A Logical Database Mining Query Language ^{*}

Luc De Raedt

Institut für Informatik, Albert-Ludwig-University
Universitätsgelände Flugplatz
D-79085 Freiburg, Germany
`deraedt@informatik.uni-freiburg.de`

Abstract. A novel query language for database mining, called RDM, is presented. RDM provides primitives for discovering frequent patterns and association rules, for manipulating example sets, for performing predictive and descriptive induction and for reasoning about the generality of hypotheses. RDM is designed for querying deductive databases and employs principles from inductive logic programming. Therefore RDM allows to query patterns that involve multiple relations as well as background knowledge. The embedding of RDM within a programming language such as PROLOG puts database mining on similar grounds as constraint programming. An operational semantics for RDM is outlined and an efficient algorithm for solving RDM queries is presented. This solver integrates Mitchell's versionspace approach with the well-known APRIORI algorithm by Agrawal *et al.*

Keywords : database mining query language, inductive logic programming, relational learning, inductive database, APRIORI, versionspaces.

1 Introduction

Imielinski and Mannila [11] present a database perspective on knowledge discovery. From this perspective, knowledge discovery is regarded as a querying process to a database mining system. Querying for knowledge discovery requires an extended query language (w.r.t. database languages), which supports primitives for the manipulation, mining and discovery of rules, as well as data. The integration of such rule querying facilities provides new challenges for database technology.

Mannila and Toivonen [14] formulate the general pattern discovery task as follows. Given a database r , a language $\mathcal{L}$ for expressing patterns, and a selection predicate q , find the theory of r with respect to $\mathcal{L}$ and q , i.e. $Th(\mathcal{L}, r, q) = \{\phi \in \mathcal{L} \mid q(r, \phi) \text{ is true}\}$. This formulation of pattern discovery is generic in that it makes abstraction of several specific tasks including the discovery of association rules, frequent patterns, inclusion dependencies, functional dependencies,

^{*} An early extended abstract of this paper appeared as part of [7]. The present paper expands significantly on this work. The work was also presented at the JICSLP Workshop in Manchester 1998, and the German Machine Learning Workshop, Magdeburg, 1999.

frequent episodes, ... Also, efficient algorithms for solving these tasks are known (cf. [14]).

So far, the type of selection predicate that has been considered is simple and typically relies on the frequency of patterns. However, knowledge discovery is often regarded as a cyclic and iterative process (cf.[10]), where one will consider a number of different selection predicates, patterns, example sets and versions of the database. Therefore, there is a need for considering and combining different theories $Th(\mathcal{L}_i, q_i, r_i)$. This practical necessity is also the main motivation underlying the work on inductive databases [15] and database mining query languages [11, 17]. Indeed, a number of extensions to database languages such as SQL have been introduced with the aim of supporting the generation and manipulation of the theories $Th(\mathcal{L}_i, q_i, r_i)$.

This paper presents a novel query language, called RDM – Relational Database Mining, as well as its execution mechanism. RDM supports a larger variety of selection predicates than previous approaches. In addition to providing an operator to manipulate patterns, it also allows to manipulate sets of examples, and provides selection predicates that impose a minimum or maximum frequency threshold. This allows RDM to address descriptive as well as predictive induction. In descriptive induction one is interested in frequent patterns, whereas in predictive induction, one aims at discovering patterns that are frequent on the positive examples but infrequent on the negative ones. Further primitives in RDM are based on a notion of generality (among patterns) and on coverage (w.r.t. specific examples). Queries in RDM will consist of a number of primitives. Each primitive will impose a specific constraint on the patterns to be discovered and will result in a theory $Th(\mathcal{L}_i, q_i, r_i)$. RDM will then efficiently compute the intersection of all these theories. RDM's execution mechanism integrates the version-space approach by Tom Mitchell [19] with the levelwise algorithm in APRIORI [1]. Directly computing the intersection of the theories contrasts with the approach taken in the MINE RULE operator by Meo *et al.* [17] and the approach of Boulicaut *et al.* [2] in that these latter approaches typically generate one theory $Th(\mathcal{L}_i, q_i, r_i)$ and then repeatedly modify it.

As RDM aims at providing a general query language for database mining, it employs patterns in the form of DATALOG queries [5, 6, 4] as in the field of relational learning or inductive logic programming [20]. This allows RDM to mine multiple relations in a database at once and to express patterns beyond the scope of propositional approaches (as shown by [6]). Embedding RDM within a programming language such as PROLOG [3] puts database mining on the same methodological grounds as constraint programming. Indeed, each of the different selection predicates or primitives in a query or program imposes certain constraints on the patterns. As in constraint programming, we have to specify the semantics of these primitives as well as develop efficient solvers for queries and programs. An operational semantics for RDM will be specified and an efficient solver will be presented.

The paper is organised as follows : in Section 2, we define the basic RDM primitives based on logic, in Section 3, we show RDM at work through a number

of database mining queries, in Section 4, we present a solver for simple queries, in Section 5, we discuss how to extend the language and solver, and finally, in Section 6, we conclude and touch upon related work.

2 Terminology

2.1 DATALOG

As [5, 4] we will use DATALOG (e.g. [6]) to represent both data and patterns. In DATALOG, *terms* are either constants (written in lowercase) or variables (starting with a capital). An *atom* is then of the form $p(t_1, \dots, t_n)$ where p is a predicate (or relation) symbol of arity n and the t_i are different terms. A *definite clause* is a universally quantified formula of the form $H : -A_1, \dots, A_n$ where H and the A_i are different atoms. The formula can be read as H is A_1 and ... and A_n . If $n = 0$ then the clause is a *fact* (this corresponds to a tuple in a relational database). A (deductive) database is then a set of definite clauses. A substitution $\theta = \{V_1 = t_1, \dots, V_k = t_k\}$ is an assignment of terms t_i to variables V_i . A substitution θ can be applied to a clause or query c resulting in $c\theta$ where all variables V_i have been substituted by the corresponding terms t_i . DATALOG queries are of the form $-B_1, \dots, B_m$ where the B_i are atoms. Using resolution theorem provers, one can answer a query q by generating all answersubstitutions θ for which $q\theta$ follows from the database. It will be convenient to represent the set of all answersubstitutions for a query q and a database r as $anserset(q, r)$. Sometimes we will only be interested in the projection $\theta_W = \{V_i = t_i \mid V_i \in W\}$ of a substitution $\theta = \{V_1 = t_1, \dots, V_k = t_k\}$ on a set of variables $W \subset \{V_1, \dots, V_k\}$ or in the projection $anserset_W(q, r) = \{\theta_W \mid \theta \in anserset(q, r)\}$ of an $anserset(q, r)$.

2.2 RDM primitives

Definition 1. A pattern $\phi(Q, K)$ consists of a DATALOG query Q and a set of variables K , called the key, occurring in Q .

The idea is that the key represents the identifier of the entities being considered. The key thus corresponds to the notion of example and will reflect what is being counted. This is similar to the GROUP BY phrase in the MINE RULE operator of [17]. One pattern about transactions involving beer and sausage is e.g. $\phi(\text{beer}(X), \text{sausage}(X)), \{X\}$. Notice that we omit the ':' of the DATALOG query for readability reasons.

Data mining with RDM is then the search for patterns that satisfy certain requirements. Three primitives are foreseen to specify requirements in RDM. The first concerns the frequency of patterns, which is defined as follows (following [5, 4]):

Definition 2. $frequency(\phi(Q, K), r) = |anserset_K(Q, r)|$ where $\phi(Q, K)$ is a pattern and r a database.

The frequency is defined as the number of substitutions for the key for which the query succeeds. This primitive allows to impose constraints on the frequency of the patterns, e.g. $\text{frequency}(\phi(Q, K), r) > \text{min}$ or $\text{frequency}(\phi(Q, K), r) < \text{max}$, where Q and K are variables (as we shall illustrate below).

The second primitive concerns the syntactic form of queries in patterns. The primitive $\preceq$ specifies the relation of the pattern to specified queries. This corresponds to encoding the *is more general than* relation.

Definition 3. $Q_1 \preceq Q_2$ if and only Q_1 is more general than Q_2 .

We will also be using $\prec$ which has the classical semantics. In the case of using DATALOG patterns, $\preceq$ can best be implemented by Plotkin's θ -subsumption [21], which states that Q_1 is more general than Q_2 if and only if there exists a substitution θ such that $Q_1\theta \subseteq Q_2$. E.g. the query $(\text{beer}(X), \text{diapers}(X))$ θ -subsumes the pattern $(\text{beer}(Y), \text{diapers}(Y), \text{sausage}(Y))$ with $\theta = \{X = Y\}$. This primitive allows to express which patterns one is interested in.

The third primitive concerns the notion of an example. An example in RDM corresponds to a substitution θ that substitutes all variables in the key K with constants in the pattern $\phi(Q, K)$. Using this primitive, it is possible to specify that a pattern should cover a specific example :

Definition 4. An example θ is covered by a pattern $\phi(Q, K)$ and a database r , notation $r \models Q\theta$, if and only if the query $Q\theta$ succeeds in the database r .

We will also be using the notation $r \not\models Q\theta$ to reflect the fact that the example θ is not covered. E.g. given a database r containing the facts

$\text{diapers}(\text{t133}), \text{beer}(\text{t133}), \text{sausage}(\text{t133})$

we have that the pattern $\phi((\text{beer}(T), \text{sausage}(T)), \{T\})$ covers the example $\{T = \text{t133}\}$ because the query $:- \text{beer}(\text{t133}), \text{sausage}(\text{t133})$ succeeds in the database.

Finally, RDM provides a number of extensions of the above primitives in order to provide more flexibility. The first extension allows RDM to work with sets of examples E and to extend the frequency and coverage primitives to work with sets of examples. This results in the following definitions :

Definition 5. $\text{frequency}(\phi(Q, K), r, E) = |\text{answerset}_K(Q, r) \cap E|$ where $\phi(Q, K)$ is a pattern, r a database, and E a set of examples.

Using this extension, we can specify RDM queries to discover patterns that are frequent on a set of positive examples but infrequent on a set of negative examples.

Definition 6. A set of examples E is covered by a pattern $\phi(Q, K)$ and a database r , notation $r \models E$, if and only if $\text{frequency}(\phi(Q, K), r, E) = |E|$.

Similarly, we will say that a set of examples E is not covered if the frequency is 0.

The second extension allows to employ negation in various forms, e.g. to specify that a query is not more specific than a given other query e.g. `not Q  $\preceq$  (sausage(X), beer(X))`.

The final extension concerns embedding the primitives in a high-level declarative programming language. Doing this puts data mining on the same methodological grounds as constraint programming. For the purposes of RDM, it seems most appropriate to embed its primitives within the PROLOG programming language because RDM assumes DATALOG as the underlying database model.

Further advantages of embedding RDM within PROLOG are that this naturally allows us to formulate complex RDM queries. E.g. the query

```
?- Q  $\preceq$  (beer(X), diapers(X), mustard(X), sausage(X)),
frequency( $\phi(Q, \{X\}), r) > 10$  .
```

would return all patterns involving beer, diapers, mustard and sausage, whose frequency on database *r* exceeds 10 where the key is given by the $\{X\}$. This shows that the embedding of RDM within PROLOG allows to use variables (written in capitals) that range over the query-part of a pattern (such as *Q*). Integrating PROLOG's and RDM's execution mechanism allows us to get answers for these variables. One such answer might be *Q* = (beer(*X*), diapers(*X*), mustard(*X*)) stating that for this particular instantiation of the variable *Q* all the primitives hold.

The embedding also allows us to define complex predicates in terms of the primitives. E.g. association rules can now be defined as follows in PROLOG.

```
associationrule(  $\phi$ (Condition,K),  $\phi$ (Conclusion,K), R, Acc, Freq) :-
    Condition  $\prec$  Conclusion,
    Freq is frequency( $\phi$ (Conclusion,K),R),
    Acc is frequency( $\phi$ (Condition,Key),R) / Freq.
```

The predicate `associationrule` will produce association rules of the database *R* of the form *Conclusion* if *Condition* and will compute their accuracy and frequency. E.g. the query

```
?- C  $\preceq$  (beer(X), diapers(X), mustard(X), sausage(X)) ,
associationrule(  $\phi(Q, \{X\}), \phi(C, \{X\}), r, Acc, Freq) ,
Freq > 10, Acc > 0.5$  .
```

would return all association rules with a frequency on *r* of more than 10 and an accuracy of more than 50 per cent. E.g. the pattern `beer(X),diapers(X) if beer(X)`, which for this case could be simplified by omitting `beer(X)` in the conclusion part.

This type of query is similar to the queries one can express within Imielinski and Virmani's M-SQL [12]. Using this same type of queries it is possible to

emulate the essence of Dehaspe and De Raedt’s WARMR [4] and De Raedt and Dehaspe’s Claudien [8].

Observe also that it is easy to enhance RDM by providing further primitives such as e.g. *sample*(E, P) with E an example set and P a probability, which would generate a sample of the examples (drawn according to probability P). This and other extensions are however quite straightforward, and merely provide further syntactic sugar. Therefore we will not discuss such extensions any further here.

So far, we have defined a semantics for RDM and embedded it within PROLOG. It is straightforward to implement the *specification* of the primitives within PROLOG (cf. [7] for such a high level implementation). However, a direct implementation will not work efficiently, and therefore a more advanced implementation will be sketched in Section 4. It should also be mentioned that not all database mining queries are safe, but that it is possible to specify safe queries using modedecoration (as typically done in Prolog manuals). For safe execution, it is e.g. required that for all patterns $\phi(Q, K)$ searched for the data mining query in RDM - PROLOG must include $Q \preceq (lit_1, \dots, lit_n)$ before any further reference to Q is made. This is necessary in order to specify the range of DATALOG queries searched for. Also, without bounding Q in this manner, many of the predicates defined above would not operate correctly. We will not go into further (implementation) details of the semantics at this point.

3 Data mining with RDM

We now basically have all primitives needed to generate database mining queries. In this section, we demonstrate the expressive power of RDM by examples.

First, as already sketched above, one can naturally discover all frequent patterns bounded from below by some DATALOG query. The RDM query shown earlier demonstrated this for the case of item-sets. Due to use of first order language for expressing patterns, it is also possible to simulate the behaviour of the WARMR system [4, 5]. E.g. the query

```
?- (atomel(C,A1,c), atomel(C,A2,h), bond(C,A1,A2))  $\preceq$  Q,
Q  $\preceq$  (atomel(C,A1,c), atomel(C,A2,h), bond(C,A1,A2),
atomtype(A1,t1), ..., atomtype(A1,tn), atomtype(A2,t1),
atomtype(A2,t2)), frequency( $\phi(Q, \{C\}), r$ ) > 10 .
```

could be used to find frequent patterns in molecules that involve bonds in which a carbon and a hydrogen atom occur and further properties of the atoms (such as types) are specified.

Further details of the use of such first order patterns and the relation to existing frameworks for frequent pattern discovery are provided in a recent paper by Dehaspe and Toivonen [6]. They show that nearly any type of frequent pattern considered in the data mining literature can be expressed in the presented first order framework. Hence, it is easy to simulate these in RDM.

A second type of query is to generate association rules as sketched in section 2.

The third type of queries is aimed at predictive data mining instead of descriptive datamining. It therefore employs both positive and negative examples. E.g.

```
?- Q ≤ (Item-1(X), ... , Item-n(X)),
N is frequency(φ(Q,{X}),r,NegSet ),
N < 10,
P is frequency(φ(Q,{X}),r,PosSet ),
Acc is P / (P+N), Acc > 0.9.
```

This query would generate condition parts of rules that are at least 90 per cent correct, and that cover at most 10 negative examples in the *NegSet* (which we did not represent explicitly here). Once such rules are found, they can (partly) define a predicate with the key as argument(s). E.g. all DATALOG queries Q generated may be turned into clauses *positive(X) :- Q* that could be asserted in the database for further use. This is akin to what many classification systems search for. Also, rather than involving a minimum frequency, this type of rule querying imposes a maximum frequency (on the negative examples).

Fourthly, sometimes predictive data mining approaches employ seed examples. These seed examples are used to constrain the rules searched for. E.g. when positive examples are used as seeds, the induced rule should cover the positive. This can easily be expressed with the covers primitive. In typical database mining situations, these different constructs will be combined in order to find relevant (sets of) patterns. E.g.

```
?- Q ≤ (item-1(T),... , item-n(T)),
r |= Q{T = t190}, positive seed
r ≠ Q{T = t133}, negative seed
P is frequency(φ(Q,{T}),r,PosSet ),
N is frequency(φ(Q,{T}),r, NegSet ),
N < 10, Acc is P / (P + N), Acc > 0.9 .
```

So far, we have only allowed queries that involve a single pattern. However, it may be useful to search for multiple patterns simultaneously. E.g.

```
?- Q ≤ (item-1(T),... , item-n(T)),
P is frequency(φ(Q,{T}),r,PosSet ),
N is frequency(φ(Q,{T}),r, NegSet ),
Acc is P / (P + N), Acc > 0.9,
coveredexamples(φ(Q,{T}), r, PosSet, Covered),
delete(Covered,PosSet,Uncovered),
Q2 ≤ (item-1(T),... , item-n(T)),
P2 is frequency(φ(Q2,{T}),r,Uncovered),
```

P2 > 100 .

This query would first find a DATALOG query Q that is 90 per cent correct on the example set and then try to find a pattern $Q2$ whose frequency exceeds 100 on the positive examples not covered by Q .

Of course, the most expressive way of using the query language is to embed the primitives in simple Prolog programs. Using this facility it is easy to emulate inductive logic programming systems (such as e.g. Quinlan's FOIL [22]) in a one page meta-program.

4 Answering database mining queries

Obviously, plainly using the primitives as they have been defined will be disastrous from the efficiency point of view. This is because all patterns in the language would simply be enumerated using and tested afterwards. The question then is : how can we efficiently produce answers to the queries formulated above ?

Let us here concentrate on simple database mining queries. Simple queries search for a single pattern only.

The key insight that leads to efficient algorithms for answering database mining queries is that 1) the space of patterns is partially ordered by the $\preceq$ relation, and 2) that all primitives provided in RDM behave monotonically with regard to $\preceq$. Therefore each constraint on a pattern allows to prune the search. Furthermore, a set of constraints defines a version space of acceptable patterns. This version space is completely determined by its S and G sets (as defined by Mitchell 1982).

We have the following kind of basic constraints or information elements (the extensions can be dealt with analogously) :

- $Query \preceq q$ (resp. $q \preceq Query$) which states that the target Query is more general (resp. more specific) than a given query q ; similar elements can be defined wrt. $\prec$.
- $frequency(\phi(Q, K), r) > t$ (resp. $\geq$) : which specifies a constraint on the frequency of the pattern (on the specified Data)
- $r \models Q\theta$ which expresses that the query should cover the example θ
- also, each of the above primitives can be negated

Information elements of the first type (i.e. concerning $\preceq$) are directly handled by Mellish's description identification algorithm [18], which extends Mitchell's version space algorithm. Fact is that the statements of the form $Query \preceq q$ (resp. $notQuery \preceq q$) can be interpreted as $Query$ covers the 'positive example' q (resp. 'negative example'). In this view the query q is regarded an example (in a different example space than that of RDM). Mellish's description identification algorithm extends Mitchell's well-known version space algorithm by handling statements of the form $q \preceq Query$ and its negation. These additional information elements are dual in the original ones, and so is its algorithmic treatment.

Thus it is straightforward to process the elements concerning $\preceq$ with the version space algorithm. First define :

$$\begin{aligned} glb(a, b) &= \max \{d \in \mathcal{L} \mid a \preceq d \text{ and } b \preceq d\} \\ lub(a, b) &= \min \{d \in \mathcal{L} \mid d \preceq a \text{ and } d \preceq b\} \\ mgs(a, b) &= \max \{d \in \mathcal{L} \mid a \preceq d \text{ and } \text{not}(d \preceq b)\} \\ msg(a, b) &= \min \{d \in \mathcal{L} \mid d \preceq a \text{ and } \text{not}(b \preceq d)\} \end{aligned}$$

The *max* and *min* functions will return the set of maximal (resp. minimal) elements within the partial order defined by $\preceq$. We can then use the following algorithm :

```

S := {⊤}; G := {⊥};
for all constraints i do
  case i of q ⪯ Query :
    S := {s ∈ S | q ⪯ s}
    G := max {glb(q, g) | g ∈ G and ∃s ∈ S : glb(q, g) ⪯ s}
  case i of Query ⪯ q :
    G := {g ∈ G | g ⪯ q}
    S := min {lub(q, s) | s ∈ S and ∃g ∈ G : g ⪯ lub(q, s)}
  case i of not Query ⪯ q :
    S := {s ∈ S | not(s ⪯ q)}
    G := max {m | ∃g ∈ G : m ∈ mgs(g, q) and ∃s ∈ S : m ⪯ s}
  case i of not q ⪯ Query :
    G := {g ∈ G | not(q ⪯ g)}
    S := min {m | ∃s ∈ S : m ∈ msg(s, q) and ∃g ∈ G : g ⪯ m}
  case i of r ⊨ Queryθ :
    G := {g ∈ G | r ⊨ gθ}
    S := min {s' | r ⊨ s'θ and ∃s ∈ S : s' ⪯ s and ∃g ∈ G : g ⪯ s}
  case i of r ⊭ Queryθ :
    S := {s ∈ S | r ⊭ sθ}
    G := max {g' | r ⊭ g'θ and ∃g ∈ G : g ⪯ g' and ∃s ∈ S : g' ⪯ s}
  case i of frequency(ϕ(Query, Key), r) > f :
    G := {g ∈ G | frequency(ϕ(g, Key), r) > f}
    S := min {s' | frequency(ϕ(s', Key), r) > f and
      ∃s ∈ S : s' ⪯ s and ∃g ∈ G : g ⪯ s'}
  case i of frequency(ϕ(Query, Key), r) < f :
    S := {s ∈ S | frequency(ϕ(s, Key), r) < f}
    G := max {g' | frequency(ϕ(g', Key), r) < f and
      ∃g ∈ G : g ⪯ g' and ∃s ∈ S : g' ⪯ s}

```

When analysing the above algorithm, the glb, lub, mgs and msg operators can be straightforwardly and efficiently implemented. However, the key implementation issue is how to compute the S set for the case of $\text{frequency}(\phi(Q, K), r) > f$. Similar questions arise for the dual constraint and the use of covers. In the data

mining literature, a lot of attention has been devoted to this, resulting in algorithms such as APRIORI that will only evaluate those patterns on the database that could possibly be frequent. It turns out that we can find the S set for this case in two different manners, which correspond to traversing the space in two dual ways. When working general to specific, one can integrate the basic ideas of the APRIORI approach into versionspaces.

We now sketch how this could be realized by looking at the case that a minimum frequency threshold is imposed on the patterns. The other cases however can be handled in a similar way.

Before specifying the two dual versions, we need a notion of a refinement and a generalization operator.

Definition 7. A refinement operator $\rho_s(Q) = \max\{Q' \in \mathcal{L} \mid Q \prec Q'\}$, where Q is a query. A generalization operator $\rho_g(Q) = \min\{Q' \in \mathcal{L} \mid Q' \prec Q\}$.

Let us then first sketch the top-down algorithm to update S and G when imposing a minimal frequency threshold.

```

NewG := {g ∈ G | frequency(g, r) > f}
I := G − NewG; the set of infrequent patterns
F := NewG; the set of frequent patterns
L0 := NewG; i := 0; Li contains frequent queries for further refinement
while Li ≠ ∅ do
  Li+1 := {p | ∃q ∈ Li : p ∈ ρs(q) and ∃s ∈ S : p ⪯ s}
  NewIs := {p | p ∈ Li+1 and ρg(p) ∩ I ≠ ∅}
  I := NewIs ∪ I ; Li+1 := Li+1 − NewIs;
  for all q ∈ Li+1 do
    if frequency(q, r) > f
      then add q to F else add q to I and delete q from Li+1
  i := i + 1 ;
endwhile
S := min(F)1 ; G := NewG;

```

Let us first consider the case where $S = \{\perp\}$ and $G = \{\top\}$ and where we work with itemsets. In this case the refinement operator will merely add a single item to a query and the generalization operator will delete a single item from the itemset (in all possible manners). In this case, the above algorithm will behave roughly as the APRIORI algorithm. The L_i will then contain only itemsets of size i and the algorithm will keep track of the set of frequent item sets F as well as the infrequent ones ². The algorithm will then repeatedly compute a set of candidate refinements L_{i+1} , delete those item sets that cannot be frequent by looking at the frequency of its generalizations, and evaluate the resulting possibly frequent itemsets on the database. This process then continues until L_i becomes empty.

² Explicitly keeping track of I is not done by APRIORI, but is similar to some of its extensions, cf. [5]

The basic modifications to run it in our context are concerned with the fact that we need not consider any element that is not in the already computed version space (i.e. any element not between an element of the G and the S set). Secondly, we have to compute the updated S set, which should contain all frequent elements whose refinements are all infrequent.

Finding the updated G and S sets can also be realized in the dual manner. In this case, one will initialize L_0 with the elements of S that are infrequent and proceed otherwise completely dual. The resulting algorithm is shown below :

```

 $G := \{g \in G \mid \text{frequency}(g, r) > f\}$ 
 $FS := \{s \in S \mid \text{frequency}(s, r) > f\}$ 
 $I := S - FS$  the set of infrequent patterns
 $F := FS$  ;3 the set of frequent patterns
 $L_0 := I$ ;  $i := 0$ ;  $L_i$  contains infrequent queries for further generalization
while  $L_i \neq \emptyset$  do
     $L_{i+1} := \{p \mid \exists q \in L_i : p \in \rho_g(q) \text{ and } \exists g \in \text{New}G : g \preceq p\}$ 
     $\text{New}Fs := \{p \mid p \in L_{i+1} \text{ and } \rho_s(p) \cap F \neq \emptyset\}$ 
     $F := \text{New}Fs \cup F$ ;  $L_{i+1} := L_{i+1} - \text{New}Fs$ ;
    for all  $q \in L_{i+1}$  do
        if  $\text{frequency}(q, r) > f$ 
        then add  $q$  to  $F$  and remove from  $L_{i+1}$  else add  $q$  to  $I$ 
     $i := i + 1$ 
endwhile
 $S := \min(F)$  ;

```

Simple queries query for a single pattern and the above algorithm shows how such queries can be answered. However, more complex queries would involve multiple patterns. The algorithm for computing answers to simple queries can be upgraded towards this type of queries. The easiest way to handle such queries is by processing the literals in the query from left to right (as in Prolog) and working with multiple version spaces. Algorithms for realizing this more efficiently are however the subject of future work.

5 Extensions to the basic engine and language

5.1 Optimization primitives

Two primitives that seem especially useful are *min* and *max*. Indeed, one could imagine being interested in those patterns that satisfy a number of constraints and in addition have maximum frequency on a certain dataset or are minimally general. Let us therefore define :

Definition 8. *An optimization literal $\text{max}(p(\phi(Q, K)), \text{rel})$ (resp. min) takes as argument a predicate p and a relation rel to be optimized. The predicate p imposes constraints on the pattern $\phi(Q, K)$ and rel specifies the criterion that should be maximized (resp. minimized). The criterion is either generality or $\text{frequency}(\phi(Q, K), r, E)$. The optimization literal then succeeds for those patterns satisfying the constraints imposed by p and being optimal w.r.t. rel .*

As an example consider the following query :

```
patt( $\phi(Q, K)$ ) :-
     $Q \preceq (\text{beer}(X), \text{diapers}(X), \text{mustard}(X), \text{sausage}(X)),$ 
     $\text{frequency}(\phi(Q, K), r) > 100.$ 

?-min(pat( $\phi(Q, K)$ ),  $\text{frequency}(\phi(Q, K), r)$ ).
```

This query would generate the pattern with minimum frequency (but higher than 100) over the items beer, diapers, mustard and sausage. For this particular query, one might want also to use two different databases (one within p and one within min).

Within the sketched solver for simple queries, it is relatively easy to accommodate this type of constructs. Indeed, because the primitives w.r.t. frequency and generality behave monotonically w.r.t. $\preceq$, one only needs to consider the maximum (resp. minimum) elements in the solutions to p . So, to answer a query containing an optimization literal, one first computes the versionspace (i.e. the S and G sets) for p . If the criterion to be optimized is generality, then the solutions are given directly by either S or G (depending on whether one wishes to minimize or to maximize). If on the other hand one wishes to optimize w.r.t. frequency then one needs to compute (or remember) the frequency of all elements in either G or S. The answers to the queries are then those patterns within either G or S that are optimal.

5.2 Heuristic solvers

So far, we have described *complete* solvers, which will find all solutions within the specified constraints. However, completeness often comes at a (computational) cost. Therefore, complete solvers may not always be desirable. There are at least two situations already encountered where this might be the case.

First, the provided primitives for optimisation were so far quite simple. Also, the criterion one may want to optimize is not necessarily frequency. Indeed, one is often more interested in accuracy, or entropy, etc. To optimize w.r.t. e.g. accuracy one cannot employ the above sketched method because accuracy involves combining maximum frequency on positives and minimum frequency on negatives. Hence, optimal patterns might lie in the middle of the versionspace.

Second, there is a discrepancy between answering simple queries and answering queries over multiple inter-related patterns. There is good hope and evidence that the former solver is reasonably efficient, but it is also clear that the latter one is much less efficient, because it merely enumerates all possibilities.

Therefore, we need to extend the current solver with *heuristic* methods. This situation is again akin to what happens in constraint logic programming (cf. [16]). The effect of heuristic methods would be that queries would get a heuristic answer, that it might be that some solutions are missed, and also - in the case of optimizations - that suboptimal solutions might be generated. Of course, in such cases the user should be aware of this. When allowing for heuristic methods, it

would be possible to extend the database mining engine with various well-known database mining algorithms, e.g. with a beam-search procedure to greedily find the most interesting clauses in predictive modelling.

5.3 The knowledge discovery cycle

Knowledge discovery in databases typically proceeds in an iterative manner. Data are selected, possibly cleaned, formatted, input in a data mining engine, and results are analysed and interpreted. As first results often can be improved, one would typically re-iterate this process until some kind of a local optimum has been reached (cf. [10]).

Because knowledge discovery is an iterative process data mining tools should support this process. One consequence of the iterative nature of knowledge discovery in our context is that many of the queries formulated to the database mining engine will be related. Indeed, one can imagine that various queries are similar except possibly for some parameters such as thresholds, data sets, pattern syntax, etc. The relationships among consecutive queries posed to the data mining engine should provide ample opportunities for optimization. The situation is - to some extent - akin to the way that views are dealt with in databases (cf. e.g. [9]). Views in databases are similar to patterns in data mining in that both constructs are virtual data structures, i.e. they do not physically exist in the database. Both forms of data can be queries and it is the task of the engines to efficiently answer questions concerning these virtual constructs.

Answering queries involving views can be realized in essentially in two different ways. First, one can *materialize* views, which means that one generates the tuples in the view relation explicitly, and then processes queries as if a normal relation was queried. Second, one can perform *query modification*, which means that any query to a view is "unfolded" into a query over the base relations. The advantage of materialization is that new queries are answered much faster whereas the disadvantage is that one needs to recompute or update the view whenever something changes in the underlying base relations. At a more general level, this corresponds to the typical computation versus storage trade-off in computer science.

These two techniques also apply to querying patterns in data mining. Indeed, if consecutive queries are inter-related, it would be useful to store the results (and possibly also the intermediate results) of one query in order to speed up the computation of the latter ones. This corresponds to materializing the patterns (together with accompanying information). Doing this would result in effective but fairly complicated solvers.

6 Related Work

RDM is related to other proposals for database mining query languages such as e.g. [17, 11, 2]. However, it differs from these proposals in a number of respects. First, due to the use of deductive databases as the underlying database model,

RDM allows to perform pattern discovery over multiple relations as in the field of inductive logic programming (cf. [5]). Secondly, a number of new and useful primitives are foreseen. Using RDM one is not restricted to finding frequent patterns, but one may also look for infrequent ones with regards to certain sets of (negative) examples. One can also require that certain examples are (resp. are not) covered by the patterns to be induced. Thirdly and most importantly, RDM allows to combine different primitives when searching for patterns. Finally, its embedding within PROLOG puts database mining on the same methodological grounds as constraint programming.

As another contribution, we have outlined an efficient algorithm for answering complex database mining queries. This algorithm integrates the principles of APRIORI with those of versionspaces and thus provide evidence that RDM can be executed efficiently. It also provides a generalized theoretical framework for datamining. The resulting framework is somewhat related to the levelwise techniques sketched by [14] who also link the levelwise algorithm to the S set of Mitchell's versionspace approach but do not further exploit the versionspace model as we do here. A prototype implementation of RDM along these lines already exists and might be put in the public domain soon.

There is plenty of further work that remains. First, a better implementation of the solver needs to be implemented. Secondly, the theoretical issues involved in answering complex database mining queries need to be addressed. Thirdly, the relationship with constraint logic programming needs to be examined further.

Finally, the author hopes that this work also provides a new perspective on inductive logic programming. The reason for this hope is that it puts inductive logic programming on the same methodological grounds as constraint logic programming. So far, inductive logic programming has mainly used the representations offered by computational logic but has basically employed the machine learning methodology. Exploiting the view of inductive logic programming as constraint processing one might bridge this gap.

Acknowledgements

This work was initially supported by the Fund for Scientific Research, Flanders, and by the ESPRIT IV project no 20237 on Inductive Logic Programming II. The author is grateful to Jean Francois Boulicaut, Maurice Bruynooghe and Luc Dehaspe for interesting discussions on this, to Hendrik Blockeel for comments on this paper, and especially to Imielinski and Mannila for inspiring this work through their CACM paper.

References

1. R. Agrawal, T. Imielinski, A. Swami. Mining association rules between sets of items in large databases. In *Proceedings of ACM SIGMOD Conference on Management of Data*, 1993.

2. J-F. Boulicaut, M. Klemettinen, H. Mannila. Querying inductive databases: a case study on the MINE RULE operator. in: *Proceedings of the Second European Symposium on Principles of Data Mining and Knowledge Discovery*, Lecture Notes in Artificial Intelligence, Vol. 1510, , Springer-Verlag, 1998.
in Proceedings of PKDD 98, Lecture Notes in AI, 1998.
3. I. Bratko. *Prolog Programming for Artificial Intelligence*. Addison-Wesley, 1990. 2nd Edition.
4. L. Dehaspe and L. De Raedt, WARMR : Wanted Association Rules over Multiple Relations, In *Proceedings of the International Workshop on Inductive Logic Programming*, Lecture Notes in Artificial Intelligence, Vol. 1297, Springer Verlag, 1997.
5. L. Dehaspe, H. Toivonen and R.D. King. Finding frequent substructures in chemical compounds, in *Proceedings of KDD-98*, 1998.
6. L. Dehaspe, H. Toivonen. Discovery of Frequent Datalog Patterns, in *Data Mining and Knowledge Discovery* , Vol. 3, 1999.
7. L. De Raedt, An inductive logic programming query language for database mining (Extended Abstract), in Calmet, J. and Plaza, J. (Eds.) *Proceedings of Artificial Intelligence and Symbolic Computation*, Lecture Notes in Artificial Intelligence, Vol. 1476, Springer Verlag, 1998.
8. L. De Raedt and L. Dehaspe. Clausal discovery. *Machine Learning*, 26:99–146, 1997.
9. Elmasri, R. and Navathe, S. Fundamentals of database systems. Benjamin Cummings, 2nd Edition, 1994.
10. Fayyad, U., Piatetsky-Shapiro, G. and Smyth, P. Advances in Knowledge Discovery, The MIT Press, 1996.
11. T. Imielinski and H. Mannila. A database perspective on knowledge discovery. *Communications of the ACM*, 39(11):58–64, 1996.
12. T. Imielinski, A. Virmani, and A. Abdulghani. A discovery board application programming interface and query language for database mining. In *Proceedings of KDD 96*. AAAI Press, 1996.
13. N. Lavrač and S. Džeroski. *Inductive Logic Programming: Techniques and Applications*. Ellis Horwood, 1994.
14. H. Mannila and H. Toivonen, Levelwise search and borders of theories in knowledge discovery, *Data Mining and Knowledge Discovery*, Vol. 1, 1997.
15. H. Mannila. Inductive databases. in *Proceedings of the International Logic Programming Symposium*, MIT Press, 1997.
16. Marriott, K. and Stuckey, P. J. Programming with constraints : an introduction. The MIT Press. 1998.
17. R. Meo, G. Psaila and S. Ceri, An extension to SQL for mining association rules. *Data Mining and Knowledge Discovery*, Vol. 2, 1998.
18. C. Mellish. The description identification algorithm. *Artificial Intelligence*, Vol. 52, 1990.
19. T. Mitchell. Generalization as Search, *Artificial Intelligence*, Vol. 18, 1980.
20. S. Muggleton and L. De Raedt. Inductive logic programming : Theory and methods. *Journal of Logic Programming*, 19,20:629–679, 1994.
21. G. Plotkin, A note on inductive generalization, *Machine Intelligence*, Vol. 3, 1970.
22. J.R. Quinlan. Learning logical definitions from relations. *Machine Learning*, 5:239–266, 1990.
23. W. Shen, K. Ong, B. Mitbender, and C. Zaniolo. Metaqueries for data mining. In U. Fayyad, G. Piatetsky-Shapiro, P. Smyth, and R. Uthurusamy, editors, *Advances in Knowledge Discovery and Data Mining*, pages 375–398. The MIT Press, 1996.