Query Languages Supporting Descriptive Rule Mining: A Comparative Study

Marco Botta[1], Jean-François Boulicaut[2], Cyrille Masson[2], and Rosa Meo[1]

[1] Universitá di Torino, Dipartimento di Informatica, corso Svizzera 185, I-10149, Torino, Italy
[2] Institut National des Sciences Appliquées de Lyon, LISI/LIRIS, Bat. Blaise Pascal, F-69621 Villeurbanne cedex, France

Abstract. Recently, inductive databases (IDBs) have been proposed to tackle the problem of knowledge discovery from huge databases. With an IDB, the user/analyst performs a set of very different operations on data using a query language, powerful enough to support all the required manipulations, such as data preprocessing, pattern discovery and pattern post-processing. We provide a comparison between three query languages (`MSQL`, `DMQL` and `MINE RULE`) that have been proposed for descriptive rule mining and discuss their common features and differences. These query languages look like extensions of SQL. We present them using a set of examples, taken from the real practice of rule mining. In the paper we discuss also OLE DB for Data Mining and Predictive Model Markup Language, two recent proposals that like the first three query languages respectively provide native support to data mining primitives and provide a description in a standard language of statistical and data mining models.

1 Introduction

Knowledge Discovery in Databases (KDD) is a complex process which involves many steps that must be done sequentially. When considering the whole KDD process, the proposed approaches and querying tools are still unsatisfactory. The relation among the various proposals is also sometimes unclear because, at the moment, a general understanding of the fundamental primitives and principles that are necessary to support the search of knowledge in databases is still lacking.

In the cInQ project[1], we want to develop a new generation of databases, called "*inductive databases*", as suggested in [5]. This kind of databases integrates *raw data* with *knowledge* extracted from *raw data*, materialized under the form of patterns, into a common framework that supports the KDD process. In this way, the KDD process consists essentially in a querying process, enabled by a powerful query language that can deal with both raw data and patterns. A few query languages can be considered as candidates for inductive databases. Most proposals emphasize one of the different phases of the KDD process. This paper

[1] Project (IST 2000-26469) partially funded by the EC IST Programme - FET.

is a critical evaluation of three proposals in the light of the IDBs' requirements: `MSQL` [6,7], `DMQL` [10,11] and `MINE RULE` [12,13]. In the paper we discuss also `OLE DB for Data Mining` (`OLE DB DM`) by Microsoft and Predictive Model Markup Language (`PMML`) by Data Mining Group [18]. `OLE DB DM` is an Application Programming Interface whose aim is to ease the task of developing data mining applications over databases. It is related to the other query languages because like them it provides native support for data mining primitives. `PMML`, instead, is a standard markup language, based on XML, and describes statistical and data mining models.

The paper is organized as follows. Section 2 summarizes the desired properties of a language for mining inside an inductive database. Section 3 introduces the main features of the analyzed languages, whereas in Section 4 some real examples of queries are discussed, so that the comparison between the languages is straightforward. Finally Section 5 draws some conclusions.

2 Desired Properties of a Data Mining Query Language

A *query language* for IDBs, is an extension of a database query language that includes primitives for supporting the steps of a KDD process, that are:

- The selection of data to be mined. The language must offer the possibility to select (e.g., via standard queries but also by means of sampling), to manipulate and to query data and views in the database. It must also provide support for multi-dimensional data manipulation.
  `DMQL`, `MINE RULE` and `OLE DB DM` allow the selection of data. Neither of them has primitives for sampling. All of them allow multi-dimensional data manipulation (because this is inherent to SQL).
- The specification of the type of patterns to be mined. Clearly, real-life KDD processes need for different kinds of patterns like various types of descriptive rules, clusters or predictive models.
  `DMQL` considers different patterns beyond association rules.
- The specification of the needed background knowledge (e.g., the definition of a concept hierarchy).
  Even though both `MINE RULE` and `MSQL` can treat hierarchies if the relationship '`is-a`' is represented in a companion relation, `DMQL` allows its explicit definition and use during the pattern extraction.
- The definition of constraints that the extracted patterns must satisfy. This implies that the language allows the user to define constraints that specify the interesting patterns (e.g., using measures like frequency, generality, coverage, similarity, etc).
  `DMQL`, `MSQL` and `MINE RULE` allow the specification of various kinds of constraints based on rule elements, rule cardinality and aggregate values. They allow the specification of primitive constraints based on support and confidence measures. `DMQL` allows some other measures like novelty.

- The satisfaction of the *closure property* (by storing the results in the database). **All of them satisfy this property**.
- The post-processing of results. The language must allow to browse the patterns, apply selection templates, *cross over* patterns and data, e.g., by selecting the data in which some patterns hold, or aggregating results.
  `MSQL` is richer than the other languages in its offer of few post-processing primitives (it has a dedicated operator, `SelectRules`). `DMQL` allows some visualization options. However, all the languages are quite poor for rule post-processing.

3 Query Languages for Rule Mining

3.1 `MSQL`

`MSQL` [7,14] has been designed at the Rutgers University, New Jersey, USA. Rules in `MSQL` are based on descriptors, each descriptor being an expression of the form $(A_i = a_{ij})$, where A_i is an attribute and a_{ij} is a value or a range of values in the domain of A_i. A conjunctset is the conjunction of an arbitrary number of descriptors, provided that there is no pair of descriptors built on the same attribute. In practice, `MSQL` extracts propositional rules like $\mathcal{A} \Rightarrow \mathcal{B}$, where $\mathcal{A}$ is a conjunctset and $\mathcal{B}$ is a descriptor (it follows that only a single proposition is allowed in the consequent). We say that a tuple t of a relation R satisfies a descriptor $(A_i = a_{ij})$ if the value of A_i in t is equal to a_{ij}. Moreover, t satisfies a conjunctset C if it satisfies all the descriptors of C. Finally, t satisfies a rule $\mathcal{A} \Rightarrow \mathcal{B}$ if it satisfies all the descriptors in $\mathcal{A}$ and $\mathcal{B}$, but it violates the rule $\mathcal{A} \Rightarrow \mathcal{B}$ if it does not satisfy $\mathcal{A}$ or $\mathcal{B}$. Notice that support of a rule is defined as the number of tuples satisfying $\mathcal{A}$ in the relation on which the mining has been performed, and the confidence is the ratio between the number of tuples satisfying both $\mathcal{A}$ and $\mathcal{B}$ and the support of the rule. An example of a rule extracted from $Emp(emp\_id, job, sex, car)$ relation containing employee data is $(job = doctor) \wedge (sex = male) \Rightarrow (car = BMW)$.

The main features of `MSQL`, as stated by the authors, are:

- *Ability to nest SQL expressions* such as sorting and grouping in a `MSQL` statement and allowing nested SQL queries by means of the `WHERE` clause.
- *Satisfaction of the closure property* and availability of operators to further manipulate results of previous `MSQL` queries.
- *Cross-over between data and rules* with operations allowing to identify subsets of data satisfying or violating a given set of rules.
- *Distinction between rule generation and rule querying.* This allows splitting rule generation, that is computationally expensive from rule post-processing, that must be as interactive as possible.

`MSQL` comprises four basic statements (see Section 4 for examples):

- `Create Encoding` that encodes continuous valued attributes into discrete values. Notice that during mining, the discretization is done "on the fly", so that it is not necessary to materialize a separate copy of the table.

– A `GetRules` query computes rules from the data and materializes them into a rule database. Its syntax is as follows:

```
[Project Body, Consequent, confidence, support]
GetRules(C) [as R1] [into <rulebase_name>]
[where (RC|PC|MC|SQ)]
[sql-group-by clause] [using-encoding-clause]
```

A `GetRules` query can deal with different conditions on the rules:

- Rule format condition (RC), that enables to restrict the items occuring in the rules elements. RC has the following format:
  ```
  Body { in | has | is } <descriptor-list>
  Consequent { in | is } <descriptor-list>
  ```
- Pruning condition (PC), that defines thresholds for support and confidence values, and constraints on the length of the rules. PC has the format:
  ```
  confidence <relop> <float-val in [0.0,1.0]>
  support <relop> <integer>
  support <relop> <float-val in [0.0,1.0]>
  length <relop> <integer>
  relop ::= { < | <= | = | >= | > }
  ```
- Mutex condition (MC), that avoids two given attributes to occur in the same rule (useful when we know some functional dependencies between attributes). Its syntax is:
  ```
  Where <other-conditions>
      { AND | OR }    mutex(method, method [, method])
      [{ AND | OR}    mutex(method, method [, method])]
  ```
- Subquery conditions (SQ), which are subqueries connected with the conventional `WHERE` keyword using `IN` and `(NOT) EXISTS`.

– A `SelectRules` query can be used for rule post-processing, i.e., querying previously extracted rules. Its syntax is as follows:

```
SelectRules(rulebase_name) [where <conditions>]
```

where `<conditions>` concerns the body, the consequent, the support and/or the confidence of the rules.

– `Satisfies` and `Violates`, that allow to cross-over data and rules. These two statements can be used together with a database selection statement, inside the `WHERE` clause of a query.

3.2 MINE RULE

`MINE RULE` has been designed at the Politecnico di Torino and the Politecnico di Milano, Italy [12,13]. This operator extracts a set of association rules from the database and stores them back in the database in a separate relation.

An association rule extracted by `MINE RULE` from a source relation is defined as follows. Let us consider a source relation over the schema $\mathcal{S}$. Let $\mathcal{R}$ and $\mathcal{G}$

be two disjoint subsets of $\mathcal{S}$ called respectively the schema of the rules and the grouping attributes. An association rule is extracted from (or satisfied by) at least a group of the source relation, where each group is a partition of the relation by the values of the grouping attributes $\mathcal{G}$. An association rule has the form $\mathcal{A} \Rightarrow \mathcal{B}$, where $\mathcal{A}$ and $\mathcal{B}$ are sets of rule elements ($\mathcal{A}$ is the body of the rule and $\mathcal{B}$ the head). The elements of a rule are taken from the tuples of one group. In particular, each rule element is a projection over (a subset of) $\mathcal{R}$. Note that however, for a given `MINE RULE` statement, the schema of the body and head elements is unique, even though they may be different.

An example of a rule extracted from the relation $Emp(emp\_id, job, sex, car)$ grouped by emp_id with the schema of the body (job, sex) and the schema of the head (car) is the following: $\{(doctor, male)\} \Rightarrow \{(BMW)\}$. This rule is extracted from within tuples, because each group coincides with a tuple of the relation. Instead, for the relation $Sales(transaction\_id, item, customer, payment)$, collecting data on customers purchases, grouped by *customer* and with the rule schema $(item)$ (where body and head schemas are coincident) a rule could be $\{(pasta), (oil), (tomatoes)\} \Rightarrow \{(wine)\}$.

The `MINE RULE` language is an extension of SQL. Its main features are:

– *Selection of the relevant set of data* for a data mining process. This feature is applied at different granularity levels, that is at the row level (selection of a subset of the rows of a relation) or at the group level (*group condition*). The *grouping condition* determines which data of the relation can take part to an association rule. This feature is similar to the grouping conditions that we can find in conventional SQL. The definition of groups, i.e. the partitions from which the rules are extracted, is made at run time and is not decided a priori with the key of the source relation (as in `DMQL`).
– Definition of the *structure of the rules*. This feature defines single-dimensional association rules (i.e., rule elements are the different values of the same dimension or attribute), or multi-dimensional rules (rule elements involve the value of more than one attribute). The structure of the rules can also be constrained by specifying the cardinality of the rule's body and head.
– Definition of *constraints applied at different granularity levels*. Constraints belong to two categories: constraints applied at the rule level (*mining conditions*), and constraints applied at the *cluster* level (*cluster conditions*). A mining condition is a constraint that is evaluated and satisfied by each tuple whose attributes, as rule elements, are involved in the rule. A cluster condition is a constraint evaluated for each cluster. Clusters are subgroups (or partitions) of the main groups that are created keeping together tuples of the same group that present common features (i.e., the value of the clustering attributes). In presence of clusters, rule body and head are extracted from a pair of clusters of the same group satisfying the cluster conditions. For instance, clusters and cluster condition may be exploited in order to extract association rules in which body and head are ordered and therefore constitute the elementary patterns of sequences.

– Definition of *rule evaluation measures*. Practically, the language allows to define support and confidence thresholds.

The general syntax of `MINE RULE` follows:

```
<MineRuleOp> := MINE RULE <TableName> AS
 SELECT DISTINCT <BodyDescr>, <HeadDescr> [,SUPPORT] [,CONFIDENCE]
 [WHERE <WhereClause>]
 FROM <FromList> [ WHERE <WhereClause> ]
 GROUP BY <AttrList> [ HAVING <HavingClause> ]
 [ CLUSTER BY <AttrList> [ HAVING <HavingClause> ]]
 EXTRACTING RULES WITH SUPPORT:<real>, CONFIDENCE:<real>

<BodyDescr>:= [ <CardSpec> ] <AttrList> AS BODY
<BodyDescr>:= [ <CardSpec> ] <AttrList> AS HEAD
<CardSpec>:=<Number> .. (<Number> | n)
<AttrList>:=<AttrName>[,<AttrList>]
```

3.3 DMQL

`DMQL` has been designed at the Simon Fraser University, Canada [10,11]. In `DMQL`, an association rule is a relation between the values of two sets of predicates that are evaluated on the relations of the database. These predicates are of the form $P(X, c)$ where P is a predicate that takes the name of an attribute of the underlying relation, X is a variable and c is a constant value belonging to the attribute's domain. The predicate is satisfied if in the relation there exists a tuple identified by the variable X whose homonymous attribute takes the value c. Notice that it is possible for the predicates to be evaluated on different relations of the database. For instance, `DMQL` can extract rules like $town(X,' London') \Rightarrow buys(X,' DVD')$ where `town` and `buys` may be two attributes of different relations and `X` is an attribute present in the both relations. Rules may belong to different categories: a single-dimensional rule contains multiple occurrences of a single predicate (e.g., *buys*) while a multi-dimensional rule involves more predicates, each of which occurs only once in the rule. However, the presence of one or more instances of a predicate in the same rule can be specified by the name of the predicate followed by +. Another important feature of `DMQL` is that it allows to guide the discovery process by using metapatterns. Metapatterns are a kind of templates that restricts the syntactical aspect of the association rules to be extracted. Moreover, they represent a way to push some hypotheses of the user and it is possible to incorporate some further constraints in them. An example of metapattern could be $town(X : customer, London) \wedge income(X, Y) \Rightarrow buys(X, Z)$, which restricts the discovery to rules with a body concerning town and income levels of the customers and a head concerning one item bought by those customers. Furthermore, a metapattern can allow the presence of non instantiated predicates that the mining task will take care to instantiate to the name of a valid attribute of the underlying relation. For instance, if we want to extract association rules describing the customers traits that are frequently related to the purchase of certain items by those customers we could use the following metapattern to guide the association rule mining:

$P(X : \mathit{customer}, W) \wedge Q(X, Y) \Rightarrow \mathit{buys}(X, Z)$

where P and Q are predicate variables that can be instantiated to the relevant attributes of the relations under examination, X is a key of the *customer* relation, W, Y and Z are object variables that can assume the values of the respective predicates for customer X.

DMQL consists of the specification of four major primitives in data mining, that are the following:

- *The set of relevant data w.r.t. a data mining process.*
  This primitive can be specified like in a conventional relational query extracting the set of relevant data from the database.
- *The kind of knowledge to be discovered.*
  This primitive may include association rules, classification rules (rules that assign data to disjoint classes according to the value of a chosen classifying attribute), characteristics (descriptions that constitute a summarization of the common properties in a given set of data), comparisons (descriptions that allow to compare the total number of tuples belonging to a class with different contrasting classes), generalized relations (obtained by generalizing a set of data corresponding to low level concepts with data corresponding to higher level concepts according to a specified concept hierarchy).
- *The background knowledge.*
  This primitive manages a set of concept hierachies or generalization operators which assist the generalization process.
- *The justification of the interestingness of the knowledge (i.e., thresholds).*
  This primitive is included as a set of different constraints depending on the kind of target rules. For association rules, e.g., besides the classical support and confidence thresholds, DMQL allows the specification of noise (the minimum percentage of tuples in the database that must satisfy a rule so that it is not discarded) and rule novelty, for selecting the most specific rules.

The DMQL grammar for extracting association rules is an extension of the conventional SQL grammar. Thus, we can find in it traditional relational operators like HAVING, WHERE, ORDER BY and GROUP BY, but we can also specify the database, select the relevant attributes of the database relation and the concept hierarchy, define thresholds and guide the mining process using a metapattern. The general syntax of a DMQL query is:

use database ⟨*database_name*⟩
{**use hierarchy** ⟨*hierarchy_name*⟩ **for** ⟨*attribute_or_dimension*⟩ }
in relevance to ⟨*attribute_or_dimension_list*⟩
mine associations [as ⟨*pattern_name*⟩] [**matching** ⟨*metapattern*⟩]
from ⟨*relation(s)/cube(s)*⟩ [**where** ⟨*condition*⟩]
[**order by** ⟨*order_list*⟩]
[**group by** ⟨*grouping_list*⟩][**having** ⟨*condition*⟩]
with ⟨*interest_measure*⟩ **threshold** = value

3.4 `OLE DB DM`

`OLE DB DM` has been designed at Microsoft Corporation [17] [16]. It is an extension of the `OLE DB` Application Programming Interface (API) that allows any application to easily access a relational data source under the Windows family OS. The main motivation of the design of OLE DB for DM is to ease the development of data mining projects with applications that are not stand-alone but are tightly-coupled with the DBMS. Indeed, research work in data mining focused on scaling analysis and algorithms running outside the DBMS on data exported from the databases in files. This situation generates problems in the deployment of the data mining models produced because the data management and maintenance of the model occurs outside of the DBMS and must be solved by ad-hoc solutions. `OLE DB DM` aims at ease the burden of making data sources communicate with data mining algorithms (also called mining model provider).

The key idea of `OLE DB DM` is the definition of a data mining model, i.e. a special sort of table whose rows contain an abstraction, a synthetic description of input data (called case set). The user can populate this model with predicted or summary data obtained running a data mining algorithm, specified as part of the model, over the case set. Once the mining task is done, it is possible to use the data mining model, for instance to predict some values over new cases, or browse the model for post-processing activities, such as reporting or visualization.

The representation of the data in the model depends on the format of data produced by the algorithm. This one could produce output data for instance by using `PMML` (Predictive Model Markup Language [18]). `PMML` is a standard proposed by `DMG` based on `XML`. It is a mark-up language for the description of statistical and data mining models. `PMML` describes the inputs of data mining models, the data transformations used for the preparation of data and the parameters used for the generation of the models themselves.

`OLE DB DM` provides an SQL-like language that allows client applications to perform the key operations in the `OLE DB DM` framework: definition of a data mining model (with the `CREATE MINING MODEL` statement), execution of an external mining algorithms on data provided by a relational source and population of the data mining model (`INSERT INTO` statement), prediction of the value of some attributes on new data (`PREDICTION JOIN`), browsing of the model (`SELECT` statement).

Thus, elaboration of an `OLE DB DM` model can be done using classical SQL queries. Once the mining algorithm has been executed, it is prossible to do some crossing-over between the data mining model and the data fitting the mining model using the `PREDICTION JOIN` statement. This is a special form of the SQL join that allows to predict the value of some attributes in the input data (test data) according to the model, provided that these attributes were specified as prediction attributes in the mining model.

The grammar for the creation of a data mining model is the following:

```
<dm_create>::=CREATE MINING MODEL <identifier> (<col_def_list>)
              USING <algorithm> [(<algo_param_list>)]

<col_def_list>::= <col_def> |<col_def_list> , <col_def>
<col_def>::= <col_def_reg> | <col_def_tbl>
<col_def_reg>::= <identifier> <col_type> [<col_distribution>]
          [<col_binary>] [<col_content>] [<col_content_qual>]
        [<col_qualif>] [<col_prediction>] [<relation_clause>]

<col_def_tbl> ::= <identifier> TABLE <col_prediction>
( <col_def_list> )

// 2 algorithms currently implemented in SQL server 2000
<algorithm> ::= MICROSOFT_DECISION_TREES | MICROSOFT_CLUSTERING

<algo_param_list>::=<algo_param> | <algo_param>,<algo_param_list>
<algo_param>::= <identifier> = <value>

<col_type>::= LONG | BOOLEAN | TEXT | DOUBLE | DATE

<col_distribution>-> NORMAL | UNIFORM

<col_binary>::= MODEL_EXISTENCE_ONLY | NOT NULL

<col_content>::= DISCRETE | CONTINUOUS
          | DISCRETIZED ( [<disc_method> [, <numeric_const>]] )
          | SEQUENCE_TIME

<disc_method>::=AUTOMATIC | EQUAL_AREAS | THRESHOLDS | CLUSTERS

<col_content_qual>::= ORDERED | CYCLICAL

<col_qualif>::= KEY | PROBABILITY | VARIANCE | STDEV | STDDEV
              | PROBABILITY_VARIANCE | PROBABILITY_STDEV
              | PROBABILITY_STDDEV | SUPPORT

<col_prediction>::= PREDICT | PREDICT_ONLY

<relation_clause>::= <related_to_clause> | <of_clause>

<related_to_clause>::=RELATED TO <identifier> | RELATED TO KEY

<of_clause>::= OF <identifier> | OF KEY
```

Notice that the grammar allows to specify many kinds of qualifiers for an attribute. For instance, it allows to specify the role of an attribute in the model (key), the type of an attribute, if the attribute domain is ordered or cyclical, if it is continuous or discrete (and in this latter case the type of discretization used), if the attribute is a measurement of time, and its range, etc. It is possible to give a probability and other statistical features associated to an attribute value. The probability specifies the degree of certainty that the value of the attribute is correct.

`PREDICT` keyword specifies that it is a prediction attribute. This means that the content of the attribute will be predicted on test data by the data mining algorithm according to the values of the other attributes of the model.

`RELATED TO` allows to associate the current attribute to other attributes, for instance for a foreign key relationship or because the attribute is used to classify the values of another attribute.

Notice that <`col_def_tbl`> production rule allows a data mining model to contain nested tables. Nested tables are tables stored as the single values of a column in an outer table. The input data of a mining algorithm are often obtained by gathering and joining information that is scattered in different tables of the database. For instance, customer information and sales information are generally kept in different tables. Thus, when joining the customer and the sales tables, it is possible to store in a nested table of the model all the items that have been bought by a given customer. Thus, nested tables allow to reduce redundant information in the model.

Notice that `OLE DB DM` seems particularly tailored to predictive tasks, i.e. to predict the value of an attribute in a relational table. Indeed, the current implementation of `OLE DB DM` in Microsoft SQL Server 2000, only two algorithms are provided (Microsoft Decision Trees and Microsoft Clustering) and both of them are designed for attribute prediction. Instead, algorithms that use data mining models for the discovery of association rules, therefore for tasks without a direct predictive purpose, seems not currently supported by `OLE DB DM`. However, according to the specifications [17], `OLE DB DM` should be soon extended for association rules mining.

Notice also that it is possible to directly create a mining model that conforms to the `PMML` standard using the following statement:

```
<pmml_create>::=CREATE MINING MODEL <id> FROM PMML <string>
```

We recall here the schema used by `PMML` for the definition of models based on association rules.

```
<!ENTITY  \% FIELD-USAGE-TYPE "(active |
                               predicted |
                               supplementary)" >

<!ENTITY  \% OUTLIER-TREAT-METHOD "( asIs |
                                     asMissingValues |
                                     asExtremeValues ) " >
```

```
 <!ENTITY  \% MISS-VALUE-TREAT-METHOD "(asIs | asMean |
                                      asMode | asMedian |
                                      asValue) " >

<!ELEMENT MiningField (Extension*)>
<!ATTLIST MiningField
  name                    \%FIELD-NAME;                #REQUIRED
  usageType               \%FIELD-USAGE-TYPE;          "active"
  outliers                \%OUTLIER-TREAT-METHOD;      "asIs"
  lowValue                \%NUMBER;                    #IMPLIED
  highValue               \%NUMBER;                    #IMPLIED
  missingValueReplacement CDATA                        #IMPLIED
  missingValueTreatment   \%MISS-VALUE-TREAT-METHOD; #IMPLIED

<!ELEMENT MiningSchema   (MiningField+) >
```

Notice that according to this specification it is possible to specify the schema of a model giving the name, type, range of values of each attribute. Furthermore, it is possible to specify the treatment method if the value of the attribute is missing, or if it is an outlier w.r.t. the predicted value for that attribute.

3.5 Feature Summary

Table 1 summarizes the different features of an ideal query language for rule mining and shows how the studied proposals satisfy them as discussed in previous Sections. Notice that the fact that OLE DB DM supports or not some of

Table 1. Summary of the main features of the different languages. [1]Depending on the algorithm. [2]Only association rules. [3]Association rules and elementary sequential patterns. [4]Concept hierarchies. [5]Selectrules, satisfies and violates. [6]Operators for visualization. [7]PREDICTION JOIN. [8]Algorithm parameters

Feature	MSQL	MINE RULE	DMQL	OLE DB DM
Satisfaction of the closure property	Yes	Yes	Yes	Yes[1]
Selection of source data	No	Yes	Yes	Yes
Specification of different types of patterns	No[2]	Some[3]	Yes	Not directly[1]
Specification of the Background Knowledge	No	No	Some[4]	No
Post-processing of the generated results	Yes[5]	No	Some[6]	Some[7]
Specification of constraints	Yes	Yes	Yes	No[8]

the features reported in Table 1 depends strictly by the data mining algorithm referenced in the data mining model. Instead, `OLE DB DM` guarantees naturally the selection of source data, since this feature is its main purpose.

When considering different languages, it is important to identify precisely the kind of descriptive rules that are extracted. All the languages can extract intra-tuple association rules, i.e. rules that associate values of attributes of a tuple. The obtained association rules describe the common properties of (a sufficient number of) tuples of the relation. Instead, only `DMQL` and `MINE RULE` can extract inter-tuple association rules, i.e. rules that associate the values of attributes of different tuples and therefore describe the properties of a set of tuples. Nested tables in the data mining model of `OLE DB DM` could ease the extraction of inter-tuple association rules by the data mining algorithm. Indeed, nested tables include in an unique row of the model the features of different tuples of the source, original tables. Thus, intra-tuple association rules seem to constitute the common "core" of the expressive capabilities of the three languages.

The language capability of dealing with inter-tuple rules affects the representation of the input for the mining engine. As already said, `MSQL` considers only intra-tuple association rules. As illustrated in the next section, this limit may be overcome by a change of representation of the input relation, i.e., by inclusion of the relevant attributes of different tuples in a unique tuple of a new relation. However, this can be a tedious and long pre-processing work. Furthermore in these cases, the `MSQL` statements that catch the same semantics of the analogous statements in `DMQL` and `MINE RULE`, can be very complex and difficult to understand.

As a last example of the different capabilities of the languages, we can mention that while `DMQL` and `MINE RULE` effectively use aggregate functions (resp. on rule elements and on clusters) for the extraction of association rules, `MSQL` provides them only as a post-processing tool over the results.

4 Comparative Examples

We describe here a complete KDD process centered around the classical basket analysis problem that will serve as a running example throughout the paper.

We are considering information of relations *Sales*, *Transactions* and *Customers* shown in Figure 1. In relation *Sales* we have stored information on sold items in the purchase transactions; in relation *Transactions* we identify the customers that have purchased in the transactions and record the method of payment; in relation *Customers* we collect information on the customers.

From the information of these tables we want to look for association rules between bought items and customer's age for payments with credit cards. The discovered association rules are meant to predict the age of customers according to their purchase habits. This data mining step requires at first some manipulations as a preprocessing step (selection of the items bought by credit card and encoding of the `age` attribute) in order to prepare data for the successive pattern extraction; then the actual pattern extraction step may take place.

transaction_id	item
1	ski_pants
1	hiking_boots
2	col_shirts
2	brown_boots
3	col_shirts
3	brown_boots
4	jackets
5	col_shirts
5	jackets
6	hiking_boots
6	brown_boots
7	ski_pants
7	hiking_boots
7	brown_boots
8	ski_pants
8	hiking_boots
8	brown_boots
8	jackets
9	hiking_boots
10	ski_pants
11	ski_pants
11	brown_boots
11	jackets

transaction_id	customer	payment
1	c1	credit_card
2	c2	credit_card
3	c3	cash
4	c4	credit_card
5	c5	credit_card
6	c6	cash
7	c7	credit_card
8	c8	credit_card
9	c9	credit_card
10	c3	credit_card
11	c2	cash

customer_id	customer_age	job
c1	26	employee
c2	35	manager
c3	48	manager
c4	39	engineer
c5	46	teacher
c6	25	student
c7	29	employee
c8	24	student
c9	28	employee

Fig. 1. `Sales` table (on the left); `Transactions` table (on the right above); `Customers` table (on the right below)

Suppose that by inspecting the result of a previous data mining extraction step, we are now interested in investigating the purchases that violate certain extracted patterns. In particular, we are interested in obtaining association rules between sets of bought items in the purchase transactions that violate the rules with `'ski_pants'` in their antecedent. To this aim, we can cross-over between extracted rules and original data, selecting tuples of the source table that violate the interesting rules, and perform a second mining step, based on the results of the previous mining step: from the selected set of tuples, we extract the association rules between two sets of items with a high confidence threshold. Finally, we allow two post-processing operations over the extracted association rules: selection of rules with 2 items in the body and selection of rules with a maximal body among the rules with the same consequent.

4.1 `MSQL`

The first thing to do is to represent source data in a suitable format for `MSQL`. Indeed, `MSQL` expects to receive a unique relation obtained by joining the source relations *Sales*, *Transactions* and *Customers* on attributes *transaction_id* and *customer_id*. Furthermore, the obtained relation must be encoded in a binary format such that each tuple represents a transaction with as many boolean

Table 2. `Boolean_Sales` transactional table used with `MSQL`

t_id	ski_pants	hiking_boots	col_shirts	brown_boots	jackets	customer_age	payment
t1	1	1	0	0	0	26	credit_card
t2	0	0	1	1	0	35	credit_card
t3	0	0	1	1	0	48	cash
t4	0	0	0	0	1	39	credit_card
t5	0	0	1	0	1	46	credit_card
t6	0	1	0	1	0	25	cash
t7	1	1	0	1	0	29	credit_card
t8	1	1	0	1	1	24	credit_card
t9	0	1	0	0	0	28	credit_card
t10	1	0	0	0	0	41	credit_card
t11	1	0	0	1	1	36	cash

attributes as are the possible items that a customer can purchase. We obtain the relation in Table 2.

This data trasformation puts in evidence the main weakness of `MSQL`. `MSQL` is designed to discover the propositional rules satisfied by the values of the attributes inside a tuple of a table. If the number of possible items on which a propositional rule must be generated is very large (as, for instance the number of different products in markets stores) the obtained input table is very large, not easily maintainable and user-readable because it contains for each transaction all the possible items even if they have not been bought. Boolean table is an important fact to take into consideration because its presence is necessary for `MSQL` language, otherwise it cannot work (and so this language is not very much flexible in its input); furthermore, boolean table requires a data transformation which is expensive (especially considering that the volume of tables is huge) and must be performed each time a new problem/source table is submitted.

Pre-processing Step 1: Selection of the Subset of Data to be Mined. We are interested only in clients paying with a credit card. `MSQL` requires that we make a selection of the subset of data to be mined, before the extraction task. The relation on which we will work is supposed to have been correctly selected from the pre-existing set of data in Table 2, by means of a view, named $View\_on\_Sales$.

Pre-processing Step 2: Encoding Age. `MSQL` provides methods to declare encodings on some attributes. It is important to note that `MSQL` is able to do discretization “on the fly”, so that the intermediate encoded value will not appear in the final results. The following query will encode the `age` attribute:

```
CREATE ENCODING e_age ON View_on_Sales.customer_age AS
BEGIN
   (MIN, 9, 0), (10, 19, 1), (20, 29, 2), (30, 39, 3),     (1)
   (40, 49, 4), (50, 59, 5), (60, 69, 6), (70, MAX,7), 0
END;
```

The relation obtained after the two pre-processing steps is shown in Table 3.

Table 3. View_on_Sales transactional table after the pre-processing phase

t_id	ski_pants	hiking_boots	col_shirts	brown_boots	jackets	e_age	payment
t1	1	1	0	0	0	2	credit_card
t2	0	0	1	1	0	3	credit_card
t4	0	0	0	0	1	3	credit_card
t5	0	0	1	0	1	4	credit_card
t7	1	1	0	1	0	2	credit_card
t8	1	1	0	1	1	2	credit_card
t9	0	1	0	0	0	2	credit_card
t10	1	0	0	0	0	4	credit_card

Rules Extraction over a Set of Items and Customers' Age. We want to extract rules associating a set of items to the customer's age and having a support over 2 and a confidence over (or equal to) 50%.

```
GETRULES(View_on_Sales) INTO SalesRB
WHERE BODY has {(ski_pants=1) OR (hiking_boots=1) OR      (2)
  (col_shirts=1) OR (brown_boots=1) OR (jackets=1)} AND
  Consequent is {(Age = *)} AND support>2 AND confidence>=0.5
USING e_age FOR customer_age
```

This example puts in evidence a limit of MSQL: if the number of items is high, the number of predicates in the WHERE clause increases correspondingly! The resulting rules are shown in Table 4.

Table 4. Table SalesRB produced by MSQL in the first rule extraction phase

Body	Consequent	Support	Confidence
(ski_pants=1)	(customer_age=[20,29])	3	75%
(hiking_boots=1)	(customer_age=[20,29])	4	100%
(brown_boots=1)	(customer_age=[20,29])	3	66%
(ski_pants=1) $\wedge$ (hinking_boots=1)	(customer_age=[20,29])	3	100%

Crossing-over: Looking for Exceptions in the Original Data. We select tuples from *View_on_Sales* that violate all the extracted rules with ski_pants in the antecedent (the first and last rule in Table 4).

```
INSERT INTO Sales2 AS
SELECT * FROM View_on_Sales
WHERE VIOLATES ALL (                                       (3)
      SELECTRULES(SalesRB) WHERE BODY HAS {(ski_pants=1)})
```

We obtain results given in Table 5.

Rules Extraction over Two Sets of Items. MSQL does not support a conjunction of an arbitrary number of descriptors in the consequent. Therefore, in this step we can extract only association rules between one set of items in the antecedent and a single item in the consequent. The resulting rule set is only $(brown\_boots = 1) \Rightarrow (color\_shirts = 1)$ with support=1 and confidence=100%.

Table 5. Tuples (in Sales2) violating all rules (in SalesRB) with ski_pants in the antecedent

t_id	ski_pants	hiking_boots	col_shirts	brown_boots	jackets	e_age
t2	0	0	1	1	0	3
t4	0	0	0	0	1	3
t5	0	0	1	0	1	4
t9	0	1	0	0	0	2
t10	1	0	0	0	0	4

```
GETRULES(Sales2) INTO SalesRB2
WHERE (Body has {(hiking_boots=1) OR (col_shirts=1)
 OR (brown_boots=1)}
              AND Consequent is {(jackets=1)}
 OR Body has {(col_shirts=1) OR (brown_boots=1) OR (jackets=1)}
              AND Consequent is {(hiking_boots=1)}
 OR Body has {(brown_boots=1) OR (jackets=1)               (4)
   OR (hiking_boots=1)}
              AND Consequent is {(col_shirts=1)}
 OR Body has {(jackets=1) OR (hiking_boots=1)
  OR (col_shirts=1)}
              AND Consequent is {(brown_boots=1)})
 AND support>=0.0 AND confidence>=0.9
USING e_age FOR customer_age
```

Notice that in this statement the `WHERE` clause allows several different conditions on the `Body` and on the `Consequent`, because we wanted to allow in the `Body` a proposition on every possible attribute except one that is allowed to appear in the `Consequent`. Writing this statement was possible because the total number of items is small in this toy example but would be impossible for a real example in which the number of propositions in the `WHERE` clause explodes.

Post-processing Step 1: Manipulation of Rules. Select the rules with 2 items in the body.

As `MSQL` extracts rules with one item in the consequent and it provides the primitive `length` applied to the itemsets originating rules, we specify that the total length of the rules is 3.

```
SelectRules(SalesRB) where length=3                          (5)
```

The only rule satisfying this condition is:
$(ski\_pants = 1) \wedge (hiking\_boots = 1) \Rightarrow (customer\_age = [20; 29])$

Post-processing Step 2: Extraction of Rules with a Maximal Body. It is equivalent to require that there is no pair of rules with the same consequent, such that the body of the first rule is included in the body of the second one.

```
SELECTRULES(SalesRB) AS R1
WHERE NOT EXISTS (SELECTRULES(SalesRB) AS R2
                  WHERE R2.body has R1.body              (6)
                    AND NOT (R2.body is R1.body)
                    AND R2.consequent is R1.consequent )
```

There are two rules satisfying this condition:
$(ski\_pants = 1) \wedge (hiking\_boots = 1) \Rightarrow (customer\_age = [20; 29])$
$(brown\_boots = 1) \Rightarrow (customer\_age = [30, 39])$

Pros and Cons of MSQL. Clearly, the main advantage of MSQL is that it is possible to query rules as well as data, by using **SelectRules** on rulebases and **GetRules** on data. Another good point is that MSQL has been designed to be an extension of classical SQL, making the language quite easy to understand. For example, it is quite simple to test rules against a dataset and to make crossing-over between the original data and query results, by using SATISFIES and VIOLATES. To be considered as a good candidate language for inductive databases, it is clear that MSQL, which is essentially built around the extraction phase, should be extended, particularly with a better handling of pre- and post-processing steps. For instance, even if it provides some pre-processing operators like ENCODE for discretization of quantitative attributes, it does not provide any support for complex pre-processing operations, like sampling. Moreover, tuples on which the extraction task must be performed are supposed to have been selected in advance. Concerning the extraction phase, the user can specify some constraints on rules to be extracted (e.g., inclusion of an item in the body or in the head, rule's length, mutually exclusive items, etc) and the support and confidence thresholds. It would be useful however to have the possibility to specify more complex constraints and interest measures, for instance user defined ones.

4.2 MINE RULE

MINE RULE does not require a specific format for the input table. Therefore we can suppose to receive data either in the set of normalized relations *Sales*, *Transactions* and *Customers* of Figure 1 or in a view obtained joining them. This view is named *SalesView* and is shown in Table 6 and we assume it is the input of the mining task. Using a view is not necessary but it allows to make SQL querying easier by gathering all the necessary information in one table eventhough all these data are initially scattered in different tables. Thus, the user can focus the query writing on the constraints useful for its mining task.

Pre-processing Step 1: Selection of the Subset of Data to be Mined. In contrast to MSQL, MINE RULE does not require to apply some pre-defined view on the original data. As it is designed as an extension to SQL, it perfectly nests SQL, and thus, it is possible to select the relevant subset of data to be mined by specifying it in the FROM.. WHERE.. clauses of the query.

Table 6. `SalesView` view obtained joining the input relations

transaction_id	item	customer_age	payment
1	ski_pants	26	credit_card
1	hiking_boots	26	credit_card
2	col_shirts	35	credit_card
2	brown_boots	35	credit_card
3	col_shirts	48	cash
3	brown_boots	48	cash
4	jackets	39	credit_card
5	col_shirts	46	credit_card
5	jackets	46	credit_card
6	hiking_boots	25	cash
6	brown_boots	25	cash
7	ski_pants	29	credit_card
7	hiking_boots	29	credit_card
7	brown_boots	29	credit_card
8	ski_pants	24	credit_card
8	hiking_boots	24	credit_card
8	brown_boots	24	credit_card
8	jackets	24	credit_card
9	hiking_boots	28	credit_card
10	ski_pants	48	credit_card
11	ski_pants	35	cash
11	brown_boots	35	cash
11	jackets	35	cash

Pre-processing Step 2: Encoding Age. Since `MINE RULE` does not have an encoding operator for performing pre-processing tasks, we must discretize the interval values.

Rules Extraction over a Set of Items and Customers' Age. In `MINE RULE`, we specify that we are looking for rules associating one or more items (rule's body) and customer's age (rule's head):

```
MINE RULE SalesRB AS
SELECT DISTINCT 1..n item AS BODY, 1..1 customer_age AS HEAD,
       SUPPORT, CONFIDENCE
FROM SalesView WHERE payment='credit_card'                (7)
GROUP BY t_id
EXTRACTING RULES WITH SUPPORT: 0.25, CONFIDENCE: 0.5
```

If we want to store results in a database supporting the relational model, extracted rules are stored into the table $SalesRB(r\_id, b\_id, h\_id, sup, conf)$ where r_id, b_id, h_id are respectively the identifiers assigned to rules, body itemsets and head itemsets. The body and head itemsets are stored respectively in tables $SalesRB\_B(b\_id, item)$ and $SalesRB\_H(h\_id, customer\_age)$. Tables $SalesRB$, $SalesRB\_B$ and $SalesRB\_H$ are shown in Figure 2.

Rule_id	Body_id	Head_id	Support	Confidence
1	1	5	37.5%	75%
2	2	5	50%	100%
3	3	5	37.5%	66%
4	4	5	37.5%	100%

Body_id	item
1	ski_pants
2	hiking_boots
3	brown_boots
4	ski_pants
4	hinking_boots

Head_id	customer_age
5	[20,29]

Fig. 2. Normalized tables containing rules produced by MINE RULE in the first rule extraction phase

Crossing-over: Looking for Exceptions in the Original Data. We want to find transactions of the original relation whose tuples violate all rules with `ski_pants` in the body. As rule components (bodies and heads) are stored in relational tables, we use an SQL query to manipulate itemsets. The corresponding query is the following:

```
SELECT * FROM SalesView AS S1 WHERE NOT EXISTS
  (SELECT * FROM SalesRB AS R1,
                SalesRB_B AS R1_B, SalesRB_H AS R1_H
   WHERE R1.b_id=R1_B.b_id AND R1.h_id=R1_H.h_id AND
   S1.customer_age=R1_H.customer_age AND S1.item=R1_B.item (8)
   AND EXISTS(SELECT * FROM SalesRB_B AS R2_B
       WHERE R2_B.b_id=R1_B.b_id AND R2_B.item='ski_pants')
   AND NOT EXISTS
         (SELECT * FROM SalesRB_B AS R3_B
          WHERE R1_B.b_id=R3_B.b_id AND NOT EXISTS
            (SELECT * FROM SalesView AS S2
             WHERE S2.t_id=S1.t_id AND S2.item=R3_B.item)))
```

This query is hard to write and to understand. It aims at selecting tuples of the original *SalesView* relation, renamed S1, such that there are no rules with ski_pants in the antecedent, that hold on them. These properties are verified by the first two nested SELECT clauses. Furthermore, we want to be sure that the above rules are satisfied by tuples belonging to the same transaction of the original tuple in S1. In other words, that there are no elements of the body of the rule that are not satisfied by tuples of the same original transaction. Therefore, we verify that each body element in the rule is satisfied by a tuple of the *SalesView* relation (renamed S2) in the same transaction of the tuple in S1 we are considering for the output.

Rules Extraction over Two Sets of Items. This is the classical example of extraction of association rules, formed by two sets of items. Using MINE RULE it is specified as follows:

```
MINE RULE SalesRB2 AS
SELECT DISTINCT 1..n item AS BODY, 1..n item AS HEAD,
       SUPPORT, CONFIDENCE                                (9)
FROM Sales2
GROUP BY t_id
EXTRACTING RULES WITH SUPPORT: 0.0, CONFIDENCE: 0.9
```

In this simple toy database the result coincides with the one generated by MSQL.

Post-processing Step 1: Manipulation of Rules. Once again, as itemsets corresponding to rule's components are stored in tables (*SalesRB_B*, *SalesRB_H*), we can select rules having two items in the body with a simple SQL query.

```
SELECT * FROM SalesRB AS R1 WHERE 2=                     (10)
  (SELECT COUNT(*) FROM SalesRB_B R2 WHERE R1.b_id=R2.b_id)
```

Post-processing Step 2: Selection of Rules with a Maximal Body. We select rules with a maximal body for a given consequent. As rules' components are stored in relational tables, we use again a SQL query to perform such a task.

```
SELECT * FROM SalesRB AS R1    # We select the rules in R1
WHERE NOT EXISTS               # such that there are no
 (SELECT * FROM SalesRB AS R2  # other rules (in R2) with
  WHERE R2.h_id=R1.h_id        # the same head, a different
    AND NOT R2.b_id=R1.b_id    # body such that it has no
    AND NOT EXISTS (SELECT *   # items that do not occur in
      FROM SalesRB_B AS B1     # the body of the R1 rule
      WHERE R1.b_id=B1.b_id AND NOT EXISTS (SELECT *
           FROM SalesRB_B AS B2                          (11)
           WHERE B2.b_id=R2.b_id AND B2.item=B1.item)))
```

This rather complex query aims at selecting rules such that there are no rules with the same consequent and a body that strictly includes the body of the former rule. The two inner sub-queries are used to check that rule body in R1 is a superset of the rule body in R2. These post-processing queries probably could be simpler if SQL-3 standard for the ouput of the rules were adopted.

Pros and Cons of MINE RULE. The first advantage of MINE RULE is that it has been designed as an extension to SQL. Moreover, as it perfectly nests SQL, it is possible to use classical statements to pre-process the data, and, for instance, select the subset of data to be mined. Like MSQL, data pre-processing is limited to operations that can be expressed in SQL: it is not possible to sample data before extraction, and the discretization must be done by the user. Notice however, that, by using the CLUSTER BY keyword, we can specify on which subgroups of a group association rules must be found. Like MSQL, MINE RULE allows the user to specify some constraints on rules to be extracted (on items belonging to head or body, their cardinality as well as more complex constraints based on the use of a taxonomy). The interested reader is invited to read [12,13] to have an illustration of these latter capabilities. Like `MSQL`, `MINE RULE` is essentially designed around the extraction step, and it does not provide much support for the other KDD steps (e.g., post-processing tasks must be done with SQL queries). Finally, according to our knowledge, `MINE RULE` is one of the few languages that have a well defined semantics [13] for all its operations. Indeed, it is clear that a clean theoretical background is a key issue to allow the generation of efficient optimizers.

4.3 `DMQL`

`DMQL` can work with traditional databases, so it can receive as input either the source relations *Sales*, *Transactions* and *Customers* shown in Figure 1 or the view obtained by joining them and shown in Table 6. As already done with the examples on `MINE RULE`, let us consider that the view *SalesView* is given as input, so that the reader's attention is more focused on the constraints that are strictly necessary for the mining task.

Pre-processing Step 1: Selection of the Subset of Data to be Mined. Like `MINE RULE`, `DMQL` nests SQL for relational manipulations. So the selection of the relevant subset of data (i.e. clients buying products with their credit card) will be done via the use of the `WHERE` clause of the extraction query.

Pre-processing Step 2: Encoding Age. `DMQL` does not provide primitives to encode data like `MSQL`. However, it allows us to define a hierarchy to specify ranges of values for customer's age, as follows:

```
define hierarchy age_hierarchy for customer_age on SalesView as
level1:{min...9}$<$level0:all
level1:{10...19}$<$level0:all                             (12)
      ...
level1:{60...69}$<$level0:all
level1:{70...max}$<$level0:all
```

Rules Extraction over a Set of Items and Customers' Age. `DMQL` allows the user to specify templates of rules to be discovered, called *metapatterns*, by using the **matching** keyword. These metapatterns can be used to impose strong syntactic constraints on rules to be discovered. So we can specify that we are looking for rule bodies relative to bought items and rule heads relative to customer's age. Moreover, we can specify that we desire to use the predefined hierarchy for the age attribute.

use database Sales_db
use hierarchy age_hierarchy **for** customer_age
mine association as SalesRB
matching with $item^{+}(X, \{I\}) \Rightarrow customer\_age(X, A)$ (13)
from SalesView
where payment='credit_card'
group by t_id
with support threshold=25%
with confidence threshold=50%

where the above metarule with the notation $\{I\}$ matches with rules with repeated *item* predicate like $item(X, I_1) \wedge item(X, I_2) \cdots item(X, I_j)$ where $\{I_1, I_2, \cdots I_j\} = I$ are different elements of the I set obtained as input by the `WHERE` predicate clause. The result is shown in Table 7.

Table 7. Results produced by `DMQL` in the first rule extraction phase (`SalesRB`)

item$^+$(X,{I})	customer_age(X,A)	Support	Confidence
item(X,ski_pants)	customer_age(X,20...29)	37.5%	75%
item(X,hiking_boots)	customer_age(X,20...29)	50%	100%
item(X,brown_boots)	customer_age(X,20...29)	37.5%	66%
item(X,ski_pants)∧item(X,hiking_boots)	customer_age(X,20...29)	37.5%	100%

Crossing-over: Looking for Exceptions in the Original Data. Like `MINE RULE`, `DMQL` does not provide support for crossing-over patterns and data: it requires SQL queries as already shown with `MINE RULE` (query (8)).

Rules Extraction over Two Sets of Items. This phase is performed by the following `DMQL` statement:

use database Sales_db
mine association as SalesRB2
matching with $item^+(X, \{I\}) \Rightarrow item^+(X, \{J\})$ (14)
from Sales2
group by t_id
with confidence threshold=90%

Post-processing Step 1: Selection of the Rules with Two Items in the Body. Like `MINE RULE`, `DMQL` does not provide support for operations of rules manipulation. As we do not have direct access the rules and thus do not the exact storage format of rules, we make the assumption the rules are stored in the same way than in `MINE RULE`, and that allows us to compare the languages in the same conditions of storage format. So, for this step, an SQL query similar to query (10) shown in the examples of `MINE RULE` is therefore needed.

Post-processing Step 2: Selection of the Rules with a Maximal Body. Like `MINE RULE`, `DMQL` does not provide support for operations of rules manipulation such as the selection of the most general rules. For the same reason as the previous post-processing step, an SQL query analogous to query (11) is therefore required.

Pros and Cons of `DMQL`. Like `MINE RULE`, one of the main advantages of `DQML` is that it completely nests classical SQL, and so it is quite easy for a new user to learn and use the language. Moreover, `DMQL` is designed to work with traditional databases and datacubes. Concerning the extraction step, `DMQL` allows to impose strong syntactic contraints on patterns to be extracted, by means of metapatterns allowing the user to specify the form of extracted rules. Another advantage of `DMQL` is that we can include some background knowledge in the process, by defining hierarchies on items occurring in the database and mining rules across different levels of hierarchies. Once rules are extracted, we can perform roll-up and drill-down manipulations on extracted rules. Clearly, analogously to the other languages studied so far, the main drawback of DMQL is that the language capabilities are essentially centered around the extraction phase, and the language relies on SQL or additional tools to perform pre- and post-processing operations. Finally, we can notice that, beyond association rules, DMQL can perform other mining tasks, such as classification.

4.4 OLE DB DM

OLE DB DM is designed for a simple use of relational data already available via OLE DB. Thus, it can work with relational data. Creating a view is not necessary because putting the data in the right format is exactly one of the purposes of the definition and population of the mining model.

Pre-processing Step 1: Selection of the Subset of Data to be Mined. In the OLE DB DM framework, selection of data to be mined is done in the definition of the data mining model and in the following insertion of data in it. Conceptually, it is very similar to the creation of a view. Here the mining model is named [SalesRB] in analogy to the previous examples for the other languages.

Creation of the mining model:

```
CREATE MINING MODEL [SalesRB](
[transaction_id] LONG KEY,
[customer_age]   LONG DISCRETIZED PREDICT,
[items]          TABLE (
   [item] TEXT KEY
   )
)
USING [My_assoc_Algo] (min_support=2, min_confidence=0.5)
```

Notice that we used a nested table [items] to specify bought items by a customer in a transaction and make reference to a mining algorithm, My_assoc_Algo, for the extraction of association rules.

Insertion of data in the data mining model:

```
INSERT INTO [SalesRB]
([transaction_id],[customer_age],[items])
 SHAPE
 {SELECT [transaction_id],[customer_age]
    FROM Transactions,Customers
    WHERE Transactions.customer=Customers.customer_id
           AND Transactions.payment="credit_card"
  APPEND(
  {SELECT [item] FROM Sales
   ORDER BY [tr_id]}
   RELATE [transaction_id] TO [tr_id])
   AS [items]}
```

Notice that selection of the interesting source data (purchases made by credit card) is done in this step. Notice also that `APPEND` keyword builds the nested table `[items]` containing items in source relation `Sales` purchased in a transaction. The relationship between the transaction identifier in `Sales` and the analogous identifier in the model is done by means of the `RELATE` keyword.

Pre-processing Step 2: Encoding Age. The definition of the data mining model allows specification of discretized attributes and of discretization method used. However, discretization itself must be provided by the data mining algorithm provider.

Rules Extraction over a Set of Items and Customer's Age. In SQL Server 2000, no algorithm for association rule mining is currently available, but the specification of `OLE DB DM` claims that association rule mining algorithm can be supported. Here, we supposed that the user has implemented an association rule mining algorithm, named `My_assoc_Algo`, which takes as input parameters of minimal support and confidence and refers to the content of the `[items]` nested tables to elaborate association rules.

The results of the association rule mining process could be stored by the algorithm in a relational table and described by the following `PMML` representation.

```
<Item id="1" value="ski_pants" />
<Item id="2" value="hiking_boots" />
<Item id="3" value="brown_boots" />

<Itemset id="1" support="0.5" numberOfItems="1">
  <ItemRef itemRef="1">
</Itemset>
<Itemset id="2" support="0.5" numberOfItems="1">
  <ItemRef itemRef="2">
</Itemset>
<Itemset id="3" support="0.375" numberOfItems="1">
  <ItemRef itemRef="3">
</Itemset>
<Itemset id="4" support="0.375" numberOfItems="2">
     <ItemRef itemRef="1" />
     <ItemRef itemRef="2" />
</Itemset>

<Item id="4" value="[20,29]" />

<Itemset id="5" support="0.5" numberOfItems="1">
     <ItemRef itemRef="4" />
</Itemset>
```

```
<AssociationRule support="0.375" confidence="0.75"
 antecedent="1" consequent="5" />
<AssociationRule support="0.50" confidence="1.0"
 antecedent="2" consequent="5" />
<AssociationRule support="0.375" confidence="0.66"
 antecedent="3" consequent="5" />
<AssociationRule support="0.375" confidence="1.0"
 antecedent="4" consequent="5" />
```

Notice that such a PMML description is very similar to the rules storage structure of MINE RULE.

Crossing-over: Looking for Exceptions in the Original Data. For this task, we must write a query in classical SQL. Since the association rules produced by the algorithm could be stored in the PMML format, which is quite close of the storage format of MINE RULE, we can say that the query will be very similar to the one used with MINE RULE.

Concerning post-processing tasks, or the usage of the rules after their proper extraction, notice that OLE DB DM only provides some facilities for prediction, with PREDICTION JOIN. However, this is not useful here.

Rules Extraction over Two Sets of Items. We want to perform a new mining task here, so we must define a new mining model. This one is analogous to the model used in previous step with the exception of customers' age that is not needed in this case. Indeed, the difference of this mining task with respect to previous one lies in the proper execution of the mining algorithm that associates an itemset to another itemset and not to the customers' age. For sake of space we do not report this new model here.

Post-processing Step 1: Manipulation of Rules. Here again, we need to access rules' components. Since the OLE DB DM suggests that bodies and heads of rules are stored following the PMML format, the query will be very similar to the one used with MINE RULE.

Post-processing Step 2: Selection of Rules with a Maximal Body. Again, since the rules could be stored following the PMML format, we can use the same kind of queries used for MINE RULE.

Pros and Cons of OLE DB for DM. The first advantage of OLE DB DM is that it is a first temptative of industrial standard and that it begins to be implemented in some commercial application (like SQL Server 2000). It is designed as an extension to SQL and so a DBA can write queries that are similar to classical SQL queries and that define and populate data mining models. But, the main problem is that the language of OLE DB DM is not really a language for Data Mining like the other three. It is particularly targeted at making the communication between relational databases and data mining algorithms easier. So it can work with a lot of different algorithms, provided that the algorithms are compliant to OLE DB DM mining model. However, it provides no facilities to handle typical constraints of the association rule mining problem, such as constraints on items, frequency and confidence. More generally, all these types of constraints must be given as parameters to the mining algorithm. Moreover, accessing the mining results and browsing of extracted patterns must be managed by the algorithm provider, which makes a general method for post-processing difficult to define. Finally, there is no formal semantics like in `MINE RULE`.

5 Conclusions

We have considered three languages, `MSQL`, `MINE RULE` and `DMQL` and an `API` for data mining, `OLE DB DM`, with an SQL-like language for the deployment of a data mining model. All of them request the extraction from a relational database of data mining patterns, and in particular of association rules. They satisfy the "closure property", a crucial property for inductive databases. We have compared the various features of these languages with the desired properties of an ideal query language for inductive databases dedicated to association rules. We have prepared a benchmark and tested the languages against it. The benchmark is constituted by an hypothetical KDD scenario, taken from the data mining practice, in which we have formulated a collection of queries. We have tested the possibility and the ease for the user to express the chosen queries in the above mentioned languages. The outcome is that no language presents all the desired properties. `MSQL` seems the one that offers the larger number of primitives tailored for post-processing and an on-the-fly encoding, specifically designed for efficiency. `DMQL` allows the extraction of different data patterns, the definition and use of hierarchies, and some visualization primitives. `MINE RULE` is the only one that allows to dynamically partition the source relation into a first and a second level grouping (the clusters) from which more sophisticated rule constraints can be applied. Furthermore, to the best of our knowledge, it looks as the only language having an algebraic semantics, an important factor for an in-depth study of optimization issues. `OLE DB DM` is an API, that allows any application to access by means of SQL-like queries to a relational data source, and to be coupled with specialized mining algorithms. The main motivation of the design of `OLE DB DM` is to ease the communication between a data mining application, the DBMS providing data and a set of available, advanced data mining algorithms. However, at the moment, it does not provide any specific feature tailored to any particular data mining task that is not predictive.

However, it is clear that one of the main limits of all the proposed languages is the weak support of rule post-processing. In particular, in all the languages post-processing capabilities are limited to a few predefined built-in primitives. Instead, it would be desirable that the grammar of the languages would accept a certain degree of extensibility. Indeed, for instance, it is not possible to introduce user-defined functions in the statements. These ones would allow the user to provide the implementation of user-defined sophisticated constraints, based, for instance, on new pattern evaluation measures.

Furthermore, the research on condensed representations for frequent itemsets [2,3] has been proved useful not only for mining frequent itemsets and frequent association rules from dense databases but also for sophisticated post-processing [1,15]. Indeed, one of the problems in association rule mining from real-life data is the huge number of extracted rules. However, many of the rules are redundant and might be useless. Thus, a condensed representation would help visualizing the result and focusing the user attention on the relevant rules. For example, Bastide et al., [1], presents an algorithm to extract a minimal cover of the set of frequent association rules.

Another crucial issue relative to query language for data mining is the optimization for sequences of queries (e.g., deciding of query containment). To the best of our knowledge, the materialization of condensed representations of the frequent itemsets seems to be quite useful [9,4] but still needs further work.

Last but not least, an important issue is the simplicity of the language and its ease of use. Indeed, we think that a good candidate language for data mining should be flexible enough to specify a variety of different mining tasks in a declarative fashion. To the best of our knowledge, the implementation of these languages tackles the mentioned problems (including the lack of instruments dedicated to post-processing) by being embedded in a data mining system, which provides a graphical front end to the language.

References

1. Bastide, Y., Pasquier, N., Taouil, R., Stumme, G., Lakhal, L.: Mining minimal non-redundant association rules using frequent closed itemsets. Proc. CL'00 (2000), London (UK). Springer-Verlag LNCS 1861. pp. 972–986.
2. Boulicaut, J-F., Bykowski, A.: Frequent closures as a concise representation for binary data mining. Proc. PAKDD'00 (2000), Kyoto (JP). Springer-Verlag LNAI 1805. pp. 62–73.
3. Boulicaut J-F., Bykowski, A., Rigotti, C.: Free-sets: a condensed representation of boolean data for the approximation of frequency queries. Data Mining and Knowledge Discovery (2003). *7(1)*5–22.
4. Giacometti, A., Laurent, D., Diop, C.T.: Condensed representations for sets of mining queries. Proc. KDID'02 (2002), Helsinki (FIN). An extended version appears in this volume.
5. Imielinski, T., Mannila, H.: A Database Perspective on Knowledge Discovery. Communications of the ACM (1996). *3(4)*58–64.
6. Imielinski, T., Virmani, A., Abdulghani, A.: DataMine: Application Programming Interface and Query Language for Database Mining. Proc. KDD'96 (1996), Portland (USA). AAAI Press. pp. 256–261.
7. Imielinski, T., Virmani, A.: MSQL: A Query Language for Database Mining. Data Mining and Knowledge Discovery (1999). *3(4)*373–408.
8. Jeudy, B., Boulicaut, J-F.: Optimization of association rule mining queries. Intelligent Data Analysis (2002). *6(4)*341–357.
9. Jeudy, B., Boulicaut, J-F.: Using condensed representations for interactive association rule mining. Proc. PKDD'02 (2002), Helsinki (FIN). Springer-Verlag LNAI 2431. pp. 225–236.

10. Han, J., Fu, Y., Wang, W., Koperski, K., Zaiane, O.: DMQL: A Data Mining Query Language for Relational Databases. Proc. of SIGMOD Workshop DMKD'96 (1996), Montreal (Canada). pp. 27–34.
11. Han, J., Kamber, M.: Data Mining – Concepts and Techniques. Morgan Kaufmann Publishers (2001).
12. Meo, R., Psaila, G., Ceri, S.: A New SQL-like Operator for Mining Association Rules. Proc. VLDB'96 (1996), Bombay (India). Morgan Kaufmann. pp. 122–133.
13. Meo, R., Psaila, G., Ceri, S.: An Extension to SQL for Mining Association Rules. Data Mining and Knowledge Discovery (1998). *2(2)*195–224.
14. Virmani, A.: Second Generation Data Mining. PhD Thesis, Rutgers University, 1998.
15. Zaki, M.J.: Generating non-redundant association rules. Proc. SIGKDD'00 (2000), Boston (USA). ACM Press. pp. 34–43.
16. Netz, A., Chaudhuri, S., Fayyad, U., Bernhardt, J.:Integrating Data Mining with SQL Databases: OLE DB for Data Mining. Proc ICDE'01 (2001), Heidelberg (Germany). IEEE Computer Society. pp. 379–387
17. OLEDB for Data Mining specifications, available at http://www.microsoft.com/data/oledb/dm/
18. Predictive Model Mark-up Language, available at http://www.dmg.org/pmmlv2-0.htm