Online Generation of Association Rules

Charu C. Aggarwal and Philip S. Yu
IBM T. J. Watson Research Center
Yorktown Heights, NY 10598

Abstract

We have a large database consisting of sales transactions. We investigate the problem of online mining of association rules in this large database. We show how to preprocess the data effectively in order to make it suitable for repeated online queries. The preprocessing algorithm takes into account the storage space available. We store the preprocessed data in such a way that online processing may be done by applying a graph theoretic search algorithm whose complexity is proportional to the size of the output. This results in an online algorithm which is practically instantaneous in terms of response time. The algorithm also supports techniques for quickly discovering association rules from large itemsets. The algorithm is capable of finding rules with specific items in the antecedent or consequent. These association rules are presented in a compact form, eliminating redundancy. We believe that the elimination of redundancy in online generation of association rules from large itemsets is interesting in its own right.

1 Introduction

The importance of discovering association rules as a tool for knowledge discovery in databases has recently been recognized. By using the data from bar code companies or sales data from catalog companies, it is possible to gain valuable information about customer buying behavior in the form of association rules. Such information can be used to make decisions such as shelving in a supermarket, designing well targeted marketing programs etc.

Let $I = \{i_1, i_2, \ldots, i_m\}$ be a set of literals called items. The database consists of a set of sales transactions $\mathcal{T}$. Each transaction $T \in \mathcal{T}$ is a set of items, such that $T \subseteq I$. In this paper, we consider the 0-1 case only; in other words a 0-1 variable indicates whether or not an item was bought. A transaction T is said to contain the set of items X if and only if $X \subseteq T$.

An *association rule* is a condition of the form $X \Rightarrow Y$ where $X \subseteq I$ and $Y \subseteq I$ are two sets of attributes. The intuitive implication of the association rule is that a presence of the set of items X in a transaction set also indicates a possibility of the presence of the itemset Y. Two notions for establishing the strength of a rule are those of *minimum support* and *minimum confidence*, which were first introduced in [2].

The *support* of a rule $X \Rightarrow Y$ is the fraction of transactions which contain both X and Y.

The *confidence* of a rule $X \Rightarrow Y$ is the fraction of transactions containing X which also contain Y. Thus, if we say that a rule has 90% confidence then it means that 90% of the tuples containing X also contain Y.

Starting with pioneering work in Agrawal et. al.[2], a host of work has been done in this area with a focus on finding association rules from very large sets of transaction data. The primary idea proposed in [2] was an itemset approach in which first all large itemsets are generated, and then these large itemsets are used in order to determine data dependencies. Subsequent work has primarily concentrated on this approach.

The itemset approach is as follows. Generate all combinations of items that have fractional transaction support above a certain user-defined threshold called *minsupport*. We call all such combinations *large itemsets*. Given an itemset S satisfying the support constraint, we can use it to generate rules of the type $S - X \Rightarrow X$ for each $X \subset S$. Once these rules have been generated, only those rules above a certain user defined threshold called *minconfidence* need be retained.

Faster algorithms for mining association rules were proposed in [3], while a hash-based algorithm was established in [17]. Generalized association rules were presented in [21]. Methods for mining quantitative association rules were established in [22]. Other related work may be found in [9, 11, 19]. An up-to-date survey on some of the work done in data mining may be found in [6].

In this paper we consider the problem of online mining of association rules. The idea in online mining is that an end user ought to be able to query the database for association rules at differing values of support and confidence without excessive I/O or computation. In the itemset method, multiple passes have to be made over the database, for each differing value of *minsupport* and *minconfidence*, starting from scratch. Some sampling techniques exist which reduce the number of passes over the database to two [19, 23]. For very large databases, this may involve a considerable I/O and in some situations it may lead to unacceptable response times for online queries.

The problem of mining association rules is especially suitable for an online approach. It is hard for a user to guess apriori how many rules might satisfy a given level of support and confidence. Typically one may be interested in only a few rules. This makes the problem all the more diffi-

Rule	Support	Confidence
$X \Rightarrow YZ$	$s(X \cup Y \cup Z)$	$s(X \cup Y \cup Z)/s(X)$
$XY \Rightarrow Z$	$s(X \cup Y \cup Z)$	$s(X \cup Y \cup Z)/s(X \cup Y)$
$XZ \Rightarrow Y$	$s(X \cup Y \cup Z)$	$s(X \cup Y \cup Z)/s(X \cup Z)$
$X \Rightarrow Y$	$s(X \cup Y)$	$s(X \cup Y)/s(X)$
$X \Rightarrow Z$	$s(X \cup Z)$	$s(X \cup Z)/s(X)$

Table 1: Redundancy in rule generation

cult, since a user may need to run the query multiple times in order to find appropriate levels of minsupport and minconfidence in order to mine the rules. In other words, the problem of mining association rules may require considerable manual parameter tuning by repeated queries before useful business information can be gleaned from the transaction database.

Another issue is that while mining association rules, a large percentage of the rules may be redundant. It is useful to eliminate redundant rules simply from the point of view of compactness in representation to an online user. For example, if the rule $X \Rightarrow YZ$ is true at a given value of *minsupport* and *minconfidence*, then rules such as $XY \Rightarrow Z$, $XZ \Rightarrow Y$, $X \Rightarrow Y$, and $X \Rightarrow Z$ are redundant. This can be easily seen from the Table 1 in which one can see that both the support and confidence values of the rule $X \Rightarrow YZ$ are less than the support and confidence values for the rules $X \Rightarrow Y$, $X \Rightarrow Z$, $XY \Rightarrow Z$, and $XZ \Rightarrow Y$. In fact, in most cases, the number of redundant rules is significantly larger than the number of essential rules, and having too many redundant rules defeats the primary purpose of data mining in the first place. We note that this kind of redundancy arises when we consider rules which have more than one item in the consequent.

In recent years, an important application of database systems has been Online Analytical Processing (OLAP). The primary idea behind this approach has been the "preprocess once query many" paradigm. The idea is that it is time consuming to compute results from raw transaction data each time a user makes a query. By preprocessing the data set just once, a user may be able to query the system efficiently multiple times at the cost of a single phase of preprocessing. Considerable work has been done in online analytical processing, as applied to the data cube [5, 7, 8, 10, 20]. This paper also discusses an approach for online mining by using one phase of preprocessing.

1.1 Contributions of this paper

In this paper, we present an intuitive framework for performing online mining of association rules. Past work has concentrated on a two phase approach:

(1) **Large Itemset Generation:** Controlling parameter *minsupport.*

(2) **Rule Generation:** Controlling parameter *minconfidence.*

The bottleneck in this procedure is the first step, since most algorithms require multiple I/O passes in order to perform this step. Thus, the natural solution is to prestore as many itemsets as possible with the least support value possible given the memory available. This approach however, has some obvious drawbacks. On the one hand, one might want to store as many of such itemsets as possible as constrained by the memory space or preprocessing time available, so that important information will not be lost. On the other hand, if too many itemsets are prestored, then the second phase of rule generation becomes the bottleneck. For example, while trying to mine rules containing specific sets of items, the number of relevant large itemsets may be a very small fraction of the total number of itemsets prestored. Yet, one may need to look at each and every prestored itemset in order to find the relevant large itemsets. Consequently, it becomes important to organize the itemsets along with support information in such a way that the online time required to mine the rules is small and is dependent on the number of large itemsets corresponding to a user query, rather than the number of itemsets prestored. In this paper we shall discuss such a method. From now on, we shall refer to the prestored itemsets as primary itemsets. The primary threshold is the minimum level of support for any prestored itemset. Thus the primary itemsets comprise all itemsets whose support is at least equal to the value of the primary threshold. At this stage we would also like to make a careful distinction between primary itemsets and large itemsets. A large itemset corresponds to an itemset for a user query, and is a subset of the primary itemsets. More specifically, the contributions of this work are as follows:

(1) We devise a framework for organizing the primary itemsets in such a way that online rules with very limited I/O on the prestored data. The online time for mining the rules is independent of the size of the transaction data as well as the number of itemsets prestored. In fact, we shall see that the time required to process a query is completely dependent upon the size of the output. This feature is especially suitable for the online case.

(2) We give a technique which can quickly predict the size of the output at a given level of user specified parameters. For a given level of user-specified *minsupport* and *minconfidence*, both the number of itemsets as well as the number of rules can be predicted. A reverse query such as predicting the level of *minsupport* for which a particular number of itemsets exist can also be performed.

(3) We discuss the issue of efficiency in the generation of the rules. Since we include the possibility of generating rules with more than one item in the consequent, it may often be cumbersome (at least from an online perspective) to look at each of the subsets of the large itemsets as a possibility for the antecedent. A large number of possibilities can be pruned by careful order of examination. It is also possible to efficiently generate only rules with exactly one item in the consequent. Such rules are called *single-consequent* rules.

(4) We discuss the issue of generating rules with specific of items in them. The items may occur in the antecedent or consequent.

(5) We discuss the issue of redundancy in the rules generated from large itemsets. We discuss the level to which essential rules may often get buried in hordes of redundant rules. Compactness of representation to an online

user is a very useful feature. This segment of the paper has both theoretical and practical significance.

(6) We present an algorithm for finding the primary itemsets which automatically decides which itemsets to prestore depending upon available memory capacity. For the sake of high level discussion, we shall fix the maximum number of itemsets rather than the memory space occupied by the itemsets. This is a slightly different problem from that discussed in Agrawal et. al. [2], where one needs to find the itemsets with support above a particular value. The value of the primary threshold at which the best fit to this maximum number of itemsets may be found is not known in advance. One may perform a binary search on the support value in order to find the value of the primary threshold. We propose techniques for improving the efficiency beyond simply performing a simple binary search.

We should note that it is not possible to perform online mining of association rules at support levels less than the primary threshold. This is not necessarily a severe restriction since the primary itemsets are obtained within the preprocessing time constraints, which are significantly more liberal than online time constraints. Thus, most useful itemsets are typically prestored.

1.2 Kinds of online queries

Assume that the kinds of online queries that such a system can support are as follows.

(1) Find all association rules above a certain level of *minsupport* and *minconfidence*.

(2) At a certain level of *minsupport* and *minconfidence*, find all association rules concerned with the set of items X.

(3) Find the *number* of association rules/itemsets in any of the cases (1), (2) above.

(4) At what level of *minsupport* do exactly k itemsets exist containing the set of items Z.

(5) For a particular level of *minconfidence* c, at what level of *minsupport* do exactly k single-consequent rules exist, which involve the set of items Z.

1.3 Overview

We introduce the concept of an adjacency lattice of itemsets. This adjacency lattice is crucial to performing effective online data mining. The adjacency lattice could be stored either in main memory or on secondary memory. We shall discuss more details about how this lattice is actually constructed in a later section. The idea of the adjacency lattice is to prestore a number of large itemsets at a level of support possible given the available memory. These itemsets are stored in a special format (called the adjacency lattice) which reduces the disk I/O required in order to perform the analysis. In fact, if enough main memory is available for the entire adjacency lattice, then no I/O may need to be performed at all.

We shall see that this structure is useful for both finding the itemsets quickly and also using the itemsets in order to generate the rules. Redundancy in rules is eliminated, so that an online user may be presented with the most compact representation possible.

Itemset	Support
A	1%
B	2%
C	2%
D	1%
AB	0.5%
AC	0.7%
BD	0.6%
BC	0.4%
ABC	0.3%

Table 2:

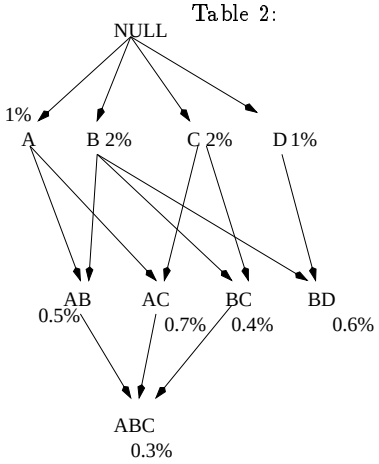


Figure 1: The adjacency lattice

2 The adjacency lattice

Before we consider making a more detailed description, we shall discuss the concept of an adjacency lattice of itemsets. For future reference, we shall denote the adjacency lattice by L.

An itemset X is said to be *adjacent* to an itemset Y if one of them can be obtained from the other by adding a single item. Specifically, an itemset X is said to be a parent of the itemset Y if Y can be obtained from X by adding a single item to the set X. Equivalently, Y may be considered to be a child of X. Thus, an itemset may possibly have more than one parent and more than one child. In fact, the number of parents of an itemset X is exactly equal to the cardinality of the set X. This observation follows from the fact that for each element i_r in an itemset X, $X - \{i_r\}$ is a parent of X. It is easy to see that if a directed path exists from the vertex corresponding to Z to the vertex corresponding to X in the adjacency lattice, then $X \supseteq Z$. In such a case, X is said to be an ancestor of Z, and Z is said to be a descendant of X.

The adjacency lattice L is constructed as follows: Construct a graph with a vertex $v(I)$ for each primary itemset I. Each vertex I has a label corresponding to the value of its support. This label is denoted by $S(I)$. For any pair of vertices corresponding to itemsets X and Y, a directed edge exists from $v(X)$ to $v(Y)$ if and only if X is a parent

```
Algorithm FindItemsets(ItemSet: I, Support: s)
begin
LIST= v(I); OutputList = φ;
while LIST ≠ φ do
   begin
   Select a vertex v(R) from LIST;
   { Assume that the children of a vertex are arranged
    in decreasing order of support }
   while the next child v(T) of v(R)
                        satisfies S(T) ≥ s do
      begin
      if v(T) ∉ OutputList do
         begin
         LIST=LIST ∪v(T);
         OutputList = OutputList ∪ (v(T), S(T));
         cardinality=cardinality+1;
         end
      end;
   Delete the vertex v(R) from LIST
   end;
end;
```

Figure 2: The search algorithm for generating large itemsets

of Y. We denote the corresponding edge by $E(X, Y)$. The vertex $v(X)$ is referred to as the *tail* of the edge $E(X, Y)$, while the vertex $v(Y)$ is referred to as the *head*.

Consider for example the group of primary itemsets illustrated in Table 2. The corresponding adjacency lattice is illustrated in Figure 1. Each vertex has a label corresponding to the value of its support. We make the following simple observations for the adjacency lattice L:

Remark 2.1 *The adjacency lattice L is a directed acyclic graph.*

Remark 2.2 *For each vertex $v(J)$ in L which is a descendent of $v(I)$, we must have $S(J) \leq S(I)$.*

The truth of Remark 2.2 follows from the fact that for each vertex $v(J)$ which is a descendent of $v(I)$, the corresponding itemsets must satisfy $J \supseteq I$. Since the adjacency lattice is the primary structure which is used to represent the preprocessed data, it is useful to measure the memory which such a structure might require. We shall proceed to show that the space required to store the adjacency lattice is not the bottleneck, and is almost of the same order as the space required to hold the itemsets themselves.

Theorem 2.1 *The number of edges in the adjacency lattice is equal to the sum of the number of items in the primary itemsets.*

Proof: The number of edges may be obtained by summing the number of parents of each primary itemset. The number of parents of a primary itemset is equal to the number of items in it. The result follows. ■

3 Online generation of itemsets

In order to find all itemsets which contain a set of items I and satisfy a level of *minsupport* s, we need to solve the following search problem in the adjacency lattice.

Problem 3.1 *For a given itemset I (including $\{\}$), find all itemsets J such that $v(J)$ is reachable from $v(I)$ by a directed path in the lattice L, and satisfies $S(J) \geq s$.*

```
Algorithm FindSupport(ItemSet: Z, Cardinality: k)
begin
LIST= v(Z); OutputList = φ;
cardinality= 0
while (LIST ≠ φ) and (cardinality≤ k) do
   begin
   Select a vertex v(R) from LIST with largest value of S(R);
   OutputList = OutputList ∪ (v(R), S(R));
   cardinality=cardinality+1;
   for each child v(T) of v(R) do
      begin
      if v(T) ∉ OutputList do
         LIST=LIST ∪v(T);
      end;
   Delete the vertex v(R) from LIST
   end;
return (min{S(R) : (v(R), S(R)) ∈ OutputList}, OutputList)
end;
```

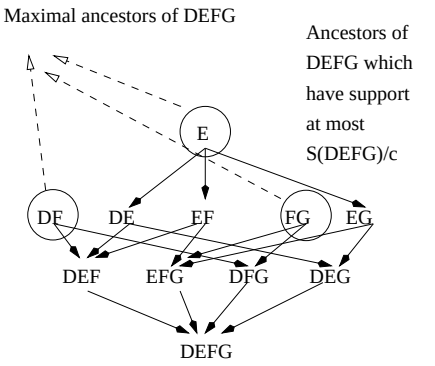
Figure 3: Finding the level of support for a fixed number of itemsets

It is important to understand that the number of vertices reachable from a given vertex may be quite large, though the number of vertices which satisfy the level of *minsupport* s may be small. The idea is to use the lattice organization to restrict the number of vertices examined. Thus, when a user makes multiple queries to the database, this pre-processed data helps avoid the reading of the entire database from scratch. We shall now discuss the search algorithm which given the parameters I and s, finds all the itemsets containing I and having a support level of at least s. This algorithm is illustrated in Figure 2. The algorithm *FindItemsets* starts at a given itemset I and LIST= $\{v(I)\}$. The algorithm then adds all of its children $v(J)$ with support $S(J) \geq s$ to LIST unless the vertex has been visited before. The vertex $v(I)$ is then deleted from LIST. This process is repeated until LIST is empty. Thus, all the vertices which are the unvisited children of a given vertex in LIST are recursively searched unless their support value is less than s. The itemsets for every vertex which is visited are also added to the *OutputList*. At the same time, a count of the cardinality of *OutputList* is maintained in order to handle the feature where a user may wish to find the cardinality of the itemsets. At termination of the algorithm the *OutputList* contains all the itemsets J with support $S(J) \geq s$ and satisfying $J \supseteq I$.

3.1 Finding the level of support for a fixed number of itemsets

A useful online feature is to find the level of support at which exactly k itemsets (each of which contains the items $Z = \{i_1 \ldots i_r\}$) exist. This can be accomplished by making a few changes to the search algorithm of Figure 2. The resulting algorithm is illustrated in Figure 3. The primary idea is that while selecting a vertex $v(R)$ on LIST which is to be examined in the current iteration, we always pick the vertex with the highest value of support. At that time, we add this vertex to *OutputList*. The algorithm terminates when k vertices have been found. It can be proved that at each stage of this algorithm, *OutputList* maintains $r \leq k$ itemsets containing Z with the highest support value.

Theorem 3.1 *The algorithm FindSupport(Z, k) finds the k itemsets containing Z and having the highest value of support. If less than k such itemsets are represented in the*



Essential Rules	Redundant Rules f
E=>DFG	DE=>FG, DEF=>G, EF=>DG, EFG=>D, EG=>DF, DEG=>F
DF=>EG	DEF=>G, DFG=>E
FG=>DE	DFG=>E , EFG=>D

Figure 4: An illustration of the boundary itemset

adjacency lattice, then the algorithm finds all the itemsets containing Z.

Proof: The proof of this theorem is by induction. The induction hypothesis is that the $r \leq k$ items maintained in the *OutputList* are the r itemsets containing Z with the highest value of support. The induction hypothesis is trivially true when $OutputList = \phi$. Each time an itemset is added to *OutputList* we pick the itemset on LIST with the highest support value. Any other itemset which we add to *OutputList* in the future, is either already on LIST, or is a descendent of some itemset currently in LIST. From Remark 2.2, the result immediately follows. ■

3.2 Finding the level of support for a fixed number of single-consequent rules

A single consequent rule is one in which the consequent contains only one item. It is also possible to use the algorithm described above to find the level of support at which a particular number (say k) of single-consequent rules exist for a prespecified level of confidence c. This can be achieved by making a minor modification to the procedure *FindSupport* of Figure 3. In this case, each time a vertex $v(X)$ is selected from LIST, all the single-consequent rules which can be generated from $v(X)$ at confidence level c are added to *OutputList*. The count of the number of rules is maintained. The first time the count exceeds k, the procedure is terminated. The proof of correctness of this method is exactly analogous to the proof of Theorem 3.1.

4 Online generation of rules from itemsets

In the previous section, we discussed how large itemsets may be generated from the adjacency lattice. In this section, we discuss how rules may be generated from these itemsets. To generate the rules, we utilize the following observation:

For each rule $A \Rightarrow B$ at confidence level c, the label (support) on the vertex $v(A \cup B)$ is at most $1/c$ times the label (support) on the vertex $v(A)$. Thus, the confidence of a rule may be obtained by comparing the labels on two vertices which satisfy an ancestor-descendant relationship in the adjacency lattice.

Conversely, let $\mathcal{X} = \{X_1, \ldots X_k\}$ be the itemsets generated in the first phase of the online processing algorithm. Let c be the level of *minconfidence* at which it is desired to mine the association rules. For each $X_i \in \mathcal{X}$, rules may be generated by applying a reverse search algorithm starting from $v(X_i)$ and finding all ancestors of $v(X_i)$ which have support at most $S(X_i)/c$. For each such ancestor $v(Y)$ of $v(X_i)$, it is possible to generate rules of the form $Y \Rightarrow X_i - Y$. Thus the problem of finding all rules generated from a large itemset X is reduced to the following graph search problem in the adjacency lattice:

Problem 4.1 *Find all ancestor vertices of $v(X)$ which have support at least $S(X)/c$.*

Unfortunately, many of the generated rules will turn out to be redundant. For example, if a rule $X \Rightarrow YZ$ is included in the output, then the rule $XY \Rightarrow Z$ can be regarded as redundant.

Definition 4.1 *Let $A \Rightarrow B$ and $C \Rightarrow D$ be two association rules. The rule $C \Rightarrow D$ is redundant with respect to the rule $A \Rightarrow B$ if the support and confidence of the former are both always at least as large as the support and confidence of the latter, independent of the nature of the transaction data.*

We shall first classify the different kinds of redundancy as follows:

Theorem 4.1 Simple Redundancy: *Let $A \Rightarrow B$ and $C \Rightarrow D$ be two rules satisfying $A \cup B = C \cup D = X$. The rule $C \Rightarrow D$ bears simple redundance with respect to the rule $A \Rightarrow B$, if $C \supset A$. In other words, if the rule $A \Rightarrow B$ is true at a certain level of support and confidence, then so is $C \Rightarrow D$, independent of the nature of the transaction data.*

Proof: Omitted. See [1]. ■

Thus, in simple redundancy, the support value for the two rules is the same, but the confidence value for one is larger than the confidence value for the other. The support values for the rules are the same since they are generated from the same itemset. As an example, the rule $AB \Rightarrow C$ bears simple redundance with respect to the rule $A \Rightarrow BC$. We shall now discuss the case when one rule dominates the other based upon both support and confidence.

Theorem 4.2 Strict Redundancy: *We consider two rules generated from itemsets X_i and X_j respectively such that $X_i \supset X_j$. Let $A \Rightarrow B$ and $C \Rightarrow D$ be rules satisfying $A \cup B = X_i$, $C \cup D = X_j$, and $C \supseteq A$. Then the rule $C \Rightarrow D$ is redundant with respect to the rule $A \Rightarrow B$.*

Proof: Omitted. See [1]. ■

Thus, in strict redundancy, one rule dominates the other based upon both support as well as confidence. As an example, the rule $X \Rightarrow Y$ bears strict redundancy with respect to the rule $X \Rightarrow YZ$. We shall introduce some additional definitions and notation here for the sake of future discussion.

```
Algorithm FindBoundary(ItemSet: X, Confidence: c)
begin
LIST= v(X); BoundaryList = φ;
while LIST ≠ φ do
   begin
   Select a vertex v(R) from LIST;
   for each parent v(T) of v(R) do
      begin
      if v(T) has not yet been visited and S(T) ≤ S(X)/c do
         LIST=LIST ∪v(T);
      end;
   Delete the vertex v(R) from LIST
   if v(R) is maximal add v(R) to BoundaryList
   end;
end;
```

Figure 5: Finding the boundary itemset

```
Algorithm GenerateRules(Set of Itemsets: 𝒳, c)
begin
RuleSet = φ
for each X_i ∈ 𝒳 do ℱ(X_i, c) = FindBoundary(X_i, c)
for each X_i ∈ 𝒳 do
   begin
   𝒫(X_i, c) = ℱ(X_i, c)
   for each child X_j ∈ 𝒳 of X_i do
      𝒫(X_i, c) = 𝒫(X_i, c) − ℱ(X_j, c)
   For each itemset Y ∈ 𝒫(X_i, c) do
      RuleSet = RuleSet ∪ {Y ⇒ X_i − Y}
   end;
return RuleSet
end;
```

Figure 6: Generating the rules from the boundary itemsets

Definition 4.2 *A rule is defined to be essential at support level s and confidence level c if it does not satisfy simple or strict redundancy with respect to any other rule which has support at least s and confidence at least c.*

As we shall see, the number of redundant rules may often be a significant fraction of the total number of rules. We shall prove a result which quantifies the number of redundant rules corresponding to a single rule $X \Rightarrow Y$. For ease in notation, we shall denote the number of items in an itemset X by $|X|$.

Theorem 4.3 *The number of rules bearing simple redundancy with respect to $X \Rightarrow Y$ is $2^{|Y|} - 2$. The number of rules bearing either simple or strict redundancy with respect to the rule $X \Rightarrow Y$ is $3^{|Y|} - 2^{|Y|} - 1$.*

Proof: Omitted. See [1]. ■

As an example, consider the rule $A \Rightarrow BC$. There are $2^2 - 2$ simple redundant rules, namely $AC \Rightarrow B$, and $AB \Rightarrow C$. The strict redundant rules are $A \Rightarrow B$, and $A \Rightarrow C$. Thus the total number of redundant rules is $3^2 - 2^2 - 1 = 4$. Clearly, as the number of items in the consequent increases, the number of redundant rules explodes exponentially.

Definition 4.3 *A vertex $v(Y)$ is a maximal ancestor of $v(X)$ at confidence level c if and only if $S(Y)/S(X) \leq 1/c$, and no strict ancestor $v(Z)$ of $v(Y)$ satisfies $S(Z)/S(X) \leq 1/c$.*

Maximal ancestors are very relevant to the process of finding rules which avoid simple redundancy.

Theorem 4.4 *Let $v(Y)$ be a maximal ancestor of $v(X)$ at a level of confidence c. Then the rule $Y \Rightarrow X - Y$ cannot exhibit simple redundancy with respect to any other rule at confidence level c and any support level $s \leq S(X)$. Conversely, if the rule $Y \Rightarrow Z$ does not exhibit simple redundancy with respect to any other rule at confidence level c, then $v(Y)$ must be a maximal ancestor of $v(Y \cup Z)$.*

Proof: Omitted. See [1]. ■

Thus, finding maximal ancestors of large itemsets is necessary and sufficient to generate rules which avoid simple redundancy. As an illustration, consider the example in Figure 4. Only the relevant segment of the adjacency lattice is illustrated in the figure. Suppose that we wish to generate all the rules at a particular confidence level c from an itemset $DEFG$. Also, assume that the itemsets which have support at most $S(DEFG)/c$ are DEF, EFG, DFG, DEG, DF, DE, EF, EG, FG, and E. Thus, a total of 10 rules (corresponding to these 10 itemsets) can be generated, each of which satisfy the confidence level c. However, as we see from Figure 4, only three of these rules are essential, while the rest bear simple redundancy to one or more of these rules. These three rules are generated by picking the three maximal ancestors of $DEFG$ from these 10 itemsets and generating the corresponding rules. Thus the problem of generating nonredundant rules with confidence level c from a large itemset X reduces to the following graph search problem.

Problem 4.2 *Find all maximal ancestors of $v(X)$ with support at most $S(X)/c$.*

We shall refer to all the maximal ancestors of a vertex as the *boundary itemsets* for the corresponding itemset at the given level of confidence.

Definition 4.4 *The boundary for an itemset X at level of confidence c is the set of all maximal ancestors of X at confidence level c, and is denoted by $\mathcal{F}(X,c)$.*

Finding the boundary for a given itemset X is simple enough by using a reverse search algorithm on the corresponding adjacency lattice starting at $v(X)$, as illustrated in Figure 5. This algorithm does not incorporate the constraints on having particular items in the antecedent or consequent. We shall discuss this issue in a later subsection.

In order to actually generate rules from the itemsets $\mathcal{X} = \{X_1, X_2, \ldots X_k\}$, we apply the following method. For each itemset $X_i \in \mathcal{X}$, we find the boundary itemset $\mathcal{F}(X_i, c)$ and for each $Y \in \mathcal{F}(X_i, c)$, we generate the rule $Y \Rightarrow X_i - Y$. Unfortunately, this may result in strict redundancy while generating rules from two different itemsets X_i and X_j which satisfy $X_i \subset X_j$. First, we will discuss some simple results.

Theorem 4.5 *Let X be an itemset, and let $X_1, X_2, \ldots X_k$ be the children of X. Let Y be any itemset in $\mathcal{F}(X,c) - \cup_{i=1}^{k} \mathcal{F}(X_i, c)$. Then, the rule $Y \Rightarrow X - Y$ cannot bear strict redundancy with respect to any other rule. Conversely, let X_i be a child of X such that Y lies in both $\mathcal{F}(X,c)$ and $\mathcal{F}(X_i,c)$. Then the rule $Y \Rightarrow X - Y$ is strictly redundant with respect to one or more rules.*

Proof: Omitted. See [1]. ■

Thus, we have effectively shown in the above theorem that in order to avoid strict redundancy, it is necessary and sufficient to prune the boundary of an itemset X so that it does

not share any itemsets with the boundary of any itemset $X_k \in \mathcal{X}$ which is a child of X. In other words, for each child $X_k \in \mathcal{X}$ of X, we remove from $\mathcal{F}(X, c)$, all member itemsets in $\mathcal{F}(X_k, c)$. Then these pruned boundaries may be used in order to generate the rules. The resulting algorithm is illustrated in Figure 6. This algorithm uses as input the itemsets $\mathcal{X}$ which are generated in the first phase of the algorithm at the appropriate level of *minsupport*. The algorithm *FindBoundary* of Figure 5 may be used as a subroutine in order to generate all the boundary itemsets. These boundary itemsets are then pruned and the rules are generated by using each of the itemsets corresponding to the boundary in the antecedent.

4.1 Rules with constraints in the antecedent and consequent

It is easy enough to adapt the above rule generation method so that particular items occur in the antecedent and/or consequent. Consider for example the case when we are generating rules from a large itemset X. Suppose that we desire the antecedent to contain the set of items P and the consequent to contain the set of items Q. (We assume that $P \cup Q \subseteq X$.) We shall refer to P as the *antecedent inclusion set*, and Q as the *consequent inclusion set*. In this case, we need to redefine the notion of maximality and boundary itemsets. A vertex $v(Y)$ is defined to be a maximal ancestor of $v(X)$ at confidence level c, antecedent inclusion set P, and consequent inclusion set Q if and only if $P \subseteq Y$, $Q \subseteq X - Y$, $S(Y)/S(X) \leq 1/c$, and no strict ancestor of Y satisfies all of these constraints. Equivalently, the boundary set contains all the itemsets corresponding to maximal ancestors of X. It is easy to modify the algorithm discussed in Figure 5, so that it takes the antecedent and consequent constraints into account. The only difference is that we add an unvisited vertex $v(T)$ to LIST if and only if $S(T) \leq S(X)/c$, and $T \supseteq P$. Also, a vertex $v(R)$ is added to *BoundaryList*, only if it satisfies the modified definition of maximality.

5 Generation of the adjacency lattice

In this section we discuss the construction of the adjacency lattice. The process of constructing the adjacency lattice requires us to first find the primary itemsets. There are two main constraints involved in choosing the number of itemsets to prestore:

(1) **Memory Limits:** In order to avoid I/O one may wish to store the primary itemsets and corresponding adjacency lattice in main memory.[1] Recall that Theorem 2.1 characterizes the size required by the adjacency lattice for this purpose. Assume that we desire to find N itemsets. Note that because of ties in the support values of the primary itemsets, support values may not exist for which there are exactly N itemsets. Thus, we assume that for some slack value N_s, we wish to find a primary threshold value for which the number of itemsets is between $N - N_s$ and N.

(2) **Preprocessing Time:** There may be some practical limits as to how much time one is willing to spend in preprocessing. Consequently, even if it is not possible to find N itemsets within the preprocessing time, it ought to be able to terminate the algorithm with some value of the primary threshold for which all itemsets with support above that value have been found.

A simple way of finding the primary itemsets is by using a binary search algorithm on the value of the primary threshold, using the *DHP* method discussed in Chen et. al. [17] as a subroutine. This method is somewhat naive and simplistic, and is not necessarily efficient, since it requires multiple executions of the *DHP* method. This method of finding the primary threshold is discussed in the algorithm *NaiveFindThreshold* of Figure 7. The time complexity of the procedure can be improved considerably by utilizing a few simple ideas:

(1) It is not necessary to execute the *DHP* subroutine to completion in each and every iteration. For estimates which are lower bounds on the correct value(s) of the primary threshold, it is sufficient to terminate the procedure as soon as N or more large itemsets have been generated at the level of support being considered.

(2) It is not necessary to start the *DHP* procedure from scratch in each iteration of the binary search procedure. It is possible to reuse information between iterations. Let $\mathcal{I}(s)$ denote the itemsets which have support at least s. It is possible to speed up the preprocessing algorithm by reusing the information available in $\mathcal{I}(Low)$. Generating k-itemsets in $\mathcal{I}(Mid)$ is only a matter of picking those k-itemsets in $\mathcal{I}(Low)$ which have support at least Low. This does not mean that every itemset in $\mathcal{I}(Mid)$ can be immediately generated using this method. Recall (from (1) above) that the *DHP* algorithm is often terminated before completion, if more than N itemsets have been generated in that iteration. Consequently, not all itemsets in $\mathcal{I}(Low)$ may be available, but only those k-itemsets for which $k \leq k_0$, for some k_0 are available. Thus, we have all

```
Function NaiveFindThreshold(NumberofItemsets: N, Slack: N_s)
begin
  High = max_i{Support of item i}
  Low = 0; Generated = 0;
  while (Generated ∉ (N − N_s, N))
  begin
    Mid = (High + Low)/2;
    Generated = DHP(Mid);
  end;
  return(Mid);
end

Algorithm ConstructLattice(NumberofItemsets: N, Slack: N_s)
begin
  p = NaiveFindThreshold(N, N_s)
  For each itemset X = {i_1, ... i_r} with S(X) ≥ p do
  Add the vertex v(X) to the adjacency lattice with label S(X)
  Add the edge E(X − {i_k}, X) for each k ∈ {1, ..., r}
end
```

Figure 7: Constructing the adjacency lattice

[1] Storing the adjacency lattice on disk is not such a bad option after all. The total I/O is still proportional to the size of the output, rather than the number of itemsets prestored. Recall that the graph search algorithms used in order to find the large itemsets and association rules visit only a small fraction of the vertices in the adjacency lattice.

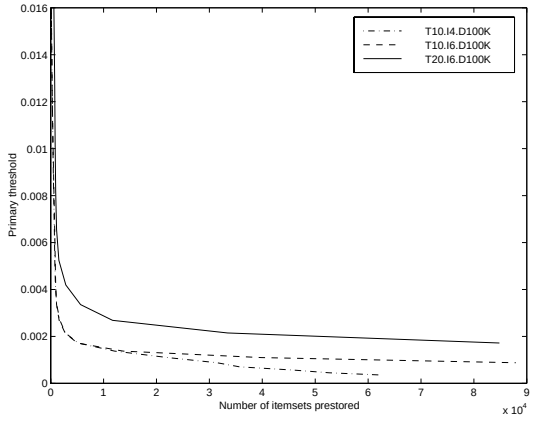


Figure 8: Threshold varation with itemsets prestored

DataSet	Conf.	Sup.	DHP	Online
T10.I4.D100K	90%	0.3%	100 sec.	instantaneous
T10.I6.D100K	90%	0.3%	130 sec.	instantaneous
T10.I6.D100K	90%	0.2%	240 sec.	2 seconds
T20.I6.D100K	90%	0.5%	100 sec.	instantaneous

Table 3: Sample illustrations of the order of magnitude advantage of online processing

those k-itemsets in $\mathcal{I}(Mid)$ available for which $k \leq k_0$. These itemsets need not be generated again.

6 Empirical Results

We ran the simulation on an IBM RS/6000 530H workstation with a CPU clock rate of 33MHz, 64 MB of main memory and running AIX 4.1.4. We tested the algorithm empirically for the following objectives:

(1) **Preprocessing sensitivity:** The preprocessing technique is sensitive to the available storage space. The larger the available space, the lower the value of the primary threshold. We tested how the primary threshold value varied with the storage space availability. We also tested how the running time of the preprocessing algorithm scaled with the storage space.

(2) **Online processing time:** We tested how the online processing times scaled with the size of the output. We also made an order of magnitude comparison between using an online approach and a more direct approach.

(3) **Level of redundancy:** We tested how the level of redundancy in the generated output set varied with user specified levels of support and confidence. We showed that the level of redundancy in the rules is quite high. Thus redundancy elimination is an important issue for an online user looking for compactness in representation of the rules.

6.1 Generating the synthetic data sets

The synthetic data sets were generated using a method similar to that discussed in Agrawal et. al. [3]. Generating the data sets was a two stage process:

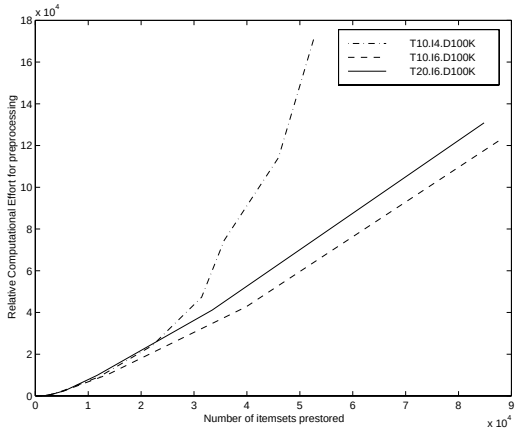


Figure 9: Computation variation with itemsets prestored

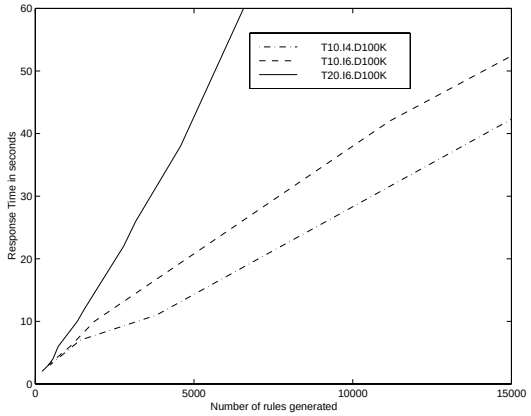


Figure 10: Online response time variation with rules generated

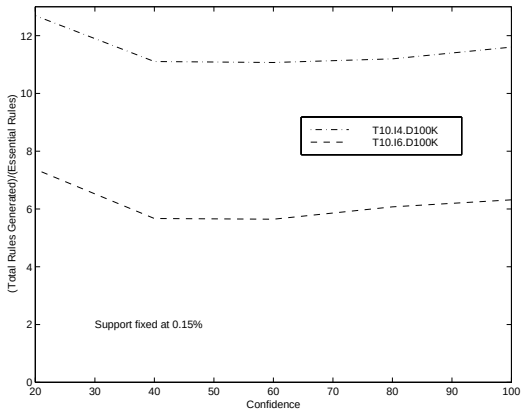


Figure 11: Redundancy level variation with confidence

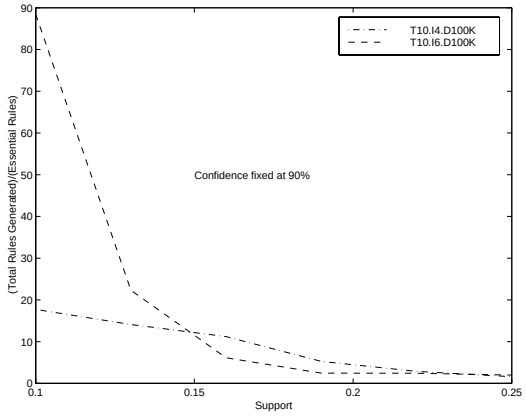


Figure 12: Redundancy level variation with support

(1) **Generating maximal potentially large itemsets:** The first step was to generate $L = 2000$ maximal "potentially large itemsets". These potentially large itemsets capture the consumer tendencies of buying certain items together. We first picked the size of a maximal potentially large itemset as a random variable from a poisson distribution with mean μ_L. Each successive itemset was generated by picking half of its items from the current itemset, and generating the other half randomly. This method ensures that large itemsets often have common items. Each itemset I has a weight w_I associated with it, which is chosen from an exponential distribution with unit mean.

(2) **Generating the transaction data:** The large itemsets were then used in order to generate the transaction data. First, the size S_T of a transaction was chosen as a poisson random variable with mean μ_T. Each transaction was generated by assigning maximal potentially large itemsets to it in succession. The itemset to be assigned to a transaction was chosen by rolling an L sided weighted die depending upon the weight w_I assigned to the corresponding itemset I. If an itemset did not fit exactly, it was assigned to the current transaction half the time, and moved to the next transaction the rest of the time. In order to capture the fact that customers may not often buy all the items in a potentially large itemset together, we added some noise to the process by corrupting some of the added itemsets. For each itemset I, we decide a noise level $n_I \in (0, 1)$. We generated a geometric random variable G with parameter n_I. While adding a potentially large itemset to a transaction, we dropped $\min\{G, |I|\}$ random items from the transaction. The noise level n_I for each itemset I was chosen from a normal distribution with mean 0.5 and variance 0.1.

We shall also briefly describe the symbols that we have used in order to annotate the data. The three primary factors which vary are the average transaction size μ_T, the size of an average maximal potentially large itemset μ_L, and the number of transactions being considered. A data set having $\mu_T = 10$, $\mu_L = 4$, and $100K$ transactions is denoted by T10.I4.D100K.

We tested how the primary threshold varied with the number of itemsets prestored. This result is illustrated in Figure 8. The figure shows that the primary threshold initially drops considerably as the number of primary itemsets increases, but it bottoms out after a while. We also illustrate the variation of the computational effort required with the available storage space in Figure 9. We note that for the itemset T10.I4.D100K, the computational effort required in order to find additional large itemsets after finding 20000 itemsets increases considerably with the number of itemsets prestored. This is because for this particular data set, the average size of a maximal potentially large itemset (or basket) is only 4. Consequently, the total number of possible large itemsets is relatively limited. On the other hand, the computational effort for preprocessing required by the data sets T20.I6.D100K and T10.I6.D100K is relatively similar. This shows that the computational effort required to find a specific number of primary itemsets is more sensitive to the size of a typical basket in the data, rather than to the size of a transaction.

We also tested the variation in the online running time of the algorithm with the number of rules generated. We ran the online queries for varying levels of input parameters in order to test the correlation between the running time and the number of rules generated. This is illustrated in Figure 10. This result is significant in that it shows that the running time of the algorithm increases linearly with the number of rules generated for all the data sets used. The absolute magnitude of time required in order to generate the rules was an order of magnitude smaller than the time required using a direct itemset generation approach like *DHP*. A brief summary of some sample relative findings is illustrated in Table 3.

We also discuss the level of redundancy present in the rule generation procedure. Figures 11 and 12 illustrate that the number of redundant rules is often much larger than the number of essential rules. The benchmark for measuring the level of redundancy is referred to as the redundancy ratio, and is defined as follows:

$$\text{Redundancy Ratio} = \frac{\text{Total Rules Generated}}{\text{Essential Rules}} \quad (1)$$

Thus, when the redundancy ratio is K, then the number of redundant rules is $K - 1$ times the number of essential rules. The redundancy ratio has been plotted on the Y-axis in Figures 11 and 12. We see that in most cases the number of redundant rules is significantly larger than the number of essential rules. This illustrates the level to which useful rules often get buried in large numbers of redundant rules. Also, the redundancy level is much more sensitive to the support rather than the confidence. The lower the level of support, the higher the redundancy level.

7 Conclusions and Summary

In this paper we investigated the issue of online mining of association rules. The two primary issues involved in online processing are the running time and compactness in representation of the rules. We discussed an OLAP-like approach for online mining association rules which avoids redundancy.

Acknowledgements

We would like to thank V. S. Jaychandran and Joel Wolf for their extensive comments and suggestions.

References

[1] Aggarwal C. C., and Yu P. S. Online Generation of Association Rules. *IBM Research Report* RC 20899.

[2] Agrawal R., Imielinski T., and Swami A. Mining association rules between sets of items in very large databases. *Proceedings of the ACM SIGMOD Conference on Management of data,* pages 207-216, Washington D. C., May 1993.

[3] Agrawal R., and Srikant R. Fast Algorithms for Mining Association Rules in Large Databases. *Proceedings of the 20th International Conference on Very Large Data Bases,* pages 478-499, September 1994.

[4] Agrawal R., and Srikant R. Mining Sequential Patterns. *Proceedings of the 11th International Conference on Data Engineering,* pages 3-14, March 1995.

[5] Agrawal S., Agrawal R., Deshpande P. M., Gupta A., Naughton J. F., Ramakrishnan R., and Sarawagi S. On the Computation of Multidimensional Aggregates. *Proceedings of the 22nd International Conference on Very Large Databases.* pages 506-521.

[6] Chen M. S., Han J., and Yu P. S. Data Mining: An Overview from Database Perspective. *IEEE Transactions on Knowledge and Data Engineering.* Volume 8, Number 6, December 1996. pages 866-883.

[7] Dyreson C. Information Retreival from an Incomplete Data Cube. *Proceedings of the 22nd International Conference on Very Large Databases.* pages 532-543, Mumbai, India, 1996.

[8] Gupta A., Harinarayan V., and Quass D. Aggregate-query processing in data warehousing environments. *Proceedings of the 21st Conference on Very Large Databases,* Zurich, Switzerland, September 1995.

[9] Han J. and Fu Y. Discovery of Multiple-Level Associaition Rules from Large Databases. *Proceedings of the 21st International Conference on Very Large Data Bases.* Zurich, Switzerland, 1995, pages 420-431.

[10] Harinarayan V., Rajaraman A., and Ullman J. Implementing Data Cubes Efficiently. *Proceedings of the 1996 ACM SIGMOD conference on Management of Data.* Montreal, Canada, June 1996, pages 205-227.

[11] Houtsma M., and Swami A. Set-oriented Mining for Association Rules in Relational Databases. *Proceedings of the 11th International Conference on Data Engineering.* March 1995, pages 25-33.

[12] Kaufman L., and Rousseeuw P. J. *Finding Groups in Data - An Introduction to Cluster Analysis.* Wiley Series in Probability and Mathematical Statistics, 1990.

[13] Klementtinen M., Mannila H., Ronkainen P., Toivonen H., and Verkamo A. I. Finding interesting rules from large sets of discovered association rules. *Proceedings of the Conference on Information and Knowledge Managements.* Gaithersburg, MD, USA 28 Nov. 2 Dec. 1994.

[14] Lent B., Swami A., and Widom J. Clustering Association Rules. *Proceedings of the Thirteenth International Conference on Data Engineering.* pages 220-231, Birmingham, UK, April 1997.

[15] Mannila H., Toivonen H., and Verkamo A. I. Efficient algorithms for discovering association rules. *AAAI Workshop on Knowledge Discovery in Databases,* pages 181-192, Seattle, Washington, July 1994.

[16] Ng R. T., and Han J. Efficient and Effective Clustering Methods for Spatial Data Mining. *Proceedings of the 20th International Conference on Very Large Data Bases.* Santiago, Chile, 1994, pages 144-155.

[17] Park J. S., Chen M. S., and Yu P. S. An Effective Hash Based Algorithm for Mining Association Rules. *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data.* pages 175-186, May 1995.

[18] Piatetsky-Shapiro G. Discovery, Analysis and Presentation of Strong Rules. *Knowledge Discovery in Databases,* 1991.

[19] Savasere A., Omiecinski E., and Navathe S. An Efficient Algorithm for Mining Association Rules in Large Data Bases. *Proceedings of the 21st International Conference on Very Large Data Bases.* Zurich, Switzerland, 1995, pages 432-444.

[20] Shukla A., Deshpande P. M., Naughton J. F. and Ramasamy K. Storage Estimation for Multidimensional Aggregates in the Presence of Hierarchies. *Proceedings of the 22nd International Conference on Very Large Databases.* pages 522-531, Mumbai, India, 1996.

[21] Srikant R., and Agrawal R. Mining Generalized Association Rules. *Proceedings of the 21st International Conference on Very Large Data Bases*, pages 407-419, September 1995.

[22] Srikant R., and Agrawal R. Mining quantitative association rules in large relational tables. *Proceedings of the 1996 ACM SIGMOD Conference on Management of Data.* Montreal, Canada, June 1996.

[23] Toivonen H. Sampling Large Databases for Association Rules. *Proceedings of the 22nd International Conference on Very Large Databases.* pages 134-145, Mumbai, India, 1996.

[24] Ziarko W. The Discovery, Analysis, and Representation of Data Dependencies in Databases. *Knowledge Discovery in Databases,* 1991.