Multirelational Association Rule Mining

Anton Flank

7th September 2004

Abstract

Data mining is a broad term used to describe various methods for discovering patterns in data. A kind of pattern often considered is association rules, probabilistic rules stating that objects satisfying description A also satisfy description B with certain support and confidence. Association rule mining has previously been restricted to data in a single table, due to the computational complexity of multirelational algorithms. Recently, many methods for mining relational data in multiple tables have been proposed. This thesis explores some of these concepts, aimed at mining multirelational association rules from a standard relational database.

Mining is here formulated as a search problem in the space of database queries and $\mathcal{L}$, a subset of tuple calculus that is decidable for satisfiability and syntactic query difference, is used as query language. The search space is ordered based on subsumption by arranging queries in a refinement graph which is built top down. New queries are formed by selecting a node in the graph by a uniform cost strategy and applying a refinement operator, making the query more specific by adding conditions. Support and confidence threshold parameters are used to sort out uninteresting rules. Rules are presented as plain English (or Swedish) sentences, and the user has the option of guiding the search by expressing a positive or negative opinion about discovered rules.

Contents

Chapter 1

Introduction

Data mining means to mine patterns from data as an important step in the larger process of knowledge discovery in databases. The term data mining is therefore somewhat of a misnomer, but the name stuck. In this work we focus on mining of *association rules*. For an example of a association rule, consider a database containing transactional data from a store selling computer equipment. Then the following association rule might be discovered: 30% of the customers buying a color printer will also buy a scanner.

Traditionally, mining has been performed on data stored in a single table. Recently however, many methods for mining multirelational data have been proposed. In this thesis we shall assume that data is stored in a standard relational database and that mining must be available on data stored in multiple tables.

1.1   Organization of this thesis

Chapter two of this thesis contains a background section that tries to provide a brief overview of the field of data mining. The concepts of association rules are presented in detail as it is central to the work presented here. To make the overview broader, other techniques such as decision tree induction and clustering are also covered briefly. Chapter 3 describes what motivates moving up to handling multirelational data, and the approach we take. Concepts such as query subsumption and the query language $\mathcal{L}$ are presented. Chapter 4 gives an overview of the implementation of a prototype system and also a test run. Finally, chapter 5 contains the discussion, with possible extensions of the system as well as a comparison to prior related work. The last section contains the conclusions.

Chapter 2

An overview of Data Mining

Data mining is often viewed as a step in the larger process of *Knowledge Discovery in Databases*, or KDD for short, which consists of a series of steps: cleaning, integration, selection, transformation, data mining, evaluation and finally presentation [10]. It is tightly coupled to the use of data warehouses. It employs algorithms that guide the search and is therefore different from *online analytical processing*, or OLAP, although the two can and often are used together [10].

From the start, one of the major aims of the research has been to make mining of very large, real life databases practical. Very large can nowadays mean a terrabyte or more. Loosely speaking, the techniques comes from *statistics* and *machine learning*, with the same intention of finding interesting patterns hidden in the data. Traditionally many such algorithms have made the assumption that all data will fit in main memory, so much effort has gone into handling larger databases, improving on efficiency and parallelization [4]. Many of the algorithms used in the field are prime candidates for parallel implementations, because often similar operations are performed on a large set of tuples with little or no interdependence.

Data mining is used to build *predictive* and *descriptive* models. A predictive model is used to explicitly predict values. As an example, based on the customers who have responded to an offer, the model can predict what other customers are most likely to respond to the same offer. Descriptive models on the other hand describe patterns in existing data. It can provide valuable information, such as identifying different customer segments [8].

2.1 Transaction data

Analysis of transaction data is very common in data mining. In its barest form, a *transaction* consists of a transaction id and a set of the *items*[1] that are part of the transaction. Usually there will also be additional information, such as the date of the sale, and information about the customer and salesperson.

---

[1] Also called a *market basket*.

Transaction id	items
1	{printer, laptop, scanner}
2	{printer, scanner}
3	{printer, screen}
4	{laptop, keyboard, modem}

Figure 2.1: Transactions.

An example of transaction data is in figure 2.1. In transaction data analysis one is interested in finding out which sets of products that are frequently bought together. The knowledge found through this kind of analysis can assist in planning store layout or to decide which customers are likely to respond to a offer.

2.2 Association rules

Patterns mined from transaction data often come in the form of *association rules*, which have the following definition [14].

Let $\mathcal{I} = \{i_1, i_2, \ldots, i_m\}$ be the set of all items. Let $\mathcal{D}$ be the set of all transactions, where every transaction $\mathcal{T}$ is a set of items $\mathcal{T} \subseteq \mathcal{I}$. An association rule is of the form $X \Rightarrow Y$ where $X \subset \mathcal{I}$, $Y \subset \mathcal{I}$, and $X \cap Y = \emptyset$. For example $\{printer\} \Rightarrow \{keyboard\}$ is an association rule from the data in figure 2.1.

The *support* for a set of items $\mathcal{T}$ is the probability of that itemset appearing in a transaction: $P(\mathcal{T})$. The support for an association rule $X \Rightarrow Y$ is the probability $P(X \cup Y \subseteq \mathcal{I})$. To compute the support, the number of transactions containing all the items in both X and Y is divided by the total number of transactions. In figure 2.1, the rule $\{printer\} \Rightarrow \{scanner\}$ has support $\frac{1}{2}$.

The *confidence* of an association rule is defined as $P(Y|X)$. It is computed by dividing the number of transactions containing Y, by the number of transactions containing X. The example rule has confidence $\frac{2}{3}$, and so it can be said that "67% of people buying a printer will also buy a scanner."

To weed out uninteresting rules, threshold values are applied, so that for a rule to be reported, its support and confidence must exceed the user supplied parameters α and β respectively. Such rules are called *significant* rules. Even significant rules may not necessarily be interesting as they might simply represent common sense knowledge. In the end only the user can subjectively decide what rules are interesting. However, using objective measurements is a requirement for not overwhelming the user with useless rules.

The association rules handled here are probabilistic so they do not share all the properties of rules which require strict satisfaction (functional dependencies). In particular, from the presence of a rule $X \Rightarrow A$ it can not be inferred that the rule $X \cup Y \Rightarrow A$ holds, because it might not have sufficient support. In the same manner it does not follow from the rules $X \Rightarrow Y$ and $Y \Rightarrow Z$ that the rule $X \Rightarrow Z$ holds, because it might not have minimum confidence [5].

Support and confidence are the most commonly used measures of the importance of a rule. There are also other measures, for example *lift*, which can contribute useful knowledge. Assume the following probabilities for two itemsets

X and Y in market basket data: $P(X) = 0.6, P(Y) = 0.75, P(X \cap Y) = 0.4$. The following rule is then discovered: $X \Rightarrow Y$, with 40% support and 66% confidence. This rule can be misleading in that it might lead the user to believe that a purchase of X implies a purchase of Y. In fact it is the other way around, a purchase of X decreases the likelihood of purchasing Y, because they are negatively correlated. Correlation is measured by:

$$corr_{X,Y} = \frac{P(X \cup Y)}{P(X)P(Y)} = P(Y|X)/P(Y),$$

also called the *lift* of the rule $X \Rightarrow Y$. If this value is less than 1 the occurrence of X is negatively correlated with, it discourages, the occurrence of Y. A value greater than one means that the occurrence of X implies the occurrence of Y. If the lift equals 1, then X and Y are independent [10].

2.3 Apriori

Discovering association rules is usually divided into two steps. In the first step, all sets of items with *support* $\geq \alpha$ are found, called the *frequent* itemsets. In the second step the frequent itemsets are used to discover the association rules.

Perhaps the most well known algorithm in the data mining field is the Apriori algorithm for finding frequent itemsets. It was developed by the Quest team at IBM [1]. From that research has then sprung a large number of research papers and patents. IBM currently holds around 40 patents related to data mining.

The algorithm description of Apriori is given in figure 2.2, where L_k denotes the set of frequent itemsets with k items. C_k are called candidate itemsets, it is all the itemsets that have a potential to be frequent.

```
 1: L_1 := { frequent 1-itemsets }
 2: for k := 2; L_{k-1} ≠ ∅; k := k + 1 do
 3:    C_k := apriori_gen(L_{k-1})
 4:    forall transactions t ∈ D do
 5:       C_t := subset(C_k, t)
 6:       forall candidates c ∈ C_t do
 7:          c.count := c.count + 1
 8:       end
 9:    end
10:    L_k := {c ∈ C_k | c.count ≥ α}
11: end
12: Answer := ⋃_k L_k
```

Figure 2.2: Algorithm Apriori.

A key observation exploited in the algorithm is that all subsets of a frequent itemset are also frequent. The first step when generating the frequent k-itemsets is therefore to join together frequent itemsets with $k-1$ items, to form C_k. This is the function of the `apriori_gen` function on line 3. The second step is then to delete all itemsets from C_k that have a subset that is not frequent. This is the job of the `subset` function on line 5.

The following example is taken from [5].

Let $L_3 = \{\{1,2,3\}, \{1,2,4\}, \{1,3,4\}, \{1,3,5\}, \{2,3,4\}\}$. After step 3, $C_4 = \{\{1,2,3,4\}, \{1,3,4,5\}\}$. The itemset $\{1,4,5\}$ is not in L_3, so $\{1,3,4,5\}$ can not be frequent and will therefore be pruned from C_4 in step 5. The algorithm concludes *a priori* that the only possible large itemset of size 4 is $\{1,2,3,4\}$ without considering the transactions in the database, thus the name of the algorithm.

A key issue of the performance of the whole Apriori algorithm is the performance of the subset function. A fast way of determining which candidate itemsets that are contained in a transaction is needed. This can be achieved by storing the candidate itemsets in a structure called a *hash tree*. Itemsets are contained in the leaf nodes of the tree. All nodes are initially created as leaf nodes but can later be converted to internal nodes if they contain to many itemsets. Internal nodes are hash tables where each bucket points to another node. In the root node the branch to follow is decided by applying a hashing function to the first (lowest numbered) item in the itemset. At the next level of the tree, the hashing function is applied to the second item and so on.

New improvements of the Apriori algorithm are constantly being invented, improving on parallelization and handling larger databases. One possible improvement is to only use the database to count the support of 1-itemsets. Then, after the first pass, the database is not used to compute the support at all. This is achieved by maintaining bookkeeping about which itemsets are present in each transaction. In early passes it may be the case that the data structures involved for bookkeeping on the frequent itemsets will be larger than the actual database. However, in later passes it is expected that these structures will be much smaller than the database. Therefore, in a hybrid solution, plain Apriori is used for the early passes, but when it is expected that the bookkeeping data structures will fit in main memory, a switch is made to the alternative algorithm. Experiments show that with this improvement, Apriori scales linearly with the number of transactions [5].

When the database is large, different sampling techniques can be used. The basic idea is to run the algorithm on a randomly selected subset of the transactions. There is a risk that some itemsets are missed, because they happen to not be frequent in the subset. However, by using lower thresholds on support and confidence the risk can be lessened. After the itemsets have been discovered, a scan through the whole database is made to compute the actual support of the frequent itemsets [10].

2.4   Concept hierarchies

Often there are hierarchies present among the items in transaction data. For example, some items in a computer store could be divided into categories as shown in figure 2.3. If the application area has natural hierarchy structures, rules including items from across hierarchies may be interesting, while rules including only items from the same category are not [9]. For example, the rule {"Color printer"} $\Rightarrow$ {"High res scanner"} might be interesting. Going up the hierarchy makes it more likely to find significant rules, the rule {"Scanner"} $\Rightarrow$ {"Printer"} is more likely to have sufficient support than the previous rule. On the other hand, rules discovered at the top levels of the hierarchy are more likely to just represent common sense knowledge [10].
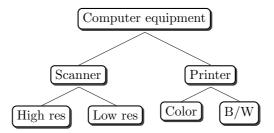
Figure 2.3: Concept hierarchy.

The way to mine rules across hierarchies is usually in a top down manner. This is because as the child nodes represent refinements of the parent node, the support is decreasing as we go down the tree. A problem is that it is expected that items at deeper levels will occur less frequently, so it might be a good idea to have lower support thresholds at deeper levels. This will complicate the search procedure however, because then the child of a node that is infrequent can still be frequent if the threshold for the child level is lower than at the parent level [10].

2.5 Classification, prediction and clustering

Both classification and prediction are used to build models that help predict future data values. The difference is that classification predicts the categorical label of a tuple, while prediction models a continuous-valued function [10].

The process of classification begins by identifying one of the attributes of the tuples as the *class label*. The data set that is used to build the model is called the *training* data set. Because the tuples in the training data has a provided class label this is a supervised learning method. In the second step, the model is evaluated. Usually this involves a *test* data set that is independent of the training data. The model is used to classify the test data and the result is compared to the class labels of the test data. If a high ratio of correctly classified tuples is obtained, the model can be used to classify new tuples with unknown class label. It is important to keep test and training data separate. Most classification methods are susceptible to *overfitting*, that is given enough training they learn the structure of the training data. A learned model can, for example, be represented by classification rules, decision trees or mathematical formulas. In this section the decision tree method for classification is covered, followed by the k-means algorithm as an example of clustering.

2.5.1 Decision trees

A decision tree is a tree where every internal node represents a question regarding an attribute of the tuple and the leaf nodes contain a class label. An example of a decision tree is shown in figure 2.4. Here the tree is used to predict which customers are likely to respond to an offer.

To classify a tuple, we start with answering the question in the root node, and choosing the path accordingly. The questions must divide the tuples into

disjoint sets so that only a single path can be taken from every node. Eventually a leaf node is reached and the tuple is then assigned to that class.

In general decision trees can have nodes with an arbitrary number of children. However, all trees can be converted to a binary tree and questions at internal nodes are then reduced to having yes/no answers. Questions could also potentially involve an arbitrary number of attributes, but to maintain interpretability it is favorable to use questions about a single attribute.

Decision trees have a number of favorable properties. Classification is fast and straight forward, usually only a few simple questions to answer and then a classification is obtained. The answer is also easy to interpret, it is easy to see how the classification algorithm came up with its answer by following the path from the root to the leaf node [6].
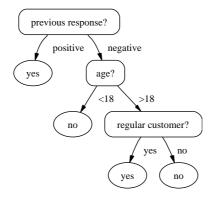

Figure 2.4: A decision tree

Starting with a training set of labeled tuples, building a decision trees is usually done top down in a greedy manner. At every step the tuples remaining are investigated. If all have the same class label we obviously are done and can create a leaf node with that class label. Otherwise we have the option to either create a leaf node with the class label that best represents the remaining tuples, or to split the data. If the data is split, the procedure is then called recursively for the two datasets that results from the split.

To split the data in a optimal way, it should be split so as to make the data that goes to the children as "pure" as possible. This is the area of *information theory*. The most frequently used measure is the *entropy impurity* [6]. For N samples and m different class label values, where $P(\omega_j)$ designates the probability of a tuple having class label j,

$$i(N) = -\sum_j P(\omega_j) \log P(\omega_j)$$

$i(N)$ is zero when all tuples belong to the same class, and peaks when the class label values have equal frequency. To split the data is then a question of evaluating each possible question and choosing the one that results in the greatest decrease in impurity.

An important question is to decide when to stop splitting. If splitting is carried out until all leaf nodes represents a set of tuples in the test data that

share the same class label value, that is zero impurity, the tree will classify the test data with total accuracy, but will be useless on new unseen tuples.

2.5.2 k-means clustering

Whereas classification is the task of assigning class labels to tuples, clustering is the task of discovering classes. There are no labeled data to train the algorithm on, and so it is said that clustering is an unsupervised learning.

This is done by grouping the data into clusters. A general description of a cluster is that the tuples that lie in the same cluster should be similar, and they should be dissimilar to tuples that are not in the same cluster [10]. Clustering can be used as a preprocessing step for classification. It can also be used as a tool by itself, to identify different segments in the data.

The most often used measure of evaluating a clustering is distance. We then naturally want the distance between tuples belonging to the same cluster to be less than the distance to tuples in different clusters. When distance is used as measure, a *metric* is needed. A general form is the *Minkowski metric* [6]:

$$d(\mathbf{x}, \mathbf{x}') = \left(\sum_{k=1}^{d} |\mathbf{x}_k - \mathbf{x}'_k|^q \right)^{1/q}$$

When $q = 2$ it is more known as the Euclidean metric. Setting $q = 1$ gives the *Manhattan* or "city block" distance.

So how do we assess the result of a clustering algorithm? The simplest and most widely used criterion function for clustering is the sum-of-squared-error function

$$J_e = \sum_{i=1}^{c} \sum_{\mathbf{x} \in \mathcal{D}_i} \|\mathbf{x} - \mathbf{m}_i\|^2,$$

where c is the number of clusters, $\mathcal{D}_\rangle$ is the tuples in cluster i and m_i is the mean value of the tuples in that cluster.

Input n, number of clusters.
                  Samples, unlabeled data samples.
Output   Cluster centers.

  1: Initialize all cluster centers μ_i to a random sample.
  2: **repeat**
  3:    **forall** samples **do**
  4:      Classify the sample according to nearest μ.
  5:    **end**
  6:    **forall** μ_i **do**
  7:      Recompute μ_i as the mean value of the samples belonging to that cluster.
  8:    **end**
  9: **until** No change in $\mu_1, \mu_2, \ldots, \mu_n$.
10: **return** $\mu_1, \mu_2, \ldots, \mu_n$.

Figure 2.5: Algorithm, k-means clustering

An example of a clustering algorithm is the k-means algorithm. It is given in figure 2.5. The k-means algorithm is an example of an iterative optimization technique. Like all algorithms in this category it is only guaranteed to find a locally optimal solution, but not the global optimum.

A major drawback of the k-means algorithm is the need to specify the number of clusters. A direct approach is to execute the algorithm for increasing values of k. By observing the behavior of the error criterion, a good value of k can often be determined if the data is well behaved. It is then expected that the error decreases rapidly until the optimal value, where after it decreases much more slowly until finally the number of clusters equals the number of data points [6].

Chapter 3

Mining over multiple relations

Because efficiency is a high priority due to the size of the databases that are handled, data has traditionally been kept in a single table. Also a large body of work is available for methods handling data from a single relation. However, single relation approaches can often be successfully extended to handle multirelational data. This includes association rules, decision trees and distance based methods. To "upgrade" a single table algorithm, it is necessary to extend the key concepts. For example, to upgrade a clustering algorithm, it is necessary to define the notion of distance between tuples in the multirelational case. If this is done successfully, the rest of the algorithm can often be used more or less intact and will have the single table algorithm as a special case [7].

Moving up to a multirelational algorithm can have several advantages. There is no longer a need to integrate the data, by joining and aggregation, to make the data fit in a single table. Integration into a single table can cause loss of information. In some cases many tricks have to be applied to capture the correct information in a single table, sometimes it is impossible. Data integration can also result in a table that is too large to handle. The disadvantage of many multirelational algorithms is that they are computationally intensive and do not scale well. But as computers get faster and cheaper and advances in parallel computing continues it gets more natural to move up to multirelational algorithms.

In order to talk about multirelational association rules, the notion of a frequent query and query subsumption need to be defined. Instead of the frequent itemsets of the single relation case, in the multirelational case there are frequent queries. The support of a query l is simply the number of tuples it returns: $|\{x|l\}|$. To talk about the relative support of a query, the support of the query needs to be compared to something. For this purpose a *focus*[1] query is chosen. The relative support of a query is then the number of tuples returned by the query divided by the number of tuples returned by the focus query. A frequent query is one that has higher support than a user supplied constant α.

For two queries l and l', l *subsumes* l' if $|\{x|l'\}| \subseteq |\{x|l\}|$. Subsumption

---

[1]Some authors use the word *key* for this, which in the context of relational databases has obvious meaning. Therefore the word *focus* is used instead.

is a semantic notion, it holds for all possible database instances, so the tuples returned by l' are always a subset of the tuples returned by l. The query l' is also said to be a *specialization* or *refinement* of l.

A multirelational association rule $l \Rightarrow l'$ exists when l and l' are two queries and l subsumes l'. The support of the rule is given by $|\{x|l'\}|$ and confidence is given by $\frac{|\{x|l'\}|}{|\{x|l\}|}$. As in the single relation case, a significant rule is one with higher support and confidence than the user supplied constants α and β [7].

These concepts will now be illustrated with an example over the database with cars and their owners in figure 3.1. Primary keys are underlined and foreign keys are printed in **bold face**.

Person

pnr	name	gender	age
1	John	M	32
2	Kalle	M	21
3	Karin	F	32
4	Anna	F	40

Car

regnr	company	model	owner
1	Volvo	740	1
2	Volvo	V70	1
3	Volvo	V70	2
4	Volvo	240	3
5	BMW	Z4	4

Figure 3.1: Example database

Query	Phrase	Support
$l_1 = \{x\|Person(x)\}$	"People"	4
$l_2 = \{x\|Person(x) \land$ $(\exists y_1)(Car(y_1) \land y_1.owner = x.pnr \land$ $y_1.company = \text{'Volvo'})\}$	"Volvo owners"	3
$l_3 = \{x\|Person(x) \land$ $(\exists y_1)(Car(y_1) \land y_1.owner = x.pnr \land$ $y_1.company = \text{'Volvo'} \land y_1.model = \text{'V70'}\}$	"Volvo V70 owners"	2

Figure 3.2: Example queries

Some example queries are given in figure 3.2. Given the focus query is l_1 the relative support of these queries would be 1.0, 0.75 and 0.5 respectively. Three association rules are present: $l_1 \Rightarrow l_2$ ($\alpha = 0.75, \beta = 0.75$) says that 75 percent of the people own a Volvo; $l_2 \Rightarrow l_3$ ($\alpha = 0.5, \beta = 0.67$) says that 67 percent of the Volvo owners own a Volvo V70[2]; $l_1 \Rightarrow l_3$ ($\alpha = 0.5, \beta = 0.5$) says that 50 percent of the people own a Volvo V70.

---

[2]Note that the rule does not say that if a person owns a Volvo, 67 percent of the time *that* Volvo is a V70.

3.1 Our approach

In this thesis a prototype system for mining of multirelational association rules is presented where the input is a standard relational database with the relevant tables and attributes marked. Discovery of rules is accomplished by a systematic search in the space of database queries. To tackle the problem of a very large, potentially infinite, search space of queries, three basic strategies are used: first, the search is systematic and orders queries based on subsumption; second, support and confidence thresholds are used to guide the search and sort out uninteresting rules; third, mining is interactive. As rules are found they are presented to the user which then has the opportunity to guide the search. If an interesting rule is found and the user gives a positive response, the search is directed to try to produce more rules similar to the interesting one. On a negative response, search is ended in that direction. When user feedback is absent the system falls back on uninformed search. Rules are presented in plain English or Swedish sentences, making it possible for the user to quickly understand the meaning of a rule before taking action and moving on.

Focus will be on developing algorithms for mining multirelational association rules and providing clear natural language descriptions, not on handling vast amounts of data or speed of execution.

3.2 The $\mathcal{L}$ query language

All queries handled by the system are specified in the $\mathcal{L}$ query language, a subset of the tuple relational query language that disallows projection [12].

Queries have the form: $\{x|R(x)\wedge\Phi\}$, where Φ is a conjunction of conditions:

- Range conditions $R(y)$ which restricts the variable y to range over the relation R in the database schema.

- Join conditions $X\theta Y$, where $\theta \in \{<,>,\leq,\geq,=,\neq\}$ and X,Y are component references. A component reference $x.a$ selects the attribute a of the relation that the variable x ranges over.

- Simple conditions $X\theta Y$, where $\theta \in \{<,>,\leq,\geq,=,\neq\}$ and one of X,Y is a component reference and the other is a constant.

- Set conditions $X\epsilon C$, where X is a component reference, C is a set of constants, and ϵ is one of the set membership operators $\{\in,\notin\}$.

- Component conditions $s(\exists y_1)\ldots(\exists y_n)(C')$, where C' is the conjunction of any number of non-component conditions and the sign is determined by s, $\neg$ for a negated component.

$\mathcal{L}$ has a number of favorable properties. For any query it can be determined if it is satisfiable. Unsatisfiable queries will never return any tuples no matter what the state of the database is. Also, the syntactic difference between queries can be determined. Combining these two makes it possible to determine if one query subsumes the other, or if two queries are equivalent.

Queries in $\mathcal{L}$ are easily translated into SQL. For example, query l_1 from figure 3.2 is equivalent to the following SQL query:

```
SELECT distinct *
FROM   Person as x
WHERE  exists(select * from Car as y1 where
              y1.owner = x.pnr and y1.color='Blue');
```

Support for a query is computed by translating the $\mathcal{L}$ query into a SQL query of the form `select count(*) from X`, where X is the result of translating the query as above.

STEP is a system for manipulating queries in $\mathcal{L}$ and the more general $\mathcal{Q}$ query language. It is used in this system for manipulating query expressions and generating natural language descriptions. For more information about STEP, see [13] and the online demo [3].

3.3   Refinement graph

A useful ordering of the space of database queries is that of a refinement graph [7]. Part of the refinement graph for the example database in figure 3.1 is shown in figure 3.3.
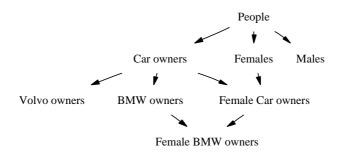

Figure 3.3: Refinement graph

A important observation is that support is nonincreasing when moving from parent to child, that is from a more general to a more specific query, in the graph. The graph itself is directed and acyclic: a node can have multiple parents and children, and there are no cycles.

For any two nodes l and l', if l subsumes l' then l is a ancestor to l' in the graph. This is a requirement for using the graph to extract all valid association rules.

Only refinement graphs with a single node at the topmost level will be considered here, where the root node contains the focus query of the search. Also, when a query l subsumes a query l', we require that l' is at a greater distance from the root node than l, this makes insertion of new nodes in the graph easier.

Extracting all valid association rules present in a set of queries is straight forward using the refinement graph, because a association rule $l \Rightarrow l'$ exists whenever l is an ancestor to l' in the graph. Usually we are also only interested in the significant rules, algorithm is given in figure 3.4.

We use a search algorithm that starts with the focus query and builds the refinement graph top down. A uniform cost strategy is used to select a node for

Input   Refinement graph

Output   All significant association rules

1: $R := \emptyset$.
2: **forall** nodes l in graph **do**
3:    **forall** Descendants l' of node in graph **do**
4:       **if** support, $|\{x|l'\}| \geq \alpha$ and
        confidence, $\frac{|\{x|l'\}|}{|\{x|l\}|} \geq \beta$ **then**
5:         $R := R \cup \{l \Rightarrow l'\}$.
6:       **end**
7:    **end**
8: **end**
9: **return** R.

Figure 3.4: Algorithm, extract association rules

expansion. Refined queries are then formed from the selected node by applying a *refinement operator* and the resulting nodes are inserted into the refinement graph.

3.4   Search strategy

Uniform cost search is used to choose which node to expand. Every node has an associated cost and the node with lowest cost is expanded first. For a new node, the cost is computed by the cost of the expanded node plus an additional cost computed from the operator used and the support of the new node. Cost is specified for the following operations:

- Adding any type of condition, except a join condition, to the query.

- Adding a join condition. The cost for this operation is kept higher than the cost for other conditions in order to favor shorter rules. Additionally it is possible to include a penalty cost for every time a join with a certain relation is used.

- The cost is modified based on the support of the resulting query. This value should be negative so that the search favors nodes with high support. By default this value is the relative support of the node multiplied by $-\frac{1}{100}$:th of the cost of adding a condition.

- The cost is modified based on the confidence of the association rule represented by the node and the parent node from which it was generated.

- A maximum cost threshold may be specified so that the search will simply disregard nodes that have a higher cost than this value.

Some of the cost modifiers are negative. This could potentially lead to a node having less cost than the one from which it was generated, and even negative cost. This is in principle not a problem, however a lower bound of zero on the cost is maintained.

3.5 Refinement operators

When a node is expanded, new nodes are generated by applying a refinement operator. To discover all valid association rules, all possible database queries given the marked attributes in the configuration must be generated. Refinement operators must therefore be applied in a systematic way so that it is known when all possible refinements of a query have been generated. It is also desirable to avoid duplicates: generating two equivalent queries. The algorithm is given in figure 3.5.

Input   Database, configuration

Output   Refinement graph

```
 1: Create root node with focus query and relation stack containing the focus relation.
 2: Initialize node queue containing the root node.
 3: while node queue not empty do
 4:     Dequeue node queue.
 5:     Let R := top of relation stack of node.
 6:     while attribute stack of R not empty do
 7:        forall child nodes in refine(node, pop(attribute stack)) do
 8:           Copy relation stack to child.
 9:           if refinement added a join with relation S then
10:              Push S on relation stack of child
11:           end
12:           insert(node, refinement graph)
13:           if Node was not duplicate and has enough support then
14:              Compute cost of child and enqueue on node queue.
15:           end
16:        end
17:     end
18:     Pop relation stack.
19: end
```

Figure 3.5: Algorithm, node generation

Note that queries with low support are not added to the node queue, because they cannot produce any refinements that have sufficient support and can therefore be discarded.

Duplicates will not be generated, except in the case of multiple joins with the same relation. This would happen for example if two relations R_1, R_2 exists with R_1 having a foreign key attribute to R_2, and R_2 a foreign key attribute to R_1. It is up to the function `insert`, that inserts new nodes into the refinement graph, to detect duplicates. Because variables in queries are existentially quantified, duplicates can not always be totally discarded, consider figure 3.6. Nodes 4 and 7 are equivalent and only one of them is kept, whichever is generated first, the other node is discarded. Nodes 2, 3 (and 5, 6) are also equivalent, however nodes 3 (or 6) cannot be discarded because then node 4 (or 7) would not be generated. The problem is currently handled by only purging duplicates resulting from adding non-join conditions; a duplicate resulting from adding a join condition is removed from the graph, but still kept in the node queue.

The function `refine` takes a query and a attribute as parameters. In the configuration the user specifies which operator is applicable to each attribute.
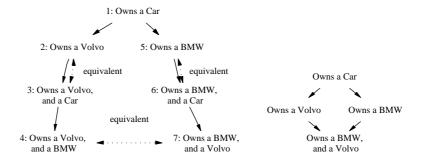
Figure 3.6: left: Search tree with equivalent queries. Right: Refinement graph

There are two basic kinds of operators, those that add simple conditions and those that add component conditions.

3.5.1 Adding simple conditions

The most basic operator adds a simple condition to the query, for example: $\{p|Person(p)\} \rightsquigarrow \{p|Person(p) \land p.age > 50\}$. That is, it adds constraints on a single attribute of a relation. Conditions added by this type of operator are either simple conditions or set conditions. Some operators can also add more than one condition in a single refinement. The following types of attributes are distinguished:

Nominal The value of the attribute belongs to a set of distinct values. No ordering among the attributes is present, so set operators are allowed, but not arithmetic operators.[3] For example the attribute `pet` might be a nominal attribute with the values $\{cat, dog\}$.

Ordinal attributes are set-valued, like nominal attributes, but there also exists an ordering among the values, partial or total, so that the arithmetic operators can be applied. For example the attribute `brightness` might take on the values $\{light, medium, dark\}$.

There are many possible ways to add conditions for ordinal valued attributes, among which the following are used in this system:

- Open ended intervals: $brightness > medium$.

- Closed intervals: $brightness > light \land brightness < dark$. When using closed intervals, two conditions will be added for every refinement.

- Pick one. Treats the attribute as a nominal attribute.

With closed intervals comes the additional choice of deciding if all possible combinations of intervals should be used, or possibly just a subset. Also, closed and open ended intervals may be used in combination.

---

[3]except possibly '=' and '$\neq$'. But there is alway an equivalent expression using the set operators

For performance it is crucial to refine the intervals in the right order so that the refinement graph can be pruned in a efficient way. For example we want to generate the query $\{p|Person(p) \wedge p.age > 10 \wedge p.age < 65\}$ first, and then generate the query $\{p|Person(p) \wedge p.age > 18 \wedge p.age < 50\}$ as it is subsumed by the first query.

Intervals introduce complications with which rules should be considered important. If there is a decrease in support of 10% of the refined query this would normally represent a interesting rule with confidence 90%. However, if the query in the child is the result of shrinking the interval in the parent by ten percent then it is only expected.

Numeric attributes like integers and floats need to be split into intervals before they can be used. For example a attribute *age* can be split into the intervals <18, >18 and <65, and >65. This can be seen as turning the attribute into a ordinal or nominal valued attribute. In the configuration this would be done by specifying the attribute as set valued with the "split points" 18 and 65.

To obtain meaningful intervals is not always straight forward. Sometimes "natural" intervals exists, for example the intervals above divides persons into children, adults and seniors. If there is no such obvious division, different clustering techniques could be used. However, as queries are refined and tables are joined, this might impact the distribution of attribute values in unforeseen ways. But running the clustering algorithm every time a join is performed might be too inefficient. Another approach is to split the attribute variable into intervals of a sufficiently small granularity and then use the the search process itself to do the clustering so to speak. Additional complications arise if the attribute is not linearly scaled. Currently these issues are ignored by requiring that intervals be specified in the input configuration.

3.5.2 Adding component conditions

The first kind of operator added simple conditions. The second kind of operator adds a component condition to the query. For example the query $\{p|Person(p)\}$ could be refined to $\{p|Person(p) \wedge (\exists c)(Car(c))\}$. This allows the construction of queries that join tables in the database.

Assuming there is at least one car in the database, the two queries are equivalent; to be meaningful, there has to be a relationship between the variables. Therefore the operator will always add a join condition to the query in the same refinement, and the example query would be refined into $\{p|Person(p) \wedge (\exists c)(Car(c) \wedge c.owner = p.id)\}$. At present, the system will only add join conditions with the operator "=", these are commonly called equi-joins.

For a join to represent useful knowledge there must be a relationship between the tables involved. The participation of entities in relationships is part of the conceptual model of the database. There can be different types of constraints present on participation that impacts the way meaningful queries can be constructed. Of most importance here are the structural constraints [9]. At the representational level, the actual tables of the database, relationships are materialized by means of foreign keys.

The user can specify an attribute to be a foreign key to an attribute in another relation, or specify a primary key attribute to be *referenced by* a foreign key attribute in another relation. Both options will result in operators adding

a join with that relation. If an attribute is referenced by another attribute, the system is by default allowed to add *multiple* joins with that relation. This is not the case with a foreign key attribute, because by definition it designates a single tuple in the other relation. Talking at the conceptual level, multiple joins should be allowed on a referenced-by attribute if it has a (x, n) structural constraint in the relationship. This is the default assumption of the system. It allows generation of queries such as "A person owning a BMW, and a Volvo".

If a foreign key or referenced-by attribute has a $(x \geq 1, n)$ structural constraint, the support of the refined query will be the same as that of the parent. In such cases the computation of support can be skipped.

Information about relationships could in part be extracted automatically from the database. Most notably this includes foreign key and `not NULL` constraints. At present however, it is up to the user to specify the relationships in the input configuration.

Negated component conditions

A negative association rule is of the form "70 percent of customers who buy X do not by Y". Such rules can potentially be very interesting. Currently however, component conditions with negative sign are not allowed in queries in this system. The problems with negation are partly technical; query refinement is not as straight forward as with positive component conditions. This is because adding conditions to a negated component condition will weaken the negation, thus making the query more general. This violates how the system currently works under the assumption that operators adds conditions to a query and thereby refining it.

Disregarding the technical issues there is a serious problem that has to do with how to evaluate negative information. Consider the case with itemsets. It is expected that only a small fraction of the items will ever be bought together. If there is 1000 items in the inventory only a very small fraction of the 2^{1000} combinations of all items will appear even once in the database. A naive approach would generate a very large amount of negative association rules. Therefore prior knowledge must be exploited in order to find interesting negative associations. One method is to use concept hierarchies [9], support for this is however not currently implemented in this system.

3.6 Node insertion

As new nodes are generated by the algorithm in figure 3.5, they are inserted into the refinement graph by the function `insert`, see figure 3.7. Initially it is called with the root node of the graph as the first argument, it may later be called recursively. The second argument is the newly generated node.

The algorithm traverses the refinement graph top down and determines the parents of the new node in the graph; a parent node is a node that subsumes the new node, and does not have any children that subsumes the new node. For a example of a generated graph see figure 4.3 on page 24.

A query subsumed by an infrequent query is itself infrequent, this fact is used to prune infrequent queries. A node inserted that is later found to be infrequent after querying the database is kept in the graph. Later, if a query

Input   Node N in graph; new node

Output   refinement graph with new node inserted

 1: **forall** children of N **do**
 2:    **if** child not already visited, and child subsumes new node **then**
 3:       **if** child has low support **then**
 4:          **return** (from all recursive calls) infrequent query found.
 5:       **else**
 6:          Call recursively `insert`(child, new node).
 7:       **end**
 8:    **end**
 9: **end**
10: Mark N as visited.
11: **if** new node was not subsumed by any child **then**
12:    **if** new node subsumes N **then**
13:       **return** (from all recursive calls) duplicate query found.
14:    **else**
15:       Add N to the parents of new node.
16:    **end**
17: **end**

Figure 3.7: Algorithm, node insert

is subsumed by an infrequent query in the graph, it can be discarded directly without querying the database.

The algorithm only determines the parents of a new node but not the children. This puts restraints on what refinements are allowed and on the order of insertion. To generate the complete refinement graph, nodes must be inserted level by level, and a query must not subsume a query at equal or lower distance from the root of the graph. As an illustration, consider again the refinement graph from the example database in figure 3.3 on page 13. If the query "Car owners" had been inserted after the query "Female car owners" into the graph, the fact that the first query subsumes the second would not be caught. Because we are doing uniform cost, not level by level search, some edges from the refinement graph could be missing. The only solution to this problem so far has been to reinsert all nodes in the graph, level by level. This would minimally have to be done once, before gathering all rules from the graph. In practice it pays off to do it more often, because as the refinement graph is used to prune infrequent queries an incomplete graph leads to unnecessary database queries. A more satisfactory solution would of course be a insertion algorithm that maintains the complete refinement graph regardless of the order of insertion, or perhaps a revised search strategy.

There are some strategies that can be used to avoid unnecessary subsumption tests. As the number of tests for every insertion grows with the size of the graph, and testing in itself is a NP hard problem in the length of the queries [12], it is worthwhile to try to use them only when needed. First, the node that was used to generate the new node certainly subsumes the new node. That node and all of its ancestors are therefore marked as "subsumers" before executing the algorithm. When a check is made to see if the new node is subsumed by a node that is marked in this way, the check immediately returns true. Second, if a node l subsumes a node l', then all ancestors of l in the graph also subsumes

l'. Therefore a first step in algorithm 3.7 would be to mark all ancestors of N as subsumers.

3.7 Natural language descriptions

The system generates natural language descriptions of the rules that are found. Descriptions of $\mathcal{L}$ queries are generated by STEP from entries in its phrasal lexicon [13]. For the descriptions to correctly capture the meaning of the rules, entries for all attributes included in the search must be specified. Each entry consists of a query and a set of patterns, here are some entries for the database in figure 3.1:

```
<{x | Person(x)}: H[e sing   :'person'],
                  H[e pl     :'people']>

<{x | Car(x)}: H[e sing :'car'],
               H[e pl    :'cars']>

<{x | Car(x), x.model = $c1}: C[e  :'of $c1 model']>

<{x | Person(x), {Car(y1), y1.owner = x.pnr, _}}:
   C[e  :'owning D({y1 | Car(y1), _}, (e sing indef))']>
```

Each entry consists of a query and a set of *patterns*, where the type of the pattern can be for example `H` for head or `C` for complement. The *features* controls the applicability of a pattern, including selecting the language, singular/plural form and definite/indefinite form. More examples of entries can be found in appendix A.

We will not delve into the details of how exactly the natural language generation process works. Generating a description for a query is then, from the point of view of this system, simple. It requires a call to a function that takes two arguments, the query and the set of features. Given the above entries, the description of the query $\{x | Person(x) \wedge (\exists y)(Car(y) \wedge y.model = \text{`v70'} \wedge y.owner = x.pnr)\}$ with the features (`e`, `sing`, `def`) would read: "A person owning a car of v70 model."

Descriptions of multirelational association rules are always a single sentence and has three main parts. A description of the left hand side rule (LHS), a description of the right hand side rule (RHS), and the confidence of the rule. Additionally some template text is added to form a readable sentence. An example of a description is:

> *85%* of the *the tall people* was *a man.*

Here the three main components are in *italics* and the rest is template text. The LHS of the rule is presented in definite plural form, while descriptions of the RHS is in indefinite singular form.

The conditions shared among the LHS and RHS cause repetition in the description:

> Among the tall, old people with a red house and a cat, 60% of
> the time it was a tall, old person with a red house and a black cat.

To avoid repetition, all simple conditions and set conditions shared with the LHS are removed before describing the RHS. Applying this to the above rule gives:

> Among the tall, old people with a red house and a cat, 60% of the time it was a person with a black cat.

3.8   User interaction

One of the goals was to make the mining interactive. As rules are found they are presented to the user which then has the opportunity to express opinions about the rule. A negative response ends the search in that direction, while a positive response makes the system strive to produce more rules similar to the one presented.

The user must specify on which queries to take action. There are three ways of selecting queries. It can be based on the specific queries that are part of a presented rule or by a query manually specified by the user. It can also be based on specific relations or attributes that are used in a rule.

There are two types of actions the system can take. For undesirable rules it can kill the selected nodes; a killed node is removed from the node queue and not further considered for expansion. Alternatively, the system can introduce a cost modifier for the selected queries. Desirable queries get a negative cost modifier while undesirable queries get a positive modifier.

A session could look like this, triggered by the system having found a new rule:

```
(support 15%): 80% of the customers living in Umeå was a female.

Options: 1:Interesting  2:Not interesting  3:Continue

>> 2

Select queries at least as specific as:
1: "A customer living in Umeå"
2: "A female customer living in Umeå"
3: Select all queries containing attribute Person.gender.
4: Specify query or attribute manually.

>> 2

Action:
1: Kill
2: Low priority
3: Enter priority manually

>> 1
```

To present such a dialog every time a new rule is found would quickly become tedious for the user. A way of lessening the number of presented rules is to only present rules where the RHS is a newly generated node and the LHS is the direct parent of that node. Another way is to use threshold values on support and confidence for presenting a rule.

Chapter 4

Implementation

A prototype system was implemented that is presented in this section. A high level overview of the architecture is shown in figure 4.1.
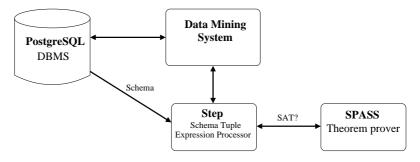

Figure 4.1: System architecture

The system uses a PostgreSQL database to store the data. Queries sent to the database will mostly be count queries to compute the support of queries. It uses STEP for equivalence/subsumption tests and for generating natural language descriptions of queries. STEP uses the theorem prover SPASS to determine satisfiability of queries [2].

4.1 Results

The system was tested on a toy database, containing transactional data. The schema for the database is shown in figure 4.2, only the attributes used in the mining are shown. Primary keys are <u>underlined</u> and foreign keys are printed in **bold face**.

The first thing to decide is *what* we want to mine for. In this case the focus will be on the `Transaction` relation, because we are interested in what items are bought together. Note that any relation can serve as the focus, for example `Customer` could be an interesting focus as well.

Next comes the configuration of the system. The complete configuration file is given in appendix A. Which attributes should be included in the search? We are interested in what cities items are sold, how/if the gender of our customers

Customer (cid, gender)

Item (iid, name)

Store (sid, **city**)

City (cityid, name)

Transaction (tid, **customer**, **store**, time)

Contains (**transaction**, **item**)

Figure 4.2: Schema for the test database.

influence what they buy, and at what time of day transactions take place. In
addition to the foreign key relationships, the types of the remaining attributes
have to be specified. Attributes `gender` in `Customer`, and `name` in `City` are nom-
inal valued. Once this is specified, the system will figure out the actual values
from the database. From the attribute `time` in `Transaction` we are interested
in the time of day that the transactions take place. We choose to divide it into
intervals: {*morning, before noon, after noon, evening*}, and therefore specify it
as an ordinal attribute with the "split points" {9, 12, 18}.

Configuration of natural language generation comes next. For example de-
scriptions for the intervals of the time attribute could be provided:

```
<{x | transaction(x), x.time = $c1 }: C[e :'at time $c1']>
<{x | transaction(x), x.time <= 9  }: C[e :'in the morning']>
<{x | transaction(x), x.time <= 12 }: C[e :'before noon']>
<{x | transaction(x), x.time <= 18 }: C[e :'in the afternoon']>
<{x | transaction(x), x.time > 18  }: C[e :'in the evening']>
```

Finally, the threshold values must be chosen. We chose to set a minimum
support value of 5%, and a minimum confidence threshold of 70%.

A testrun was made with this setup on a tiny database with 5000 trans-
actions, with a total number of 8000 item in all transactions together. The
database server was a UltraSparc I, 167MHz with 512Mb of RAM, running on
another host in the network while SPASS was running on the same host as the
data mining system, a Intel Pentium 4, 3GHz with 1Gb of RAM. The system
was run until it had expanded 1000 nodes, counting nodes for which all children
was generated, and nodes that was killed. This involved a total of 272 SQL
queries and 16509 calls to SPASS. The total execution time was 21 minutes,
real time, or about 1.3 seconds per node. Of the total runtime, 39 percent was
spent waiting for SQL queries to finish and 60 percent was spent waiting for
SPASS calls to finish.

Examples of rules that were discovered:

(support 0.1) 88.5% of the transactions containing a Printer, by a female
customer, in a city named Umea, was a transaction containing a Scanner.

(support 0.1) 80.7% of the transactions by a customer, containing a Scan-
ner, taking place in the morning, was a transaction by a female customer.

(support 0.1) 80.6% of the transactions by a customer, taking place in the
morning, was a transaction by a female customer.

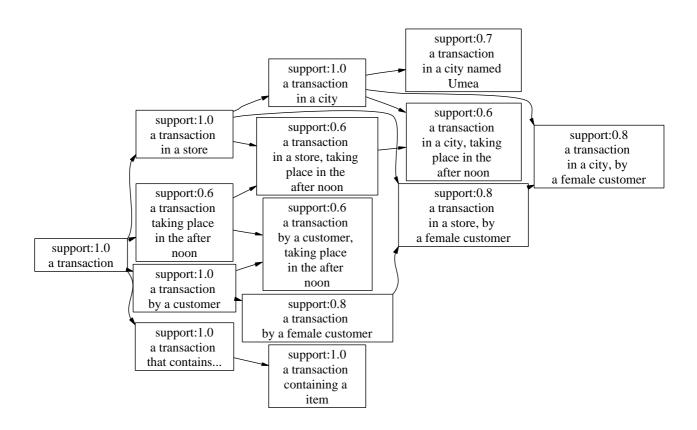See figure 4.3 for parts of the generated refinement graph.

Figure 4.3: Refinement graph over queries in transaction example.

Chapter 5

Discussion

5.1   Comparison to prior work

This work is related to other efforts in multirelational data mining. Other approaches taken in this field includes graph-based data mining and approaches influenced by inductive logic programming which often use Datalog as their query language. One example of an early such system is WARMR which is presented in this section.

WARMR is a system for discovering frequent Datalog queries [11]. Although a discussion of the foundations of Datalog is beyond the scope of this thesis it can be said that it shares the same expressive power as relational algebra (and calculus), but Datalog also has recursion. Concepts such as *ancestor* which can not be expressed in relational algebra can therefore be expressed in Datalog.

The WARMR system is similar to this system in that it uses the notion of a focus query, and it uses refinement operators such that a query subsumes any queries that are refined from it. WARMR is more general than this system in that it uses Datalog which is more expressive than $\mathcal{L}$. However, Datalog is often too expressive, so there is a need to restrict how queries are formed. To restrict the refinement that can be made to a query, a *declarative language bias* is used. In addition to specifying the focus, the bias settings also specify input-output modes for variables. Atoms are allowed to appear multiple times in a query as long as the modes are obeyed:

+ Variable is *input*. It has to be bound before the atom is encountered.

− Variable is *output*. It is bound by the atom.

± Variable can be both input and output.

Assuming the language bias specification is $\{p(+,-), q(-)\}$, valid queries include $? - q(X), p(X, Y), p(Y, Z)$. But the query $? - p(X, Y)$ is invalid.

Specifying correct declarative biases, WARMR has the Apriori algorithm as a special case. It also has a number of approaches for discovering frequent itemsets, such as concept hierarchies and sequential pattern discovery as special cases [7]. The algorithm for WARMR is given in figure 5.1.

Input   Database **r**; Declarative language bias $\mathcal{L}$ and *focus*; threshold *minfreq*;

Output   All queries $Q \in \mathcal{L}$ with frequency $\geq$ *minfreq*.

1: Initialize level $d := 1$
2: Initialize the set of candidate queries $\mathcal{Q}_1 := \{? - focus\}$
3: Initialize the set of (in)frequent queries $\mathcal{F} := \emptyset; \mathcal{I} := \emptyset$
4: **while** $\mathcal{Q}_d$ not empty **do**
5:    Find frequency of all queries $Q \in \mathcal{Q}_d$
6:    Move those with frequency below *minfreq* to $\mathcal{I}$
7:    Update $\mathcal{F} := \mathcal{F} \cup \mathcal{Q}_d$
8:    Compute new candidates: $\mathcal{Q}_{d+1} = \mathtt{WARMRgen}(\mathcal{L}; \mathcal{I}; \mathcal{F}; \mathcal{Q}_d)$
9:    Increment d
10: **end**
11: **return** $\mathcal{F}$

Figure 5.1: Algorithm, WARMR

1: Initialize $\mathcal{Q}_{d+1} := \emptyset$.
2: **forall** $Q_j \in Q_d$, and for each refinement $Q'_j \in \mathcal{L}$ of Q_j **do**
3:    Add Q'_j to $\mathcal{Q}_{d+1}$, unless:
      1. Q'_j is more specific than some query $\in \mathcal{I}$, or
      2. Q'_j is equivalent to some query $\in \mathcal{Q}_{d+1} \cup \mathcal{F}$.

4: **end**
5: **return** $\mathcal{Q}_{d+1}$

Figure 5.2: Algorithm, WARMRgen

5.2 Future work

Evaluation of the system on a larger database is needed. Finding a large relational dataset to perform testing on would be the next step in the future work. As well as testing performance it is also needed to evaluate how the natural language descriptions and user interaction work in a more realistic setting.

Regarding the algorithms a important issue is to look for ways of improving the node insertion algorithm. At present, a complete rebuild of the refinement graph is required at least once in order to compute the complete set of association rules, but must in practice be done often in order to lessen database scans. If possible, a algorithm that maintains the complete refinement graph regardless of the order of node insertion would be preferred, otherwise it may be necessary to revise the search strategy. Uniform cost search is flexible, expanding only the most interesting nodes and it also allows a natural way for user interaction by adding cost modifiers. On the other hand a level-wise approach, like breadth first or beam search, would allow the node insertion algorithm to be used in its present state, without any rebuilding of the refinement graph.

More understanding of the complexity of the algorithms is needed. We would like to be able to determine under what circumstances mining is practical. Important factors are the size of the database, and the size and complexity of the

search space. The size of the search space is influenced by the number and types of marked attributes, and the query language. Extending allowed queries to the whole $\mathcal{L}$ language by allowing negated components would for example increase the size of the search space. However, if threshold levels and the properties of the data are such that large parts of the search space can be pruned away at a relatively early stage, mining may be possible even on a very large database and complex search space. It is therefore a hard task to determine when mining is actually practical.

We expect that parallelization will be a important factor in speeding up the mining. At present the system alternates between node insertion in the refinement graph and database querying, but the two tasks might be performed simultaneously for different queries. The task of node insertion itself may be parallelized further. The main focus is however on minimizing database scans, and here it will be important to compute the support of as many $\mathcal{L}$ queries within a single database scan as possible. Currently the system always issues one SQL query for every $\mathcal{L}$ query, something that certainly can be improved.

Making the mining interactive is one of the main goals of the system. One of the main problems here is that too many similar rules are presented, quickly making the user bored. We will look for ways of minimizing the number of dialogs presented, but without making the user lose too much control over the search process. A evaluation on a real database would doubtlessly reveal more issues that needs to be handled in this area.

5.3   Conclusions

This thesis has explored some of the concepts of multirelational association rule mining, where mining is performed over multiple tables in a relational database. Mining is viewed as a search in the space of database queries and $\mathcal{L}$ is used as query language. Queries are ordered based on subsumption in a refinement graph, which starting with the focus query is built top down. A uniform cost strategy is used to select the next node for expansion, new refined queries are then formed by applying a refinement operator. Threshold parameters for support and confidence helps to sort out uninteresting rules. Also, the mining is interactive, inviting the user to guide the search in the right direction. Rules are presented as plain English or Swedish sentences, and the user can guide the search by expressing positive or negative opinions about discovered rules. A prototype system was implemented that uses the STEP system and a standard relational database.

5.4   Acknowledgements

Thanks goes to my advisor Michael Minock for his help during the writing of this thesis.

Bibliography

[1] IBM Quest group. http://www.almaden.ibm.com/software/quest/. Accessed june 2004.

[2] SPASS theorem prover. http://spass.mpi-sb.mpg.de. Accessed june 2004.

[3] Step, live demonstration. http://www.cs.umu.se/ mjm/step/. Accessed june 2004.

[4] R. Agrawal, A. Arning, T. Bollinger, M. Mehta, J. Shafer, and R. Srikant. The quest data mining system. In *In Proc. of the 2nd Int'l Conference on Knowledge Discovery in Databases and Data Mining, Portland, Oregon*, August 1996.

[5] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, pages 487–499. Morgan Kaufmann Publishers Inc., 1994.

[6] Richard O. Duda, Peter E. Hart, and David G. Stork. *Pattern Classification (2nd Edition)*. Wiley-Interscience, 2000.

[7] Sašo Džeroski. Multi-relational data mining: an introduction. *SIGKDD Explor. Newsl.*, 5(1):1–16, 2003.

[8] Herb Edelsten. Mining large databases - a case study. Technical report.

[9] Ramez Elmasri and Shamkant B. Navathe. *Fundamentals of Database Systems*. Addison Wesley, third edition, 2000.

[10] Jiawei Han and Micheline Kamber. *Data Mining: Concepts and Techniques*. Morgan Kaufmann Publishers, 2001.

[11] H. Toivonen L. Dehaspe. Discovery of frequent datalog patterns. *Data mining and Knowledge Discovery*, 3(1):7–36, 1999.

[12] Michael Minock. Data integration under the schema tuple query assumption. Technical report, june 2003.

[13] Michael Minock. Natural language access to relational databases through STEP. Technical report, Department of Computing Science, Umeå University, 2004.

[14] T. Imielinski R. Agrawal and A. Swami. Mining associations between sets of items in massive databases. In *Proceedings of the ACM SIGMOD Int'l Conference on Management of Data, Washington D.C.*, pages 207–216, May 1993.

Appendix A

Example configuration

This is the configuration file used in the example in section 4.1.

```
1   (defconstant +minimum-support-percent+ 0.05)
2   (defconstant +confidence-important-threshold+ 0.70)
3   (defconstant +cost-add-cond+ 100)
4   (defconstant +cost-add-join+ 150)
5   (defconstant +cost-reuse-join+ 1000)
6   (defconstant +cost-support-percent+ -1)
7   (defconstant +cost-important-rule+ -500)
8
9   (defconstant +maximum-cost+ 1000000)
10  (defconstant +set-default+ :pick-one)
11
12  (setf *schema*
13        '((Customer
14            ((cid :numeric :pk) name (gender :string)
15             (age :numeric) (height :numeric) address))
16          (Item ((iid :numeric :pk) (name :string) price))
17          (Store ((sid :numeric :pk) city size))
18          (Inventory (store item))
19          (City ((cityid :numeric :pk) name population latitude longitude))
20          (Contains (transaction item))
21          (Transaction ((tid :numeric :pk) customer (time :numeric) store))
22          (Store ((sid city)))
23          ))
24
25  (setf *fds* '((Customer ( ((cid) (name gender age height address)) ))
26                (Item ( ((iid) (name price)) ))
27                (Store ( ((sid) (city size)) ))
28                (City ( ((cityid) (name population latitude longitude)) ))
29                (Transaction ( ((tid) (customer time store)) ))
30                (Store ( ((sid) (city)) ))
31                ))
32
33  (setf *dm-schema*
34        '((Customer ((gender :set) ))
35          (Item ( (name :set) ))
36          (Transaction ((tid :referenced-by (Contains transaction)
37                             :multiple-append)
38                        (time :set (:split-points 9 12 18))
39                        (customer :foreign-key (Customer cid))
40                        (store :foreign-key (Store sid))
41                        ))
42          (City ( (name :set) ))
43          (Contains ( (item :foreign-key (Item iid)) ))
44          (Store ( (city :foreign-key (City cityid)) ))
45          ))
46
47  (add-patterns "
48  <{x | transaction(x)}: H[e sing indef :'transaction'],
49                         H[e pl def    :'transactions']>
50
```

```
51    <{x | transaction(x), x.time = $c1 }: C[e :'at time $c1']>
52    <{x | transaction(x), x.time <= 9  }: C[e :'taking place in the morning']>
53    <{x | transaction(x), x.time <= 12 }: C[e :'taking place before noon']>
54    <{x | transaction(x), x.time <= 18 }: C[e :'taking place in the after noon']>
55    <{x | transaction(x), x.time > 18  }: C[e :'taking place in the evening']>
56
57    <{x | transaction(x), {customer(c), x.customer = c.cid, _}}:
58      C[e :'byD({c | customer(c), _ }, (e indef sing))']>
59
60    <{x | transaction(x), {store(s), city(c),
61                          x.store = s.sid, c.cityid = s.city, _}}:
62      C[e :'inD({c | city(c), _}, (e indef sing))']>
63
64    <{x | transaction(x),
65          { contains(b), item(i), b.transaction = x.tid, b.item = i.iid, _ }}:
66      C[e :'containingD({i | item(i), _}, (e indef sing))']>
67
68    <{x | transaction(x),
69          { contains(b), item(i), contains(b2), item(i2), contains(b3), item(i3),
70            b.transaction = x.tid, b.item = i.iid, i.name = $c1,
71            b2.transaction = x.tid, b2.item = i2.iid, i2.name = $c2,
72            b3.transaction = x.tid, b3.item = i3.iid, i3.name = $c3,
73            i.iid <> i2.iid, i2.iid <> i3.iid, i3.iid <> i.iid }}:
74      C[e :'containing a $c1, a $c2 and a $c3']>
75
76    <{x | transaction(x),
77          { contains(b), item(i), contains(b2), item(i2),
78            b.transaction = x.tid, b.item = i.iid, i.name = $c1,
79            b2.transaction = x.tid, b2.item = i2.iid, i2.name = $c2,
80            i.iid <> i2.iid }}:
81      C[e :'containing a $c1 and a $c2']>
82
83    <{x | customer(x)}: H[e sing indef:'customer'],
84                        H[e pl def:'customers']>
85
86    <{x | customer(x), x.gender = 'F'}: O[e sing :'female'],
87                                        O[e pl   :'females']>
88
89    <{x | customer(x), x.gender = 'M'}: O[e sing :'male'],
90                                        O[e pl :'males']>
91
92    <{x | city(x)}:  H[e sing:'city'],
93                     H[e pl:'cities']>
94
95    <{x | city(x), x.name = $c1 }: C[e :'named $c1']>
96
97    <{x | store(x)}: H[e sing :'store'],
98                     H[e pl   :'stores']>
99
100   <{x | item(x)}: H[e sing  :'item'],
101                   H[e pl    :'items']>
102
103   <{x | item(x), x.name = $c1}: O[e :'$c1']>")
```