Mining Association Rules between Sets of Items in Large Databases

Rakesh Agrawal Tomasz Imielinski* Arun Swami

IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120

Abstract

We are given a large database of customer transactions. Each transaction consists of items purchased by a customer in a visit. We present an efficient algorithm that generates all significant association rules between items in the database. The algorithm incorporates buffer management and novel estimation and pruning techniques. We also present results of applying this algorithm to sales data obtained from a large retailing company, which shows the effectiveness of the algorithm.

1 Introduction

Consider a supermarket with a large collection of items. Typical business decisions that the management of the supermarket has to make include what to put on sale, how to design coupons, how to place merchandise on shelves in order to maximize the profit, etc. Analysis of past transaction data is a commonly used approach in order to improve the quality of such decisions. Until recently, however, only global data about the cumulative sales during some time period (a day, a week, a month, etc.) was available on the computer. Progress in bar-code technology has made it possible to store the so called *basket* data that stores items purchased on a per-transaction basis. Basket data type transactions do not necessarily consist of items bought together at the same point of time. It may consist of items bought by a customer over a period of time. Examples include monthly purchases by members of a book club or a music club.

*Current address: Computer Science Department, Rutgers University, New Brunswick, NJ 08903

Proceedings of the 1993 ACM SIGMOD Conference
Washington DC, USA, May 1993

Several organizations have collected massive amounts of such data. These data sets are usually stored on tertiary storage and are very slowly migrating to database systems. One of the main reasons for the limited success of database systems in this area is that current database systems do not provide necessary functionality for a user interested in taking advantage of this information.

This paper introduces the problem of "mining" a large collection of basket data type transactions for association rules between sets of items with some minimum specified confidence, and presents an efficient algorithm for this purpose. An example of such an association rule is the statement that 90% of transactions that purchase bread and butter also purchase milk. The antecedent of this rule consists of bread and butter and the consequent consists of milk alone. The number 90% is the confidence factor of the rule.

The work reported in this paper could be viewed as a step towards enhancing databases with functionalities to process queries such as (we have omitted the confidence factor specification):

- Find all rules that have "Diet Coke" as consequent. These rules may help plan what the store should do to boost the sale of Diet Coke.
- Find all rules that have "bagels" in the antecedent. These rules may help determine what products may be impacted if the store discontinues selling bagels.
- Find all rules that have "sausage" in the antecedent and "mustard" in the consequent. This query can be phrased alternatively as a request for the additional items that have to be sold together with sausage in order to make it highly likely that mustard will also be sold.
- Find all the rules relating items located on shelves A and B in the store. These rules may help shelf planning by determining if the sale of items on shelf A is related to the sale of items on shelf B.

- Find the "best" k rules that have "bagels" in the consequent. Here, "best" can be formulated in terms of the confidence factors of the rules, or in terms of their support, i.e., the fraction of transactions satisfying the rule.

The organization of the rest of the paper is as follows. In Section 2, we give a formal statement of the problem. In Section 3, we present our algorithm for mining association rules. In Section 4, we present some performance results showing the effectiveness of our algorithm, based on applying this algorithm to data from a large retailing company. In Section 5, we discuss related work. In particular, we put our work in context with the rule discovery work in AI. We conclude with a summary in Section 6.

2 Formal Model

Let $\mathcal{I} = I_1, I_2, \ldots, I_m$ be a set of binary attributes, called items. Let T be a database of transactions. Each transaction t is represented as a binary vector, with $t[k] = 1$ if t bought the item I_k, and $t[k] = 0$ otherwise. There is one tuple in the database for each transaction. Let X be a set of some items in $\mathcal{I}$. We say that a transaction t *satisfies* X if for all items I_k in X, $t[k] = 1$.

By an *association rule*, we mean an implication of the form $X \Longrightarrow I_j$, where X is a set of some items in $\mathcal{I}$, and I_j is a single item in $\mathcal{I}$ that is not present in X. The rule $X \Longrightarrow I_j$ is satisfied in the set of transactions T with the confidence factor $0 \leq c \leq 1$ iff at least $c\%$ of transactions in T that satisfy X also satisfy I_j. We will use the notation $X \Longrightarrow I_j \mid c$ to specify that the rule $X \Longrightarrow I_j$ has a confidence factor of c.

Given the set of transactions T, we are interested in generating all rules that satisfy certain additional constraints of two different forms:

1. *Syntactic Constraints*: These constraints involve restrictions on items that can appear in a rule. For example, we may be interested only in rules that have a specific item I_x appearing in the consequent, or rules that have a specific item I_y appearing in the antecedent. Combinations of the above constraints are also possible — we may request all rules that have items from some predefined itemset X appearing in the consequent, and items from some other itemset Y appearing in the antecedent.

2. *Support Constraints*: These constraints concern the number of transactions in T that support a rule. The *support* for a rule is defined to be the fraction of transactions in T that satisfy the union of items in the consequent and antecedent of the rule.
   Support should not be confused with confidence. While confidence is a measure of the rule's strength, support corresponds to statistical significance.
   Besides statistical significance, another motivation for support constraints comes from the fact that we are usually interested only in rules with support above some minimum threshold for business reasons. If the support is not large enough, it means that the rule is not worth consideration or that it is simply less preferred (may be considered later).

In this formulation, the problem of rule mining can be decomposed into two subproblems:

1. Generate all combinations of items that have fractional transaction support above a certain threshold, called *minsupport*. Call those combinations *large* itemsets, and all other combinations that do not meet the threshold *small* itemsets.
   Syntactic constraints further constrain the admissible combinations. For example, if only rules involving an item I_x in the antecedent are of interest, then it is sufficient to generate only those combinations that contain I_x.

2. For a given *large* itemset $Y = I_1 I_2 \ldots I_k$, $k \geq 2$, generate all rules (at the most k rules) that use items from the set $I_1, I_2, \ldots, I_k$. The antecedent of each of these rules will be a subset X of Y such that X has $k-1$ items, and the consequent will be the item $Y - X$. To generate a rule $X \Longrightarrow I_j \mid c$, where $X = I_1 I_2 \ldots I_{j-1} I_{j+1} \ldots I_k$, take the support of Y and divide it by the support of X. If the ratio is greater than c then the rule is satisfied with the confidence factor c; otherwise it is not.
   Note that if the itemset Y is large, then every subset of Y will also be large, and we must have available their support counts as the result of the solution of the first subproblem. Also, all rules derived from Y must satisfy the support constraint because Y satisfies the support constraint and Y is the union of items in the consequent and antecedent of every such rule.

Having determined the large itemsets, the solution to the second subproblem is rather straightforward. In the next section, we focus on the first subproblem. We develop an algorithm that generates all subsets of a given set of items that satisfy transactional support requirement. To do this task efficiently, we use some estimation tools and some pruning techniques.

3 Discovering large itemsets

Figure 1 shows the template algorithm for finding *large* itemsets. Given a set of items $\mathcal{I}$, an itemset $X + Y$ of items in $\mathcal{I}$ is said to be an extension of the itemset X if $X \cap Y = \emptyset$. The parameter *dbsize* is the total number of tuples in the database.

The algorithm makes multiple passes over the database. The *frontier set* for a pass consists of those itemsets that are extended during the pass. In each pass, the support for certain itemsets is measured. These itemsets, called *candidate itemsets*, are derived from the tuples in the database and the itemsets contained in the frontier set.

Associated with each itemset is a counter that stores the number of transactions in which the corresponding itemset has appeared. This counter is initialized to zero when an itemset is created.

```
procedure LargeItemsets
begin
 let Large set L = ∅;
 let Frontier set F = {∅};

 while F ≠ ∅ do begin

  -- make a pass over the database
  let Candidate set C = ∅;
  forall database tuples t do
   forall itemsets f in F do
    if t contains f then begin
     let C_f = candidate itemsets that are extensions
          of f and contained in t; -- see Section 3.2
     forall itemsets c_f in C_f do
      if c_f ∈ C then
       c_f.count = c_f.count + 1;
      else begin
       c_f.count = 0;
       C = C + c_f;
      end
    end

  -- consolidate
  let F = ∅;
  forall itemsets c in C do begin
   if count(c)/dbsize > minsupport then
    L = L + c;
   if c should be used as a frontier -- see Section 3.3
    in the next pass then
     F = F + c;
  end
 end
end
```

Figure 1: Template algorithm

Initially the frontier set consists of only one element, which is an empty set. At the end of a pass, the support for a candidate itemset is compared with *minsupport* to determine if it is a *large* itemset. At the same time, it is determined if this itemset should be added to the frontier set for the next pass. The algorithm terminates when the frontier set becomes empty. The support count for the itemset is preserved when an itemset is added to the large/frontier set.

We did not specify in the template algorithm what candidate itemsets are measured in a pass and what candidate itemsets become a frontier for the next pass. These topics are covered next.

3.1 Number of passes versus measurement wastage

In the most straightforward version of the algorithm, every itemset present in any of the tuples will be measured in one pass, terminating the algorithm in one pass. In the worst case, this approach will require setting up 2^m counters corresponding to all subsets of the set of items $\mathcal{I}$, where m is number of items in $\mathcal{I}$. This is, of course, not only infeasible (m can easily be more than 1000 in a supermarket setting) but also unnecessary. Indeed, most likely there will very few large itemsets containing more than l items, where l is small. Hence, a lot of those 2^m combinations will turn out to be small anyway.

A better approach is to measure in the kth pass only those itemsets that contain exactly k items. Having measured some itemsets in the kth pass, we need to measure in $(k+1)$th pass only those itemsets that are 1-extensions (an itemset extended by exactly one item) of large itemsets found in the kth pass. If an itemset is small, its 1-extension is also going to be small. Thus, the frontier set for the next pass is set to candidate itemsets determined large in the current pass, and only 1-extensions of a frontier itemset are generated and measured during a pass.[1] This alternative represents another extreme — we will make too many passes over the database.

These two extreme approaches illustrate the tradeoff between number of passes and wasted effort due to measuring itemsets that turn out to be small. Certain measurement wastage is unavoidable — if the itemset A is large, we must measure AB to determine if it is large or small. However, having determined AB to be small, it is unnecessary to measure ABC, ABD, $ABCD$, etc. Thus, aside from practical feasibility, if we measure a large number of candidate itemsets in a pass, many of them may turn out to be small anyhow —

[1] A generalization of this approach will be to measure all up to g-extensions ($g > 0$) of frontier itemsets in a pass. The frontier set for the next pass will then consist of only those large candidate itemsets that are *precisely* g-extensions. This generalization reduces the number of passes but may result in some itemsets being unnecessarily measured.

wasted effort. On the other hand, if we measure a small number of candidates and many of them turn out to be large then we need another pass, which may have not been necessary. Hence, we need some careful estimation before deciding whether a candidate itemset should be measured in a given pass.

3.2 Determination of candidate itemsets

One may think that we should measure in the current pass only those extensions of frontier itemsets that are *expected* to be large. However, if it were the case and the data behaved according to our expectations and the itemsets expected to be large indeed turn out to be large, then we would still need another pass over the database to determine the support of the extensions of those large itemsets. To avoid this situation, in addition to those extensions of frontier itemsets that are expected to be large, we also measure the extensions $X + I_j$ that are expected to be small but such that X is expected to be large and X contains a frontier itemset. We do not, however, measure any further extensions of such itemsets. The rationale for this choice is that if our predictions are correct and $X + I_j$ indeed turns out to be small then no superset of $X + I_j$ has to be measured. The additional pass is then needed only if the data does not behave according to our expectation and $X + I_j$ turns out to be large. This is the reason why not measuring $X + I_j$ that are expected to be small would be a mistake — since even when the data agrees with predictions, an extra pass over the database would be necessary.

Expected support for an itemset

We use the statistical independence assumption to estimate the support for an itemset. Suppose that a candidate itemset $X + Y$ is a k-extension of the frontier itemset X and that $Y = I_1 I_2 \ldots I_k$. Suppose that the itemset X appears in a total of x tuples. We know the value of x since X was measured in the previous pass (x is taken to be *dbsize* for the empty frontier itemset). Suppose that $X + Y$ is being considered as a candidate itemset for the first time after c tuples containing X have already been processed in the current pass. Denoting by $f(I_j)$ the relative frequency of the item I_j in the database, the expected support $\bar{s}$ for the itemset $X + Y$ is given by

$$\bar{s} \;=\; f(I_1) \times f(I_2) \times \ldots \times f(I_k) \times (x - c)/dbsize$$

Note that $(x - c)/dbsize$ is the *actual* support for X in the remaining portion of the database. Under statistical independence assumption, the *expected* support for $X + Y$ is a product of the support for X and individual relative frequencies of items in Y.

If $\bar{s}$ is less than *minsupport*, then we say that $X + Y$ is expected to be small; otherwise, it is expected to be large.

Candidate itemset generation procedure

An itemset not present in any of the tuples in the database never becomes a candidate for measurement. We read one tuple at a time from the database and check what frontier sets are contained in the tuple read. Candidate itemsets are generated from these frontier itemset by extending them recursively with other items present in the tuple. An itemset that is expected to be small is not further extended. In order not to replicate different ways of constructing the same itemset, items are ordered and an itemset X is tried for extension only by items that are later in the ordering than any of the members of X. Figure 2 shows how candidate itemsets are generated, given a frontier itemset and a database tuple.

procedure Extend(X: itemset, t: tuple)
begin
  let item I_j be such that $\forall I_l \in X,\ I_j \geq I_l$;
  forall items I_k in the tuple t such that $I_k > I_j$ **do begin**
    output(XI_k);
    if (XI_k) is expected to be large **then**
      Extend(XI_k, t);
  end
end

Figure 2: Extension of a frontier itemset

For example, let $\mathcal{I} = \{A, B, C, D, E, F\}$ and assume that the items are ordered in alphabetic order. Further assume that the frontier set contains only one itemset, AB. For the database tuple $t = ABCDF$, the following candidate itemsets are generated:

ABC	expected large: continue extending
$ABCD$	expected small: do not extend any further
$ABCF$	expected large: cannot be extended further
ABD	expected small: do not extend any further
ABF	expected large: cannot be extended further

The extension $ABCDF$ was not considered because $ABCD$ was expected to be small. Similarly, $ABDF$ was not considered because ABD was expected to be small. The itemsets $ABCF$ and ABF, although expected to be large, could not be extended further because there is no item in t which is greater than F. The extensions $ABCE$ and ABE were not considered because the item E is not in t.

3.3 Determination of the frontier set

Deciding what itemsets to put in the next frontier set turns out to be somewhat tricky. One may think that it is sufficient to select just maximal (in terms of set inclusion) large itemsets. This choice, however, is incorrect — it may result in the algorithm missing some large itemsets as the following example illustrates:

Suppose that we extended the frontier set AB as shown in the example in previous subsection. However, both ABD and $ABCD$ turned out to be large at the end of the pass. Then ABD as a non-maximal large itemset would not make it to the frontier — a mistake, since we will not consider $ABDF$, which could be large, and we lose completeness.

We include in the frontier set for the next pass those candidate itemsets that were expected to be small but turned out to be large in the current pass. To see that these are the only itemsets we need to include in the next frontier set, we first state the following lemma:

Lemma. If the candidate itemset X is expected to be small in the current pass over the database, then no extension $X + I_j$ of X, where $I_j > I_k$ for any I_k in X is a candidate itemset in this pass.

The lemma holds due to the candidate itemset generation procedure.

Consequently, we know that no extensions of the itemsets we are including in the next frontier set have been considered in the current pass. But since these itemsets are actually large, they may still produce extensions that are large. Therefore, these itemsets must be included in the frontier set for the next pass. They do not lead to any redundancy because none of their extensions has been measured so far. Additionally, we are also complete. Indeed, if a candidate itemset was large but it was not expected to be small then it should not be in the frontier set for the next pass because, by the way the algorithm is defined, all extensions of such an itemset have already been considered in this pass. A candidate itemset that is small must not be included in the next frontier set because the support for an extension of an itemset cannot be more than the support for the itemset.

3.4 Memory Management

We now discuss enhancements to handle the fact that we may not have enough memory to store all the frontier and candidate itemsets in a pass. The large itemsets need not be in memory during a pass over the database and can be disk-resident. We assume that we have enough memory to store any itemset and all its 1-extensions.

Given a tuple and a frontier itemset X, we generate candidate itemsets by extending X as before. However, it may so happen that we run out of memory when we are ready to generate the extension $X+Y$. We will now have to create space in memory for this extension.

```
procedure ReclaimMemory
begin
 -- first obtain memory from the frontier set

  while enough memory has not been reclaimed do
    if there is an itemset X in the frontier set
      for which no extension has been generated then
        move X to disk;
    else
      break;

  if enough memory has been reclaimed then return;

  -- now obtain memory by deleting some
  -- candidate itemsets

  find the candidate itemset U
      having maximum number of items;
  discard U and all its siblings;

  let Z = parent(U);

  if Z is in the frontier set then
    move Z to disk;
  else
    disable future extensions of Z in this pass;
end
```

Figure 3: Memory reclamation algorithm

Figure 3 shows the memory reclamation algorithm. Z is said to be the *parent* of U if U has been generated by extending the frontier set Z. If U and V are 1-extensions of the same itemset, then U and V are called *siblings*.

First an attempt is made to make room for the new itemset by writing to disk those frontier itemsets that have not yet been extended. Failing this attempt, we discard the candidate itemset having maximum number of items. All its siblings are also discarded. The reason is that the parent of this itemset will have to be included in the frontier set for the next pass. Thus, the siblings will anyway be generated in the next pass. We may avoid building counts for them in the next pass, but the elaborate book-keeping required will be very expensive. For the same reason, we disable future extensions of the parent itemset in this pass. However, if the parent is a candidate itemset, it continues to be measured. On the other hand, if the parent is a frontier itemset, it is written out to disk creating more memory space.

It is possible that the current itemset that caused the memory shortage is the one having maximum number of items. In that case, if a candidate itemset needs to be deleted, the current itemset and its siblings are the ones that are deleted. Otherwise, some other candidate itemset has more items, and this itemset and its siblings are deleted. In both the cases, the memory reclamation algorithm succeeds in releasing sufficient memory.

In addition to the candidate itemsets that were expected to be small but turn out to be large, the frontier set for the next pass now additionally includes the following:

- disk-resident frontier itemsets that were not extended in the current pass, and
- those itemsets (both candidate and frontier) whose children were deleted to reclaim memory.

If a frontier set is too large to fit in the memory, we start a pass by putting as many frontiers as can fit in memory (or some fraction of it).

It can be shown that if there is enough memory to store one frontier itemset and to measure all of its 1-extensions in a pass, then there is guaranteed to be forward progress and the algorithm will terminate.

3.5 Pruning based on the count of remaining tuples in the pass

It is possible during a pass to determine that a candidate itemset will eventually not turn out to be large, and hence discard it early. This pruning saves both memory and measurement effort. We refer to this pruning as the *remaining tuples optimization.*

Suppose that a candidate itemset $X + Y$ is an extension of the frontier itemset X and that the itemset X appears in a total of x tuples (as discussed in Section 3.2, x is always known). Suppose that $X + Y$ is present in the cth tuple containing X. At the time of processing this tuple, let the count of tuples (including this tuple) containing $X + Y$ be s.

What it means is that we are left with at most $x - c$ tuples in which $X + Y$ may appear. So we compare *maxcount* $=(x - c + s)$ with *minsupport* $\times$ *dbsize.* If *maxcount* is smaller, then $X + Y$ is bound to be small and can be pruned right away.

The remaining tuples optimization is applied as soon as a "new" candidate itemset is generated, and it may result in immediate pruning of some of these itemsets. It is possible that a candidate itemset is not initially pruned, but it may satisfy the pruning condition after some more tuples have been processed. To prune such "old" candidate itemsets, we apply the pruning test whenever a tuple containing such an itemset is processed and we are about to increment the support count for this itemset.

3.6 Pruning based on synthesized pruning functions

We now consider another technique that can prune a candidate itemset as soon as it is generated. We refer to this pruning as the *pruning function optimization.*

The pruning function optimization is motivated by such possible pruning functions as *total transaction price.* Total transaction price is a cumulative function that can be associated with a set of items as a sum of prices of individual items in the set. If we know that there are less than *minsupport* fraction of transactions that bought more than τ dollars worth of items, we can immediately eliminate all sets of items for which their total price exceeds τ. Such itemsets do not have to be measured and included in the set of candidate itemsets.

In general, we do not know what these pruning functions are. We, therefore, synthesize pruning functions from the available data. The pruning functions we synthesize are of the form

$$w_1 I_{j_1} \ + \ w_2 I_{j_2} \ + \ \ldots \ + \ w_m I_{j_m} \ \leq \ \tau$$

where each binary valued $I_{j_i} \in \mathcal{I}$. Weights w_i are selected as follows. We first order individual items in decreasing order of their frequency of occurrence in the database. Then the weight of the ith item I_{j_i} in this order

$$w_i \ = \ 2^{i-1}\epsilon$$

where ϵ is a small real number such as 0.000001. It can be shown that under certain mild assumptions[2] a pruning function with the above weights will have optimal pruning value — it will prune the largest number of candidate itemsets.

A separate pruning function is synthesized for each frontier itemset. These functions differ in their values for τ. Since the transaction support for the item XY cannot be more than the support for itemset X, the pruning function associated with the frontier set X can be used to determine whether an expansion of X should be added to the candidate itemset or whether it should be pruned right away. Let $z(t)$ represent the value of the expression

$$w_1 I_{j_1} \ + \ w_2 I_{j_2} \ + \ \ldots \ + \ w_m I_{j_m}$$

for tuple t. Given a frontier itemset X, we need a procedure for establishing τ_X such that *count*(t | tuple t contains X and $z(t) > \tau_X$) $<$ *minsupport.*

Having determined frontier itemsets in a pass, we do not want to make a separate pass over the data just to determine the pruning functions. We should collect

[2] For every item pair I_j and I_k in $\mathcal{I}$, if frequency(I_j) $<$ frequency(I_k), then for every itemset X comprising items in $\mathcal{I}$, it holds that frequency($I_j X$) $<$ frequency($I_k X$).

information for determining τ for an itemset X while X is still a candidate itemset and is being measured in anticipation that X may become a frontier itemset in the next pass. Fortunately, we know that only the candidate itemsets that are expected to be small are the ones that can become a frontier set. We need to collect τ information only for these itemsets and not all candidate itemsets.

A straightforward procedure for determining τ for an itemset X will be to maintain *minsupport* number of largest values of z for tuples containing X. This information can be collected at the same time as the support count for X is being measured in a pass. This procedure will require memory for maintaining *minsupport* number of values with each candidate itemset that is expected to be small. It is possible to save memory at the cost of losing some precision (i.e., establishing a somewhat larger value for τ). Our implementation uses this memory saving technique, but we do not discuss it here due to space constraints.

Finally, recall that, as discussed in Section 3.4, when memory is limited, a candidate itemset whose children are deleted in the current pass also becomes a frontier itemset. In general, children of a candidate itemset are deleted in the middle of a pass, and we might not have been collecting τ information for such an itemset. Such itemsets inherit τ value from their parents when they become frontier.

4 Experiments

We experimented with the rule mining algorithm using the sales data obtained from a large retailing company. There are a total of 46,873 customer transactions in this data. Each transaction contains the department numbers from which a customer bought an item in a visit. There are a total of 63 departments. The algorithm finds if there is an association between departments in the customer purchasing behavior.

The following rules were found for a minimum support of 1% and minimum confidence of 50%. Rules have been written in the form $X \Longrightarrow I|(c,s)$, where c is the confidence and s is the support expressed as a percentage.

[Tires] ⇒ [Automotive Services] (98.80, 5.79)
[Auto Accessories], [Tires] ⇒
    [Automotive Services] (98.29, 1.47)
[Auto Accessories] ⇒ [Automotive Services] (79.51, 11.81)
[Automotive Services] ⇒ [Auto Accessories] (71.60, 11.81)
[Home Laundry Appliances] ⇒
    [Maintenance Agreement Sales] (66.55, 1.25)
[Children's Hardlines] ⇒
    [Infants and Children's wear] (66.15, 4.24)
[Men's Furnishing] ⇒ [Men's Sportswear] (54.86, 5.21)

In the worst case, this problem is an exponential problem. Consider a database of m items in which every item appears in every transaction. In this case, there will be 2^m large itemsets. To give an idea of the running time of the algorithm on actual data, we give below the timings on an IBM RS-6000/530H workstation for finding the above rules:

real 2m53.62s
user 2m49.55s
sys 0m0.54s

We also conducted some experiments to asses the effectiveness of the estimation and pruning techniques, using the same sales data. We report the results of these experiments next.

4.1 Effectiveness of the estimation procedure

We measure in a pass those itemsets X that are expected to be large. In addition, we also measure itemsets $Y = X + I_j$ that are expected to be small but such that X is large. We rely on the estimation procedure given in Section 3.2 to determine what these itemsets X and Y are. If we have a good estimation procedure, most of the itemsets expected to be large (small) will indeed turn out to be large (small).

We define the accuracy of the estimation procedure for large (small) itemsets to be the ratio of the number of itemsets that actually turn out to be large (small) to the number of itemsets that were estimated to be large (small). We would like the estimation accuracy to be close to 100%. Small values for estimation accuracy for large itemsets indicate that we are measuring too many unnecessary itemsets in a pass — wasted measurement effort. Small values for estimation accuracy for small itemsets indicate that we are stopping too early in our candidate generation procedure and we are not measuring all the itemsets that we should in a pass — possible extra passes over the data.

Figure 4 shows the estimation accuracy for large and small itemsets for different values of *minsupport*. In this experiment, we had turned off both remaining tuple and pruning function optimizations to isolate the effect of the estimation procedure. The graph shows that our estimation procedure works quite well, and the algorithm neither measures too much nor too little in a pass.

Note that the accuracy of the estimation procedure will be higher when the data behaves according to the expectation of statistical independence. In other words, if the data is "boring", not many itemsets that were expected to be small will turn out to be large, and the algorithm will terminate in a few passes. On the other hand, the more "surprising" the data is, the lower will be the estimation accuracy and the more passes it will take our algorithm to terminate. This behavior seems to be quite reasonable — waiting longer "pays off" in the form of unexpected new rules.

We repeated the above experiment with both the remaining tuple and pruning function optimizations turned on. The accuracy figures were somewhat better, but closely tracked the curves in Figure 4.

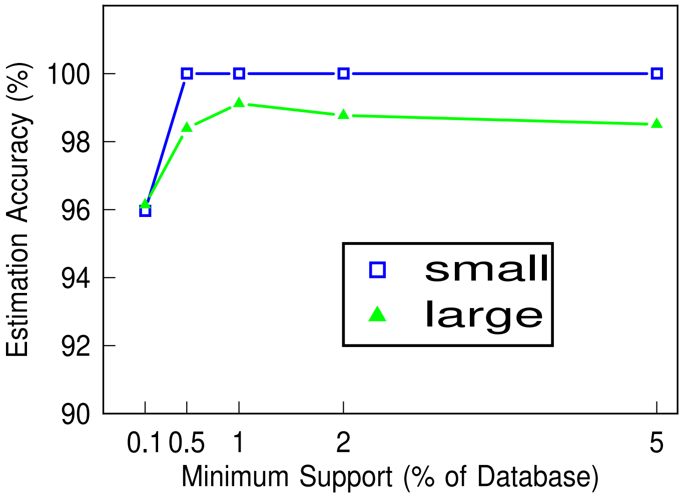

Figure 4: Accuracy of the estimation procedure

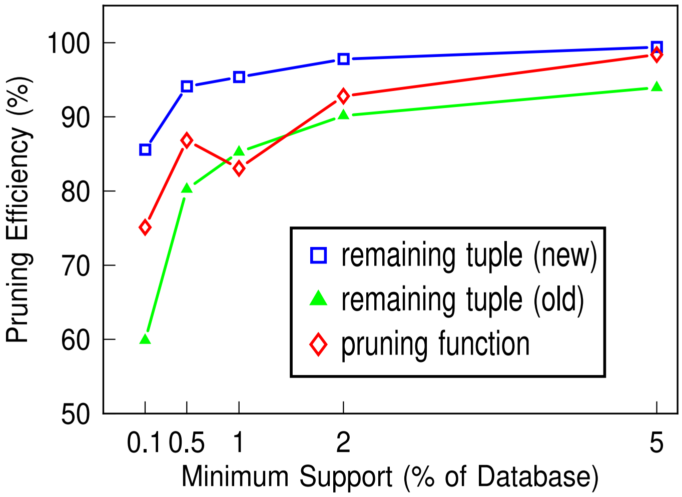

Figure 5: Efficiency of the pruning techniques

4.2 Effectiveness of the pruning optimizations

We define the efficiency of a pruning technique to be the fraction of itemsets that it prunes. We stated in Section 3.5 that the remaining tuple optimization is applied to the new candidate itemsets as soon as they are generated. The unpruned candidate itemsets are added to the candidate set. The remaining tuple optimization is also applied to these older candidate itemsets when we are about to increment their support count. Figure 5 shows the efficiency of the remaining tuple optimization technique for these two types of itemsets. For the new itemsets, the pruning efficiency is the ratio of the new itemsets pruned to the total number of new itemsets generated. For the old itemsets, the pruning efficiency is the ratio of the old candidate itemsets pruned to the total number of candidate itemsets added to the candidate set. This experiment was run with the pruning function optimization turned off. Clearly, the remaining tuple optimization prunes out a very large fraction of itemsets, both new and old.

The pruning efficiency increases with an an increase in *minsupport* because an itemset now needs to be present in a larger number of transactions to eventually make it to the large set. The candidate set contains itemsets expected to be large as well as those expected to be small. The remaining tuple optimization prunes mostly those old candidate itemsets that were expected to be small; Figure 4 bears out that most of the candidate itemsets expected to be large indeed turn out to be large. Initially, there is a large increase in the fraction of itemsets expected to be small in the candidate set as *minsupport* increases. This is the reason why initially there is a large jump in the pruning efficiency for old candidate itemsets as *minsupport* increases.

Figure 5 also shows the efficiency of the pruning function optimization, with the remaining tuple optimization turned off. It plots the fraction of new itemsets pruned due to this optimization. The effectiveness of the optimization increases with an increase in *minsupport* as we can use a smaller value for τ. Again, we note that this technique alone is also quite effective in pruning new candidate itemset.

We also measured the pruning efficiencies for new and old itemsets when both the remaining tuple and pruning function optimizations were turned on. The curves for combined pruning tracked closely the two curves for the remaining tuple optimization. The pruning function optimization does not prune old candidate itemsets. Given the high pruning efficiency obtained for new itemsets just with the remaining tuple optimization, it is not surprising that there was only slight additional improvement when the pruning function was also turned on. It should be noted however that the remaining tuple optimization is a much cheaper optimization.

5 Related Work

Discovering rules from data has been a topic of active research in AI. In [11], the rule discovery programs have been categorized into those that find *quantitative* rules and those that find *qualitative* laws.

The purpose of quantitative rule discovery programs is to automate the discovery of numeric laws of the type commonly found in scientific data, such as Boyle's law $PV = c$. The problem is stated as follows [14]: Given m variables $x_1, x_2, \ldots, x_m$ and k groups of observational data $d_1, d_2, \ldots, d_k$, where each d_i is a set of m values — one for each variable, find a formula $f(x_1, x_2, \ldots, x_m)$ that best fits the data and symbolically reveals the relationship among variables. Because too many formulas might fit the given data, the domain knowledge is generally used to provide the bias toward the formulas that are appropriate for the domain. Examples of some well-known systems in this category include ABACUS[5], Bacon[7], and COPER[6].

Business databases reflect the uncontrolled real world, where many different causes overlap and many patterns are likely to co-exist [10]. Rules in such data are likely to have some uncertainty. The qualitative rule discovery programs are targeted at such business data and they generally use little or no domain knowledge. There has been considerable work in discovering classification rules: Given examples that belong to one of the pre-specified classes, discover rules for classifying them. Classic work in this area include [4] [9].

The algorithm we propose in this paper is targeted at discovering qualitative rules. However, the rules we discover are not classification rules. We have no pre-specified classes. Rather, we find all the rules that describe association between sets of items. An algorithm, called the KID3 algorithm, has been presented in [10] that can be used to discover the kind of association rules we have considered. The KID3 algorithm is fairly straightforward. Attributes are not restricted to be binary in this algorithm. To find the rules comprising $(A = a)$ as the antecedent, where a is a specific value of the attribute A, one pass over the data is made and each transaction record is hashed by values of A. Each hash cell keeps a running summary of values of other attributes for the tuples with identical A value. The summary for $(A = a)$ is used to derive rules implied by $(A = a)$ at the pass. To find rules by different fields, the algorithm is run once on each field. What it means is that if we are interested in finding all rules, we must make as many passes over the data as the number of combinations of attributes in the antecedent, which is exponentially large. Our algorithm is linear in number of transactions in the database.

The work of Valiant [12] [13] deals with learning boolean formulae. Our rules can be viewed as boolean implications. However, his learnability theory deals mainly with worst case bounds under any possible probabilistic distribution. We are, on the other hand, interested in developing an efficient solution and actual performance results for a problem that clearly has the exponential worst case behavior in number of itemsets.

There has been work in the database community on inferring functional dependencies from data, and efficient inference algorithms have been presented in [3] [8]. Functional dependencies are very specific predicate rules while our rules are propositional in nature. Contrary to our framework, the algorithms in [3] [8] consider strict satisfaction of rules. Due to the strict satisfaction, these algorithms take advantage of the implications between rules and do not consider rules that are logically implied by the rules already discovered. That is, having inferred a dependency $X \rightarrow A$, any other dependency of the form $X + Y \rightarrow A$ is considered redundant and is not generated.

6 Summary

We introduced the problem of mining association rules between sets of items in a large database of customer transactions. Each transaction consists of items purchased by a customer in a visit. We are interested in finding those rules that have:

- Minimum transactional support s — the union of items in the consequent and antecedent of the rule is present in a minimum of $s\%$ of transactions in the database.
- Minimum confidence c — at least $c\%$ of transactions in the database that satisfy the antecedent of the rule also satisfy the consequent of the rule.

The rules that we discover have one item in the consequent and a union of any number of items in the antecedent. We solve this problem by decomposing it into two subproblems:

1. Finding all itemsets, called *large* itemsets, that are present in at least $s\%$ of transactions.
2. Generating from each large itemset, rules that use items from the large itemset.

Having obtained the large itemsets and their transactional support count, the solution to the second subproblem is rather straightforward. A simple solution to the first subproblem is to form all itemsets and obtain their support in one pass over the data. However, this solution is computationally infeasible — if there are m items in the database, there will be 2^m possible itemsets, and m can easily be more than 1000. The algorithm we propose has the following features:

- It uses a carefully tuned estimation procedure to determine what itemsets should be measured in a pass. This procedure strikes a balance between the number of passes over the data and the number of itemsets that are measured in a pass. If we measure a large number of itemsets in a pass and many of them turn out to be small, we have wasted measurement effort. Conversely, if we measure a small number of itemsets in a pass and many of them turn out to be large, then we may make unnecessary passes.

- It uses pruning techniques to avoid measuring certain itemsets, while guaranteeing completeness. These are the itemsets that the algorithm can prove will not turn out to be large. There are two such pruning techniques. The first one, called the "remaining tuple optimization", uses the current scan position and some counts to prune itemsets as soon as they are generated. This technique also establishes, while a pass is in progress, that some of the itemsets being measured will eventually turn out to be large and prunes them out. The other technique, called the "pruning function optimization", synthesizes pruning functions in a pass to use them in the next pass. These pruning functions can prune out itemsets as soon as they are generated.
- It incorporates buffer management to handle the fact that all the itemsets that need to be measured in a pass may not fit in memory, even after pruning. When memory fills up, certain itemsets are deleted and measured in the next pass in such a way that the completeness is maintained; there is no redundancy in the sense that no itemset is completely measured more than once; and there is guaranteed progress and the algorithm terminates.

We tested the effectiveness of our algorithm by applying it to sales data obtained from a large retailing company. For this data set, the algorithm exhibited excellent performance. The estimation procedure exhibited high accuracy and the pruning techniques were able to prune out a very large fraction of itemsets without measuring them.

The work reported in this paper has been done in the context of the Quest project [1] at the IBM Almaden Research Center. In Quest, we are exploring the various aspects of the database mining problem. Besides the problem of discovering association rules, some other problems that we have looked into include the enhancement of the database capability with classification queries [2] and queries over large sequences. We believe that database mining is an important new application area for databases, combining commercial interest with intriguing research questions.

Acknowledgments. We thank Mike Monnelly for his help in obtaining the data used in the performance experiments. We also thank Bobbie Cochrane, Bill Cody, Christos Faloutsos, and Joe Halpern for their comments on an earlier version of this paper.

References

[1] Rakesh Agrawal, Tomasz Imielinski, and Arun Swami, "Database Mining: A Performance Perspective", *IEEE Transactions on Knowledge and Data Engineering*, Special Issue on Learning and Discovery in Knowledge-Based Databases, (to appear).

[2] Rakesh Agrawal, Sakti Ghosh, Tomasz Imielinski, Bala Iyer, and Arun Swami, "An Interval Classifier for Database Mining Applications", *VLDB-92*, Vancouver, British Columbia, 1992, 560–573.

[3] Dina Bitton, "Bridging the Gap Between Database Theory and Practice", Cadre Technologies, Menlo Park, 1992.

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

[5] B. Falkenhainer and R. Michalski, "Integrating Quantitative and Qualitative Discovery: The ABACUS System", *Machine Learning*, 1(4): 367–401.

[6] M. Kokar, "Discovering Functional Formulas through Changing Representation Base", *Proceedings of the Fifth National Conference on Artificial Intelligence*, 1986, 455–459.

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

[8] Heikki Mannila and Kari-Jouku Raiha, "Dependency Inference", *VLDB-87*, Brighton, England, 1987, 155-158.

[9] J. Ross Quinlan, "Induction of Decision Trees", *Machine Learning*, **1**, 1986, 81–106.

[10] G. Piatetsky-Shapiro, *Discovery, Analysis, and Presentation of Strong Rules*, In [11], 229–248.

[11] G. Piatetsky-Shapiro (Editor), *Knowledge Discovery in Databases*, AAAI/MIT Press, 1991.

[12] L.G. Valiant, "A Theory of Learnable", *CACM*, **27**, 1134–1142, 1984.

[13] L.G. Valiant, "Learning Disjunctions and Conjunctions", *IJCAI-85*, Los Angeles, 1985, 560–565.

[14] Yi-Hua Wu and Shulin Wang, *Discovering Functional Relationships from Observational Data*, In [11], 55–70.