical Results

To assess the relative performance of the algorithms for discovering large itemsets, we performed several experiments. We first describe the synthetic datasets used in the performance evaluation. Then we show the performance results on synthetic data and discuss the trends in performance. We obtained similar results on real-life datasets (Agrawal and Srikant 1994; Mannila *et al.* 1994). Finally, we describe how the best performance features of Apriori and AprioriTid can be combined into an AprioriHybrid algorithm and demonstrate its scale-up properties.

The experiments were performed on an IBM RS/6000 530H workstation. To keep the comparison fair, we implemented all the algorithms using the same basic data structures.

12.4.1 Synthetic Data

To evaluate the performance of the algorithms over a large operating region, we developed synthetic transactions data. These transactions attempt to mimic the transactions in the retailing environment. Our model of the "real" world is that people tend to buy sets of items together. Each such set is potentially a maximal large itemset. An example of such a set might be sheets, pillow case, comforter, and ruffles. However, some people may buy only some of the items from such a set. A transaction may contain more than one large itemset. For example, a customer might place an order for a dress and jacket when ordering sheets and pillow cases, where the dress and jacket together form another large itemset. In our model, transaction sizes are small with respect to the total number of items, i.e. the data is sparse. The transaction sizes are typically clustered around a mean and a few transactions have many items. Typical sizes of large itemsets are also clustered around a mean, with a few large itemsets having a large number of items.

To create synthetic datasets, we used the following method. First we generated 2000 potentially large itemsets from 1000 items. We picked the size of a set from a Poisson distribution with mean equal to $|I|$ = 2, 4, or 6, and we randomly assigned items to the set. To model that large itemsets often have common items, some fraction of items in subsequent itemsets were chosen from the previous itemset generated. Each itemset has a weight associated with it, which corresponds to the probability that this itemset will be picked. This weight is picked from an exponential distribution with unit mean, and is then normalized so that the sum of the weights for all the itemsets is 1.

Then we generated $|\mathcal{D}|$ = 100,000 transactions. The average size $|T|$ of a transaction was 5, 10 or 20, and the size was picked from a Poisson distribution. Each transaction was assigned a series of fractions of potentially large itemsets, to model that all the items in a large itemset are not always bought together.

The number of transactions was to set to 100,000 because, as we will see in Section 12.4.2, SETM could not be run for larger values. However, for our scale-up experiments, we generated datasets with up to 10 million

Table 12.2
Parameters and sizes of datasets.

Name	$\|T\|$	$\|I\|$	$\|\mathcal{D}\|$	MB
T5.I2.D100K	5	2	100K	2.4
T10.I4.D100K	10	4	100K	4.4
T20.I4.D100K	20	4	100K	8.4
T20.I6.D100K	20	6	100K	8.4

Table 12.3
Execution times for T10.I4.D100K (sec).

Algorithm	Minimum Support (%)				
	2.0	1.5	1.0	0.75	0.5
SETM	41	91	659	929	1639
Apriori	3.8	4.8	11.2	17.4	19.3

transactions (838MB for $|T| = 20$). Table 12.2 summarizes the dataset parameter settings. A more detailed description of the synthetic data generation can be found in (Agrawal and Srikant 1994).

12.4.2 Experiments with Synthetic Data

Figure 12.5 shows the execution times for the four synthetic datasets given in Table 12.2 for decreasing values of minimum support. As the minimum support decreases, the execution times of all the algorithms increase because of increases in the total number of candidate and large itemsets.

Apriori outperforms AIS for all problem sizes, by factors ranging from 2 for high minimum support to more than an order of magnitude for low levels of support. For small problems, AprioriTid did about as well as Apriori, but its performance degraded to be about twice as slow for large problems.

For SETM, we have only plotted the execution times for the dataset T5.I2.D100K in Figure 12.5. The execution times for SETM for T10.I4. D100K are given in Table 12.3. We did not plot the execution times in Table 12.3 on the corresponding graphs because they are too large compared to the execution times of the other algorithms. For the two datasets with transaction sizes of 20, SETM took too long to execute and we aborted those runs as the trends were clear. Clearly, Apriori outperforms SETM by more than an order of magnitude for large datasets.

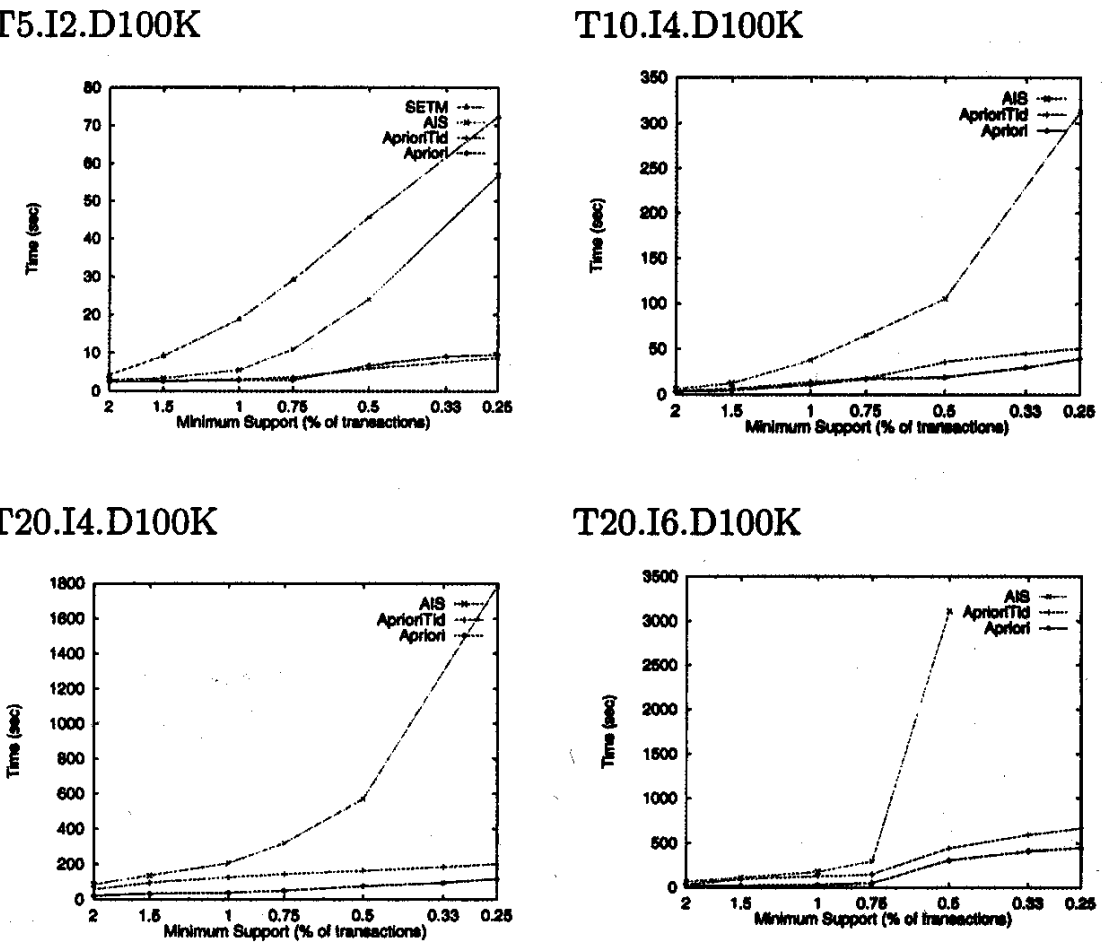

Figure 12.5
Execution times: synthetic data.

12.4.3 Explanation of the Relative Performance

To explain these performance trends, we show in Figure 12.6 the sizes of the large and candidate sets in different passes for the T10.I4.D100K dataset for the minimum support of 0.75%. L_k is the lower bound for all the curves. Note that the Y-axis in this graph has a log scale.

The problem with the SETM algorithm is the size of its $\widehat{C}_k$ sets. Recall that the size of the set $\widehat{C}_k$ is given by $\sum_{c \in C_k}$ support-count(c). Thus, the sets $\widehat{C}_k$ are roughly S times bigger than the corresponding C_k sets, where S is the average support count of the candidate itemsets. Unless the problem size is very small, the $\widehat{C}_k$ sets have to be written to disk, and externally sorted twice, causing the SETM algorithm to perform poorly. For datasets with more transactions, the performance gap between SETM and the other algorithms will become even larger.

The problem with AIS is that it generates too many candidates that

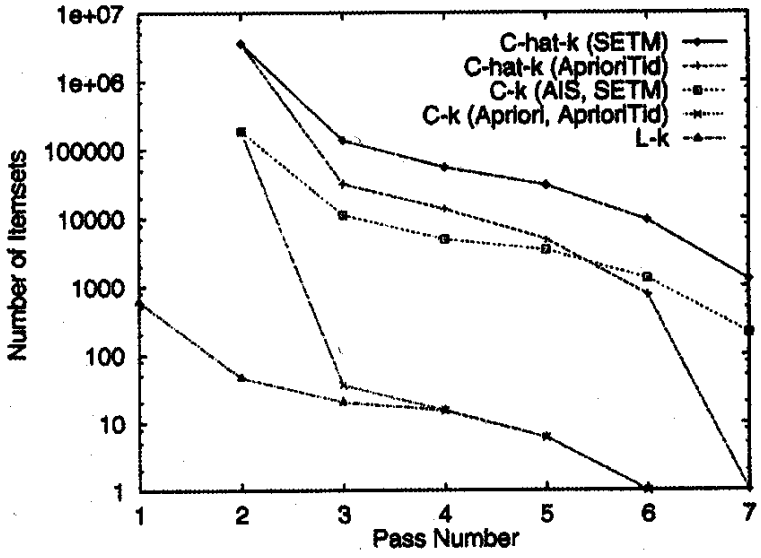

Figure 12.6
Sizes of the large and candidate sets (T10.I4.D100K, minsup = 0.75%).

later turn out to be small, causing it to waste too much effort. Apriori also counts too many small sets in the second pass (recall that C_2 is really a cross-product of L_1 with L_1). However, this wastage decreases dramatically from the third pass onward.

AprioriTid also has the problem of SETM that $\widehat{C}_k$ tends to be large. However, the apriori candidate generation used by AprioriTid generates significantly fewer candidates than the transaction-based candidate generation used by SETM. As a result, the $\widehat{C}_k$ of AprioriTid has fewer entries than that of SETM. In addition, unlike SETM, AprioriTid does not have to sort $\widehat{C}_k$. Thus, AprioriTid does not suffer as much as SETM from maintaining $\widehat{C}_k$.

AprioriTid has the nice feature that it replaces a pass over the original dataset by a pass over the set $\widehat{C}_k$. Hence, AprioriTid is very effective in later passes when the size of $\widehat{C}_k$ becomes small compared to the size of the database. Thus, we find that AprioriTid beats Apriori when its $\widehat{C}_k$ sets can fit in memory and the distribution of the large itemsets has a long tail. When $\widehat{C}_k$ doesn't fit in memory, there is a jump in the execution time for AprioriTid, such as when going from 0.75% to 0.5% for datasets with transaction size 10 in Figure 12.5. In this region, Apriori starts beating AprioriTid.

12.4.4 Algorithm AprioriHybrid

Based on the observations above, we can design a hybrid algorithm, which we call AprioriHybrid, that uses Apriori in the initial passes and switches to AprioriTid when it expects that the set $\widehat{C}_k$ at the end of the pass will fit in memory.

We use the following heuristic to estimate if $\widehat{C}_k$ would fit in memory in the next pass. At the end of the current pass, we have the counts of the candidates in C_k. From this, we estimate what the size of $\widehat{C}_k$ would have been if it had been generated. This size, in words, is ($\sum_{\text{candidates } c \in C_k} \text{support}(c)$ + number of transactions). If $\widehat{C}_k$ in this pass was small enough to fit in memory, and there were fewer large candidates in the current pass than the previous pass, we switch to AprioriTid.

We have run performance tests with the datasets described earlier, and AprioriHybrid performs better than Apriori and AprioriTid in almost all cases. AprioriHybrid does a little worse than Apriori when the pass in which the switch occurs is the last pass; AprioriHybrid thus incurs the cost of switching without realizing the benefits. AprioriHybrid did up to 30% better than Apriori, and up to 60% better than AprioriTid.

12.4.5 Scale-up Experiment

Figure 12.7 shows how AprioriHybrid scales up as the number of transactions is increased from 100,000 to 10 million transactions. We used the combinations (T5.I2), (T10.I4), and (T20.I6) for the average sizes of transactions and itemsets respectively. All other parameters were the same as for the data in Table 12.2. The sizes of these datasets for 10 million transactions were 239MB, 439MB and 838MB respectively. The minimum support level was set to 0.75%. The execution times are normalized with respect to the times for the 100,000 transaction datasets in the first graph and with respect to the 1 million transaction dataset in the second. As shown, the execution times scale quite linearly. Although we do not give the graphs, similar experiments showed that the Apriori algorithm also scales linearly.

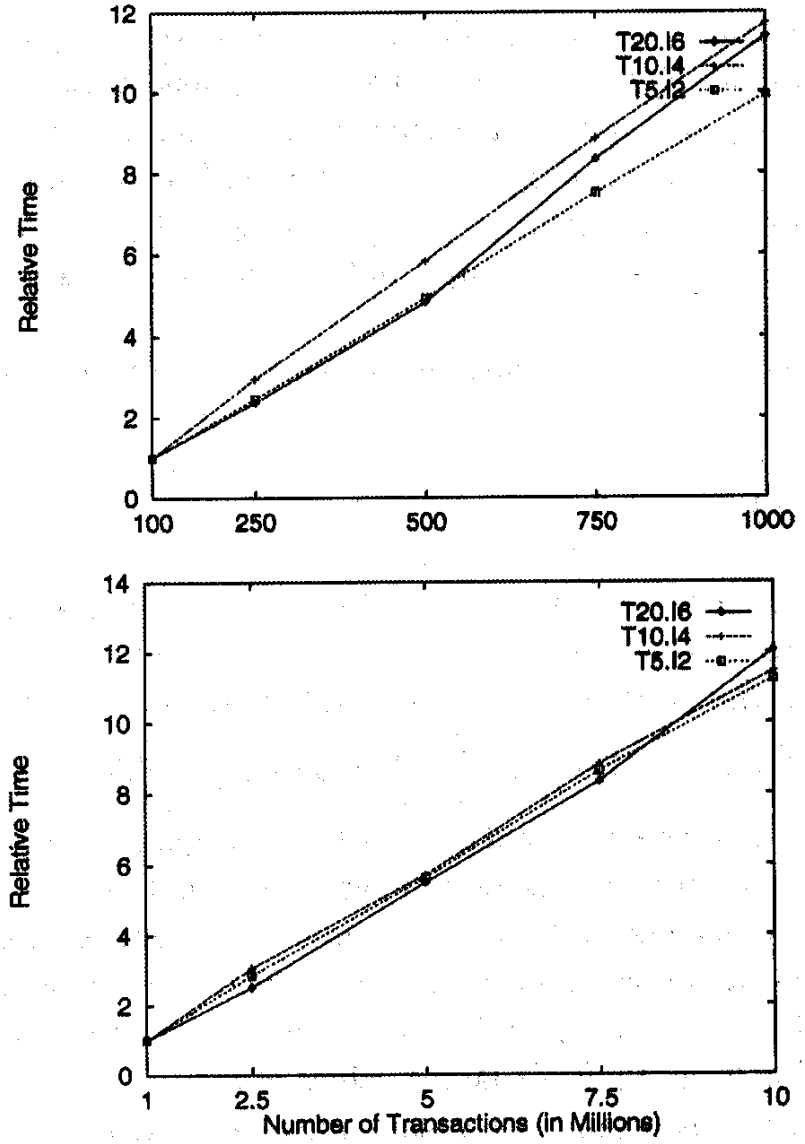

Figure 12.7
Number of transactions scale-up.

12.5 Theoretical Analyses

The algorithms presented in the previous sections perform quite well in practice. Their running time is bounded by $O(||C|| \cdot |\mathcal{D}|)$, where $||C||$ denotes the sum of the sizes of candidates considered and $|\mathcal{D}|$ denotes the size of the database.

12.5.1 A Lower Bound

The quantity $||C||$ can be exponential in the number of items, as all itemsets can be large. If there are only a few large sets, the above

algorithms still investigate several candidates. Next we show that this is to some degree inevitable. Namely, we now give an information-theoretic lower bound for finding one association rule in a restricted model of computation where the only way of getting information from a database $\mathcal{D}$ is by asking questions of the form "is the set X large." This model is realistic in the case the database $\mathcal{D}$ is large and stored using a database system.

Assume the database $\mathcal{D}$ has m items. In the worst case one needs at least

$$\log \binom{m}{k} \approx k \log(m/k)$$

questions of the form "is the set X large" to locate one maximal large set, where k is the size of the large set.

The proof of this claim is simple. Consider a database with exactly 1 maximal large set of size k. There are $\binom{m}{k}$ different possible answers to the problem of finding the maximal large set. Each question of the form "is the set X large" provides at most 1 bit of information.

This lower bound is not optimal for small values of k. For example, assume that there is exactly one large set of size 1. Then any algorithm for finding this set has to use at least $\Theta(m)$ queries of the above type. However, the bound is fairly tight for larger values of k.

Loveland (1987) has considered the problem of finding "critical sets." Given a function $f : \mathcal{P}(R) \to \{0, 1\}$ that is *upwards monotone* (i.e., if $f(Y) = 1$ and $Y \subseteq X$, then $f(X) = 1$), a set X is *critical* if $f(X) = 1$, but $f(Z) = 0$ for all subsets Z of X. Thus maximal large itemsets are complements of critical sets of the upwards monotone function $f(X) = 0$, if X is large, and $f(X) = 1$, otherwise. For example for $k = m/2$, the lower bound above matches exactly the upper bound provided by one of Loveland's algorithms.

12.5.2 Probabilistic Analysis of Random Databases

The number of large sets is an important factor influencing the running times of the algorithms. We now show that in one model of random databases all large itemsets have small size.

Consider a random database $\mathcal{D} = \{T_1, \ldots, T_n\}$ over items $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$; assume that each transaction T_k of the database contains any

item i_j with probability q, and assume that the entries are independent. Then the probability that T_k contains i_j for all i_j in a given itemset X is q^h, where $h = |X|$. The number x of such transactions has a binomial distribution with parameters n and q^h.

The Chernoff bounds (Alon and Spencer 1992; Hagerup and Rüb 1989/90) state that for all $a > 0$ we have

$$Pr[x > nq^h + a] < e^{-2a^2/n}.$$

We thus obtain

$$Pr[x > sn] = Pr[x > nq^h + n(s - q^h)] < e^{-2n(s-q^h)^2}$$

where n is the number of transactions $|\mathcal{D}|$ and s is the minimum support *minsup*. Thus the expected number of large itemsets of size h is bounded by $m^h e^{-2n(s-q^h)^2}$, where m is the number of items. This is less than 1 provided $s > \sqrt{(h \ln m)/n} + q^h$. For large databases and thus for large n the first term is very small. Hence if $s > q^h$, the expected number of large sets of size is small. (For $s = minsup = 0.01$ and $q = 0.1$, this means $h \geq 2$; for $s = 0.0001$ and $q = 0.1$, this means $h \geq 4$.) Thus a random database typically has only very few large itemsets. Of course, databases occurring in practice are not random.

12.5.3 Analysis of Sampling

The running times of the algorithms depended linearly on the size of the database. One possibility of lowering this factor is to use only a sample of transactions. We show that small samples are sometimes quite good for finding large itemsets.

Let s be the support of a given set X of items. Consider a random sample with replacement of size h from the database. Then the number of transactions in the sample that contain X is a random variable x with binomial distribution of h trials, each having success probability s. We can again use the Chernoff bounds. The probability that the estimated support is off by at least α is

$$Pr[x > h(s + \alpha)] < e^{-2\alpha^2 h^2/h} = e^{-2\alpha^2 h},$$

i.e., bounded by a quantity exponential in h.

Table 12.4 presents sufficient sample sizes, given values for α and probabilities of error more than α. For accuracy to be within support

Table 12.4
Sufficient sample sizes, given values for α and probabilities of error more than α.

$\alpha =$	1%	0.1%	0.01%	0.001%
$Pr[error > \alpha] \approx 1\%$	23000	$2.3 \cdot 10^6$	$2.3 \cdot 10^8$	$2.3 \cdot 10^{10}$
$Pr[error > \alpha] \approx 5\%$	15000	$1.5 \cdot 10^6$	$1.5 \cdot 10^8$	$1.5 \cdot 10^{10}$
$Pr[error > \alpha] \approx 10\%$	11500	$1.15 \cdot 10^6$	$1.15 \cdot 10^8$	$1.15 \cdot 10^{10}$

of an itemset $\pm$ 1%, samples of some dozens of thousands of examples can be sufficient. For association rules with support in fractions of a percent (where we would like accuracy to be within 0.01% or 0.001%), these bounds indicate that sampling is not effective. Note that the completeness guarantee of finding *all* the rules satisfying the minimum support and confidence constraints is lost when we use sampling.

12.6 Conclusions and Open Problems

Association rules are a simple and natural class of database regularities, useful in various analysis and prediction tasks. We have considered the problem of finding all the association rules satisfying user-specified support and confidence constraints that hold in a given database.

We presented two new algorithms, Apriori and AprioriTid, for discovering all significant association rules between items in a large database of transactions. We compared these algorithms to algorithms introduced in earlier work, the AIS (Agrawal *et al.* 1993) and SETM (Houtsma and Swami 1993) algorithms. We presented experimental results, using synthetic data, showing that the proposed algorithms always outperform AIS and SETM. We obtained similar results with real data (Agrawal and Srikant 1994; Mannila *et al.* 1994). 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.

We showed how the best features of the two proposed algorithms can be combined into a hybrid algorithm, called AprioriHybrid, which then becomes the algorithm of choice for this problem. The scale-up properties of AprioriHybrid and Apriori demonstrate the feasibility of using these algorithms in real applications involving very large databases. However, the implementation of AprioriHybrid is more complex than Apriori. Hence the somewhat worse performance of Apriori may be an acceptable tradeoff in some situations. We have also analyzed the theo-

retical properties of the problem of finding association rules.

Several problems remain open and are subject to further research.

- 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. We would like to be able to find association rules that use such hierarchies.
- We did not consider the quantities or values of the items bought in a transaction, which are important for some applications. Finding such rules needs further work.
- We may be interested in only those rules in which certain items appear in the consequent and/or antecedent. Pushing constraints on antecedents into the computation is quite straightforward, but the exploitation of constraints on the consequent is an interesting problem. More generally, the question is about using the user input or domain knowledge to improve the execution efficiency of the mining process.

References

Agrawal, R., Imielinski, T., and Swami, A. 1993. Mining Association Rules between Sets of Items in Large Databases. In *Proceedings, ACM SIGMOD Conference on Management of Data,* 207–216. Washington, D.C.

Agrawal, R., and Srikant, R. 1994. Fast Algorithms for Mining Association Rules. IBM Research Report RJ9839, June 1994, IBM Almaden Research Center, San Jose, Calif..

Alon, N.; and Spencer, J. H. 1992. *The Probabilistic Method.* New York: John Wiley Inc.

Brachman, R., et al. 1993. Integrated Support for Data Archeology. Presented at the AAAI Workshop on Knowledge Discovery in Databases, Washington, D.C.

Breiman, L.; Friedman, J. H.; Olshen, R. A.; and Stone, C. J. 1984. *Classification and Regression Trees.* Belmont, Calif.: Wadsworth.

Cheeseman, P.; Kelly, J.; Self, M.; Stutz, J.; Taylor, W.; and Freeman, D. 1988. AutoClass: A Bayesian Classification System. In *Proceedings, Fifth International Conference on Machine Learning,* 54–64. San Mateo, Calif.: Morgan Kaufmann.

Fisher, D. H. 1987. Knowledge Acquisition Via Incremental Conceptual Clustering. *Machine Learning* 2(2): 139–172.

Hagerup, T., and Rüb, C. 1989/90. A Guided Tour of Chernoff Bounds. *Information Processing Letters* 33: 305–308.

Houtsma, M.; and Swami, A. 1993. Set-Oriented Mining of Association Rules. Research Report RJ 9567, Oct. 1993, IBM Almaden Research Center, San Jose, Calif.

Klemettinen, M., Mannila, H., Ronkainen, P., Toivonen, H., and Verkamo, A. I. 1994. Finding Interesting Rules from Large Sets of Discovered Association Rules. In *CIKM'94: Conference on Information and Knowledge Management,* 401–407. Gaithersburg, Md.

Langley, P.; Simon, H.; Bradshaw, G.; and Zytkow, J. 1987. *Scientific Discovery: Computational Explorations of the Creative Process.* Cambridge, Mass.: The MIT Press.

Loveland, D. W. 1987. Finding Critical Sets. *Journal of Algorithms* 8: 362–371.

Mannila, H., and Räihä, K.-J. 1992. On the Complexity of Inferring Functional Dependencies. *Discrete Applied Mathematics* 40: 237–243.

Mannila, H., and Räihä, K.-J. 1994. Algorithms for Inferring Functional Dependencies from Relations. *Data & Knowledge Engineering* 12(1): 83–99.

Mannila, H., Toivonen, H., and Verkamo, A. I. 1994. Efficient Algorithms for Discovering Association Rules. In Knowledge Discovery in Databases, Tech. Report WS-94-03, American Association for Artificial Intelligence, Menlo Park, Calif.

Muggleton, S., and Feng, C. 1992. Efficient Induction of Logic Programs. In *Inductive Logic Programming,* 281–298, ed. S. Muggleton. London: Academic Press.

Piatetsky-Shapiro, G. 1991. Discovery, Analysis, and Presentation of Strong Rules. In *Knowledge Discovery in Databases,* 229–248, ed. G. Piatetsky-Shapiro and W. Frawley. Menlo Park, Calif.: AAAI Press.

Piatetsky-Shapiro, G., and Frawley, W., eds, 1991. *Knowledge Discovery in Databases.* Menlo Park, Calif.: AAAI Press.

Piatetsky-Shapiro, G., and Matheus, C. J. 1994. The Interestingness of Deviations. Presented at the AAAI Workshop on Knowledge Discovery in Databases, Seattle, Wash.

Quinlan, J. R. 1993. *C4.5: Programs for Machine Learning.* San Mateo, Calif.: Morgan Kaufmann.

Spiegelhalter, D.; Dawid, A.; Lauritzen, S.; and Cowell, R. 1993. Bayesian Analysis in Expert Systems. *Statistical Science* 8(3): 219–283.