Mining Association Rules with Item Constraints

Ramakrishnan Srikant and **Quoc Vu** and **Rakesh Agrawal**
IBM Almaden Research Center
650 Harry Road, San Jose, CA 95120, U.S.A.
{srikant,qvu,ragrawal}@almaden.ibm.com

Abstract

The problem of discovering association rules has received considerable research attention and several fast algorithms for mining association rules have been developed. In practice, users are often interested in a subset of association rules. For example, they may only want rules that contain a specific item or rules that contain children of a specific item in a hierarchy. While such constraints can be applied as a post-processing step, integrating them into the mining algorithm can dramatically reduce the execution time. We consider the problem of integrating constraints that are boolean expressions over the presence or absence of items into the association discovery algorithm. We present three integrated algorithms for mining association rules with item constraints and discuss their tradeoffs.

1. Introduction

The problem of discovering association rules was introduced in (Agrawal, Imielinski, & Swami 1993). Given a set of transactions, where each transaction is a set of literals (called items), an association rule is an expression of the form $X \Rightarrow Y$, where X and Y are sets of items. The intuitive meaning of such a rule is that transactions of the database which contain X tend to contain Y. An example of an association rule is: "30% of transactions that contain beer also contain diapers; 2% of all transactions contain both these items". Here 30% is called the *confidence* of the rule, and 2% the *support* of the rule. Both the left hand side and right hand side of the rule can be sets of items. The problem is to find all association rules that satisfy user-specified minimum support and minimum confidence constraints. Applications include discovering affinities for market basket analysis and cross-marketing, catalog design, loss-leader analysis, store layout and customer segmentation based on buying patterns. See (Nearhos, Rothman, & Viveros 1996) for a case study of an application in health insurance, and (Ali, Manganaris, & Srikant 1997) for case studies of applications in predicting telecommunications order failures and medical test results. There has been considerable work on developing fast algorithms for mining association rules, including (Agrawal *et al.* 1996) (Savasere, Omiecinski, & Navathe 1995) (Toivonen 1996) (Agrawal & Shafer 1996) (Han, Karypis, & Kumar 1997).

Taxonomies (*is-a* hierarchies) over the items are often available. An example of a taxonomy is shown in Figure 1. This taxonomy says that Jacket *is-a* Outerwear, Ski Pants *is-a* Outerwear, Outerwear *is-a* Clothes, etc. When taxonomies are present, users are usually interested in generating rules that span different levels of the taxonomy. For example, we may infer a rule that people who buy Outerwear tend to buy Hiking Boots from the fact that people bought Jackets with Hiking Boots and Ski Pants with Hiking Boots. This generalization of association rules and algorithms for finding such rules are described in (Srikant & Agrawal 1995) (Han & Fu 1995).

In practice, users are often interested only in a subset of associations, for instance, those containing at least one item from a user-defined subset of items. When taxonomies are present, this set of items may be specified using the taxonomy, e.g. all descendants of a given item. While the output of current algorithms can be filtered out in a post-processing step, it is much more efficient to incorporate such constraints into the associations discovery algorithm. In this paper, we consider constraints that are boolean expressions over the presence or absence of items in the rules. When taxonomies are present, we allow the elements of the boolean expression to be of the form ancestors(item) or descen-

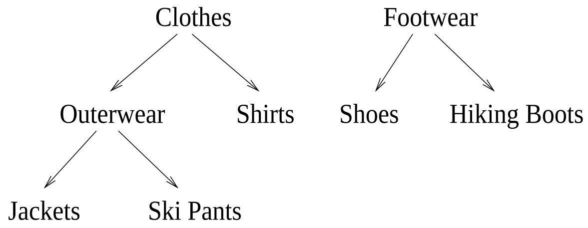

Figure 1: Example of a Taxonomy

dants(item) rather than just a single item. For example,

(Jacket $\wedge$ Shoes) $\vee$ (`descendants`(Clothes) $\wedge$ $\neg$ `ancestors`(Hiking Boots))

expresses the constraint that we want any rules that either (a) contain both Jackets and Shoes, or (b) contain Clothes or any descendants of clothes and do not contain Hiking Boots or Footwear.

Paper Organization We give a formal description of the problem in Section 2. Next, we review the Apriori algorithm (Agrawal *et al.* 1996) for mining association rules in Section 3. We use this algorithm as the basis for presenting the three integrated algorithms for mining associations with item constraints in Section 4. However, our techniques apply to other algorithms that use apriori candidate generation, including the recently published (Toivonen 1996). We discuss the tradeoffs between the algorithms in Section 5, and conclude with a summary in Section 6.

2. Problem Statement

Let $\mathcal{L} = \{l_1, l_2, \ldots, l_m\}$ be a set of literals, called items. Let $\mathcal{G}$ be a directed acyclic graph on the literals. An edge in $\mathcal{G}$ represents an *is-a* relationship, and $\mathcal{G}$ represents a set of taxonomies. If there is an edge in $\mathcal{G}$ from p to c, we call p a *parent* of c and c a *child* of p (p represents a generalization of c.) We call x an *ancestor* of y (and y a *descendant* of x) if there is a directed path from x to y in $\mathcal{G}$.

Let $\mathcal{D}$ be a set of transactions, where each transaction T is a set of items such that $T \subseteq \mathcal{L}$. We say that a transaction T *supports* an item $x \in \mathcal{L}$ if x is in T or x is an ancestor of some item in T. We say that a transaction T *supports* an itemset $X \subseteq \mathcal{L}$ if T supports every item in the set X.

A *generalized association rule* is an implication of the form $X \Rightarrow Y$, where $X \subset \mathcal{L}$, $Y \subset \mathcal{L}$, $X \cap Y = \emptyset$.[1] The rule $X \Rightarrow Y$ holds in the transaction set $\mathcal{D}$ with *confidence c* if $c\%$ of transactions in $\mathcal{D}$ that support X also support Y. The rule $X \Rightarrow Y$ has *support s* in the transaction set $\mathcal{D}$ if $s\%$ of transactions in $\mathcal{D}$ support $X \cup Y$.

Let $\mathcal{B}$ be a boolean expression over $\mathcal{L}$. We assume without loss of generality that $\mathcal{B}$ is in *disjunctive normal form* (DNF).[2] That is, $\mathcal{B}$ is of the form $D_1 \vee D_2 \vee \ldots \vee D_m$, where each *disjunct* D_i is of the form $\alpha_{i1} \wedge \alpha_{i2} \wedge \ldots \wedge \alpha_{in_i}$. When there are no taxonomies present, each element α_{ij} is either l_{ij} or $\neg l_{ij}$ for some $l_{ij} \in \mathcal{L}$. When a taxonomy $\mathcal{G}$ is present, α_{ij} can also be `ancestor`(l_{ij}), `descendant`(l_{ij}), $\neg$ `ancestor`(l_{ij}), or $\neg$ `descendant`(l_{ij}). There is no bound on the number of ancestors or descendants that can be included. To evaluate $\mathcal{B}$, we implicitly replace `descendant`(l_{ij}) by $l_{ij} \vee l'_{ij} \vee l''_{ij} \vee \ldots$, and $\neg$ `descendant`(l_{ij}) by $\neg(l_{ij} \vee l'_{ij} \vee l''_{ij} \vee \ldots)$, where $l'_{ij}, l''_{ij}, \ldots$ are the descendants of l_{ij}. We perform a similar operation for `ancestor`. To evaluate $\mathcal{B}$ over a rule $X \Rightarrow Y$, we consider all items that appear in $X \Rightarrow Y$ to have a value `true` in $\mathcal{B}$ and all other items to have a value `false`.

Given a set of transactions $\mathcal{D}$, a set of taxonomies $\mathcal{G}$ and a boolean expression $\mathcal{B}$, the problem of mining association rules with item constraints is to discover all rules that satisfy $\mathcal{B}$ and have support and confidence greater than or equal to the user-specified minimum support and minimum confidence respectively.

3. Review of Apriori Algorithm

The problem of mining association rules can be decomposed into two subproblems:

- Find all combinations of items whose support is greater than minimum support. Call those combinations *frequent* itemsets.
- Use the frequent itemsets to generate the desired rules. The general idea is that if, say, $ABCD$ and AB are frequent itemsets, then we can determine if the rule $AB \Rightarrow CD$ holds by computing the ratio r = support($ABCD$)/support(AB). The rule holds only if $r \geq$ minimum confidence. Note that the rule will have minimum support because $ABCD$ is frequent.

We now present the Apriori algorithm for finding all frequent itemsets (Agrawal *et al.* 1996). We will use this algorithm as the basis for our presentation. Let k-itemset denote an itemset having k items. Let L_k represent the set of frequent k-itemsets, and C_k the set of candidate k-itemsets (potentially frequent itemsets). The algorithm makes multiple passes over the data. Each pass consists of two phases. First, the set of all frequent $(k-1)$-itemsets, L_{k-1}, found in the $(k–1)$th pass, is used to generate the candidate itemsets C_k. The candidate generation procedure ensures that C_k is a superset of the set of all frequent k-itemsets. The algorithm now scans the data. For each record, it determines which of the candidates in C_k are contained in the record using a hash-tree data structure and increments their support count. At the end of the pass, C_k is examined to determine which of the candidates are frequent, yielding L_k. The algorithm terminates when L_k becomes empty.

Candidate Generation Given L_k, the set of all frequent k-itemsets, the candidate generation procedure returns a superset of the set of all frequent $(k+1)$-itemsets. We assume that the items in an itemset are lexicographically ordered. The intuition behind this procedure is that all subsets of a frequent itemset are also frequent. The function works as follows. First, in the *join* step, L_k is joined with itself:

[1] Usually, we also impose the condition that no item in Y should be an ancestor of any item in X. Such a rule would have the same support and confidence as the rule without the ancestor in Y, and is hence redundant.

[2] Any boolean expression can be converted to a DNF expression.

```
insert into C_{k+1}
select p.item_1, p.item_2, ..., p.item_k, q.item_k
from L_k p, L_k q
where p.item_1 = q.item_1, ..., p.item_{k-1} = q.item_{k-1},
      p.item_k < q.item_k;
```

Next, in the *prune* step, all itemsets $c \in C_{k+1}$, where some k-subset of c is not in L_k, are deleted. A proof of correctness of the candidate generation procedure is given in (Agrawal *et al.* 1996).

We illustrate the above steps with an example. Let L_3 be $\{\{1\ 2\ 3\}, \{1\ 2\ 4\}, \{1\ 3\ 4\}, \{1\ 3\ 5\}, \{2\ 3\ 4\}\}$. After the join step, C_4 will be $\{\{1\ 2\ 3\ 4\}, \{1\ 3\ 4\ 5\}\}$. The prune step will delete the itemset $\{1\ 3\ 4\ 5\}$ because the subset $\{1\ 4\ 5\}$ is not in L_3. We will then be left with only $\{1\ 2\ 3\ 4\}$ in C_4.

Notice that this procedure is no longer complete when item constraints are present: some candidates that are frequent will not be generated. For example, let the item constraint be that we want rules that contain the item 2, and let $L_2 = \{\ \{1\ 2\},\ \{2\ 3\}\ \}$. For the Apriori join step to generate $\{1\ 2\ 3\}$ as a candidate, both $\{1\ 2\}$ and $\{1\ 3\}$ must be present – but $\{1\ 3\}$ does not contain 2 and will not be counted in the second pass. We discuss various algorithms for candidate generation in the presence of constraints in the next section.

4. Algorithms

We first present the algorithms without considering taxonomies over the items in Sections 4.1 and 4.2, and then discuss taxonomies in Section 4.3. We split the problem into three phases:

- **Phase 1** Find all frequent itemsets (itemsets whose support is greater than minimum support) that satisfy the boolean expression $\mathcal{B}$. Recall that there are two types of operations used for this problem: candidate generation and counting support. The techniques for counting the support of candidates remain unchanged. However, as mentioned above, the apriori candidate generation procedure will no longer generate all the potentially frequent itemsets as candidates when item constraints are present.

  We consider three different approaches to this problem. The first two approaches, "MultipleJoins" and "Reorder", share the following approach (Section 4.1):

  1. Generate a set of *selected items* $\mathcal{S}$ such that any itemset that satisfies $\mathcal{B}$ will contain at least one selected item.
  2. Modify the candidate generation procedure to only count candidates that contain selected items.
  3. Discard frequent itemsets that do not satisfy $\mathcal{B}$.

  The third approach, "Direct" directly uses the boolean expression $\mathcal{B}$ to modify the candidate generation procedure so that only candidates that satisfy $\mathcal{B}$ are counted (Section 4.2).

- **Phase 2** To generate rules from these frequent itemsets, we also need to find the support of all subsets of frequent itemsets that do not satisfy $\mathcal{B}$. Recall that to generate a rule $AB \Rightarrow CD$, we need the support of AB to find the confidence of the rule. However, AB may not satisfy $\mathcal{B}$ and hence may not have been counted in Phase 1. So we generate all subsets of the frequent itemsets found in Phase 1, and then make an extra pass over the dataset to count the support of those subsets that are not present in the output of Phase 1.

- **Phase 3** Generate rules from the frequent itemsets found in Phase 1, using the frequent itemsets found in Phases 1 and 2 to compute confidences, as in the Apriori algorithm.

We discuss next the techniques for finding frequent itemsets that satisfy $\mathcal{B}$ (Phase 1). The algorithms use the notation in Figure 2.

4.1 Approaches using Selected Items

Generating Selected Items Recall the boolean expression $\mathcal{B} = D_1 \vee D_2 \vee \ldots \vee D_m$, where $D_i = \alpha_{i1} \wedge \alpha_{i2} \wedge \ldots \wedge \alpha_{in_i}$ and each element α_{ij} is either l_{ij} or $\neg l_{ij}$, for some $l_{ij} \in \mathcal{L}$. We want to generate a set of items $\mathcal{S}$ such that any itemset that satisfies $\mathcal{B}$ will contain at least one item from $\mathcal{S}$. For example, let the set of items $\mathcal{L} = \{1, 2, 3, 4, 5\}$. Consider $\mathcal{B} = (1 \wedge 2) \vee 3$. The sets $\{1, 3\}$, $\{2, 3\}$ and $\{1, 2, 3, 4, 5\}$ all have the property that any (non-empty) itemset that satisfies $\mathcal{B}$ will contain an item from this set. If $\mathcal{B} = (1 \wedge 2) \vee \neg 3$, the set $\{1, 2, 4, 5\}$ has this property. Note that the inverse does not hold: there are many itemsets that contain an item from $\mathcal{S}$ but do not satisfy $\mathcal{B}$.

For a given expression $\mathcal{B}$, there may be many different sets $\mathcal{S}$ such that any itemset that satisfies $\mathcal{B}$ contains an item from $\mathcal{S}$. We would like to choose a set of items for $\mathcal{S}$ so that the sum of the supports of items in $\mathcal{S}$ is minimized. The intuition is that the sum of the supports of the items is correlated with the sum of the supports of the frequent itemsets that contain these items, which is correlated with the execution time.

We now show that we can generate $\mathcal{S}$ by choosing one element α_{ij} from each disjunct D_i in $\mathcal{B}$, and adding either l_{ij} or all the elements in $\mathcal{L} - \{l_{ij}\}$ to $\mathcal{S}$, based on whether α_{ij} is l_{ij} or $\neg l_{ij}$ respectively. We define an element $\alpha_{ij} = l_{ij}$ in $\mathcal{B}$ to be "present" in $\mathcal{S}$ if $l_{ij} \in \mathcal{S}$ and an element $\alpha_{ij} = \neg l_{ij}$ to be "present" if all the items in $\mathcal{L} - \{l_{ij}\}$ are in $\mathcal{S}$. Then:

Lemma 1 *Let $\mathcal{S}$ be a set of items such that*

$$\forall D_i \in \mathcal{B}\ \exists\, \alpha_{ij} \in D_i \quad [(\alpha_{ij} = l_{ij} \wedge l_{ij} \in \mathcal{S}) \vee (\alpha_{ij} = \neg l_{ij} \wedge (\mathcal{L} - \{l_{ij}\}) \subseteq \mathcal{S})].$$

Then any (non-empty) itemset that satisfies $\mathcal{B}$ will contain an item in $\mathcal{S}$.

Proof: Let X be an itemset that satisfies $\mathcal{B}$. Since X satisfies $\mathcal{B}$, there exists some $D_i \in \mathcal{B}$ that is `true` for X. From the lemma statement, there exists some $\alpha_{ij} \in D_i$

$\mathcal{B}$	$= D_1 \vee D_2 \vee \ldots \vee D_m$ (m disjuncts)
D_i	$= \alpha_{i1} \wedge \alpha_{i2} \wedge \ldots \wedge \alpha_{in_i}$ (n_i conjuncts in D_i)
α_{ij}	is either l_{ij} or $\neg l_{ij}$, for some item $l_{ij} \in \mathcal{L}$
$\mathcal{S}$	Set of items such that any itemset that satisfies $\mathcal{B}$ contains an item from $\mathcal{S}$
selected itemset	An itemset that contains an item in $\mathcal{S}$
k-itemset	An itemset with k items.
L_k^s	Set of frequent k-itemsets (those with minimum support) that contain an item in $\mathcal{S}$
L_k^b	Set of frequent k-itemsets (those with minimum support) that satisfy $\mathcal{B}$
C_k^s	Set of candidate k-itemsets (potentially frequent itemsets) that contain an item in $\mathcal{S}$
C_k^b	Set of candidate k-itemsets (potentially frequent itemsets) that satisfy $\mathcal{B}$
F	Set of all frequent items

Figure 2: Notation for Algorithms

such that either $\alpha_{ij} = l_{ij}$ and $l_{ij} \in \mathcal{S}$ or $\alpha_{ij} = \neg l_{ij}$ and $(\mathcal{L} - \{l_{ij}\}) \subseteq \mathcal{S}$. If the former, we are done: since D_i is `true` for X, $l_{ij} \in X$. If the latter, X must contain some item in in $\mathcal{L} - \{l_{ij}\}$ since X does not contain l_{ij} and X is not an empty set. Since $(\mathcal{L} - \{l_{ij}\}) \subseteq \mathcal{S}$, X contains an item from $\mathcal{S}$. □

A naive optimal algorithm for computing the set of elements in $\mathcal{S}$ such that support($\mathcal{S}$) is minimum would require $\prod_{i=1}^{m} n_i$ time, where n_i is the number of conjuncts in the disjunct D_i. An alternative is the following greedy algorithm which requires $\sum_{i=1}^{m} n_i$ time and is optimal if no literal is present more than once in $\mathcal{B}$. We define $\mathcal{S} \cup \alpha_{ij}$ to be $\mathcal{S} \cup l_{ij}$ if $\alpha_{ij} = l_{ij}$ and $\mathcal{S} \cup (\mathcal{L} - \{l_{ij}\})$ if $\alpha_{ij} = \neg l_{ij}$.

```
S := ∅;
for i := 1 to m do begin  // B = D_1 ∨ D_2 ∨ ... ∨ D_m
   for j := 1 to n_i do  // D_i = α_i1 ∧ α_i2 ∧ ... ∧ α_in_i
      Cost(α_ij) := support(S ∪ α_ij) - support(S);
   Let α_ip be the α_ij with the minimum cost.
   S := S ∪ α_ip;
end
```

Consider a boolean expression $\mathcal{B}$ where the same literal is present in different disjuncts. For example, let $\mathcal{B} = (1 \wedge 2) \vee (1 \wedge 3)$. Assume 1 has higher support than 2 or 3. Then the greedy algorithm will generate $\mathcal{S} = \{2, 3\}$ whereas $\mathcal{S} = \{1\}$ is optimal. A partial fix for this problem would be to add the following check. For each literal l_{ij} that is present in different disjuncts, we add l_{ij} to $\mathcal{S}$ and remove any redundant elements from $\mathcal{S}$, if such an operation would decrease the support of $\mathcal{S}$. If there are no overlapping duplicates (two duplicated literals in the same disjunct), this will result in the optimal set of items. When there are overlapping duplicates, e.g, $(1 \wedge 2) \vee (1 \wedge 3) \vee (3 \wedge 4)$, the algorithm may choose $\{1, 4\}$ even if $\{2, 3\}$ is optimal.

Next, we consider the problem of generating only those candidates that contain an item in $\mathcal{S}$.

Candidate Generation Given L_k^s, the set of all selected frequent k-itemsets, the candidate generation procedure must return a superset of the set of all selected frequent $(k+1)$-itemsets.

Recall that unlike in the Apriori algorithm, not all subsets of candidates in C_{k+1}^s will be in L_k^s. While all subsets of a frequent selected itemset are frequent, they may not be selected itemsets. Hence the join procedure of the Apriori algorithm will not generate all the candidates.

To generate C_2^s we simply take $L_1^s \times F$, where F is the set of all frequent items. For subsequent passes, one solution would be to join any two elements of L_k^s that have $k-1$ items in common. For any selected k-itemset where $k > 2$, there will be at least 2 subsets with a selected item: hence this join will generate all the candidates. However, each $k+1$-candidate may have up to k frequent selected k-subsets and $k(k-1)$ pairs of frequent k-subsets with $k-1$ common items. Hence this solution can be quite expensive if there are a large number of itemsets in L_k^s.

We now present two more efficient approaches.

Algorithm MultipleJoins The following lemma presents the intuition behind the algorithm. The itemset X in the lemma corresponds to a candidate that we need to generate. Recall that the items in an itemset are lexicographically ordered.

Lemma 2 *Let X be a frequent $(k+1)$-itemset, $k \geq 2$.*

A. *If X has a selected item in the first $k-1$ items, then there exist two frequent selected k-subsets of X with the same first $k-1$ items as X.*

B. *If X has a selected item in the last* `min`*$(k-1, 2)$ items, then there exist two frequent selected k-subsets of X with the same last $k-1$ items as X.*

C. *If X is a 3-itemset and the second item is a selected item, then there exist two frequent selected 2-subsets of X, Y and Z, such that the last item of Y is the second item of X and the first item of Z is the second item of X.*

For example, consider the frequent 4-itemset {1 2 3 4}. If either 1 or 2 is selected, {1 2 3} and {1 2 4} are two subsets with the same first 2 items. If either 3 or 4 is selected, {2 3 4} and {1 3 4} are two subsets with the same last 2 items. For a frequent 3-itemset {1 2 3} where 2 is the only selected item, {1 2} and {2 3} are the only two frequent selected subsets.

Generating an efficient join algorithm is now straightforward: Joins 1 through 3 below correspond directly to the three cases in the lemma. Consider a candidate $(k+1)$-itemset X, $k \geq 2$. In the first case, Join 1 below will generate X. (Join 1 is similar to the join step of the Apriori algorithm, except that it is performed on a subset of the itemsets in L_2^s.) In the second case, Join 2 will generate X. When $k \geq 3$, we have covered all possible locations for a selected item in X. But when $k = 2$, we also need Join 3 for the case where the selected item in X is the second item. Figure 3 illustrates this algorithm for $\mathcal{S} = \{2, 4\}$ and an L_2^s with 4 itemsets.

// **Join 1**
$L_k^{s\,\prime} := \{p \in L_k^s \mid$ one of the first $k-1$ items of p is in $\mathcal{S}\}$
`insert into` C_{k+1}^s
`select` $p.\text{item}_1, p.\text{item}_2, \ldots, p.\text{item}_k, q.\text{item}_k$
`from` $L_k^{s\,\prime}\ p,\ L_k^{s\,\prime}\ q$
`where` $(p.\text{item}_1 = q.\text{item}_1,\ \ldots,\ p.\text{item}_{k-1} = q.\text{item}_{k-1},$
$\quad p.\text{item}_k < q.\text{item}_k)$

// **Join 2**
$L_k^{s\,\prime\prime} := \{p \in L_k^s \mid$ one of the last `min`$(k-1,2)$ items of p is in $\mathcal{S}\}$
`insert into` C_{k+1}^s
`select` $p.\text{item}_1, q.\text{item}_1, q.\text{item}_2, \ldots, q.\text{item}_k$
`from` $L_k^{s\,\prime\prime}\ p,\ L_k^{s\,\prime\prime}\ q$
`where` $(p.\text{item}_1 < q.\text{item}_1,\ p.\text{item}_2 = q.\text{item}_2,\ \ldots,$
$\quad p.\text{item}_k = q.\text{item}_k)$

// **Join 3** $(k = 2)$
`insert into` C_3^s
`select` $q.\text{item}_1, p.\text{item}_1, p.\text{item}_2$
`from` $L_2^{s\,\prime}\ p,\ L_2^{s\,\prime\prime}\ q$
`where` $(q.\text{item}_2 = p.\text{item}_1)$ `and`
$\quad (q.\text{item}_1,\ p.\text{item}_2$ are not selected);

Note that these three joins do not generate any duplicate candidates. The first $k-1$ items of any two candidate resulting from Joins 1 and 2 are different. When $k = 2$, the first and last items of candidates resulting from Join 3 are not selected, while the first item is selected for candidates resulting from Join 1 and the last item is selected for candidates resulting from Join 1. Hence the results of Join 3 do not overlap with either Join 1 or Join 2.

In the prune step, we drop candidates with a selected subset that is not present in L_k^s.

Algorithm Reorder As before, we generate C_2^s by taking $L_1^s \times F$. But we use the following lemma to simplify the join step.

Lemma 3 *If the ordering of items in itemsets is such that all items in $\mathcal{S}$ precede all items not in $\mathcal{S}$, the join procedure of the Apriori algorithm applied to L_k^s will generate a superset of L_{k+1}^s.*

$L_2^{s\,\prime}$	$L_2^{s\,\prime\prime}$	Join 1	Join 2	Join 3
{2 3}	{1 2}	{2 3 5}		{1 2 3}
{2 5}	{1 4}		{1 3 4}	{1 2 5}
	{3 4}			

Figure 3: MultipleJoins Example

L_2^s	Join 1
{2 1}	{2 1 3}
{2 3}	{2 1 5}
{2 5}	{2 3 5}
{4 1}	{4 1 3}
{4 3}	

Figure 4: Reorder Example

The intuition behind this lemma is that the first item of any frequent selected itemset is *always* a selected item. Hence for any $(k+1)$-candidate X, there exist two frequent selected k-subsets of X with the same first $k-1$ items as X. Figure 4 shows the same example as shown in Figure 3, but with the items in $\mathcal{S}$, 2 and 4, ordered before the other items, and with the Apriori join step.

Hence instead of using the lexicographic ordering of items in an itemset, we impose the following ordering. All items in $\mathcal{S}$ precede all items not in $\mathcal{S}$; the lexicographic ordering is used when two items are both in $\mathcal{S}$ or both not in $\mathcal{S}$. An efficient implementation of an association rule algorithm would map strings to integers, rather than keep them as strings in the internal data structures. This mapping can be re-ordered so that all the frequent selected items get lower numbers than other items. After all the frequent itemsets have been found, the strings can be re-mapped to their original values. One drawback of this approach is that this re-ordering has to be done at several points in the code, including the mapping from strings to integers and the data structures that represent the taxonomies.

4.2 Algorithm Direct

Instead of first generating a set of selected items $\mathcal{S}$ from $\mathcal{B}$, finding all frequent itemsets that contain one or more items from $\mathcal{S}$ and then applying $\mathcal{B}$ to filter the frequent itemsets, we can directly use $\mathcal{B}$ in the candidate generation procedure. We first make a pass over the data to find the set of the frequent items F. L_1^b is now the set of those frequent 1-itemsets that satisfy $\mathcal{B}$. The intuition behind the candidate generation procedure is given in the following lemma.

Lemma 4 *For any $(k+1)$-itemset X which satisfies $\mathcal{B}$, there exists at least one k-subset that satisfies $\mathcal{B}$ unless each D_i which is true on X has exactly $k+1$ non-negated elements.*

We generate C^b_{k+1} from L^b_k in 4 steps:

1. $C^b_{k+1} := L^b_k \times F$;
2. Delete all candidates in C^b_{k+1} that do not not satisfy $\mathcal{B}$;
3. Delete all candidates in C^b_{k+1} with a k-subset that satisfies $\mathcal{B}$ but does not have minimum support.
4. For each disjunct $D_i = \alpha_{i1} \wedge \alpha_{i2} \wedge \ldots \wedge \alpha_{in_i}$ in $\mathcal{B}$ with exactly $k+1$ non-negated elements $\alpha_{ip_1}, \alpha_{ip_2}, \ldots, \alpha_{ip_{k+1}}$, add the itemset $\{\alpha_{ip_1}\alpha_{ip_2}\ldots, \alpha_{ip_{k+1}}\}$ to C^b_{k+1} if all the α_{ip_j}s are frequent,

For example, let $\mathcal{L} = \{1, 2, 3, 4, 5\}$ and $\mathcal{B} = (1 \wedge 2) \vee (4 \wedge \neg 5))$. Assume all the items are frequent. Then $L^b_1 = \{\{4\}\}$. To generate C^b_2, we first take $L^b_1 \times F$ to get { {1 4}, {2 4}, {3 4}, {4 5} }. Since {4 5} does not satisfy $\mathcal{B}$, it is dropped. Step 3 does not change C^b_2 since all 1-subsets that satisfy $\mathcal{B}$ are frequent. Finally, we add {1 2} to C^b_2 to get {{1 2}, {1 4}, {2 4}, {3 4}}.

4.3 Taxonomies

The enhancements to the Apriori algorithm for integrating item constraints apply directly to the algorithms for mining association rules with taxonomies given in (Srikant & Agrawal 1995). We discuss the Cumulate algorithm here.[3] This algorithm adds all ancestors of each item in the transaction to the transaction, and then runs the Apriori algorithm over these "extended transactions". For example, using the taxonomy in Figure 1, a transaction {Jackets, Shoes} would be replaced with {Jackets, Outerwear, Clothes, Shoes, Footwear}. Cumulate also performs several optimization, including adding only ancestors which are present in one or more candidates to the extended transaction and not counting any itemset which includes both an item and its ancestor.

Since the basic structure and operations of Cumulate are similar to those of Apriori, we almost get taxonomies for "free". Generating the set of selected items, $\mathcal{S}$ is more expensive since for elements in $\mathcal{B}$ that include an `ancestor` or `descendant` function, we also need to find the support of the ancestors or descendants. Checking whether an itemset satisfies $\mathcal{B}$ is also more expensive since we may need to traverse the hierarchy to find whether one item is an ancestor of another.

Cumulate does not count any candidates with both an item and its ancestor since the support of such an itemset would be the same as the support of the itemset without the ancestor. Cumulate only checks for such candidates during the second pass (candidates of size 2). For subsequent passes, the apriori candidate generation procedure ensures that no candidate that contains both an item and its ancestor will be generated. For example, an itemset {Jacket Outerwear Shoes} would not be generated in C_3 because {Jacket Outerwear} would have been deleted from L_2. However, this property does not hold when item constraints are specified. In this case, we need to check each candidate (in every pass) to ensure that there are no candidates that contain both an item and its ancestor.

[3] The other fast algorithm in (Srikant & Agrawal 1995), EstMerge, is similar to Cumulate, but also uses sampling to decrease the number of candidates that are counted.

5. Tradeoffs

Reorder and MultipleJoins will have similar performance since they count exactly the same set of candidates. Reorder can be a little faster during the prune step of the candidate generation, since checking whether an k-subset contains a selected item takes O(1) time for Reorder versus O(k) time for MultipleJoins. However, if most itemsets are small, this difference in time will not be significant. Execution times are typically dominated by the time to count support of candidates rather than candidate generation. Hence the slight differences in performance between Reorder and MultipleJoins are not enough to justify choosing one over the other purely on performance grounds. The choice is to be made on whichever one is easier to implement.

Direct has quite different properties than Reorder and MultipleJoins. We illustrate the tradeoffs between Reorder/MultipleJoins and Direct with an example. We use "Reorder" to characterize both Reorder and MultipleJoins in the rest of this comparison. Let $\mathcal{B} = 1 \wedge 2$ and $\mathcal{S} = \{1\}$. Assume the 1-itemsets {1} through {100} are frequent, the 2-itemsets {1 2} through {1 5} are frequent, and no 3-itemsets are frequent. Reorder will count the ninety-nine 2-itemsets {1 2} through {1 100}, find that {1 2} through {1 5} are frequent, count the six 3-itemsets {1 2 3} through {1 4 5}, and stop. Direct will count {1 2} and the ninety-eight 3-itemsets {1 2 3} through {1 2 100}. Reorder counts a total of 101 itemsets versus 99 for Direct, but most of those itemsets are 2-itemsets versus 3-itemsets for Direct.

If the minimum support was lower and {1 2} through {1 20} were frequent, Reorder will count an additional 165 ($19 \times 18/2 - 6$) candidates in the third pass. Reorder can prune more candidates than Direct in the fourth and later passes since it has more information about which 3-itemsets are frequent. For example, Reorder can prune the candidate {1 2 3 4} if {1 3 4} was not frequent, whereas Direct never counted {1 3 4}. On the other hand, Direct will only count 4-candidates that satisfy $\mathcal{B}$ while Reorder will count any 4-candidates that include 1.

If $\mathcal{B}$ were "$1 \wedge 2 \wedge 3$" rather than "$1 \wedge 2$", the gap in the number of candidates widens a little further. Through the fourth pass, Direct will count 98 candidates: {1 2 3} and {1 2 3 4} through {1 2 3 100}. For the minimum support level in the previous paragraph, Reorder will count 99 candidates in the second pass, 171 candidates in the third pass, and if {1 2 3} through {1 5 6} were frequent candidates, 10 candidates in the fourth pass, for a total of 181 candidates.

Direct will not always count fewer candidates than Reorder. Let $\mathcal{B}$ be "$(1 \wedge 2 \wedge 3) \vee (1 \wedge 4 \wedge 5)$" and $\mathcal{S}$ be {1}. Let items 1 through 100, as well as {1 2 3}, {1 4 5} and their subsets be the only frequent sets. Then Reorder will count around a hundred candidates while Direct will count around two hundred.

In general, we expect Direct to count fewer candidates than Reorder at low minimum supports. But the candidate generation process will be significantly more expensive for Direct, since each subset must be checked against a (potentially complex) boolean expression in the prune phase. Hence Direct may be better at lower minimum supports or larger datasets, and Reorder for higher minimum supports or smaller datasets. Further work is needed to analytically characterize these trade-offs and empirically verify them.

6. Conclusions

We considered the problem of discovering association rules in the presence of constraints that are boolean expressions over the presence of absence of items. Such constraints allow users to specify the subset of rules that they are interested in. While such constraints can be applied as a post-processing step, integrating them into the mining algorithm can dramatically reduce the execution time. We presented three such integrated algorithm, and discussed the tradeoffs between them. Empirical evaluation of the MultipleJoins algorithm on three real-life datasets showed that integrating item constraints can speed up the algorithm by a factor of 5 to 20 for item constraints with selectivity between 0.1 and 0.01.

Although we restricted our discussion to the Apriori algorithm, these ideas apply to other algorithms that use apriori candidate generation, including the recent (Toivonen 1996). The main idea in (Toivonen 1996) is to first run Apriori on a sample of the data to find itemsets that are expected to be frequent, or all of whose subsets are are expected to be frequent. (We also need to count the latter to ensure that no frequent itemsets were missed.) These itemsets are then counted over the complete dataset. Our ideas can be directly applied to the first part of the algorithm: those itemsets counted by Reorder or Direct over the sample would be counted over the entire dataset. For candidates that were not frequent in the sample but were frequent in the datasets, only those extensions of such candidates that satisfied those constraints would be counted in the additional pass.

References

Agrawal, R., and Shafer, J. 1996. Parallel mining of association rules. *IEEE Transactions on Knowledge and Data Engineering* 8(6).

Agrawal, R.; Mannila, H.; Srikant, R.; Toivonen, H.; and Verkamo, A. I. 1996. Fast Discovery of Association Rules. In Fayyad, U. M.; Piatetsky-Shapiro, G.; Smyth, P.; and Uthurusamy, R., eds., *Advances in Knowledge Discovery and Data Mining.* AAAI/MIT Press. chapter 12, 307–328.

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining association rules between sets of items in large databases. In *Proc. of the ACM SIGMOD Conference on Management of Data*, 207–216.

Ali, K.; Manganaris, S.; and Srikant, R. 1997. Partial Classification using Association Rules. In *Proc. of the 3rd Int'l Conference on Knowledge Discovery in Databases and Data Mining.*

Han, J., and Fu, Y. 1995. Discovery of multiple-level association rules from large databases. In *Proc. of the 21st Int'l Conference on Very Large Databases.*

Han, E.-H.; Karypis, G.; and Kumar, V. 1997. Scalable parallel data mining for association rules. In *Proc. of the ACM SIGMOD Conference on Management of Data.*

Nearhos, J.; Rothman, M.; and Viveros, M. 1996. Applying data mining techniques to a health insurance information system. In *Proc. of the 22nd Int'l Conference on Very Large Databases.*

Savasere, A.; Omiecinski, E.; and Navathe, S. 1995. An efficient algorithm for mining association rules in large databases. In *Proc. of the VLDB Conference.*

Srikant, R., and Agrawal, R. 1995. Mining Generalized Association Rules. In *Proc. of the 21st Int'l Conference on Very Large Databases.*

Toivonen, H. 1996. Sampling large databases for association rules. In *Proc. of the 22nd Int'l Conference on Very Large Databases*, 134–145.