A Comparison between Query Languages for the Extraction of Association Rules

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

[1] Universitá di Torino, Dipartimento di Informatica, corso Svizzera 185, 10149, Torino, Italy
[2] Institut National des Sciences Appliquées de Lyon, 69621 Villeurbanne cedex, France

Abstract. Recently inductive databases (IDBs) have been proposed to afford 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 special-purpose language, powerful enough to perform all the required manipulations, such as data preprocessing, pattern discovery and pattern post-processing. In this paper we present a comparison between query languages (MSQL, DMQL and MINE RULE) that have been proposed for association rules extraction in the last years and discuss their common features and differences. We present them using a set of examples, taken from the real practice of data mining. This allows us to define the language design guidelines, with particular attention to the open issues on IDBs.

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*", suggested in [2]. 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 an ad-hoc, powerful and universal query language that can deal both with raw data or patterns. A few query languages can be considered as candidates for inductive databases. Most of the proposals emphasize one of the different phases of the KDD process. This paper is a critical evaluation of three proposals in the light of the IDBs' requirements: `MSQL` [3,4], `DMQL` [6,7] and `MINE RULE` [8,9].

The paper is organized as follows. Section 2 summarizes the desired properties of a language for mining inside an inductive database. Section 3 introduces

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

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 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` and `MINE RULE` allow the selection of data. None 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).
  All the three languages allow the specification of various kinds of constraints based on rule elements, rule cardinality and aggregate values. They allow the specification of 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 the three languages 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 two languages in its offer of few post-processing primitives (it has a dedicated operator, `SelectRules`). `DMQL` allows some visualization options. However, the three languages are quite poor for rule post-processing.

3 Query Languages for Rule Mining

MSQL has been described in [4,5], and designed at the Rutgers University, New Jersey, USA. The main features of `MSQL`, as stated by the authors, are the following: (1) *Ability to nest SQL expressions* such as sorting and grouping in a `MSQL` statement and allowing nested SQL queries. (2) Satisfaction of the *closure* property and availability of operators to further manipulate results of previous `MSQL` queries. (3) *cross-over between data and rules* with operations allowing to identify subsets of data satisfying or violating a given set of rules. (4) *distinction between rule generation and rule querying.* Indeed, as the volume of generated rules might explode, rules might be extensively generated only at querying time, and not at generation time.

The language comprises five basic statements: `GetRules` that generates rules into a rule base; `SelectRules` that queries the rule base; `Create Encoding` that efficiently encodes discrete values into continuous valued attributes; `satisfies` and `violates` that allow to cross-over data and rules, and that can be used in a data selection statement.

DMQL has been presented in [6,7] and designed at the Simon Fraser University, Canada. The language consists of the specification of four major primitives in data mining that manage: (1) the set of *relevant data* w.r.t. a data mining process; (2) the *kind of knowledge* to be discovered; (3) the *background knowledge*; (4) the *justification of the interestingness* of the knowledge (i.e., thresholds).

(1) This primitive is specified as a relational conventional query.

(2) This primitive may include association rules, classification rules (rules that assign data to disjoint classes according to the value of a chosen classifying attribute), characteristica (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).

(3) This primitive manages a set of concept hierarchies or generalization operators that assist the generalization processes.

(4) 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, i.e., electing the more specific rule.

MINE RULE proposal can be found in [8] and [9]. This operator extracts a set of association rules from the database and stores them back in the database in a separate relation. This language is an extension of SQL. Its main features are the following. (1) *Selection of the relevant set of data* for a data mining process; (2) Definition of the *structure of the rules* to be mined and of constraints applied at different granularity levels; (3) Definition of the *grouping condition* that determines which data of the relation can take part to an association rule; (4) Definition of *rule evaluation measures* (i.e., support and confidence thresholds).

The selection above mentioned as the first feature of `MINE RULE` 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 second feature defines unidimensional association rule (i.e., its elements are the different values of the same dimension or attribute), or multidimensional one (each rule element involves the value of more attributes). Furthermore, rules constraints belong to two categories: the former ones are applied at the rule level (*mining conditions*), while the second ones (*cluster conditions*), are applied at the *body* or *head* level (i.e., the sets of rule elements that compose each rule).

4 Comparative Examples

We describe here a classical basket analysis problem that will serve as a running example troughout the paper: we are considering information of Table 1 and we are looking for association rules between bought items and customer's age for payments with a credit cards. We are considering a complete KDD process. We will consider two manipulations at the pre-processing step (selection of the items bought by credit card and encoding of the `age` attribute), crossing-over between extracted rules and original data (selecting tuples of the source table that violate all the extracted rules of size 3), and two post-processing operations (selection of rules with 2 items in the body and selection of rules having a maximal body).

Table 1. 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	29	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	34	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

4.1 MSQL

Table 1 corresponds to the source data encoded in the input format used by MSQL. There are as many boolean attributes as there are possible items.

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 a pre-existing set of data, by means of a view, named **RSales**.

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

Rules extraction. We want to extract rules associating a set of items to the customer's age and having a support over 25 % and a confidence over 50 %.

```
GETRULES(RSales) INTO RSalesRB
WHERE BODY has {(ski_pants=1) OR (hiking_boots=1) OR (col_shirts=1)
      OR (brown_boots=1) OR (jackets=1)} AND
      Consequent is {(Age = *)} AND support>=0.25 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!

Crossing-over: looking for exceptions in the original data. Finally, we select tuples from **RSales** that violate all the extracted rules of size 3.

```
SELECT * FROM RSales
WHERE VIOLATES ALL (SELECTRULES(RSalesRB) WHERE length=3)
```

Post-processing step 1: manipulation of rules. We select rules with 2 items in the body. As MSQL is designed to extract rules with one item in the head and as it provides access only to the extracted rules (and not to the originating itemsets), we must specify that the total size of the rule is 3.

```
SelectRules(RSalesRB) where length=3
```

Post-processing step 2: extraction of rules with a maximal body. It is equivalent to require that there is no couple of rules with the same consequent, such that the body of one rule is included in the body of the other one.

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

Pros and cons of MSQL. Clearly, the main advantage of MSQL is that it is possible to query knowledge as well as data, by using **SelectRules** on rule-bases and **GetRules** on data (and it is possible to specify if we want rules to be materialized or not). 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 SATISFY 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

We include the same information of Table 1 into a normalized relation Sales over a schema (t_id, customer_age, item, payment).

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 nest SQL, and thus, it is possible to select the relevant subset of data to be mined by specifying it in the WHERE clause of the query.

Pre-processing step 2: encoding age. Since MINE RULE does not have an encoding operator for performing pre-processing tasks, we must encode by ourselves the interval values (2 represents an age in the interval [20-29], 3 represents an age in [30-39], and so on).

Rules extraction. 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 Sales WHERE payment='credit_card'
GROUP BY t_id
EXTRACTING RULES WITH SUPPORT: 0.25, CONFIDENCE: 0.5
```

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 resp. in tables SalesRB_B(b_id,<bodySchema>) and SalesRB_H(h_id,<headSchema>).

Crossing-over: looking for exceptions in the original data. We want to find tuples of the original relation violating all rules with 2 items in the body. As rules' components (bodies and heads) are stored in relational tables, we use an SQL query to manipulate itemsets. The correspondent query is reported here:

```
SELECT * FROM Sales AS S1 WHERE NOT EXISTS
      (SELECT * FROM SalesRB AS R1
       WHERE (SELECT customer_age FROM SalesRB_H
              WHERE h_id=R1.h_id)=S1.customer_age
         AND (SELECT COUNT(*) FROM SalesRB_B
              WHERE R1.b_id=SalesRB_B.b_id)=2
         AND NOT EXISTS (SELECT * FROM SalesRB_B AS I1
                         WHERE I1.b_id=R1.b_id AND NOT EXISTS
                           (SELECT * FROM Sales AS S2
                            WHERE S2.t_id=S1.t_id
                            AND S2.item=I1.item )));
```

This query is hard to write and to understand. It aims at selecting tuples of the original table such that there are no rules of size 3 that hold in it. To check that, we verify that the consequent of the rule occurs in a tuple associated to a transaction and that there are no items of the rule's body that do not occur in the same transaction.

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=
  (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
           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 queries probably could result 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, on their cardinality as well as more complex constraints based on the use of a taxonomy). The interested reader is invited to read [8,9] to have 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 [9] for each of its operations. Indeed, it is clear that a clean theoretical background is a key issue to allow the generation of efficient compilers.

4.3 DMQL

`DMQL` can work with traditional databases, so let's consider that we have encoded information of Table 1 into a **sales_db** database which is made of the relations **Sales**(customer_id, item) and **Customer_info**(customer_id, age, payment).

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` statement 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_info on sales as
level1:min...9<level0:all
level1:10...19<level0:all
...
level1:60...69<level0:all
level1:70...max<level0:all
```

Rules extraction. `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's bodies relative to bought items and rule's 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_info.age
mine association as SalesRB
matching with $sales^{+}(X, \{I\}) \Rightarrow customer\_info(X, A)$
from sales, customer_info
where sales.customer_id=customer_info.customer_id

AND customer_info.payment='credit_card'
with support threshold=25% **with confidence** threshold=50%

Crossing-over and post-processing operations. Like `MINE RULE`, `DMQL` does not provide support for post-processing operations and performings them requires writing SQL queries or using *ad hoc* tools provided externally.

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 many mining operations, like mining characteristic descriptions, discriminant descriptions, or classification rules.

5 Conclusions

In this paper, we have considered various features of three languages, `MSQL`, `DMQL` and `MINE RULE`, that extract association rules from a relational database and support the "closure property", a crucial property for inductive databases. Then, we have compared them with the desired properties of an ideal query language for inductive databases. Next, we have presented a set of queries taken from data mining practice and have discussed the suitability of these languages for querying inductive databases. 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 the most large set of different data patterns and the definition and use of hierarchies and of some visualization primitives during the rule extraction; `MINE RULE` is the only one that allows to dynamically partition the source relation into groups from which the rules will be extracted; a second level of grouping, the clusters, from which more sophisticated rule constraints can be applied, is also possible. Furthermore, at our knowledge, it looks as the only one with an algebraic semantics, what could become an important positive factor when query optimization issues will be addressed.

However, one of the main limits of all the three languages is the insufficient support of post-processing issues. Whatever the language is, the user must use one of the predefined built-in options. This problem becomes crucial when considering user-defined post-processing operations involving something else than rule's component, support and confidence. Instead, in our view, a good candidate language for inductive databases must be flexible enough in its grammar to let the user specify its own constraints and different post-processing operations. Another important issue is the simplicity of the language and its ease of use. Indeed, we think that a good candidate language for data mining must be flexible enough to specify a lot of different mining tasks in a declarative fashion. However the powerful declarative semantics must not affect the simplicity of use. At our knowledge, these languages tackle this problem by being embedded in a data mining tool, which provides a front end to the grammar.

Another crucial issue relative to query languages for data mining is the optimization problem. When designing a language compiler, we must pay attention to optimizations issues; for instance, trying to make profit of previously generated results, or to analyze the equivalence between queries. In these terms, at our knowledge, a lot of work must still be done. Furthermore, it could reveal as very important the ability to manipulate intermediate representation like condensed representations of frequent patterns (see, e.g., [1]). A challenge tightly linked to such a functionality would be to find ways to characterize constraints defined in an operation of rule extraction in terms such that they could eventually be exploited during the mining process.

This study allows us to conclude that the path to reach the maturity in inductive database technology is still far to be reached. However, the limits and the merits of the current query languages to give support to the knowledge discovery process have been already identified.

References

1. Boulicaut J-F., Jeudy B.: Mining free-sets under constraints. Proc. of Database Engineering & Applications Symposium, IDEAS'01, Grenoble, France (2001).
2. Imielinski, T., Mannila, H.: A Database Perspective on Knowledge Discovery. Communications of the ACM. **3:4** (1996) 58–64.
3. Imielinski, T., Virmani, A., Abdulghani, A.: DataMine: Application Programming Interface and Query Language for Database Mining. Proc. of the 2nd Int. Conf. on Knowledge Discovery and Data Mining, KDD'96. 3 (1996) 256–261.
4. Imielinski, T., Virmani, A.: MSQL: A Query Language for Database Mining. Data Mining and Knowledge Discovery. 3 (1999) 373–408.
5. Virmani, A.: Second Generation Data Mining. PhD Thesis, Rutgers Univ. (1998).
6. Han, J., Fu, Y., Wang, W., Koperski, K., Zaiane, O.: DMQL: A Data Mining Query Language for Relational Databases.
7. Han, J., Kamber, M.: Data Mining – Concepts and Techniques. Morgan Kaufmann Publishers (2001).
8. Meo, R., Psaila, G., Ceri, S.: A New SQL-like Operator for Mining Association Rules. Proc. of the 22nd Int. Conf. of Very Large Data Bases. Bombay, India (1996).
9. Meo, R., Psaila, G., Ceri, S.: An Extension to SQL for Mining Association Rules. Data Mining and Knowledge Discovery. **9:4** (1997).