Journal of Machine Learning Research 2 (2001) Unsure Submitted 30/11; Published To Be Determined

Faster Association Rules for Multiple Relations

Siegfried Nijssen SNIJSSEN@LIACS.NL

Joost Kok JOOST@LIACS.NL

Leiden Institute of Advanced Computer Science
Leiden University
Niels Bohrweg 1, 2333 CA Leiden, The Netherlands

Editor: D. Schuurmans

Abstract

Several notions have been developed which combine association rules with first order logic formulas, such as query flocks and the rules used in WARMR. Although this resulted in usable algorithms, little attention was payed until recently to the efficiency of these algorithms. In this paper we present some ideas to turn one important intermediate step in the process of discovering such rules, i.e., the discovery of an upgraded version of frequent item sets, more efficient. Using an implementation that we coined FARMER, we show that indeed a speed-up is obtained and that the performance is much more comparable to an efficient implementation of a traditional association rule algorithm. It will appear that the set of queries found by our algorithm is not always the same as that of the related WARMR algorithm. We will show for which situations this is the case and will present some restricted versions if one desires equivalence. We will give some hints on how queries can be made more usable by also analysing them for interestingness.

1. Introduction

Back in 1993 association rules were invented as a means of analysing basket databases (Agrawal et al., 1993). An example of a basket database is a supermarket database which for every transaction contains the items which are bought in that transaction. The task of the association rule algorithm is to find relevant dependencies between the items. Part of the success of association rules is the existence of an efficient algorithm, APRIORI, which can be used to perform one step in the discovery process efficient in many situations (Agrawal et al., 1996).

As appealing as association rules may initially appear, their applicability is limited in several ways. Much research has been done to overcome these limitations. We focus at one limitation of traditional association rules: their inability to deal concisely with *multi-relational* databases, which are databases with more than one table (or: *relation* in database terminology). Several approaches have been taken to deal with such databases. One approach is to translate the problem into a traditional association rule problem. However, it can be shown that this transformation is not possible without losing some information. Another approach, which does not suffer from this problem, is to use complete first-order logic formulas to denote upgraded association rules. This approach was taken in the WARMR algorithm (Dehaspe and De Raedt, 1997). Also this approach faces some problems. The most

©2001 Siegfried Nijssen.

row	items
1	{1,2,3}
2	{1,2,3,4}
3	{1,4}

Table 1: Example for APRIORI input

important problem is its inefficiency. The algorithm evaluates all formulas using PROLOG, and, most important of all, checks that formulas are the same using a mechanism called *θ-subsumption*. This mechanism is known to be NP complete (Kietz and Lübbe, 1994).

The contribution of our work consists of the development of a WARMR-like algorithm that we called FARMER (*Faster Association Rules for MultiplE Relations*). It does not use PROLOG and does not use θ-subsumption. In this way, we remove most of the efficiency problems raised by WARMR. The set of rules which is yielded by our algorithm is not always the same as the set found by WARMR; we will prove in which restricted cases our algorithm does behave the same. As any restriction to the general paradigm of WARMR could be undesirable, we investigate several variants of our algorithm; these variants remove restrictions, but also impose other (new) restrictions. We believe that in many cases, one of these variants is sufficient to solve the problem of finding rules.

One of the problems of finding association rules in general is the large number of rules which are found. Many of them are not interesting if they are considered from a statistical point of view. The output of FARMER suffers from the same problem. We will give some hints here on the filtering of statistically interesting rules. This shows how ideas originating from other association rule research can easily be adapted to the multirelational FARMER setup.

Our paper is organized as follows. Section 2 serves the purpose of making the paper self-contained. It provides background information on efficient association rule algorithms (Section 2.1); it describes techniques that have been developed to filter for statistically interesting rules (Section 2.2) and reviews all relevant information about the WARMR algorithm (Section 2.3). In Section 3 we will introduce the core of our FARMER algorithm. In Section 4 we will prove in several steps in how far FARMER is comparable to WARMR. Section 4.3 discusses some variants of FARMER; here we will use some of the observations made during our proofs. Section 4.4 provides some ideas on the incorporation of statistical tests. In Section 5 experiments show that our goal of obtaining efficiency has been reached. Section 6 concludes and gives some hints on further work.

2. Association Rules

The framework we will be building on is the framework of *association rules.* The original notion was introduced by Agrawal et al. (1993, 1996) for the purpose of basket analysis. Given is a two dimensional table in which each row consists of a set of items. An example of this is given in Table 1.

Common origins for these kind of tables include:

- a row is a client of a supermarket, the items are the products in the supermarket which the client has bought since the database was started;

- a row is a transaction of a supermarket, the items are the products in the supermarket that are bought together in that particular transaction.

From the latter interpretation most technical terms have been derived. The rows are usually called *transactions* in association rule theory. A set of products is called an *item set* (usually denoted by I). The number of transactions in which a certain item set is contained, is called the *support* of that item set (denoted by $\text{sup}(I)$). *Frequent* item sets (or: *large* item sets) are item sets for which $\text{sup}(I) \geq minsup$, where $minsup$ is a predefined threshold value.

Example 1 In Table 1, $\{1, 2, 3\}$, but also $\{1, 2\}$, is an item set. The support of $\{1, 2\}$ is 2. The support of $\{3, 4\}$ is 1. If $minsup$ is predefined to be 2, then $\{1, 2\}$ is frequent; $\{3, 4\}$ is however infrequent.

A frequent item set can be valuable knowledge. In a supermarket, such a set represents a set of products that are bought together many times. The supermarket could use this knowledge to introduce a shop layout in which clients on their way from one product to another pass profitable products. Another benefit may be drawn by giving clients which almost exactly match a frequent item set a discount on an item presumed to be missing.

The breakthrough presented by Agrawal et al. (1996) was the APRIORI algorithm, which contained a trick to efficiently discover all frequent item sets in a database. Although the number of such sets may be exponential in the number of products, in many cases all sets can be computed using a level-wise algorithm which iterates a process of generating candidate item sets and counting these item sets. In the generating step the set of all frequent item sets of size k is used to create item sets of size $k+1$ which can be frequent. The essential observation is (for item sets I_1 and I_2):

$$I_1 \subseteq I_2 \Rightarrow \text{sup}(I_1) \geq \text{sup}(I_2). \tag{1}$$

A consequence of this observation is that all subsets of a frequent item set of size $k+1$ must also be large. Good candidates at step $k+1$ can only be those sets for which all subsets were found in step k. The reduction of the search space in this way is called *pruning*.

In the counting step, the frequency of all candidate sets of a certain size is counted in the database to determine whether they are really large.

Example 2 We will use the database of Table 1 and threshold *minsup* $= 2$ to demonstrate how frequent item sets are found in a stepwise fashion.

$k = 1$	**Candidates:**	$\{1\}, \{2\}, \{3\}, \{4\}$
	Counted candidates:	$\text{sup}(\{1\}) = 3$, $\text{sup}(\{2\}) = 2$, $\text{sup}(\{3\}) = 2$, $\text{sup}(\{4\}) = 2$.
	Frequent item sets:	$\{1\}, \{2\}, \{3\}, \{4\}$
$k = 2$	**Candidates:**	$\{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}, \{2,4\}, \{3,4\}$ (note that we list each set once)
	Counted candidates:	$\text{sup}(\{1,2\}) = 2$, $\text{sup}(\{1,3\}) = 2$, $\text{sup}(\{1,4\}) = 2$, $\text{sup}(\{2,3\}) = 2$, $\text{sup}(\{2,4\}) = 1$, $\text{sup}(\{3,4\}) = 1$.
	Frequent item sets:	$\{1,2\}, \{1,3\}, \{1,4\}, \{2,3\}$
$k = 3$	**Candidates:**	$\{1,2,3\}$
	Counted candidates:	$\text{sup}(\{1,2,3\}) = 2$
	Frequent item sets:	$\{1,2,3\}$
$k = 4$	**Candidates:**	none, so stop.

As one can see in this example, not all possible item sets are counted (for example, $\{2,3,4\}$ is not), which cuts down the search space in most practical cases drastically.

After the discovery of frequent item sets, Agrawal et al. (1996) propose to process these sets to generate *association rules*. An association rule is a dependency of the form $I_1 \to I_2$, or: **if** items I_1 **then** items I_2. The *confidence* of such a rule is defined as $\text{conf}(I_1 \to I_2) = \text{sup}(I_1 \cup I_2)/\text{sup}(I_1)$. Rules for which $\text{sup}(I_1 \cup I_2) \geq \textit{minsup}$ and $\text{conf}(I_1 \to I_2) \geq \textit{minconf}$ reflect relations which occur in 'many' transactions. Here *minconf* is a predefined threshold, analogous to *minsup*. To efficiently find the set of association rules, a similar pruning algorithm can be used as for frequent item sets. It is not necessary to check for some small antecedents if some larger super set in the antecedent does not yield sufficient confidence.

What we have just discussed are the basic principles of association rules and the process of their discovery. Because of their simplicity, association rules have become very popular. To be of (more) practical use, however, still some questions have to be solved:

- *Efficiency:* How is the APRIORI algorithm efficiently implemented? Are there more efficient algorithms?
- *Usability:* Once rules have been found, how can they be presented in an understandable, clear way?
- *Generality:* How can more complicated data also be explored using APRIORI? Can additional knowledge be used in APRIORI?

We will discuss each of these in the following sections. In this discussion, we will give special attention to those ideas we will exploit in our algorithm.

2.1 Efficiency

The quest for efficiency is generally going into two directions:

- a direction in which one concentrates on the order of item set evaluation and searches efficient, special purpose datastructures to maintain data, counters and candidates;

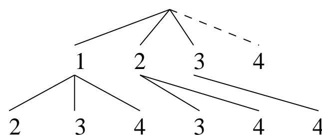


Figure 1: A small set enumeration tree containing all the candidates of size $k = 2$ in Example 2.

- a direction in which one tries to use database management systems (DBMS) as efficiently as possible to search frequent item sets.

An efficient coupling with a DBMS would be very advantageous if one would like to mine data at a very regular basis, as most data are stored in commercial databases. The search for optimization is based on the observation that there is currently no operator in the industry's standard structured query language (SQL) to efficiently evaluate the counts of a set of item sets. Proposed extensions to DBMSs are the MINE RULE operator (Meo et al., 1996, Boulicaut et al., 1998) and query flocks expressed in logic (Tsur et al., 1998). For none of these approaches a mature implementation exists. The common approach taken is not to incorporate the operator within a DBMS, but to preprocess the new operators and translate them into traditional database manipulations, for example using cursors. Such a cursor can be used to traverse a table stored in a DBMS record-by-record.

To gain more insight in the efficiency of DBMS approaches, Sarawagi et al. (1998) have implemented several such approaches. From their experiments, they concluded that the Cache-Mine strategy was the most beneficial (at least twice as fast as the other approaches). In this strategy, the database is scanned only once to build a special-purpose datastructure. The APRIORI algorithm is then applied to it. All approaches which used the DBMS during the run of APRIORI appeared to be inferior in terms of time efficiency. From these experiments, we conclude that a special purpose algorithm currently seems the most realistic approach if one wants to mine large data sets.

Much research has been done in order to develop special purpose, efficient item set evaluation schemes. One approach is to change the order in which item sets are evaluated. For example, Pijls and Bioch (1999) evaluate item sets depth-first, under the assumption that the database fits in main memory; others try to prune some additional item sets using the idea of free sets, at the cost of some precision in the resulting counts (Boulicaut et al., 2000). Minor optimizations of the original APRIORI algorithm are APRIORI-TID (Agrawal et al., 1996), which stores a set of transaction numbers with each counted item set, and a breadth-first algorithm (Pijls and Bioch, 1999, Bayardo et al., 1999) using set enumeration trees (Rymon, 1992). Especially the latter two algorithms appear to have some features which are useful in our multi-relational problem. We will therefore discuss the breadth-first algorithm in more detail here.

The breadth first-algorithm is also a bottom-up level-wise algorithm. It starts with candidates of size one, after which a process is repeated of counting $k-$candidate item sets and of using them to obtain candidate $k+1$ item sets. All these steps are performed on a tree datastructure, of which Figure 1 displays an example. Every path from the root to a

node corresponds to an item set; all leafs at the deepest level correspond to candidate item sets, in this case of the candidates of size $k = 2$ in Example 2. Paths which do not reach the deepest level are maintained only when they correspond to frequent item sets; they are displayed with dotted lines for the sake of clarity. The tree is used in the following fashion:

- In the step of candidate counting, a recursive tree traversal is performed for *every* transaction, as follows. Starting with the top level nodes, one of the following two things happens if the item in the node occurs in the transaction which is currently considered: if the node is internal, the procedure is repeated for all its children; if the node is a leaf, a count is increased which is stored in the node.
- In the step of candidate generation, for every frequent item set new children are generated, consisting of all frequent right brothers. In the example $\{2\}$ is expanded by its frequent right brothers 3 and 4. This copying mechanism in combination with the order of the items takes care of generating every item set at most once. Of every newly generated item set, at least two subsets are already known to be frequent. Frequent item sets for which no children could be generated, are marked.

In both steps, this mechanism distinguishes itself from the original APRIORI algorithm (Agrawal et al., 1996). Instead of building a new tree for each round, this procedure efficiently constructs a new set of candidates by merely copying nodes. Furthermore, during the counting phase, it passes *first* through the tree and *then* checks for the existence of candidates in the current transaction. This is in contrast with the original algorithm, where *first* a subset in the transaction is determined, and *then* a search in a hash node is performed to check whether there is a candidate to be counted. It will appear that these characteristics make this variant of APRIORI suitable for our purposes.

2.2 Usability

A major problem of the APRIORI algorithm is its dependence on thresholds. For large supports, most of the time only trivial dependencies are found; for small supports the number of frequent rules is too large to be useful. Approaches which have been taken to solve this problem are to remove the minimum support constraint (see, for example, Castelo et al., 2001), to visualize the rules (see, for an example in the text mining domain, Wong et al., 1999) or to remove frequent rules using statistical tests. We will consider the latter class of approaches in more detail. A possibility is to exploit additional knowledge. One such situation is the existence of an item hierarchy which describes relations between items, such as "salmon and sardine are both fish" in a supermarket example (Srikant and Agrawal, 1995). Other statistical tests — which are independent of additional knowledge — are *lift*, *conviction*, *interest*, χ^2 *tests*, *IS tests* and *IR tests*. An overview of some of these can be found in (Bayardo and Agrawal, 1999).

The idea of a statistical test is that even if $X \to Y$ occurs with a high support and confidence, this does not mean that X and Y are dependent on each other. For example, the principle of *lift* is that a rule $X \to Y$ with confidence c is not interesting if Y has support c in all transactions. Another measure is the IR test, introduced by Piatetsky-Shapiro (1991):

$$IR(X, Y) = \frac{\sup(X \cup Y)}{|T|} - \frac{\sup(X)}{|T|}\frac{\sup(Y)}{|T|},$$

where $|T| = \text{sup}(\emptyset)$ is the number of transactions in the database. In this formula, $\text{sup}(X)/|T|$ approximates the probability of item set X. Usually, X denotes the antecedent of the association rule; Y denotes the consequent. If $IR \gg 0$ this means that the rule occurs in many more transactions than could be explained by the coincidence of X and Y occurring together. If $IR \ll 0$ the occurrence rate is unexpectedly low.

The main drawback of the IR test with respect to rules is its symmetry: $X \rightarrow Y$ and $Y \rightarrow X$ are treated as the same rule. We choose to focus on the discovery of item sets and not on the rules which may be computed from them. We will conclude this section with a discussion of a method to determine whether item sets are interesting. We will argue that this method may especially be useful in the algorithms we are considering.

A method to determine the interestingness of an item set X can be:

$$X \text{ is interesting} \Leftrightarrow \forall Y, Z \subset X \text{ such that } Y \cup Z = X\text{: } IR(Y, Z) > \varepsilon,$$

for some threshold ε. For this kind of usage of the IR test, it is also useful to define $IR(X, Y)$ if $X \cap Y \neq \emptyset$:

$$\begin{aligned} IR(X,Y) &= \frac{\text{sup}(X \cup Y)}{|T|} - \frac{\text{sup}(X)}{\text{sup}(X \cap Y)} \frac{\text{sup}(Y)}{\text{sup}(X \cap Y)} \frac{\text{sup}(X \cap Y)}{|T|} \\ &= \frac{\text{sup}(X \cup Y)}{|T|} - \frac{\text{sup}(X)\text{sup}(Y)}{|T|\text{sup}(X \cap Y)}. \end{aligned} \tag{2}$$

The ability to incorporate overlapping sets will prove to be useful in our discussion of multi-relational item sets.

To determine whether all subsets obey the mentioned criterion, also some pruning strategy can be applied using this observation:

If for two sets X and Y the following holds:

$$\frac{\text{sup}(X \cup Y)}{|T|} - \frac{\text{sup}(X)}{\text{sup}(X \cap Y)} > \theta, \tag{3}$$

then any subdivision of $X \cup Y$ into two sets $Y \cup Z$ and $X \backslash Z$ (where $Z \subseteq X \backslash Y$) need not be checked as the following can be concluded from Formula 3:

$$\forall Z \subseteq X \backslash Y : IR(X \backslash Z, Y \cup Z) = \frac{\text{sup}(X \cup Y)}{|T|} - \frac{\text{sup}(X \backslash Z)\text{sup}(Y \cup Z)}{|T|\text{sup}(X \cap Y)} > \theta.$$

So, for some subdivision into two sets X and Y (which may overlap), we can sometimes conclude that several other subdivisions need not be checked. Note that this pruning criterion can also be applied with reversed roles for X and Y.

In the worst case, however, the complexity of this search is bad. The number of combinations of subsets X and Y which have to be considered for one item set of size n is $(3^n - 2^{n+1} - 1)/2$ as an item is only in X, only in Y or in both; X and Y may not be complete subsets of each other; X and Y are interchangeable. It is undesirable to apply such a procedure to all item sets.

$I := \cup_{i=1}^{k} I_i$, where I_i contains the frequent item sets of size i
for $i := 2$ **to** k **do**
  for all $X \in I_i$ **do**
    for all $x \in X$ **do**
      if $\sup(X \backslash \{x\}) - \sup(X) < \varepsilon$ **then**
        remove $X \backslash \{x\}$ from I_{i-1}

Figure 2: An algorithm to find maximal item sets.

A solution for this can be found by applying another item set reduction strategy which removes many small item sets. The idea is this: given a set X with support s, any subset with approximate support s is not interesting as this support can be considered as a consequence of the support of the larger set. Building on this idea, a set X is interesting if

$$\forall Y \supset X : \sup(X) - \sup(Y) > \varepsilon.$$

The list of such maximal item sets is easily incrementally computed. An outline of an algorithm is given in Figure 2. If there is an item set Y such that $\sup(X) - \sup(Y) < \varepsilon$, then also $\sup(X) - \sup(X \cup \{e\}) < \varepsilon$ for every element $e \in Y \backslash X$. A variant of this algorithm is obtained by using

$$\forall Y \supset X : \frac{\sup(X)}{\sup(Y)} > \varepsilon \tag{4}$$

as test for interestingness.

This two-step mechanism may allow us to find reasonably large sets which are unexpected.

2.3 Generality

Initially, association rules were meant to be applied to sets of items only, without the incorporation of any additional knowledge. However, it is of course not a good idea to throw away information in order to fit some standard algorithm input. Much work has therefore been done to adapt the original algorithm to practical situations. One such situation is the existence of an item hierarchy (Srikant and Agrawal, 1995). Other algorithms allow for numerical values in a database table by introducing fuzzy association rules (Kuok et al., 1998); also combinations of ideas were examined (De Graaf et al., 2001).

We will focus on another type of problem: given that the database consists of more than one table, is it still possible to find association rules? To clarify this, we will use the example in Figure 3, which is part of a hypothetical database. Apart from the products in the transactions, also a client card identifier is stored; this card can be linked to several users, of which also information is stored. Similar situations may also occur in log-files of servers: one knows the computer performing a request, but one may not have a precise idea which user is performing it.

The most straightforward solution for finding frequent item sets is to map this problem to a usual association rule problem. This is done for example in Hipp et al. (2001): here one first computes the join of all tables, and then adds as products all properties linked to a transaction. In many situations, this could be sufficient. However, one must be aware of

tr_card

tr.	card
1	c1
2	c3
3	c1
4	c2
⋮	⋮

tr_product

tr.	products
1	chips
1	beer
2	rice
3	potatoes
4	cola
⋮	⋮

card_user

card	user
c1	u1
c1	u2
c2	u3
c3	u4
⋮	⋮

user_prop

user	properties
u1	male
u1	youngchildren
u1	football
u2	female
u2	youngchildren
u2	car
u3	boy
⋮	⋮

Figure 3: A multi-relational consumer database

what happens with multiple users of a card in our example. Both properties football and car would be added to transaction 1. One loses the information in the original data that these properties did not belong to the same user. In the sequel, we will only consider algorithms which do not lose this information and do not require duplication of table values. Two major approaches have been proposed for this: *query flocks* (Tsur et al., 1998) and *ARMR*s (association rules for multiple relations, Dehaspe and De Raedt, 1997). We will treat their similarities first, before discussing the differences which lead us to prefer ARMRs here.

Both take Datalog queries as their starting point (Ullman, 1988). Datalog is a restriction of PROLOG to Horn clauses without functions and lists. The following is an example of such a clause, which we will call a *query*:

$$key(X) \leftarrow tr_product(X, chips), tr_card(X, Y), card_user(Y, Z), user_prop(Z, male).$$

Informally, this means

> *key* is true for some given transaction if that transaction contains chips, and there is an associated card having a male user.

More technically, it states that $key(X)$ is true for those values of X for which the right hand side of the arrow can be proven for some variable values of Y and Z. Every predicate refers to a table. To give an example, for a given transaction id 2 which is stored in variable X, atom $tr_product(X, chips)$ is true if there is a record $(2, chips)$ in the *tr_product* table.

Also for queries, it is possible to define a support: the support of the query is the number of assignments to X for which the right hand side of the arrow can be proven. The equivalent for the APRIORI problem of finding frequent item sets is the discovery of queries which are sufficiently true. However, which queries do we want to find? For example, the query

$$key(X) \leftarrow tr_card(X, Y), tr_card_user(Y, u1),$$

which refers to the transactions for user $u1$, may or may not be a desirable result; to reduce the search space, it is preferable that the user has the possibility to specify this. The algorithms for query flocks and ARMRs (the latter being implemented in the WARMR

algorithm) differ in the way the user specifies the queries, and, consequently, they also differ in the way the search is performed.

The following is an example of a query flock:

$$key(X) \leftarrow tr_product(X,\$1), tr_product(X,\$2), \$1 < \$2.$$

The query flock system should return all those combinations of constants which can be substituted for \$1 and \$2 such that the resulting query is frequent. The atom $\$1 < \$2$ takes care of not generating lexicographical equivalents. Except for constants, the resulting queries all have the query flock layout. Still, the APRIORI trick can be applied. If the query can be split up into smaller queries containing less $\$x$ constants, these easier queries can be used to filter combinations of constants: the easier query must be true for all Xs for which the more complex query is true, so no combination of constants can be true for the larger query if it is not true in the smaller query.

Although very appealing in many ways, it turns out that in the query flock specification setup, it is very difficult to mimic the usual item set problem. To find all item sets up to size k, one would have to present k flocks, each of which has $2k$ atoms[1]. It is unclear how to define maximal item sets, and the system is left with finding feasible easy queries.

A different approach is taken by Dehaspe and De Raedt (1997). Instead of decomposing into easy queries, they use a *mode* mechanism which allows to build complex queries from small parts. The specification of a "usual" frequent item set problem would be:

$$key(key((-T))$$
$$mode(1, tr_product(+T, \#))$$

It states that predicate *key* will be the head of a query; the $-$ indicates that the atom introduces a new variable which will be of type T. To the query's body atoms can be added of the *tr_product* predicate; an existing transaction variable must be the first parameter (indicated by $+T$), and a constant must be second parameter (indicated by $\#$). The 1 defines a restriction on the number of times the mode may be used.

In the sequel, we will call this specification the *bias* of the algorithm. The parameters in the declarations, such as $+$, $-$ and $\#$, are called *mode constraints*; with a $+$ we denote an *input* parameter, with a $-$ we denote an *output* parameter[2]. The body of the query will also be referred to as an *atom set*.

An extension of the classical problem as discussed in Section 2 is obtained by adding:

$$mode(1, tr_card(+T, -C))$$
$$mode(1, card_user(+C, -U))$$
$$mode(1, user_prop(+U, \#))$$

Given query

$$key(X) \leftarrow tr_product(X, chips), tr_card(X, Y),$$

the modes allow a *refinement* (an extension of a query with one atom) to the following query:

$$key(X) \quad \leftarrow \quad tr_product(X, chips), tr_card(X, Y), card_user(Y, Z). \tag{5}$$

1. One atom for *tr_product* and one for $\$x < \$y$.
2. A slightly different notation is used in the original publication (Dehaspe and De Raedt, 1997); we believe our notation is more straightforward.

The following refinements are invalid:

- $key(X) \leftarrow tr_product(X, chips), tr_card(X, Y), card_user(Y, X)$: the last X is not new.
- $key(X) \leftarrow tr_product(X, chips), tr_card(X, Y), card_user(Z, Z)$: Z is not an existing variable.
- $key(X) \leftarrow tr_product(X, chips), tr_card(X, Y), card_user(X, Z)$: the input variable of *card_user* should be of type C (card); however, X is outputted at a position which gives it type T (transaction) and therefore is not of the correct type.

Query (5) can be refined further:

- *Not* valid is:

  $$key(X) \leftarrow tr_product(X, chips), tr_card(X, Y), card_user(Y, Z), card_user(Y, Z_2) :$$

  the *card_user* mode starts with a number 1, which is intended to forbid the repeated application of this mode.

- A *valid* refinement is:

  $$\begin{aligned} key(X) \leftarrow\ & tr_product(X, chips), tr_card(X, Y), card_user(Y, Z), \\ & tr_product(X, beer) : \end{aligned} \tag{6}$$

  the *tr_product* mode allows the addition of the latter atom. The 1 in the *tr_product* mode is intended to forbid the repeated application with the same constant.

It is easy to see that several queries can be built which "mean" exactly the same in the traditional semantics of logic. Query (6) means the same as:

$$key(X) \leftarrow tr_product(X, chips), tr_product(X, beer), tr_card(X, Y), card_user(Y, Z), \tag{7}$$

but is constructed in another way. More complex equivalences may also occur between queries. Some bias may allow the following queries:

$$key(X) \leftarrow tr_product(X, Y), tr_product(X, chips)$$

and

$$key(X) \leftarrow tr_product(X, chips);$$

still these two queries clearly mean the same. To formally define when queries mean the same, Dehaspe and De Raedt (1997) use *θ-subsumption* between atom sets.

Definition 1 (θ-subsumption) *An atom set C subsumes an atom set D, denoted by $C \succeq D$, if there is a substitution θ, which maps variables to variables or to constants, such that $C\theta \subseteq D$. The θ-subsumption relation induces an equivalence relation $\sim$, that is defined as follows: $C \sim D$ iff $C \succeq D$ and $D \succeq C$.*

warmr-gen
$C_{k+1} = \emptyset$;
for all $c \in L_k$ **do**
  for each atom c' which may refine c **do**
    Add c' to C_{k+1} unless:
      (a) there is an element $e \in \bigcup_{i \leq k} I_i : e \succeq c'$, or
      (b) there is an element $e \in \bigcup_{i \leq k} L_i \cup C_{k+1} : e \sim c'$.

Figure 4: Candidate generation algorithm of WARMR.

It can be shown that a property similar to Formula (1) also holds for θ-subsumption on atom sets:

$$C \succeq D \Rightarrow \text{sup}(C) \geq \text{sup}(D). \quad (8)$$

Using these building blocks, it is possible to mimic the original APRIORI algorithm quite closely. Also in the WARMR algorithm, a level-wise process of generating candidate atom sets and counting these atom sets is repeated.

WARMR uses the algorithm in Figure 4 to generate a new set of candidate queries C_{k+1}. With L_k we denote the set of frequent queries of size k; I_k denotes a set of infrequent queries of size k. In this algorithm, restriction (a) removes queries which have already been determined to be infrequent. Restriction (b) removes queries which have the same meaning as previously considered frequent queries or candidates. This restriction makes sure that the equivalent queries we mentioned previously do not occur.

In the counting step, all possible constant values are assigned to the parameter variables in the *key* predicate, X in our example. For each of these assignments, all atom sets are evaluated using a standard PROLOG interpreter or compiler in the implementation of Dehaspe and De Raedt (1997). To this purpose, (a part of) the database is repeatedly added to a PROLOG knowledge base.

Although the choices made in the WARMR algorithm make it a very widely applicable algorithm, they have their drawbacks on the efficiency:

- checking $\theta-$subsumption is an NP-complete problem (Kietz and Lübbe, 1994);
- the algorithm requires a PROLOG interpreter or compiler, which for queries that only contain predicates that refer to tables may introduce unnecessary complexity.

The algorithm we will introduce in the next section differs from WARMR in these essential points. Although the algorithm loses some of the wide applicability of WARMR, we hope to get the efficiency in return which is necessary to handle large-scale databases.

3. FARMER

The FARMER algorithm for mining multirelational databases does not rely on a PROLOG database, but uses special purpose datastructures which have been developed for this algorithm. In this section we will first discuss some details of these datastructures. We will introduce some notation to make clear how these datastructures are accessed. We will use this notation in our description of the counting step of the FARMER algorithm.

Just like APRIORI and WARMR, FARMER repeats a process of generating and counting candidate queries (item sets). Each of these steps will be discussed separately. In the counting step, a tree which contains the candidate queries is evaluated using the datastructures. The tree of candidate queries is obtained using the query generation algorithm, which is discussed next. This step includes the copying of existing nodes and the addition of new nodes. However, no θ-subsumption is used to remove queries. We will investigate the effect of this choice in the next section.

All the algorithms which we discuss here have also been implemented in C++. The sources can be found at our homepage (Nijssen, 2001).

3.1 Database

Our algorithm uses special-purpose datastructures to store the database tables. The main idea we exploit is that the modes can be used to optimize the storage structures, as the modes define how a table is accessed in a query. In a preprocessing step, the table are stored in a special way, which results in a speedup for FARMER.

In our setup every table is referred to by a predicate; for each predicate a type declaration must define the types of the parameters:

$$predicate(p(t_1, \ldots, t_n)),$$

where t_i $(1 \leq i \leq n)$ is the name of a type. Every table is stored in a plain ASCII text file. Using hash tables, all attributes in the original tables are translated into IDs, which are consecutive integer numbers for all constants of the same type. We will refer to the set of records belonging to predicate p as

$$\mathcal{R}_p = \{\mathbf{r}\} = \{(r_1, \ldots, r_n)\}.$$

With $\mathbf{r}$ we will always denote a vector which is a row of a table.

Example 3 *In our running example, $\mathcal{R}$ for tr_product is defined as:*

$$\mathcal{R}_{tr_product} = \{(1, chips), (1, beer), (2, rice), \ldots\}.$$

An example of $\mathbf{r}$ *is* $\mathbf{r} = (1, chips)$*. In the sequel, we will denote all constants by numbers which may be obtained using the hash table; the example becomes:*

$$\mathcal{R}_{tr_product} = \{(1, 1), (1, 2), (2, 3), \ldots\}.$$

Our algorithm also uses mode declarations to define the building blocks of the search. Because predicates have a unique type given by their declaration, there is however no need to give types in mode declarations. Furthermore, when reading the tables from disc to main memory, we utilize the modes to choose a good datastructure:

- for a mode with n constraints without outputs, by default a binary n dimensional table is constructed in main memory. An example is given in Figure 5(a). Formally, this table is defined as follows.

	1	2	3	4	5
1	1	1	0	0	0
2	0	0	1	0	0
3	0	0	0	1	0
4	0	0	0	0	1
5	0	0	0	0	0

(a) Binary table for $mode(1, tr_product(+, \#))$

1	$\rightarrow 1 \rightarrow 2$
2	$\rightarrow 3$
3	$\rightarrow 4$
4	$\rightarrow 5$
5	

(b) Table of lists for $mode(1, tr_product(+, -))$

Figure 5: Datastructures to store tables in FARMER.

Definition 2 *Given is* $mode(k, p(c_1, \ldots, c_n)) = \mu$ *for which* $\forall i : c_i \in \{`+', `\#'\}$. *The binary table* I_μ *is defined as*

$$I_\mu(\mathbf{r}) = \begin{cases} 1 & \text{if } \mathbf{r} \in \mathcal{R}_p; \\ 0 & \text{otherwise.} \end{cases} \tag{9}$$

It can be seen in the example table that $I_\mu((1,1)) = 1$ as $(1,1) \in \mathcal{R}_{tr_product}$.

This datastructure is identical to the datastructure used in many special purpose APRIORI implementations. Of course, for sparse databases this datastructure may be very inefficient. The next datastructure can therefore also be used.

- for a mode with m outputs and $(n-m)$ other constraints, an $(n-m)$-dimensional table is constructed in which each of every input element contains a list of output values. An example is given in Figure 5(b). Formally, this table is defined as follows.

Definition 3 *Given is* $mode(k, p(c_1, \ldots, c_n)) = \mu$. *Let* C_μ^- *be the indexes of the output constraints* $C_\mu^- = \{i | c_i = `-'\}$, *let* Π_μ^- *be the attributes of a record at the output positions* $\Pi_\mu^-(r_1, \ldots, r_n) = (r_i | i \in C_\mu^-)$ *(*Π_μ^- *thus projects on the output coordinates of* $\mathbf{r}$*), and let* $\Pi_\mu^+(r_1, \ldots, r_n) = (r_i | i \notin C_\mu^-)$. *Then the table of outputs* $\mathcal{R}_\mu$ *can be defined as*

$$\mathcal{R}_\mu(\tilde{\mathbf{r}}) = \{\Pi_\mu^-(\mathbf{r}) | \mathbf{r} \in \mathcal{R}_p, \Pi_\mu^+(\mathbf{r}) = \tilde{\mathbf{r}}\}. \tag{10}$$

In the sequel, we will denote a projection of a vector, which is a vector containing some coordinates of the complete vector, with $\tilde{\mathbf{r}}$. Also for such a projection, we define a function I_μ:

$$I_\mu(\tilde{\mathbf{r}}) = 1 \Leftrightarrow \mathcal{R}_\mu(\tilde{\mathbf{r}}) \neq \emptyset.$$

In our running example, we can clarify the meaning of I_μ by considering

$$\mu = mode(1, tr_product(+, \#)).$$

For the products bought by client 1, we denote $\mathcal{R}_\mu((1)) = \{(1), (2)\}$ (client 1 buys products 1 and 2). The set is represented by a list. Furthermore, we can conclude from this table that $I_\mu((1)) = 1$ (client 1 buys a product).

These datastructures allow the retrieval of truth values and variable values in constant time.

We wish to repeat here that the predicates in FARMER only refer to one table, and that the queries will be built by adding predicates one by one. This is a restriction which turns some situations less easy to handle, but, on the other hand, forces the user to clearly think about data preparation and problem formulation. We will illustrate this using our example. If we reconsider query (5), we see that it contains consecutive *tr_card* and *card_user* predicates. If every card has at least one user, we can expect that every query containing *tr_card* can be refined with the *card_user* predicate. Some redundant computation would be done if the refinement is performed in two steps. In WARMR this situation is accounted for by allowing *look ahead* declarations. In short, this means that the user can specify that these two predicates should always be added together. Following our idea that the data must be preprocessed as much as possible before starting the datamining algorithm, we prefer another solution: one could also introduce a new predicate *tr_user* which is defined as[3]:

$$tr_user(T, U) \leftarrow tr_card(T, C), card_user(C, U).$$

A table for this new predicate can be computed in advance. Depending on where the table comes from, this may or may not be easy. In case *tr_card* and *card_user* are originally stored in a DBMS, one could first compute a join and then translate the joined table in our special purpose datastructure. In other cases, one may still use PROLOG or write — yet another — special purpose algorithm.

To allow for an easy translation into our special purpose datastructures, we have implemented a C++ library with appropriate functionality. It can be downloaded from our homepage (Nijssen, 2001).

3.2 Search

The search which FARMER performs differs in two aspects from the original WARMR algorithm:

- it does not use θ-subsumption;
- it manipulates a tree datastructure to generate the queries which are defined by the bias.

Also FARMER repeats a process of counting and generating candidates. A tree datastructure, of which an example is given in Figure 6, contains the candidates in a similar way as previously discussed for traditional item sets: the candidate queries are paths from the root to a leaf. In the example tree the atoms are generated using the bias of our running example; for the time being, however, we assume that there are only two products and two client properties. The constants for products and clients are represented by numbers. If we assume that all queries are frequent, this tree is obtained after generating all candidates with 3 atoms in the body of the query.

We will first discuss the counting algorithm in more detail, in which we use the functions to access the data. This algorithm should be applied to a tree of candidates. How these candidates are obtained, is discussed next.

3. If there is no card for the current transaction, the *tr_user* predicate does not find a user; as we knew every card had a user, we can still derive from the non-existence of a user the lack of a card.

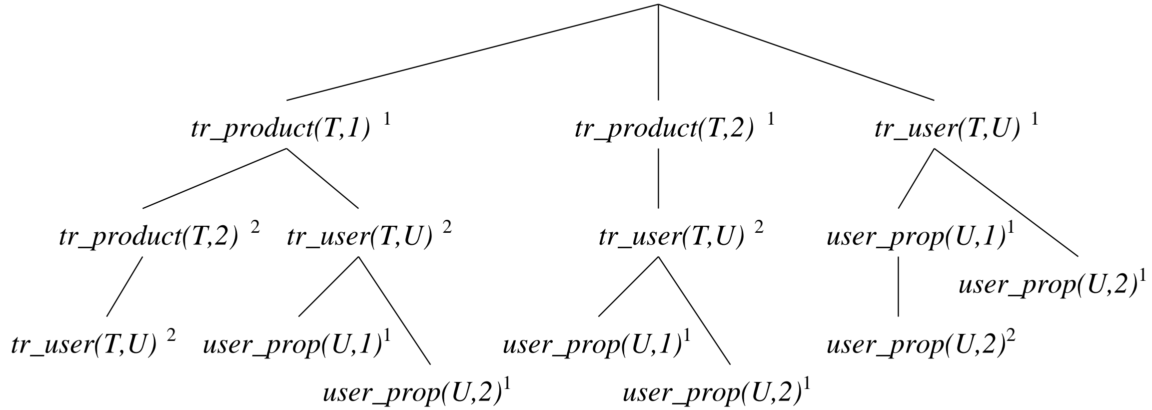


Figure 6: A tree in FARMER

3.2.1 COUNTING

Pseudo-code for the algorithm is given in Figure 7. In this pseudo-code, we use the following notation:

- a capital A refers to an atom in the tree;
- a capital $B = \mathbf{V}/\mathbf{r}$ refers to a set of variable assignments, which is a set containing mappings from variables V_i to constant values r_i. If $\mathbf{r}'$ is a vector which contains variables besides constants (typically, the parameters of an atom), with $\mathbf{r}'B$ we denote the vector in which all variables have been replaced by their corresponding constant. The resulting vector may still contain variables.
- $I(A, B)$ is an interpretation function, which returns true if an atom A can be proved to be true using assignment B:

  $$I(A, B) = I_\mu(\Pi^+_\mu(\mathbf{r}'B)),$$

  where μ is the mode of atom $A = p(\mathbf{r}')$. In the algorithm B maps all input variables to constants. Therefore $\mathbf{r}'B$ only contains variables at the output positions. Next we apply Π^+_μ to the resulting vector. The definition of Π^+_μ needs no further adaptation to incorporate this; the result is a vector which does not contain variables. If μ does not contain outputs, function Π^+_μ, which projects on no-output parameters, yields $\mathbf{r}'B$. In this case, I_μ is computed using the binary table we discussed. If μ does have outputs, the input of I_μ is a projected vector; I_μ is then evaluated using $\mathcal{R}_\mu$.
- $\mathcal{B}(A, B)$ is an assignment function. It is defined as

  $$\mathcal{B}(A, B) = \{\Pi^-_\mu(\mathbf{r}'B)/\tilde{\mathbf{r}} | \tilde{\mathbf{r}} \in \mathcal{R}_\mu\left(\Pi^+_\mu(\mathbf{r}'B)\right)\},$$

  where μ is the mode of atom $A = p(\mathbf{r}')$. Function Π^-_μ outputs an ordered set (a vector) which contains only variables: all output variables in $\mathbf{r}'$ are new and therefore are not substituted by B. These variables get a value from $\tilde{\mathbf{r}}$.

```
Count(A,B)
if I(A,B) = 1 then
  if A is a leaf then
    disable A
    increase support A
    return true
  else
    for all B' ∈ 𝓑(A,B) do (*)
      d := true
      for all A' ∈ children of A do
        if A' not disabled then
          d := Count(A',B ∪ B') ∧d
      if d = true then
        disable A
        return true
return false
```

Figure 7: The FARMER counting algorithm.

For all values of the key variables, the tree is traversed recursively. Initially all nodes are enabled. For a given node all values for the outputs are checked as long as there are children which have not been satisfied. A similar idea is also applied in Blockeel et al. (2000).

3.2.2 CANDIDATE GENERATION ALGORITHM

The generation mechanism is based on the following idea. For every atom, there is one position at which it can occur for the first time: this is the place at which all its input variables have just been defined. If a query is refined with an atom in FARMER (and also in WARMR), that atom belongs to one of two classes:

1. The atoms that could not be added earlier.

2. The atoms that could be added earlier.

Examples of these classes are given by the superscripts in Figure 6.

During the construction of the tree this subdivision is used. Given a tree and an ordered set of mode declarations, the tree is refined using the algorithm in Figure 8. The tree in Figure 6 is obtained using this mechanism when it is assumed that all queries of the following bias are frequent:

$key(key(-)).$
$mode(1, tr_product(+, \#)).$
$mode(1, tr_user(+, -)).$
$mode(1, user_prop(+, \#)).$

The superscripts also in this case denote the mechanism that was used to create a node.

```
Expand(A)
if A is internal then
  for all A' ∈ children(A) do
     Expand ( A' )
else if A is frequent then
  add as child from left to right:
     1. all valid refining atoms which use at least one variable introduced in A
     2. all frequent right brothers of A
     (*)
else
  remove A
```

Figure 8: The FARMER query refinement algorithm.

The first mechanism serves the purpose of introducing atoms which could not be added previously. The atoms are introduced in the same order as the corresponding mode declarations and a deterministic mechanism is used to go through all the possible input variables. It is important to note that every mode is applied only once with the same constants. If the mode of *tr_user* would have had been $mode(2, tr_user(+,-))$, the following would be a valid query:

$$key(X) \leftarrow tr_user(X,Y), user_prop(Y,1), tr_user(X,Z);$$

both $tr_user(X,Y)$ and $tr_user(X,Z)$ can be added to the empty query, which would require these two atoms to be generated by the same mode as brothers. We will not allow this for the moment. We will discuss this restriction in detail in the sequel.

The atoms which are newly generated are brothers of each other; the second mechanism takes care that all subsets are generated afterwards — if frequent. By keeping the children in order, every subset is generated only once, or, equivalently, only one permutation out of a set of dependent atoms is considered. If necessary, the second mechanism gives new names to output variables to make sure they remain outputs.

4. FARMER vs θ-subsumption

Due to the absence of θ-subsumption, it can easily be seen that FARMER will often not find the same set of queries as WARMR does. In this section we will show in which situations FARMER will generate the same output. We will prove this in several steps:

1. We will show that a bias of FARMER must obey some restrictions to make sure that queries found by FARMER do not θ-subsume each other.

2. We will show that for the restricted bias also the queries found by WARMR do not θ-subsume each other.

3. We will show that for each query found by WARMR there is at least one query found by FARMER which is equivalent by θ-subsumption. Combining this with 1 we can conclude that there must be exactly one in the output of FARMER.

4. We will prove the reverse of 3: for every query found by FARMER, WARMR finds an equivalent one.

5. Combining 3 and 4 we can conclude that the sets of queries are equivalent under θ-subsumption if the bias is restricted.

It will appear that both the FARMER and WARMR algorithm do not find all queries which obey the bias specification, due to the steps performed by their refinement algorithms. We will finally introduce some changes to FARMER which remove some of the restrictions on the bias, however at the cost of new restrictions on the form of the queries.

In all proofs, we assume that queries are frequent. Furthermore, it is important to recall here that for any frequent query all its subsets and their equivalent queries must also be frequent.

4.1 Restriction to avoid θ-subsumption equivalence

The θ-subsumption equivalence relation is only one method for determining that queries mean the same. A less strict relation is when queries are *alphabetic variants*, which we will denote with $\simeq$ here and is defined for two (unordered) sets of atoms as follows: $C \simeq D$ iff there exist substitutions θ_1 and θ_2 such that $C\theta_1 = D$ and $D\theta_2 = C$. Also for queries, which are *ordered* sets of atoms, these relations can be defined: theoretically, by considering queries as unordered sets; practically, by finding a permutation of atoms followed by a substitution.

The correspondence between $\sim$ and $\simeq$ can be expressed using Plotkin's reduced atom sets.

Definition 4 *An atom set D is called reduced iff ($C \subseteq D$ and $C \sim D$) imply $C = D$ (Plotkin, 1969).*

Examples of atom sets which are not reduced are:

$$D_1 = tr_user(X, Y), tr_user(X, 1).$$

$$D_2 = tr_user(X, Y), user_prop(Y, 1), tr_user(X, Z).$$

Theorem 5 *Let C and D be reduced atom sets. Then $C \simeq D$ iff $C \sim D$.*

Proof "$\Rightarrow$": this is clear as $C\theta_1 \subseteq D$ and $D\theta_2 \subseteq C$.
"$\Leftarrow$": as $C \succeq D$, $C\theta_1 \subseteq D$, and as $D \succeq C$, $C\theta_1\theta_2 \subseteq C$. Let $C' = C\theta_1\theta_2$. Because $C' \subseteq C$ and $C\theta_1\theta_2 \subseteq C'$, also $C \sim C'$ holds (by definition), and then $C = C'$ because C is reduced. Thus $C\theta_1\theta_2 = C$. In the same way, $D\theta'_1\theta'_2 = D$. As $|C\theta_1| = |C|$ and $C\theta_1 \subseteq D$, $|D| \geq |C|$. As $|D\theta'_1| = |D|$, $|C| \geq |D|$, and finally $|D| = |C|$. By combining $|C\theta_1| = |D|$ and $C\theta_1 \subseteq D$, $C\theta_1 = D$ is shown. This proves that $C \simeq D$. ■

Definition 6 *A restricted bias is a bias which does not contain two modes for the same predicate.*

We will show that for a restricted bias, FARMER will always generate reduced atom sets. Then we will show that FARMER does not generate two different atom sets that are alphabetic variants. From this we conclude that, given a restricted bias, FARMER will not generate queries that subsume each other.

Theorem 7 *For a restricted bias* FARMER *will always generate reduced queries.*

We will first illustrate the use of this theorem by an example. Assume that the following bias is used which is not reduced:

$mode(1, tr_user(+, -))$
$mode(1, tr_user(+, \#))$

A query which may be generated by this bias is:

$$key(X) \leftarrow tr_user(X, Y), tr_user(X, a).$$

This query is clearly not reduced. Note that when we say reduced queries, we actually mean the atom sets in the queries. Part of our proof will be to show that FARMER does not generate two identical atoms, as in general two identical elements can not occur in a set.
Proof Assume that D is a query in the tree, obtained by using a restricted bias. We will show that for any subset $C \subset D$, $D\theta \subseteq C$ can never be true. In order for this, there must at least be two atoms A_1 and A_2 in D which are mapped to the same atom in C: $A_1\theta = A_2\theta$. For any pair we will try to construct such a substitution. By definition of the bias, both atoms must have inputs at the same positions. Important to note is that the refinement algorithm only applies a mode once with the same input variables, so at least one input variable of A_1 and A_2 must be different. Construct a substitution θ which unifies A_1 and A_2. This substitution will always map variables to variables. Apply θ to the whole query D. Consider the set of atoms that introduced the different variables used in A_1 and A_2; then there are two possibilities:

1. This set contains one atom which has two outputting parameters. By θ these outputting variables are bound to the same variable. One of the two variables is not new, although it occurs at an output position. Such an atom can never be generated, and will therefore never be part of C, which is a subset of a query generated by FARMER.

2. This set has at least two different atoms in C. Of both atoms an output is bound to the same variable by θ. In whatever order these atoms are placed, one of them is not new although it occurs at an output position.

Thus, there can only exist reduced queries. ■

Theorem 8 *Given a restricted bias,* FARMER *will never generate two queries that are alphabetic variants.*

order(A)
add A to the end of Q'
S := atoms in Q which use at least one variable introduced in A
and no variable outside Q'
order S according to mode declarations and
a deterministic input variable numbering strategy
for all $A' \in S$ in order **do**
order(A')

Figure 9: An algorithm to order atoms in a query Q.

Proof We first note that for ordered atom sets, such as queries, a deterministic variable naming mechanism can be used. Furthermore we note that two queries must be of equal size and that the substitution can only map variables to variables. Thus, to determine whether two queries are alphabetic variants, it suffices to find a permutation of atoms, followed by a deterministic variable renaming, that makes two queries equal. We will show that FARMER generates one permutation by giving an algorithm which yields this permutation given a query.

The restricted bias is such that for every atom in an (unordered) atom set, there is only one possible mode declaration. The usage of input and output constraints determines a partial order on the atoms: some atoms must occur in a certain order. The task is to order the atoms which are not ordered by constraints; to order these other atoms, the modes are given an order which is not changed throughout the execution of the algorithm. Figure 9 shows the strategy to order the atoms in query Q. The initial value of Q' is the empty query $\leftarrow$ and the strategy should be called with **order**($key(\mathbf{V})$). This first atom is added at the head of the query. The order obtained by this strategy corresponds to the order of FARMER. The set S corresponds to the atoms generated in step 1 of the algorithm in Figure 8. The tree building mechanism in FARMER which places new nodes before copied nodes takes care of the recursion by acting as a sort of LIFO queue. ■

We immediately conclude:

Corollary 9 *Given a restricted bias* FARMER *will never generate queries that θ-subsume each other.*

4.2 FARMER and WARMR generate the same output

In four steps we will prove that the outputs of WARMR and FARMER are the same if the same restricted bias is used for both algorithms, as outlined earlier.

Theorem 10 *For a restricted bias* WARMR *will always generate reduced queries.*

To give an idea about the consequence of this theorem, consider the following modes (which do not come from our running example):

$mode(1, p(+X, -Q, -Q))$
$mode(1, p(+X, +Q, +Q))$

A query which may be constructed using these modes is:

$$key(X) \leftarrow p(X, Q, P), p(X, Q, Q).$$

This query is not reduced. However, WARMR will not prune it, as the reduced query

$$key(X) \leftarrow p(X, Q, Q).$$

is never generated and WARMR only tests for θ-subsumption with previously found queries.
Proof The proof is similar to the proof of Theorem 7. The difference lies in the observation that FARMER does not apply the same mode twice for the same constants and input variables. The refinement algorithm of WARMR does not have this restriction. If Q is a query, an atom A can be added which — apart from output variable names — is equal to an atom $A' \in Q$. However, WARMR has the restriction that queries should not θ-subsume each other. A substitution θ which maps the outputs of A to the outputs of A' is easily constructed; this substitution shows that Q expanded with A is equivalent to Q. A refinement with A will therefore also be pruned by WARMR. ■

Theorem 11 *For any query Q found by* WARMR *with a restricted bias, there is one query Q' in the output of* FARMER *such that $Q \sim Q'$.*

Proof All queries in WARMR are reduced, as are the queries in FARMER. For a query Q it thus suffices to find an alphabetic variant Q' in the output of FARMER. The ordering algorithm in Figure 9 can be used to order the atoms in Q. This order is unambiguous as all atoms in S have a different mode, or, if they belong to the same mode, different constants or different input variables. We have already shown that this ordering algorithm yields a query found by FARMER for a restricted bias. ■

Theorem 12 *For any query Q found by* FARMER, *there is one query Q' in the output of* WARMR *such that $Q \sim Q'$.*

Proof As Q and Q' are reduced, we are searching for an alphabetic variant Q' of Q. We will prove that for any query Q which is frequent, is reduced and obeys the bias, WARMR will find an alphabetic variant, so in particular also for such a query coming from FARMER's output.

We will prove this by contradiction. Assume that no alphabetic variant Q' is generated by WARMR. Consider one such variant Q' which obeys the bias restrictions, but is not found by WARMR. Decompose Q':

$$Q' = Q'', A$$

where A is the last atom of the alphabetic variant.

First we consider the case that Q'' was empty. Then we have a contradiction as $Q' = A$ could be generated by the bias.

Now we consider the case that Q'' was nonempty. First assume that Q'' was generated by WARMR. As we assumed the alphabetic variant Q' obeyed the bias, Q'' can be refined

with A. If $Q' = Q'', A$ is pruned, it must be equivalent with some other query. This is however impossible, as we assumed no alphabetic variant of Q' occurs in the set of queries considered and Q' can neither be equivalent with a smaller query as Q' is reduced.

Consequently, the only possibility is that Q'' must not have been generated by WARMR. As also any valid alphabetic variant of Q'' could be refined with A to create an alphabetic variant of Q, neither an alphabetic variant of Q'' was apparently generated.

We can recursively apply our decomposition, until we reach a query $Q' = A$ and run into a contradiction. ■

It now follows that:

Corollary 13 *For a restricted bias, a query found by* FARMER *is an equivalent of exactly one query found by* WARMR, *and vice versa.*

4.3 Removing restrictions by introducing new ones

The restriction of the bias was developed in order to make sure that queries found by FARMER do not imply each other. This was done to avoid that many queries will be found which actually mean the same. Another solution to obtain this goal may be to change the meaning of a query, such that if they are literally different, they also mean something different.

As we observed, the following queries "mean" the same in usual logic:

$$key(X) \leftarrow tr_user(X, Y), tr_user(X, a) \tag{11}$$

and

$$key(X) \leftarrow tr_user(X, a).$$

However, a different evaluation scheme for queries could also be used, using the idea of *Object Identity* (Esposito et al., 1996, 2001). The idea of object identity is:

> Within a clause, terms denoted with different symbols must be distinct.

More precisely, this means that every query is expanded by a set of literals $constraints(Q)$ before it is evaluated. These constraints are defined as:

$$constraints(Q) = \{(X \neq Y) | X, Y \in V(Q), X \neq Y\} \cup \{(X \neq x) | X \in V(Q), x \in K(Q)\},$$

where we use $V(Q)$ to denote the set of variables occurring in Q and $K(Q)$ denotes the set of constants occurring in Q. Query (11) would be expanded to:

$$key(X) \leftarrow tr_user(X, Y), tr_user(X, a), X \neq Y, X \neq a, Y \neq a.$$

Another kind of θ-subsumption relation can be defined for two atom sets which are evaluated under object identity:

Definition 14 *Query D OI-subsumes C, denoted with $C \succeq_{OI} D$ iff $C' \succeq D'$ for $D' = D, constraints(D)$ and $C' = C, constraints(C)$.*

It has been shown that under OI-subsumption every atom set is reduced. Consequently, queries which do not contain duplicates of the same atom can only subsume each other if they are alphabetic variants.

The idea of object identity is easily integrated in FARMER by means of one modification. In the counting algorithm of Figure 7, the line marked with a (*) should be expanded. Assignments B' which assign a variable to a previously used variable (as stored in B for variables, and in the path to the root of the set enumeration tree, in case of constants) may not be used. Furthermore, no assignment B' may ever be considered if A contains a constant which is used by a variable occurring in B. This forces the algorithm to choose another assignment at an earlier stage.

For queries in FARMER not to OI subsume each other, it would therefore suffice to make sure that no two alphabetic variants are generated and no atom may be created twice. For an arbitrary bias, these two requirements are not necessarily satisfied, as shown by the following two examples.

Example 4 *Consider the following bias:*

$mode(1, tr_product(+, -))$
$mode(1, tr_product(+, +))$

For this bias, FARMER *would generate:*

$$key(X) \leftarrow tr_product(X, Y), tr_product(X, Y),$$

which clearly contains two identical atoms.

Example 5 *Consider the following bias (which has not been taken from our running example):*

$mode(1, p(+, -, \#))$
$mode(1, p(+, +, \#))$

For this bias, FARMER *would generate the following two queries:*

$$key(X) \leftarrow p(X, Y, a), p(X, Y, b)$$

and

$$key(X) \leftarrow p(X, Y, b), p(X, Y, a),$$

which are clearly alphabetic variants of each other.

As can be seen in these examples, two modes which have constants at the same positions, but differ otherwise, cause trouble. We will therefore consider a restricted bias in which this is forbidden.

For such a bias, it is clear that no two identical atoms can occur. FARMER does not generate two identical atoms with equal inputs. If atoms are equal, one of these atoms must have an input where the other has an output. This is avoided by the restricted bias.

To prove that no two alphabetic variants are generated, we can use the ordering algorithm in Figure 9. An important observation in the application of this algorithm was that every atom can only be generated by one mode. For the restricted bias we use here, this is still the case: the positions at which constants occur, uniquely define the mode that introduced the atom.

Expanding beyond WARMR. As we observed, queries found by WARMR are reduced if the bias is restricted to one mode per predicate. The building procedure which uses θ-subsumption based pruning and allows for non-redundancy, may have a disadvantage.

Example 6 *Consider the following bias, which is restricted:*

$$mode(2, tr_user(+, -))$$
$$mode(1, user_prop(+, \#))$$

The following query obeys this bias:

$$key(X) \leftarrow tr_user(X, Y), user_prop(Y, 1), tr_user(X, Z), user_prop(Z, 2); \qquad (12)$$

it will however never be generated, as an intermediate step is not reduced.

We already observed that FARMER will also not generate this query. A small change however suffices to allow for this query. In the refinement algorithm of Figure 8, one line should be added at the place of the (*):

3. If the bias allows this, add a copy of A with new output variables.

However, for all restricted biases we discussed, the output of FARMER is no longer guaranteed to contain no alphabetic variants. In the ordering algorithm of Figure 9, it is no longer possible to give an order to the atom A and its copy. Besides query (12), also this query will be generated:

$$key(X) \leftarrow tr_user(X, Y), user_prop(Y, 2), tr_user(X, Z), user_prop(Z, 1).$$

Also under OI subsumption equivalence, these two queries mean the same. However, an idea similar to OI implication can also be used in some cases to make use of the fact that FARMER generates two alphabetic variants. In case constants have an order, we can use the *order* of the atoms to define a new set of *constraints* atoms:

$$constraints_2(Q) = \{(X < Y) \mid X \text{ and } Y \text{ are the } i\text{th output of } A\text{, respectively } A'\text{, where } A' \text{ is a copy } A\}.$$

Query (12) would be expanded to:

$$key(X) \leftarrow tr_user(X, Y), user_prop(Y, 1), tr_user(X, Z), user_prop(Z, 2), Y < Z.$$

This clearly means something different than:

$$key(X) \leftarrow tr_user(X, Y), user_prop(Y, 2), tr_user(X, Z), user_prop(Z, 1), Y < Z.$$

With the addition of $constraints_2$, both queries need no longer both be frequent as they mean something rather different. The counting algorithm of FARMER can easily be adapted in a similar way as for object identity.

However, does this distinction make sense? We believe that in some situations this is the case. In our example, the users could be ranked by their age; the implicit addition of

$Y < Z$ tells something about the way ages of clients relate to each other. This kind of information could have never been found by any other APRIORI-like algorithm.

In our current implementation of FARMER, we do not yet have an explicit mechanism to order constants. Currently, FARMER orders constants according to the order in which they are encountered. By influencing this encountering order, the order of constants can be influenced. A more elegant solution would be to specify the order of constants in a mode declaration.

4.4 Finding interesting queries

Our reason to prefer the WARMR approach to, for example, query flocks was its close resemblance to the traditional APRIORI algorithm. Also WARMR tries to build queries as large as possible, without requiring to specify a size at before hand. In this paragraph we present some ideas which may be used to construct an algorithm to find interesting queries in a similar way as in Section 2.2. We will assume that one of the restricted biases is used; furthermore we will only consider the FARMER algorithm.

If we consider a frequent query Q, according to the ideas in Section 2.2, we should split this query into two parts, on which the IR test can be applied. A problem with queries is that not every subset obeys the bias and is therefore generated. Therefore, we will have to restrict ourselves to valid subqueries. By considering the ordering algorithm, we conclude that for any valid subset, the order of this subquery is also the order in which the query is found by FARMER. This allows for an efficient search in the tree for the support of this subquery; remember that the subquery must exist in the tree, as all subqueries must also be frequent.

In particular, this means that for valid subqueries Q_1, Q_2 and $Q_1 \cap Q_2$ of a query Q such that $Q = Q_1 \cup Q_2$, the support can efficiently be found. This allows the application of the IR test for overlapping item sets (see Formula 2).

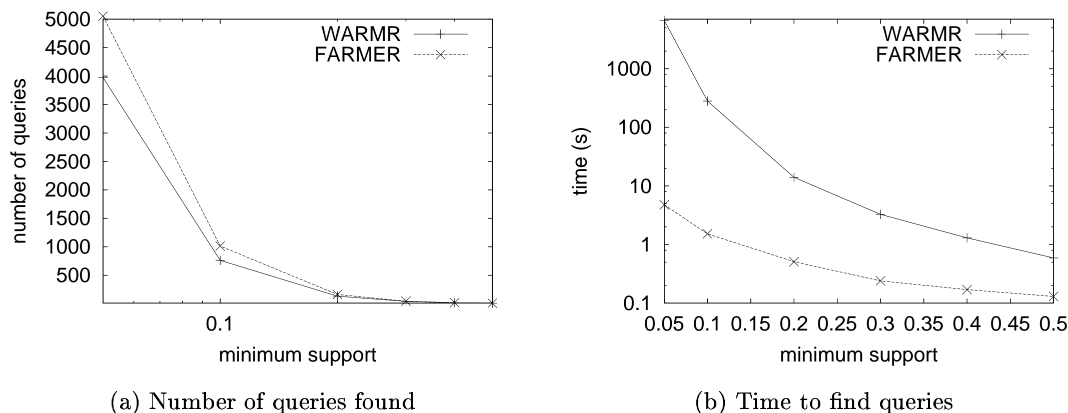
As we have already shown, the application of the IR test as discussed in Section 2.2 is a very time complex procedure. A way to reduce the effort is to reduce the number of subqueries which will be tested, by only varying Q_1. Given a valid subquery Q_1, the set $Q_2 = Q \setminus Q_1$ is easily computed; however, Q_2 may not be a valid query. The solution is to prefix Q_2 with the minimal set of atoms in Q necessary to turn it into a valid query. This minimal set is easily found by recursively determining all the atoms which introduce variables used in Q_2.

Some further optimizations are possible. The atoms in the set $Q_1 \cap Q_2$ typically map from variables to variables. If it is known at beforehand that such a mapping always exist for all atoms in the overlap and for all inputs, the support of $Q_1 \cap Q_2$ is known at beforehand to be 1. The IR test for overlapping sets then turns into the usual IR test.

5. Experimental results

Our main reason to implement the FARMER algorithm was the performance of WARMR in some situations. In this section we will see how the efficiency of FARMER and WARMR relate to each other.

We have compared FARMER and WARMR on three datasets. We should remark that in our experiments we used an implementation of WARMR that did not yet use the tree datas-



(a) Number of queries found

(b) Time to find queries

Figure 10: A comparison of FARMER and WARMR on the Bongard dataset. Note that the scales are different.

tructure discussed by Blockeel et al. (2000); a comparison of both algorithms is therefore not completely fair. Experiments of Blockeel et al. (2000) revealed speed-ups in the order of magnitude of 20 for WARMR in some situations. The time to translate an input database in our special purpose file format is included in all time comparison experiments.

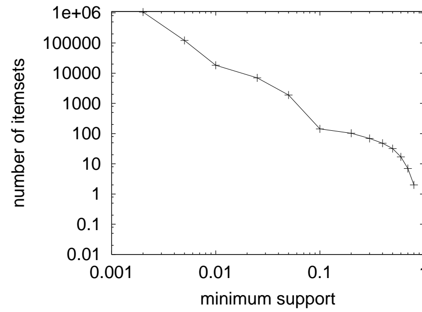
Bongard The Bongard dataset (Bongard, 1970) contains descriptions of artificial images. The task is to discover patterns in the images. Every image consists of several triangles, circles and squares that can be included into each other. We do not use a restricted bias, but use the object identity counting algorithm. We expect that FARMER generates more queries.

In Figure 10 the results of the experiments are depicted. Figure 10(a) shows the number of queries that each algorithm finds. The number of FARMER is higher in all cases, as expected. In Figure 10(b) the execution times of the algorithms are compared[4]. Paying attention to the fact that the scale is logarithmic, the speed-ups are considerable for this dataset. This is even more surprising considering the fact that FARMER generates redundant queries and has to count these redundant queries too.

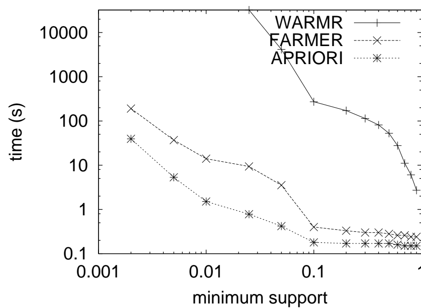
Frequent itemsets In this experiment we compare the performance of FARMER to a special purpose algorithm. As test case a binary digit dataset is used which contains 1000 binary coded numbers. The special purpose algorithm which is used as comparison is the breadth-first implementation of APRIORI by Pijls and Bioch (1999). FARMER uses many of the mechanisms introduced in that algorithm and should perform comparable to that algorithm.

In Figure 11 the results of the experiments are depicted. A characteristic of the dataset is given in Figure 11(a). The number of frequent itemsets appears to increase rapidly when

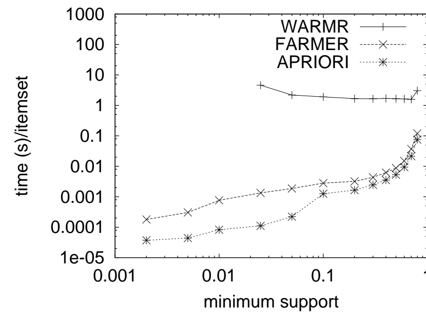
4. Experiments were carried out on a Sun Enterprise 450 4x400MHz UltraSPARC2 CPU w/4MB E-cache 4GB RAM.



(a) Number of queries



(b) Time to find all queries



(c) Time to find each query

Figure 11: Comparison of results for the digit dataset.

the minimum support is beneath 0.1. In Figure 11(b) the execution times of the algorithms are given. The time graph of FARMER is comparable to that of APRIORI, although still exponentially larger. WARMR has a completely different behavior of both other algorithms and had such high execution times that we decided not to carry out experiments for low supports.

In Figure 11(c) both previous graphs are combined and the execution time for each item set is shown. It makes clear how the algorithms react when the amount of solutions they have to find increases. While the execution times of WARMR increase, the times of the other algorithms decrease. Although the overhead for each item set is larger in FARMER, which could be explained by the additional mechanisms that are hooked in, the difference is acceptable. The decreasing trend can be explained by the increasing number of overlapping evaluations when the number of itemsets increases.

Bank In order to test whether our algorithm would work on large datasets, we have applied it to a large real-world dataset, which was released as part of the 3rd European Conference on Principles and Practice of Knowledge Discovery (Berka, 1999). The bank data set is a multi-relational dataset which consists of 8 relations; the largest of these relations contains 1,056,320 records. Each record has a fixed number of attributes, which

makes the representation in terms of item sets very dense. The data set is therefore a challenge for item set algorithms in general, and for multi-relational algorithms in particular.

We made several choices during the preparation of the dataset for usage in FARMER. Several continuous attributes were discretized in intervals. Time or day related attributes were discarded. An adapted version of the principle of object identity was used during evaluation in which object identity is only required for some types. As minimum support 10% of the number of transactions was chosen. Details of our experiments can be found in (Nijssen, 2000).

After 32 hours and 25 minutes of computation on a Sun Enterprise the algorithm finished. Memory usage reached a peak of 200MB, while processor utilization was between 20% and 25%. Most of the memory was consumed by the tree. The counting algorithm took most of the computation time. The number of frequent queries was expected to be very large; we decided to search for maximal queries using the algorithm in Figure 2. We applied the variant in Formula (4) with threshold $\varepsilon = 0.1$. After applying the algorithm 990 queries remained.

In itself, the queries which we found are not immediately useful. Some algorithm to determine their real interestingness should be applied next. We have already given some hints on how this could be done. The most important conclusion we have drawn from our experiment was that we showed that an algorithm having its origins in the world of Inductive Logic Programming can handle very large databases, contrary to common belief.

6. Conclusions and further work

We introduced an algorithm for discovering queries which reflect frequently occuring patterns. It uses a tree datastructure both to count queries and to generate queries. We showed that for a restricted type of bias, this algorithm is equivalent to the WARMR algorithm. We have given some variations of the algorithm which impose a special meaning on many queries. This removes some restrictions on the bias. Although the special meaning could be a useful extension in some cases, it could however also be an undesirable new restriction in other situations.

We have performed several experiments which lead us to believe that our algorithm is indeed a step forwards towards the implementation of an efficient algorithm for the discovery of first order logic frequent queries. We have shown that even for biases which are not reduced, a speed-up can be obtained. A problem we faced in the results of our experiments was the large number of queries. We believe that increasing the threshold is not the right solution to solve the problem. To find an efficient algorithm which determines the statistically *interesting* queries is the most important problem to solve next. We have given some hints on how such an algorithm could look like. The algorithm such as we presented it could reduce the number of resulting rules considerably; however, it could also be too time complex.

Investigating the topics presented here further, we believe several directions are worth considering:

- To implement the algorithm for finding statistically interesting queries.

- To investigate how FARMER scales up for large datasets; especially, it would be interesting to know how large a database can be with an unrestricted bias before FARMER is less efficient than WARMR. For an unrestricted bias several alphabetic variants can be found by FARMER, each of which possibly has to be counted for each transaction, which could increase counting time eventually beyond that of WARMR.

- To determine the efficiency of the interestingness algorithm, especially in combination with the previous observation: if a query is interesting, all its alphabetic variants will be so either. This could drastically reduce the efficiency of the combined algorithms; however, it could also be possible that this test for interestingness is still more efficient than repeated θ-subsumption. In this case, we may consider applying θ-subsumption on the final (small) set of queries.

- To change the queries into more general rules for which confidence can be computed.

- To apply the resulting algorithm to other real-world problems.

More generally, we believe that a closer investigation of real-world databases is necessary. We have the impression that many of the biases which could theoretically be dealt with, will not occur in practice, which would allow for further optimizations. Furthermore, in our current approach many operations are to be performed by hand: to read the tables into our algorithm, including the ordering of constants, some coding is required at a conceptual level at which it should not be performed. A specification language which defines the bias in close relation to databases would be preferable. Interesting investigations into this direction, which we believe we could benefit from, are attempts to specify biases using the *Unified Modeling Language* (Knobbe et al., 2001). Eventually this could lead to a usable as well as an efficient algorithm.

Acknowledgments

We wish to thank the anonymous reviewers of our IJCAI'01 paper (Nijssen and Kok, 2001) for their constructive remarks. Furthermore, we are grateful to Luc De Raedt and Hendrick Blockeel for arranging a meeting in Belgium to discuss the topic of multirelational association rules. We thank Walter Kosters for carefully proofreading this paper.

References

R. Agrawal, H. Mannila, R. Srikant, H. Toivonen, and A.I. Verkamo. Fast discovery of association rules. In *Advances in Knowledge Discovery and Data Mining*, pages 307–328. AAAI/MIT Press, 1996.

Rakesh Agrawal, Tomasz Imielinski, and Arun N. Swami. Mining association rules between sets of items in large databases. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216, Washington, D.C., 26–28 1993.

R. Bayardo, R. Agrawal, and D. Gunopulos. Constraint-based rule mining in large, dense databases. In *Proceedings of the Fifteenth International Conference on Data Engineering*, pages 188–197, 1999.

R.J. Bayardo and R. Agrawal. Mining the most interesting rules. In *Proceedings of the Fifth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 145–154, 1999.

P. Berka. *Workshop Notes on Discovery Challenge PKDD-99*. 1999. URL `http://lisp.vse.cz/pkdd99/chall.htm`.

H. Blockeel, L. Dehaspe, B. Demoen, G. Janssens, J. Ramon, and H. Vandecasteele. Executing query packs in ILP. In *Proceedings of the 10th International Conference on Inductive Logic Programming*, pages 60–77, 2000.

M. Bongard. *Pattern Recognition*. Hayden Book Company (Spartan Books), 1970.

J-F. Boulicaut, A. Bykowski, and C. Rigotti. Approximation of frequency queries by means of free-sets. In *Proceedings of the Fourth European Conference on Principles and Practice of Knowledge Discovery in Databases PKDD'00, Lyon, France*, volume 1910 of *Lecture Notes in Artificial Intelligence*, pages 75–85. Springer-Verlag, 2000.

J-F. Boulicaut, M. Klemettinen, and H. Mannila. Querying inductive databases: A case study on the MINE RULE operator. In *Principles of Data Mining and Knowledge Discovery*, pages 194–202, 1998.

R. Castelo, A. Feelders, and A. Siebes. Mambo: Discovering association rules based on conditional independencies. In F. Hoffman, D. Hand, N. Adams, D. Fisher, and G. Guimaraes, editors, *Proceedings 4th Symposium on Intelligent Data Analysis*, volume 2189 of *Lecture Notes in Computer Science*, pages 289–298. Springer-Verlag, 2001.

J.M. De Graaf, W.A. Kosters, and J.J.W. Witteman. Interesting fuzzy association rules in quantitative databases. In *Proceedings of the Fifth European Conference on Principles of Data Mining and Knowledge Discovery PKDD'01, Freiburg, Germany*, volume 2168 of *Lecture Notes in Artifical Intelligence*, pages 140–151. Springer-Verlag, 2001.

L. Dehaspe and L. De Raedt. Mining association rules in multiple relations. In S. Džeroski and N. Lavrač, editors, *Proceedings of the 7th International Workshop on Inductive Logic Programming*, volume 1297 of *Lecture Notes in Computer Science*, pages 125–132. Springer-Verlag, 1997.

F. Esposito, N. Fanizzi, S. Ferilli, and G. Semeraro. OI-implication: Soundness and refutation completeness. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 847–852, 2001.

F. Esposito, A. Laterza, D. Malerba, and G. Semeraro. Refinement of Datalog programs. In *Proceedings of the MLnet Familiarization Workshop on Data Mining with Inductive Logic Programing*, pages 73–94, 1996.

J. Hipp, U. Guntzer, and U. Grimmer. Integrating association rule mining algorithms with relational database systems. In *Proceedings of the International Conference on Enterprise Information Systems (ICEIS 2001), Setubal, Portugal*, 2001.

J-U. Kietz and M. Lübbe. An efficient subsumption algorithm for inductive logic programming. In S. Wrobel, editor, *Proceedings of the 4th International Workshop on Inductive Logic Programming*, volume 237, pages 97–106. Gesellschaft für Mathematik und Datenverarbeitung MBH, 1994.

A. Knobbe, A. Siebes, H. Blockeel, and D. van der Wallen. Multi-relational data mining, using UML for ILP. In *Proceedings Workshop Multi-Relational Data Mining (MRDM 2001)*, pages 49–60, 2001.

C. Kuok, A. Fu, and M. Wong. Fuzzy association rules in large databases with quantitative attributes. In *ACM SIGMOD Record 27*, pages 41–46, 1998.

R. Meo, G. Psaila, and S. Ceri. A new SQL-like operator for mining association rules. In *Proceedings of the 22th Conference on Very Large Data Bases*, pages 122–133, 1996.

S. Nijssen. Data mining using logic, Master's Thesis, Leiden University, 2000.

S. Nijssen, 2001. URL `http://www.liacs.nl/home/snijssen/farmer`.

S. Nijssen and J. Kok. Faster association rules for multiple relations. In *Proceedings of the the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 891–896, 2001.

G. Piatetsky-Shapiro. Discovery, analysis, and presentation of strong rules. In G. Piatetsky-Shapiro and W. Frawley, editors, *Knowledge Discovery in Databases*, pages 229–248. AAAI Press, 1991.

W. Pijls and J.C. Bioch. Mining frequent itemsets in memory-resident databases. In E. Postma, editor, *Proceedings Eleventh Belgium/Netherlands Artificial Intelligence Conference*, pages 75–82, 1999.

G.D. Plotkin. A note on inductive generalization. In B. Meltzer and D. Michie, editors, *Machine Intelligence 5*, pages 153–163, Edinburgh, 1969. Edinburgh University Press.

R. Rymon. Search through systematic set enumeration. In *Proceedings of the Third International Conference on Principles of Knowledge Representation and Reasoning*, pages 539–550, 1992.

S. Sarawagi, S. Thomas, and R. Agrawal. Integrating association rule mining with relational database systems: Alternatives and implications. In *Proceedings of the ACM SIGMAD International Conference on Management of Data*, pages 343–354, 1998.

R. Srikant and R. Agrawal. Mining generalized association rules. In *Proceedings of the 21th Conference on Very Large Data Bases*, pages 407–419, 1995.

D. Tsur, J.D. Ullman, S. Abiteboul, C. Clifton, R. Motwani, S. Nestorov, and A. Rosenthal. Query flocks: a generalization of association-rule mining. In *Proceedings of the 1998 ACM SIGMOD Conference on Management of Data*, pages 1–12, 1998.

J.D. Ullman. *Fundamentals of Database and Knowledge-base Systems*, volume 1. Computer Science Press, 1988.

P. Wong, P. Whitney, and J. Thomas. Visualizing association rules for text mining. In G. Wills and D. Keim, editors, *Proceedings of IEEE Information Visualization '99*. IEEE CS Press, 1999.