# IQL: A Proposal for an Inductive Query Language

Siegfried Nijssen and Luc De Raedt

Institut für Informatik, Albert-Ludwidgs-Universität,
Georges-Köhler-Allee, Gebäude 097, D-79110, Freiburg im Breisgau, Germany
Departement Computerwetenschappen, Katholieke Universiteit Leuven,
Celestijnenlaan 200A, B-3001, Leuven, Belgium
`siegfried.nijssen@cs.kuleuven.be`

**Abstract.** The overall goal of this paper is to devise a flexible and declarative query language for specifying or describing particular knowledge discovery scenarios. We introduce one such language, called IQL. IQL is intended as a general, descriptive, declarative, extendable and implementable language for inductive querying that supports the mining of both local and global patterns, reasoning about inductive queries and query processing using logic, as well as the flexible incorporation of new primitives and solvers. IQL is an extension of the tuple relational calculus that includes functions as primitives. The language integrates ideas from several other declarative programming languages, such as pattern matching and function typing. We hope that it will be useful as an overall specification language for integrating data mining systems and principles.

## 1 Introduction

The area of inductive databases [8], inductive query languages, and constraint-based mining [1] has promised a unifying theory and framework for reasoning about data mining principles and processes, which should also result in powerful inductive query languages for supporting complex data mining tasks and scenarios. The key idea is to treat patterns and models as first-class citizens that can be queried and manipulated. The slogan: "From the user point of view, there is no such thing as real discovery, just a matter of the expressive power of the available query language" has been advocated.

There has been a lot of progress in the past few years witnessed by the introduction of several inductive query languages, such as MINE RULE [12], MSQL [9], DMQL [7] and XMine [3]; Microsoft's Data Mining Extensions (DMX) of SQL Server [16]; the algebra of the 3W model [10]; the query language of the SINDBAD project [11]; the logic based query languages of MolFEA [13] and LDL-Mine [6], which all have contributed new insights: MINE RULE, MSQL, DMQL and XMine focus on the derivation of either frequent itemsets or association rules; notation wise, these languages are extensions of the industry standard SQL language. Microsoft's SQL server includes a larger set of algorithms, and provides an interface for learning, clustering and applying a wider range of data mining algorithms, including association rules, decision trees and Bayesian networks; however, it does not provide a framework for reusing frequent itemsets, or for specifying additional constraints. Similar to Microsoft's SQL server, also SINDBAD provides an extension of SQL, in which data mining algorithms are included as functions that transform relations into relations, but only little attention is devoted to the use of constraints. The approach of Calders et al. [4], on the other hand, concentrates mainly on the specification of constraints. The algebra of the 3W model supports association rule discovery as well as learning rule based classifiers, but focuses more on postprocessing than on the specification of constraints. The language of MolFEA allows for the discovery of patterns under constraints, and is more abstract, but does not integrate classification or clustering algorithms. LDL-Mine is similar in spirit to our proposal, but takes Datalog as its starting point and is less focused on the representation of constraints. The data mining algebra and the Datalog++ foundations of LDL-Mine are sufficiently complete to represent data mining algorithms themselves. Relevant is finally also the study of Siebes [15] concerned with upgrading relational algebra operators, such as selection and projection, to data mining models.

Despite this plethora of languages, there is still no comprehensive theory of inductive querying or a unifying query language that is powerful yet simple. In this paper, we propose the inductive query language IQL, which addresses some of the limitations of existing approaches, and integrates many of the ideas that have been proposed in the aforementioned papers, such as the use of *data as a bridge* [11], the conception of domains as *virtual relations* [4], and the use of *logic* as a query language [13]. In contrast to earlier work, the overall goal of this research is to devise a flexible and declarative query language for specifying KDD scenarios.

We designed IQL with the following goals with in mind:

- to provoke discussion on inductive query languages;
- to encompass a rich variety of data mining tasks, including: local pattern mining over different domains, clustering, classification, regression as well as probabilistic modeling;
- to integrate models as first-class citizens of the database;
- to support reasoning about queries, their execution and their optimization;
- to integrate data mining primitives in a database language; as database language we employ an extension of the tuple relational calculus rather than SQL because this allows us to focus more on the principles of the language than on the syntax, but our ideas extend to many other languages;
- to design an extendable language, in which other researchers can describe and possibly implement their constraints, primitives and approaches; if this succeeds, IQL might become a unifying description or specification language for data mining;
- to design an implementable language, even though we wish to stress that –at this point– we are not concerned with the efficiency of the resulting system but rather with the underlying principles.

The final language that we have in mind is very general, and includes for instance existentially and universally quantified formulas. At this point it is an open question if this very general language is implementable. For restrictions that we will point out throughout the paper, we will show that it can indeed be implemented. IQL under these restrictions will be referred to as *simplified* IQL (sIQL).

The paper is organized as follows: Section 2 provides an intuitive introduction to our query language; Section 3 introduces IQL in more detail. We discuss how IQL can be implemented in Section 4. Within IQL we believe that certain primitives should be supported. These are provided in Section 5. We sketch what kind of reasoning is supported by IQL in Section 6. A scenario is described in Section 7. A brief investigation of the possibilities to integrate IQL in other query languages is provided in Section 8. Finally we conclude in Section 9.

Given that IQL is a query language, there are many similarities between IQL and other languages for writing programs or queries. We choose to point out these relations throughout the whole paper, instead of including an additional section for related work.

## 2 Some Example Queries

The best way to introduce the ingredients of a language, and hence also those of IQL, is by providing some examples. IQL is derived from and extends the query language we introduced earlier [13]. An example inspired on that language, but rewritten in IQL is:

$$\begin{aligned}&\textbf{create table } R \textbf{ as}\\ &\{< \mathit{pattern} : S,\ \mathit{freq}1 : \mathit{freq}(S, D_1),\ \mathit{freq}2 : \mathit{freq}(S, D_2) > \ \mid\ S \in \mathit{Sequence} \wedge\\ &\qquad S \preceq \text{“}C - H - 0 - n\text{”} \ \wedge\ \mathit{freq}(S, D_1) = 0 \ \wedge\ \mathit{freq}(S, D_2) \geq 1 \ \}.\end{aligned}$$

This query generates a relation in which the tuples consist of sequential patterns and their frequency in datasets $D_1$ and $D_2$. Furthermore, all patterns must occur at least once in dataset $D_2$, must not occur in $D_1$ and must be more general than (i.e., a substring of) “$C - H - O - n$” . This query thus corresponds to a typical local pattern mining step. By convention, we write variables and relations with capitals.

As a second example, consider

$$\begin{aligned}&\textbf{create view } R' \textbf{ as } \ \{\ T + < \mathit{target} : \mathit{apply}(D, T) > \ \mid\\ &\qquad D \in \mathit{DecisionTree}[< A : \mathit{integer}, B : \mathit{integer} >, < C : \mathit{string} >] \wedge\\ &\qquad\qquad C4.5(D, R) \wedge T \in R\ \}.\end{aligned}$$

This query creates a view $R'$, which extends the relation $R$ with the attribute *target*. The value of *target* is the prediction made by a decision tree generated by C4.5 on the projection of $R$ on the attributes $A$, $B$ and $C$ (as the class attribute). So, this query does not only generate a decision tree but also applies it to a data set, which corresponds – in part – to a cross-over operation.

These two examples illustrate the following key ingredients of IQL:

- queries are generating relations of the form { tuple | condition (tuple) };
- IQL is an extension of the relational tuple calculus;
- the result of a query is a relation, hence, the closure property is satisfied;
- the values of the tuples can be complex, e.g. sequences, functions, etc.; we also allow for operations such as "+" and "-" on tuples, which add, respectively remove attributes from tuples;
- the logical connectives $\wedge, \vee, \neg$ are permitted; (in sIQL, we do not allow $\vee$;)
- IQL is able to employ functions; for instance, $\mathit{freq}(P, D)$, which computes the frequency of the pattern $P$ in the dataset $D$;
- IQL employs a typing system; for instance, the decision tree $D$ maps tuples with attributes $A$ and $B$ onto their classes $C$;
- as in [4] a virtual relation represents a domain, for instance, *DecisionTree*;
- as in the language by [13], there are some built-in predicates such as $\preceq$, which denotes generality, and $freq(P, D)$;
- calls to specific algorithms, such as C4.5, can be integrated; this may be realized using pattern matching, cf. Section 5.

Let us now define these ingredients in a more formal manner.

## 3 Manipulation of Data

To manipulate data as well as pattern and functions, we shall employ an extension of the tuple relational calculus. The tuple relational calculus is a standard theoretical query language for relational databases [5,14]. By using the relational calculus, we keep the desirable closure property: the result of each query is a relation. Furthermore, the relational calculus is based on logic and is therefore declarative.

Essential in the relational model is that data is stored in relations, each of which consists of a set of tuples. A tuple is an expression of the form $< n_1 : v_1, \ldots, n_k : v_k >$ where $n_i$ is an attribute, and $v_i$ a value out of the domain $D_i$ of the attribute $n_i$, e.g. the tuple $< a : 0,\ b : 1 >$. For reasons of convenience, we allow tuples to be joined or subtracted using the $+$ and $-$ symbols [1]. The schema of a tuple is denoted by $< n_1 : D_1, \ldots, n_k : D_k >$. For instance, in the above example, this is $< a : \mathit{boolean}, b : \mathit{boolean} >$. A relation is then a set of tuples over a particular schema, e.g. $R = \{\ < a : 0,\ b : 1 >, < a : 1,\ b : 0 >\ \}$. We will also say that the type of a relation is $\{< n_1 : D_1, \ldots, n_k : D_k >\}$. In tuple relational calculus, variables range over tuples in relations.

The *syntax* of the tuple relational calculus is then defined as follows. A query is an expression of the form $\{T|q\}$, where $T$ is a tuple and $q$ is a formula.

[1] For simplicity we shall assume that no clashes occurs (as e.g. in $< \mathit{temp} : 5 > + < \mathit{temp} : 6 >$).

A *formula* is usually built from the traditional connectives $\wedge$, $\vee$ and $\neg$, and contains variables that can be quantified using the $\exists$ and $\forall$ quantifiers. In sIQL, we restrict ourselves to formulas without the $\forall$ quantifier and $\vee$ connective. The following atoms are allowed:

– atoms of the form $e_1 \theta e_2$, where $\theta \in \{\geq, \leq, >, <, =, \neq\}$ and $e_i$ is a term. Constants, attributes $a$ of tuples $T$ (denoted by $T.a$) and tuples with one attribute can be used as terms;
– $T \in R$, where $T$ is a tuple variable, and $R$ is a relation.

From a data mining perspective, a dataset is conceived as a set of tuples, each of which contains information about an example. One crucial aspect of IQL is that we allow for arbitrary domains. For instance, we shall consider the domain of graphs, sequences, ... For such domains, there will typically be special built-in operators such as for instance the generality or covers relation $\preceq$ stressed by [13]. Similarly, we can conceive a pattern set as a set of tuples, each of which contains a pattern.

Even though we assume that the inductive database conceptually deals with domains such as graphs or sequences, this does not mean that we claim that an inductive database should be able to store such structures entirely in an attribute. For instance, an attribute in the graph domain could also be implemented as an identifier pointing to another relation storing the real graphs. At this point, we abstract from implementation details, such as how objects are incorporated or implemented, and essentially only assume that they can be manipulated and passed on using IQL calculus.

A crucial extension is that we allow for functions. To achieve this, we propose the use of a typing system which includes the following types:

– basic types;
  **Examples:** *integer*, *float*, *boolean*
– complex types;
  **Examples:** *decisionTree*, *itemset*
– a tuple type $<>$, which is taken on by every tuple. We can specialize this type; if $\tau_1, \ldots, \tau_n$ are types, and $\lambda_1, \ldots \lambda_n$ are identifiers,

$$< \lambda_1 : \tau_1, \ldots, \lambda_n : \tau_n >$$

  specifies tuples that contain at least the given attributes; so, we constrain tuples to a certain *schema*;
  **Examples:** $<$ *pattern:itemset,support:integer* $>$, $<$ *tree:decisiontree, acc* : $float >$
– if $\tau$ is the type of a tuple, then

$$\{\tau\}$$

  is the type of a relation of tuples of this type;
  **Examples:** $\{<>\}$, $\{<$ *pattern:itemset,support:integer* $>\}$
– a schema type, which allows one to pass on a list of attribute identifiers to a function; so, the expression $< A : integer, B : integer >$ is of type *schema.*
  **Example:** *schema*

– if $\tau$ is a complex type and $\theta_1, \ldots, \theta_n$ are tuple types, then

$$\tau[\theta_1, \ldots, \theta_n]$$

is a parameterized type; in contrast to functional programming languages, the parameter types are not intended to allow for generic programming; $\theta_1, \ldots, \theta_n$ are intended to associate to a model the schema of the data it was learned from, which could be used later on to constrain the applications of the model;
**Example:** *decisionTree*[< *A:integer,B:integer* >, < *C:string* >]
In this example, < *A:integer,B:integer* > is the set of attributes that are used to perform predictions by the decision tree, and < *C:string* > is the attribute that is predicted by the tree.

Using these types, we can now specify what the signature of a function is:

$$(\sigma_1, \ldots, \sigma_n) \rightarrow \theta,$$

where $\sigma_i$ is an input parameter type and $\theta$ is a return value type. In addition to the <> type, we will also allow variables in signatures, denoted by $\alpha$, $\beta$, .... These variables represent the type <>, but take on the schema of the tuple or relation that is passed to the function. The idea is that they can be used to express further constraints on the schemas of relations that are passed on to a function.
**Examples**

– $\mathit{apply} : (\mathit{decisionTree}[\alpha, \beta], \gamma) \rightarrow \mathit{integer} \quad (\alpha \subseteq \gamma)$
This signature defines a function that applies a decision tree to a tuple in a relation. The variables $\alpha$, $\beta$ and $\gamma$ in this signature, which are used in stead of the general <> type, can be used to express additional constraints. In this case, it is required that the input attributes of the decision tree should occur in the relation to which it is applied.
– $\mathit{join} : (\{\alpha\}, \{\beta\}, \mathit{schema}[\gamma]) \rightarrow \{\alpha + \beta\} \quad (\gamma \subseteq \alpha, \gamma \subseteq \beta)$
This signature defines a function that takes two relations as input, and a set of attribute identifiers that are common to both relations, and produces a relation in which tuples are joined on these common attributes. In this notation, $\gamma$ takes on the tuple schema that is passed as parameter to the function.

As we allow for functions, we can also easily deal with predicates, by conceiving predicates as functions with boolean return type. Functions and predicates are incorporated in IQL by allowing expressions of the form $f(e_1, \ldots, e_n)$ where $f$ is a function and the $e_i$ are expressions with types that should satisfy the constraints specified in the signature of the function. These expressions can occur in atoms, as well as in tuples, as they denote particular values.

As illustrated by the *join* function, the inclusion of functions in IQL means that many common operations of the relational algebra can be implemented. IQL is however more powerful than the traditional relational algebra to which additional operators are added. A query which can be expressed in IQL is

$$\{D + T | D \in R \wedge T \in f(D)\},$$

for some relation $R$ and function $f$ that returns a relation. In this query, we iterate over one relation, and apply an operator on each tuple in the relation, resulting in a relation for that particular tuple.

The final elements of IQL are:

- in addition to types, we also allow for the definition of *type classes*, which are similar to those found, for instance, in the functional programming language Haskell. Concretely, a type class specifies a set of function headers in which the class is a parameter. A type can only be an instance of a class iff all these functions are implemented for the type.
  **Examples:** For the class *classifier*, we can require that the function $\mathit{apply} : (\mathit{classifier}[\alpha, \beta], \gamma) \rightarrow \mathit{integer}$
  exists. The *classifier* type can be used for instance for relations containing multiple types of classifiers; still, we can apply the common operation *apply* to all of them.
- we introduce a *virtual relation* of schema $\{< \mathit{element}{:}\tau >\}$ for every complex type $\tau$ (similar to [4]). These relations are necessary to define the pattern type or model type of interest;
  **Example:** $\{T | T \in \mathit{Itemset} \wedge \mathit{freq}(T, D) \geq 10\}$
  Here, the schema of *Itemset* is $\{< \mathit{element{:}itemset} >\}$. Observe that we capitalize the first letter of the type name when used as a relation.
- we allow new relations to be created that contain the result of a query, and we allow for the definition of functions; views are special functions without arguments.
  **Examples**
  **create function** $f(id : Int)$ **as** $\{\ t - < id >\ |\ t \in D \wedge t.id = id\ \}$
  **create table** $F$ **as** $\{\ < \mathit{pattern} : S, id : V.id, \mathit{freq} : \mathit{freq}(S, f(V.id)) >\ |$
  $V \in ID \wedge S \in \mathit{Sequence} \wedge \mathit{freq}(S, f(V.id)) \geq 10\ \}$;
  Here $ID$ and $D$ are relations in the database.

## 4 Evaluation of Queries

Given that our language supports statements such as ‘create table’ that modify the state of the inductive database, our language can be considered imperative. Still, the queries themselves are more declarative in nature, and the question rises as to how we can evaluate them. An important property of our language is that it supports both declarative and procedural mechanisms for specifying queries. Assume that we have an algorithm for learning decision trees[2], then we can represent this using a function

$$\mathit{dtLearner} : (\{\alpha\}, \mathit{schema}[\beta], \mathit{schema}[\gamma]) \rightarrow \{< \mathit{tree} : \mathit{decisionTree}[\beta, \gamma] >\} \qquad (\beta \subseteq \alpha, \gamma \subseteq \alpha);$$

the decision tree learner takes as input the relation for which a decision tree is to be learned, and furthermore, the identifiers of attributes that are used as inputs and class attribute, respectively. It produces a relation containing decision trees (in most cases, only one decision tree). This function can be used in a query such as

**create table** $T$ **as** $\mathit{dtLearner}(R, < A : \mathit{integer} >, < B : \mathit{string} >)$

Alternatively, we could also define a function

$$\mathit{isLearnedTree} : (\{\alpha\}, \mathit{decisionTree}[\beta, \gamma]) \rightarrow \mathit{boolean} \qquad (\beta \subseteq \alpha, \gamma \subseteq \alpha),$$

which succeeds if a decision tree has been learned from certain data by an algorithm. This function can be used to allow for the query

**create table** $T$ **as**
$\{T | T \in \mathit{DecisionTree}[< A : \mathit{integer} >, < B : \mathit{string} >] \wedge \mathit{isLearnedTree}(D, T)\}.$

The main idea behind the evaluation of our query language is to rewrite the declarative query into its procedural form, not unlike the way that relational calculus is rewritten in relational algebra. We can achieve this through the *pattern matching* principle that is common in many declarative programming languages, such as Prolog and Haskell, but clearly, the pattern matching system of IQL must be more powerful than the systems used in these languages. It is an open question as to how powerful the pattern matching system for (full) IQL should be, but to illustrate that evaluation by pattern matching is possible, we will show this for the simplified IQL in the remainder of this section.

In sIQL, we propose to drive the pattern matching system by declarations of the following form:

$$\{T_1 + \cdots + T_n | \phi\} \equiv f,$$

where $\phi$ is a conjunctive formula of the form $\phi = \{T_1 \in R_1 \wedge \ldots \wedge T_n \in R_n \wedge a_1 \wedge \ldots \wedge a_m\}$; $R_1, \ldots, R_n$ are relations and $a_1, \ldots, a_m$ are atoms; on the righthand side of the declaration a function call $f$ is given. As an example, we can have the following declaration:

$$\{\underline{T} | \underline{T} \in \mathit{decisionTree}[\underline{B}, \underline{G}], \mathit{isLearnedTree}(\underline{D}, \underline{T})\} \equiv \mathit{dtLearner}(\underline{D}, \underline{B}, \underline{G}), \qquad (1)$$

In this declaration, some variables are underlined. These variables are *substitutable*. A substitution for a pattern is a set $\theta = \{V_1/T'_1, \ldots, V_n/T'_n\}$; when the substitution

[2] For reasons of simplicity, we assume that this algorithm does not have additional parameters.

is applied to the pattern, all substitutable variables $V_i$ are simultaneously replaced with corresponding new terms $T'_i$ as defined in the substitution set $\theta$.

The declarations are used to guide the rewriting of queries. If the formula $\phi$ of a pattern equals part of a query after a substitution, the matched atoms in the query are replaced with the righthand side of the declaration that matched.

For instance, for the query:

$$\{C|C \in \mathit{decisionTree}[< a : \mathit{integer} >, < b : \mathit{integer} >] \wedge \\ \mathit{isLearnedTree}(R, C) \wedge \mathit{acc}(C, R) \geq 10\},$$

we can apply substitution $\theta = \{T/C, B/ < a : \mathit{integer} >, G/ < b : \mathit{integer} >, D/R\}$ to the formula of Equation 1 to obtain a match.

The rewriting proceeds as follows. First, we compute the atoms that were not matched with the pattern, $q - \phi\theta$, to make sure that they reoccur in the rewritten query. Then, we add a new atom $(T \in f\theta)$ to this set of atoms, which ranges over the result of a function call as defined by the righthand side of the matched declaration (we abort the pattern matching if we detect a type mismatch). Finally, we have to make sure that all variables that ranged over relations that disappeared in the new query, range over the result of the function call. This can be achieved by applying a final substitution[3].

In our example, after applying substitution $\theta$, we can rewrite the query into

$$\{T|T \in \mathit{dtLearner}(R, < a : \mathit{integer} >, < b : \mathit{integer} >) \wedge \mathit{acc}(T, R) \geq 10\}.$$

For a query that does not contain virtual relations, and for which all functions are implemented, we can use a straightforward evaluation method, similar to the evaluation of *list comprehensions* in programming languages such as Haskell or Python: first, the atoms are ordered. Then, for every atom $T \in R$ ('generator expressions'), if $R$ is function call, it is evaluated; for every possible value in the resulting relation, the remainder of the atoms is evaluated. Other atoms ('guard expressions') are evaluated by performing the necessary function calls first, and testing the results of the function calls. The lefthand side of the query is evaluated for every combination of tuples that survives all guards.

## 5 Primitives and Extensions

In this section, we study a list of possible queries, and investigate how they can be represented in our language. In this discussion, we will point out whether the queries are already supported by the sIQL or if the full IQL is required.

*Condensed representations.* We have seen already that frequent pattern miners in general can be represented by introducing a type class *pattern*. Algorithms

[3] For reasons of simplicity, we assume there are no name clashes between the attributes of relations; we assume that the function that is called, returns a tuple that contains all attributes of the matched relations.

for mining using condensed representations, such as closed itemsets, can be represented by including functions

$$\mathit{isClosed} : (\mathit{itemset}, \{\mathit{itemset}\}) \rightarrow \mathit{boolean},$$

which checks if an itemset is closed within a certain database, and

$$\mathit{closedMiner} : (\{\mathit{itemset}\}, \mathit{integer}) \rightarrow \{\mathit{itemset}\},$$

which returns the set of frequent closed itemsets for a given database and support threshold. These functions are used in the declaration

$$\{\underline{I} | \underline{I} \in \mathit{Itemset}, \mathit{freq}(\underline{I}, \underline{R}) \geq \underline{T}, \mathit{isClosed}(\underline{I}, \underline{R})\} \equiv \mathit{closedMiner}(\underline{R}, \underline{T}).$$

This procedure can be repeated for every kind of condensed representation and type of pattern. Observe that if the *isClosed* function is implemented separately, we have two different ways to evaluate a closed itemset mining query: one option is to call a closed itemset miner; another option is to call a frequent itemset miner, and to postprocess the results. It is a matter of optimization which of these two options is chosen.

A useful feature of IQL could be to introduce templates. It can then be specified that for a condensed representation, the above mentioned set of two functions types and one declaration should be provided.

*Miners under multiple constraints.* Some data mining algorithms are able to deal with conjunctions of constraints of arbitrary size, for instance, the MolFEA algorithm [13]. It is obvious that such algorithms are straightforwardly represented in IQL. Evaluation within the sIQL setting is however difficult, as every pattern in the pattern matching system has a fixed size. Both a more complicated pattern matching system and a more complicated typing system are required to pass variable numbers of constraints to an algorithm.

*Top-k pattern miners.* A recent branch of research involves that of mining top $k$ patterns, where the top $k$ patterns are determined according to some convex measure, such as the $\chi^2$ test. One way that a user could specify such a query is

$$\textbf{create view } V = \{< \mathit{itemset} : I, \mathit{value} : \chi^2(I, D) > | I \in \mathit{Itemset}\}$$

$$\{I | I \in \mathit{Itemset} \wedge \mathit{rank}(I, V) \leq 10\}.$$

Here, $\chi^2$ is a function with signature

$$\chi^2 : (\mathit{itemset}, \{< \mathit{itemset} : \mathit{itemset}, \mathit{class} : \mathit{string} >\}) \rightarrow \mathit{float};$$

this function computes the correlation of an itemset in a dataset that contains at least an itemset attribute and a class attribute. The *rank* function has signature

$$\mathit{rank} : (\mathit{itemset}, \{< \mathit{itemset} : \mathit{itemset}, \mathit{value} : \mathit{float} >\}) \rightarrow \mathit{integer}$$

and returns the position of an itemset in a set of itemsets that is sorted according to associated floating point values.

The link between a top $k$ pattern miner and a declarative query is formalized by the declaration

$$\{\underline{I} | \underline{I} \in \mathit{Itemset}, \mathit{rank}(\underline{I}, \underline{V}) \leq \underline{T}\} \equiv \mathit{TopKChi2Miner}(\underline{D}, \underline{T})$$

under the constraint that $\underline{V}$ is a view of the form $\{< \mathit{itemset} : \underline{I}, \mathit{value} : \chi^2(\underline{I}, \underline{D}) > | \underline{I} \in \mathit{Itemset}\}$.

A variation of this approach, which allows us to deal with a larger number of convex measures, is to replace the $\chi^2$ function with a general function

$$\mathit{applyconvex} : (\mathit{measure}, \mathit{itemset}, \{< \mathit{itemset} : \mathit{itemset}, \mathit{class} : \mathit{string} >\}) \rightarrow \mathit{float}.$$

A benefit of using views, is that it is easy to incorporate additional constraints on top $k$ patterns. For instance, if we are interested in the top $k$ free itemsets, this can be expressed by modifying the view into

**create view** $V =$
$\{< \mathit{itemset} : I, \mathit{value} : \chi^2(I, D) > | I \in \mathit{Itemset} \wedge \mathit{isFree}(I, D)\}$

As soon as the user introduces a minimum frequency constraint in the view,

**create view** $V =$
$\{< \mathit{itemset} : I, \mathit{value} : \chi^2(I, D) > | I \in \mathit{Itemset} \wedge \mathit{freq}(I, D) \geq 10\}$,

a different query evaluation plan can emerge in which the view is first materialized; the end result can be obtained by postprocessing the materialized view.

*Classification algorithms.* We have already seen how a decision tree can be integrated in IQL. It is easy deal to with further constraints on decision trees. For instance, if we define the function $\mathit{size} : \mathit{decisiontree}[\alpha, \beta] \rightarrow \mathit{integer}$, this function can be used in a declaration:

$$\{\underline{T} | \underline{T} \in \mathit{DecisionTree}[\underline{B}, \underline{G}], \mathit{isLearnedTree}(\underline{D}, \underline{T}), \mathit{size}(\underline{T}) \leq \underline{M}\} \equiv \mathit{dtMaxLearner}(\underline{D}, \underline{B}, \underline{G}, \underline{M})\},$$

for an appropriate decision tree learner *dtMaxLearner*.

Observe that if we have two decision tree learners, this query may be evaluated in two ways:

– the specialized decision tree learner can be used;
– a general decision tree learner can be used, whose output is postprocessed.

In the second approach, the result of the query may be empty if by default a heuristic decision tree learner is used, and this algorithm finds a tree that is too large. It is an essential property of many data mining algorithms that their output is not defined otherwise than through the implementation of the algorithm itself, while in ordinary databases, the outcome of a query does not depend on the evaluation strategy (see, for instance, also [15] about this issue). If one believes that deterministic behavior is also desirable in inductive databases, there are several possible solutions:

- we can disallow predicates and declarations that could lead to alternative query execution plans for heuristic algorithms; for instance, in the case of decision trees, we could forbid the use of predicates such as *isLearnedTree* in favor of predicates such as *C4.5*;
- if multiple query evaluation plans exist within the database, we execute them all; the result of the query is the union of all executions.

Our query language also shows that for several kinds of queries on classification models currently no solvers exist, for instance:

$$\begin{aligned}&\textbf{create view } V = \{< \mathit{model}: T, \mathit{value}: \mathit{accuracy}(T,D), \mathit{value2}: \mathit{size}(T) > \mid\\ &\qquad T \in \mathit{DecisionTree}[< A: \mathit{integer}, B: \mathit{integer} >] \wedge \mathit{leafsup}(T,D) \geq 2\}\end{aligned}$$

$$\{T | T \in V \wedge \mathit{rank}(T,V) \leq 2\}.$$

This query asks for all decision trees for which the accuracy is maximal and ties are cut by taking the smallest possible tree. The search space is restricted to those trees in which each leaf contains at least two examples of the training data. Similar queries can also be posed for other types of models.

*Probabilistic Models.* In contrast to classifiers, probabilistic models do not output a single class, but a probability distribution over a set of target attributes. The type of the *apply* function is

$$\mathit{apply}: (\mathit{probmodel}[\alpha,\beta],\gamma) \rightarrow \{< \mathit{string}, \mathit{float} >\} \qquad (\alpha \subseteq \gamma, \beta \subseteq \gamma >),$$

and reflects that for every example, a distribution over the class attributes is returned. The approach for learning probabilistic models is similar to that for classification models.

*Clustering.* Clustering algorithm do not target a specific class attribute, but rather try to find meaningful groups within the data, and can easily be integrated in IQL. For instance, assume that we have a $k$-means clustering algorithm that puts examples into multiple clusters and assigns a degree of membership for each cluster (for example, according to the distance to the cluster centre). Then the following declaration formalizes such an algorithm:

$$\begin{aligned}&\{\underline{T} + \underline{L} + \underline{C} \mid \underline{C} \in \mathit{KMeansClustering}[\underline{X}],\\ &\qquad \mathit{isClustering}(\underline{C},\underline{R}), \mathit{size}(\underline{C}) = \underline{N}, \underline{T} \in \underline{R}, \underline{L} \in \mathit{apply}(\underline{C},\underline{T})\} \equiv\\ &\qquad\qquad\qquad\qquad\qquad\qquad \mathit{kMeansLearner}(\underline{T},\underline{N},\underline{X}),\end{aligned}$$

where we assume the following function types:

$$\begin{aligned}&\mathit{isClustering}: (\mathit{kMeansClustering}[\alpha], \{\beta\}) \rightarrow \mathit{boolean} \quad (\alpha \subseteq \beta)\\ &\mathit{size}: (\mathit{kMeansClustering}[\alpha]) \rightarrow \mathit{integer}\\ &\mathit{apply}: (\mathit{kMeansClustering}[\alpha], \beta) \rightarrow\\ &\qquad\qquad \beta \cup < \mathit{cluster}: \mathit{integer}, \mathit{membership}: \mathit{float} > \quad (\alpha \subseteq \beta)\end{aligned}$$

This query attaches to every example the clusters that it is part of, and the degree of this membership. Observe that in this pattern, learning and prediction are combined. For many clustering algorithms, it is difficult to separate these operations. However, if a clustering algorithm generates a model for assigning clusters to unseen examples, then it can be handled as a classifier or a probabilistic model.

*Feature Construction.* Once a set of local patterns has been mined, a common operation is to use the patterns for creating new features for a set of examples. For instance, for a frequent itemset $\{A, B\}$ in a relation $R$, one could add an attribute $AB$ to $R$ which is *true* for every tuple that contains $\{A, B\}$, and *false* otherwise.

To support this operation, we require that two types of functions are supported by the inductive database. First, a function

$$\mathit{name} : (\mathit{type}) \rightarrow \mathit{string},$$

is required for every *type*, which assigns names to data mining objects. We leave it unspecified whether this name should be interpretable; important is that it if two objects are not equivalent, they should never be given the same name.

Then, a function

$$\mathit{transpose}\colon (\{\alpha\}, \mathit{boolean}) \rightarrow \{\alpha - \beta\} \;\; (\beta = < \mathit{name} : \mathit{string}, \mathit{value} : \mathit{boolean} >, \beta \subseteq \alpha)$$

is required. This function groups all tuples in the input relation according to all attributes other than *name* and *value*, and creates a new relation in which new attributes are added for every *name*, of which the values are obtained from the *value* fields; if no value is available, the default value is used that is a parameter of the function. An example of the application of the *transpose* function is given below.

| Id | Name | Value |
|---|---|---|
| 1 | A | *true* |
| 1 | B | *true* |
| 2 | A | *true* |
| 2 | B | *false* |

$\Rightarrow$

| Id | A | B |
|---|---|---|
| 1 | *true* | *true* |
| 2 | *true* | *false* |

Only after a table is created using this function, and the query is finished, its schema is known.

To be able to create a binary value from a pattern, we assume that a function

$$\mathit{covers} : (\mathit{pattern}, \mathit{pattern}) \rightarrow \mathit{boolean}$$

is provided for patterns. Alternatively, the "$\subseteq$" symbol can be used in an infix notation.

## 6 Reasoning

IQL allows one to reason about queries, as [13]. For instance, consider the sequence:

$$\textbf{create table } R \textbf{ as } \{ \; < pattern : S, freq : freq(S, union(D_1, D_2)) > \; | \\ S \in Sequence \wedge freq(S, union(D_1, D_2)) \geq 5 \; \};$$

$$\textbf{create table } R' \textbf{ as } \{ < pattern : S, freq : freq(S, D_1) > \; | \\ S \in Sequence \wedge freq(S, D_1) \geq 5 \};$$

and assume that the queries are posed sequentially. If the inductive querying system has the following background knowledge,

$$D_1 \subseteq union(D_1, D_2)$$
$$D_2 \subseteq union(D_1, D_2)$$
$$D_1 \subseteq D_2 \implies \forall T : freq(T, D_1) \leq freq(T, D_2)$$

Then one can actually see that the answer to the first query is a superset of that of the second one. Therefore, rather than calling the frequent pattern miner again for the second query, one might simply go through the result of the first one to verify which patterns satisfy the second frequency constraint. Examples of this kind of reasoning, and a deeper discussion of these issues, is provided in [13]. Observe, however, that the frequencies of all frequent sequences have to be computed to finally answer the second query, as the frequencies in the second query may be smaller than in the first.

In IQL, this type of reasoning can be extended to constraints on other domains. For instance, a decision tree with minimum accuracy 0.9 on a dataset $R$ is also a decision tree with minimum accuracy 0.8 on the same dataset.

Due to its close connection to relational calculus, there are similar optimization possibilities in IQL as in relational calculus. For instance, consider this query:

$$\{T + < prediction : apply(C, T) > \; | \\ T \in R \wedge C \in DecisionTree[< A : Int >, < B : Int >] \wedge isLearnedTree(C, R)\};$$

to evaluate this query, the query optimizer should first construct the decision tree, and then apply it to all examples; it should not choose to construct the decision tree repeatedly for every example again.

We already pointed out that there can be multiple execution plans if multiple matching patterns and algorithms are provided. It is possible to perform query optimization by comparing execution plans.

## 7 Scenario

IQL should support the description of scenarios [2]. In this section we will demonstrate a typical scenario, in which a pattern miner is used to find frequent patterns, these frequent patterns are then used to create features, and finally a classification model is learned.

The first step in this scenario is easily described. Assume that we have a database of molecules *HIV*, and we are looking for subgraphs with a high support in *active* molecules, but a low support in the *inactive* molecules:

**create function** $\mathit{hiv}(d : \mathit{string})$ **as**
$\{\ T - < \mathit{activity} > \mid T \in \mathit{HIV} \wedge T.\mathit{activity} = d\ \}$

**create table** $R$ **as**
$\{\ S \mid S \in \mathit{Graph} \wedge \mathit{freq}(S, \mathit{hiv}(\text{“}\mathit{active}\text{”})) \geq 10 \wedge \mathit{freq}(S, \mathit{hiv}(\text{“}\mathit{inactive}\text{”})) \leq 10\ \}$

Next, we use the local patterns to create features.

**create function** $f(\mathit{Data} : \{< \mathit{id} : \mathit{integer}, \mathit{graph} : \mathit{graph}, \mathit{activity} : \mathit{string} >\})$ **as**
$\mathit{transpose}($
$\{< \mathit{id} : T.\mathit{id}, \mathit{activity} : T.\mathit{activity}, \mathit{name} : \mathit{name}(G), \mathit{value} : \mathit{covers}(G, T.\mathit{graph}) > \mid$
$T \in \mathit{Data} \wedge G \in R\}, \mathit{false})$

**create table** $\mathit{Features}$ **as** $f(\mathit{HIV})$

The result of this query is a relation in which columns denote whether a graph contains a certain subgraph or not. We can build a decision tree for this relation.

**create table** $R'$ **as**
$\{D | D \in \mathit{DecisionTree}[\mathit{schema}(\mathit{Features}) - < \mathit{id}, \mathit{activity} >, < \mathit{activity} : \mathit{boolean} >]$
$\wedge\ \mathit{isLearnedTree}(D, \mathit{Features})\}.$

Here, *schema* returns the schema of relation *Features*; the classifier should use the features in this relation, excluding the *id* and *activity* attributes. Finally, we can use this decision tree to predict the activity of molecules in a dataset $\mathit{HIV}'$.

**create table** *HIVPredictions* **as**
$\{T' + < \mathit{pred} : \mathit{apply}(D, T) > | T \in f(\mathit{HIV}') \wedge D \in R' \wedge T' \in \mathit{HIV}' \wedge T.\mathit{id} = T'.\mathit{id}\}.$

This query shows how using traditional data manipulation operations, we can associate the prediction of a molecule to its original representation, instead of its binary feature representation.

## 8 Extensions of Other Query Languages

In this paper we concentrated on an extension of the tuple relational calculus. The tuple relational calculus has the advantage that, when writing queries, we do not need to be concerned with the exact schema of relations. An open question is to what extent the principles of the IQL can be integrated in other query languages. In this section, we preliminarily investigate the issues that rise if we extend other query languages to obtain the same expressive power as the simplified version of the IQL.

*Domain Relational Calculus.* In domain relational calculus, the variables do not range over tuples in relations, but over values of attributes. Furthermore, it is common that attributes are identified by their position in tuples, and not by their names. Our example query for learning a decision tree would be formulated in domain relational calculus as follows:

$$\begin{aligned}\textbf{create view } R' \textbf{ as } \{\ & < X_1, X_2, X_3, \mathit{apply}(D, < X_1, X_1, X_3 >) > \ \mid \\ & < D > \in \mathit{DecisionTree}[\{1,2\},\{3\}] \wedge \\ & C4.5(< D >, R) \wedge < X_1, X_2, X_3 > \in R\ \}.\end{aligned}$$

We do not expect many problems to extend sIQL towards domain relational calculus. More complications can be expected when additional quantifiers and negations are allowed.

*Datalog and Prolog.* Datalog differs from domain relational calculus in several aspects. First, notation wise, the infix predicate $\in$ is not used. Second, more importantly, queries can define new relations recursively. Datalog is therefore more expressive than either tuple or domain relational calculus.

Our most important extension of the tuple relational calculus consists of adding functions. One might therefore think that our query language is very similar to Prolog. Our functions play however a slightly different role than the functions in Prolog. The functions in our language act as predicates in Prolog, and transform input into output. Furthermore, our functions can take relations as input and produce new relations as output. The main point of our pattern matching mechanism is to rewrite queries such that they use functions that are implemented in an arbitrary language. In Prolog, this behavior can only be achieved through the use of meta-predicates (such as *call*) that can be used to emulate higher order logics. To illustrate this issue, consider the query which creates a new table through a function:

$$\begin{aligned}&\textbf{create function } f(id : Int) \textbf{ as } \{\ t - < id > \ \mid t \in D \wedge t.id = id\ \} \\ &\textbf{create table } F \textbf{ as } \{\ < \mathit{pattern} : S, id : V.id, \mathit{freq} : \mathit{freq}(S, f(V.id)) > \ \mid \\ &\qquad V \in ID \wedge S \in \mathit{Sequence} \wedge \mathit{freq}(S, f(V.id)) \geq 10\ \};\end{aligned}$$

This function repeatedly creates a temporary relation that is passed to another predicate. A Prolog *freq* predicate would have to take a formula (query) as argument and materialize this formula in order to compute the frequency.

Overall, it is already feasible to integrate principles of sIQL in Datalog, but additional research is required to make this integration smoother.

*Algebra.* To give relational algebra the same expressive power as sIQL, we face similar problems as with Datalog. The most obvious way to integrate functions into the algebra, is to conceive functions as additional operators in the algebra; after all, we have already pointed out that queries expressed in relational algebra can be conceived as repeated applications of functions in sIQL. Still, we need additional formalisms to deal with functions that do not act on relations, or functions that are repeatedly applied to relations created by another function.

One way to address this problem is to add a *loop operator* $\iota$ to the relational algebra. Given a function $f(\sigma_1, \sigma_2, \ldots, \sigma_n)$, we can define that

```
**operator** $\iota_f(R_0, R_1, \ldots, R_n)$ :
Let $< \lambda_1 : \tau_1, \ldots, \lambda_m : \tau_m >$ be the schema of $R_0$
$R' = \emptyset$
**for each** $T \in R_0$ **do**
    **for** $1 \leq i \leq n$ **do**
        select tuples from $R_i$ such that $R_i.\lambda_1 = T.\lambda_1 \wedge \ldots \wedge R_i.\lambda_m = T.\lambda_m$,
            for those attributes of $T$ that also occur in $R_i$
        project $\lambda_1, \ldots, \lambda_m$ away from the selected tuples
        store the resulting relation in $x_i$
    Let $x' = f(x_1, \ldots, x_n)$
    $R' = R' \cup (T \times x')$
**return** $R'$
```

The main idea behind this loop operator is that $R_0$ contains the values of an iterator, and the relations $R_i$ $(i \geq 1)$ contain the parameters with which the function is called for each value of the iterator. For each value of the iterator a call is performed; the result is stored. If a relation $R_i$ contains multiple rows for the same iterator value, a relation with multiple tuples is passed to the function.

Given the function $f$ which selects tuples based on their class attribute, we can now formulate the following query:

$$\sigma_{freq \geq 10}(\iota_{freq}(Sequence \times ID, I(Sequence), \iota_f(ID, V))),$$

In this query $\iota_f(ID, V)$ creates a relation which for every identifier in $ID$, stores the selected part of the dataset $V$. Relation $I(Sequence)$ is the relation that associates every sequence with itself. Next $\iota_{freq}$ associates to every combination of a sequence and an identifier the corresponding frequency. Only those sequences with a frequency greater than 10 end up in the resulting relation. As this relation cannot be evaluated due to the infinity of $Sequence$, it would have to be rewritten into

$$\iota_{frequentSequenceMiner}(ID, 10, \iota_f(ID, V)).$$

How such rewriting can be achieved, and if there is an automatic way of rewriting sIQL in a well-specified relational algebera, is an open question which we will not address further in this paper.

## 9 Conclusions

We presented a relational calculus for data mining. A key ingredient was the inclusion of functions. This allowed us to integrate a large set of algorithms into IQL, including classification algorithms and clustering algorithms.

The inclusion of functions in the calculus has major other consequences. Common operators in relational algebra, such as *join* and *project*, can also be conceived as functions. We have seen that our language is more powerful than a relational algebra to which functions are added.

To evaluate queries, we proposed the use of *pattern matching*, which is common in many other declarative programming languages. We investigated how several common data mining operations can be expressed as queries in our calculus, and found that most algorithms can be integrated by making the pattern matching language more powerful. One could argue that the power of the declarative languages is determined by the power of its pattern matching language. We provided a concrete evaluation strategy for a simplified version of the IQL.

Even though IQL was presented in a rather informal way, we believe that IQL can already be used as a description language and interface to a wide variety of data mining algorithms and techniques in a uniform and theoretically appealing way. The authors would also like to herewith invite other groups interested in the development of inductive query languages to describe their favorite constraint based mining tools within IQL.

**Acknowledgments.** This work was supported by the EU FET IST project IQ ("Inductive Querying"), contract number FP6-516169. The authors would like to thank John Lloyd for interesting discussions, and visitors of the workshop for their comments.

## References

1. Bonchi, F., Boulicaut, J.-F. (eds.) KDID 2005. LNCS, vol. 3933, Springer, Heidelberg (2006)
2. Boulicaut, J.-F., De Raedt, L., Mannila, H. (eds.): Constraint-Based Mining and Inductive Databases. LNCS (LNAI), vol. 3848. Springer, Heidelberg (2006)
3. Braga, D., Campi, A., Ceri, A., Lanzi, S., Klemetinen, M.: Mining association rules from XML data. In: Kambayashi, Y., Winiwarter, W., Arikawa, M. (eds.) DaWaK 2002. LNCS, vol. 2454, Springer, Heidelberg (2002)
4. Calders, T., Goethals, B., Prado, A.: Integrating pattern mining in relational databases. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) PKDD 2006. LNCS (LNAI), vol. 4213, Springer, Heidelberg (2006)
5. Date, C.J.: An introduction to database systems. Addison-Wesley, Reading (2000)
6. Giannotti, F., Manco, G., Turini, F.: Specifying mining algorithms with iterative user-defined aggregates. IEEE Transactions Knowledge and Data Engineering , 1232–1246 (2004)
7. Han, J., Fu, Y., Koperski, K., Wang, W., Zaiane, O.: DMQL: A data mining query language for relational databases. In: Proceedings of the ACM SIGMOD Workshop on research issues on data mining and knowledge discovery, ACM Press, New York (1996)
8. Imielinski, T., Mannila, H.: A database perspective on knowledge discovery. Communications of the ACM 39(11), 58–64 (1996)
9. Imielinski, T., Virmani, A.: MSQL: A query language for database mining. Data Mining and Knowledge Discovery 2(4), 373–408 (1999)
10. Johnson, T., Lakshmanan, L.V., Ng, R.: The 3w model and algebra for unified data mining. In: Proc. VLDB Int. Conf. Very Large Data Bases, pp. 21–32 (2000)

11. Kramer, S., Aufschild, V., Hapfelmeier, A., Jarasch, A., Kessler, K., Reckow, S., Wicker, J., Richter, L.: Inductive databases in the relational model: The data as the bridge. In: Bonchi, F., Boulicaut, J-F. (eds.) KDID 2005. LNCS, vol. 3933, pp. 124–138. Springer, Heidelberg (2006)
12. Meo, R., Psaila, G., Ceri, S.: An extension to SQL for mining association rules. Data Mining and Knowledge Discovery 2(2), 195–224 (1998)
13. De Raedt, L.: A perspective on inductive databases. SIGKDD Explorations 4(2), 69–77 (2003)
14. Ramakrishnan, R., Gehrke, J.: Database Management Systems. McGraw-Hill, New York (2004)
15. Siebes, A.: Data mining in inductive databases. In: Bonchi, F., Boulicaut, J-F. (eds.) KDID 2005. LNCS, vol. 3933, Springer, Heidelberg (2006)
16. Tang, Z., MacLennan, J.: Data Mining with SQL Server 2005. Wiley, Chichester (2005)