DataMine: Application Programming Interface and Query Language for Database Mining

Tomasz Imielinski and **Aashu Virmani** and **Amin Abdulghani**
{imielins, avirmani,aminabdu}@cs.rutgers.edu
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903
Phone: (908) 445 3551, Fax: (908) 445 0537

Introduction

The main objective of the *DataMine* is to provide application development interface to develop knowledge discovery applications on the top of large databases.

Current database systems have been designed mainly to support business applications. The success of SQL capitalized on a small number of primitives which are sufficient to support a vast majority of applications today. Unfortunately this is not enough to capture the emerging family of new applications dealing with the so called *rule and knowledge discovery*. The goal of the *DataMine* and our work is to make the next step in the development of DBMS and provide much needed support for the rule discovery applications.

A typical knowledge discovery application starts with rule discovery, but rules are not necessarily the end products. For example:

1. Finding the "best" candidates for a marketing promotion package from a large population stored in the database; for example, the best candidates for certain type of insurance may be those who frequent health clubs and are under 40 etc.
2. Finding any strong rules between the age, disease, and residence area, such as "40% of heart disease case in NJ occur in patients older than 50" (rules are statements of the form "if *condition* then *consequent*".)
3. Finding the most distinctive features (as opposed to other states) of NJ heart patients

Finding rules is only the first step in a knowledge discovery application. Typically, a user wants to embed information obtained from the rules in a larger program. For instance, in target marketing applications a company may have a fixed promotion budget and can only offer some limited number of promotions. A promotion mailing application must rank the best candidates for mailing and go "down the list" of most likely candidates until all promotion offerings are taken. To accomplish such a task we need an integrated API for knowledge discovery applications, integrated with the programming language (like C) and with the database query language (such as SQL).

There is no commercial system nor research prototype today which would offer such integrated API for knowledge discovery applications. Today, most systems offer "stand alone" features using tree classifiers, neural nets, and meta-pattern generators. Such systems cannot be embedded into a large application and typically offer just one knowledge discovery feature. The situation today is thus very similar to the situation in DBMS in the early sixties when each application had to build from scratch, without the benefit of dedicated database primitives provided later by SQL and relational database APIs.

The objective of *DataMine* is to fill this gap and bring the database support for knowledge discovery applications *to the same level that exists today for business applications*. What we offer and plan to offer can be summarized as follows:

- Extension of SQL, called M-SQL to generate and selectively retrieve sets of rules from a large database.
- Embedding of M-SQL in the general host language (in a similar way as SQL is embedded in C) to provide API for Knowledge and Data Discovery applications.

Thus, just as SQL does, we are supporting two basic modes: *free form querying* and *embedded querying*. Free form querying allows the user to perform interactive and exploratory data analysis, while embedded querying provides features to run applications which rely on rule discovery, but use rules in some further computations.

Key Features

The key features of the *DataMine* system include:

- Extension of SQL to handle rule mining applications

  The proposed extension uses just one additional primitive - the *MINE* operator.

- Query Optimizer to compile M-SQL queries into efficient execution plans.

- Application Programming Interface (API) with M-SQL embedded into C++ host programming language

Both in the query language as well as in the application programming interface, a *single rule*, and a *set of rules* are the basic objects which are manipulated and queried. Rules are defined as in (Agrawal, Imielinski, & Swami 1993). A rule is an expression of the form:
Body $\Longrightarrow$ Consequent
where Body is a conjunction of descriptors and consequent is a descriptor. A descriptor is a pair (Attribute, Value), and is satisfied by those tuples in the database for which Attribute equals Value. (When the domain of A_i is continuous we also allow ranges of values to also appear as descriptors. Thus, for example, (Age IN $< 30, 40 >$) is a legal descriptor as well). Attributes could either exist in the database originally or could be a user defined method, in which case the evaluation for the method is done at run time.

Additionally, each rule is specified by two parameters: *support* and *confidence*. The **support** of the rule is defined as a number of tuples which satisfy the body of the rule. The **confidence** is defined as the ratio of the number of all tuples which satisfy both the body and the consequent to the number of all tuples which satisfy just the body of the rule.

Eager Evaluation: Since rule queries may possibly take a very long time to execute and in addition, may return very large rule sets, (the rule generation problem is in general of exponential worst case complexity in terms of number of attributes involved) we have implemented a new mode of query evaluation in which the first few rules are generated as soon as possible, and then a certain steady rate of rule production is ensured. This enables the system to keep users interested and let the user cognitively process the returned rules while other rules are generated.

Rule Query Language: The rule query language, M-SQL, (the M stands for Mining) is built as an extension of SQL by including a small set of primitives for data mining. A typical M-SQL query is of the form:

```
Select
    From Mine(C)
    Where
       D-CONDITION
       s1 < Support < s2
       c1 < Confidence < c2
```

C can be a persistent class or a table defined by a "Mine-Free" SQL expression. Mine(C) returns the set of *all* rules as defined above, which are true in C. This set could be very large, but in the presence of constraints and D-Conditions (defined below), an optimized algorithm never actually generates the whole set. Thus, *Mine* is viewed as a logical operator. Let *MUST*, *MAY* and *TARGET* be sets of descriptors. D-CONDITION is defined as a disjunction of atoms of the following form:

(MUST $\subset$ *Body* $\subseteq$ *MAY)*
AND (Consequent IN TARGET)

This allows specification of conditions imposed on the bodies as well as on consequents of target rules. For example, the query: *"Find all rules in Table T involving the Attributes(methods) Disease, Age and ClaimAmt, which have a confidence of at least 50%"* might be expressed as:

```
select *
from Mine(T) R
where
  R.Body < {(Disease=*),(Age=*),(ClaimAmt=*)}
  and {} < R.Body
  and R.Consequent IN
          {(Disease=*),(Age=*),(ClaimAmt=*)}
  and R.Confidence > 0.5
```

The "from" line in the query allows substitution of Mine(T) for R, for notation simplicity. The above query explicitly disallows rules with empty bodies. However, a rule with an empty body does state a fact about the database, for example, a rule like $\emptyset \Longrightarrow$ **(Sex = 'Male') (30%, 239)** merely states that out of the database of 239 objects, 30 percent are males. Queries are compiled and executed using an efficient algorithm for rule generation (in the case when the data mining mode is used). We describe below the two lines of development we have chosen for the system, i.e. Free Form Querying, and Embedded querying, which complement each other and are each useful in their own ways.

Free Form Querying

Free form querying works in two modes: *data mining mode* and *rule mining mode*. First, the user formulates a rule query. The rule query is specified in M-SQL described above and describes the conditions which the rule should satisfy in order to be included

in the answer. The current prototype implements only the graphical "query by example" style version of the language. Then, If a query is processed in the *data mining mode*, it is evaluated on the top of the original database and the requested rules are generated from data in the run time of the query. In the *rule mining mode*, it is assumed that rules have been generated earlier and are stored in a rule base. In this case query behaves like a pattern against which existing rules are matched and retrieved from the rulebase.

Data Mining Mode This mode forms the core of the mining engine, which is responsible for transforming the query into an optimized algorithm for rule generation depending on the search space requested by the user. If the user has some understanding of the data set, he can exercise greater control over the mining process by specifying a user defined discretization function to be applied to continuous attributes (or one of the several builtin choices), and for instance, restricting the rules from containing redundant information by specifying dependencies that exist among the attributes. On the other hand, for a novice user, who is trying to gain an understanding of the data set, the mining engine can run in a highly interactive mode, where a graphical interface is constantly updated with progress information from the system (see "Exploratory Mining Scenario" later in this paper), and the user can query the rulebase as it is being generated, narrow his preferences about what he wants, and prune the overall rule-space the system must generate. Figure 1 shows a screen snapshot of this mode.

Rule Mining Mode The rule mining mode helps to browse through the vast numbers of rules which can be generated from a database. In addition to being just a retriever/browser, this mode contains a builtin applications suite or "Macros" (described in more detail later) which run on top of the rulebase, and can help the user query the rulebase in a very effective manner. It also lets the user shuffle back and forth between the rulebase and database asking for tuples that satisfied or violated a given rule and vice-versa. Figure 2 shows one of the screen shots of this mode.

Embedded Querying: API

We demonstrate a number of pilot knowledge discovery applications which were developed using the proposed API, like classification, typicality and "characteristic of" applications. Using these

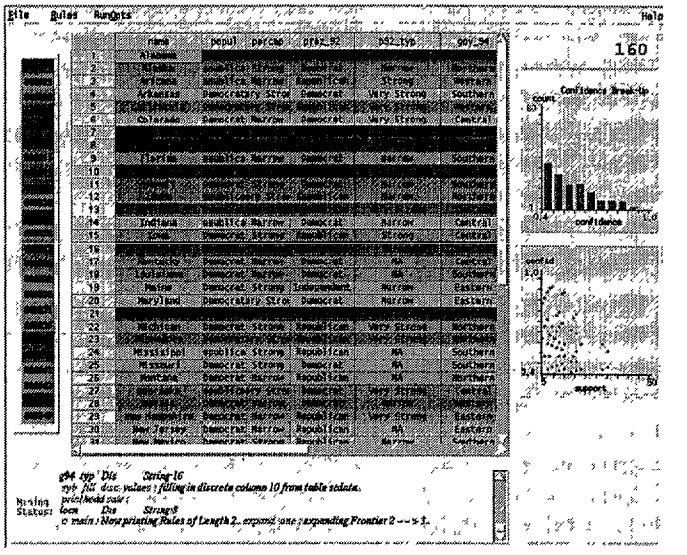

Figure 1: *DataMine system in online-mining mode.*

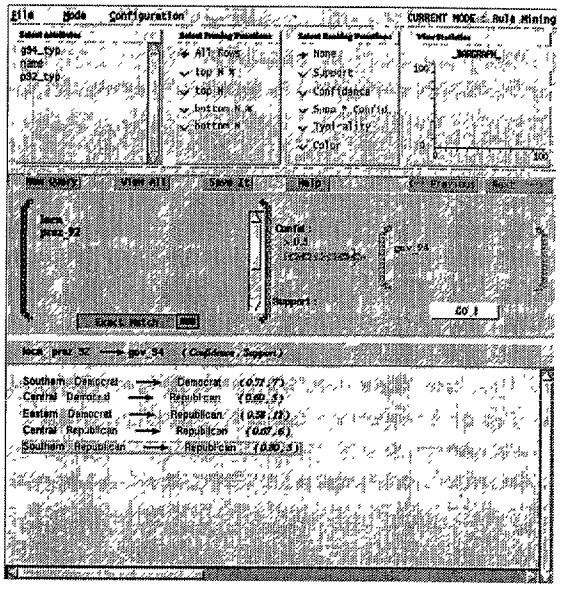

Figure 2: *Rule Manger - query based mining.*

applications one can find the "most typical" heart patients, the most characteristic features of smokers from New Jersey or the best candidates for life insurance in a given population sub-category. One can also "mine around a rule", when a user can look for rules which are similar but stronger than a given one.The key aspect of *DataMine* is that it provides application programming interface to develop new discovery applications.

Our API design mirrors the design of C++ with SQL calls, using standard host language/SQL interface involving cursors. We plan to offer a similar interface between C++ and M-SQL. This requires adding an extra class of *rules* and binding cursor variables which range through rule sets to that rule class. The rule class will be a generic library class with methods directly corresponding to the specific rule attributes. In addition, several visualization primitives are provided which to graphically display the rules, or to display a bar graph based on a certain attribute, or the confidence and support of a set of rules.

Scenarios of Use

In this section we briefly describe what you will see in the Demo of the existing prototype and some of the features which will be added very soon. The first four scenarios are fully supported, the fifth is currently "under construction".

Scenario 1: Mining around a rule

User starts from a specific rule which he wants to check/verify against the data. *DataMine* system additionally offers a possibility of *mining around a rule*, that is not only verifying whether the original rule holds but also *suggesting* other, perhaps stronger rules which are "close" to the original rule which the user wanted to verify.

Scenario 2: Rule Querying

User can query and obtain all rules having certain body, consequent, support, confidence, or a mixture of one of more the above criteria. Once the rules are generated, the user can pick a rule and further mine around it, as in the previous scenario. Alternately, (s)he can obtain all the records in the database which satisfy(violate) the rule, select a subset of these records, look for further rules there etc. This process crosses the border between the rulebase and the database many times and is highly interactive.

Scenario 3: Exploratory Mining

This mode is intended for the user who knows very little about the data, probably not enough to even formulate a query. In this mode, the system looks for all rules between all attributes in the database. This is computationally expensive, but we have implemented this process in a very incremental way to keep the user interested. Such a user will see the database with tuples (records) changing colors, while mining process is taking place. The colors reflect how many rules, discovered so far, a given record in the database satisfies. The "hottest" (red color) tuples satisfy most of the rules and attract user's interest. The user can then click on such a record and see all the rules which such a record satisfies. In addition, a scatter plot of the confidence vs. support of the rules (see fig. 1) is also constantly updated. The user can zoom a section of this plot and see the rules contained in that confidence and support range. These queries can further lead to mining around a rule if some rule looks particularly interesting to the user.

Scenario 4: Typicality and Atypicality

This is an application which we have built using our API as a proof of concept. From the user specified subset of records it selects the most "typical" and the most "atypical" records. Typical records are the records which satisfy most of the rules, atypical records violate most of the rules. Coloring is used again to make a distinction between typical and atypical records (atypical are blue, typical are red). This application helps in identifying database irregularities and can be useful, for example, in fraud detection.

Scenario 5: Application Development

User constructs a new application on the top of the set of rules generated by a rule query. The graphical API helps the user to write a C program which uses rules as first class objects.

System Platform

The *DataMine* prototype has been built keeping in mind the above features. It currently generates propositional rules matching a specified set of patterns (The patterns could be wildcard, in which case it generates all rules). The mining engine has been implemented in C (about 15,000 lines of code), and the front and back end GUI has been developed using Tcl-Tk (about 20,000 lines of code). It can read data from Ascii files or from a database directly, without any need for preprocessing by the user. The database system used currently is Sybase 10.2, but the design ensures a "loose-coupling" between the database and the engine, so that plugging in modules for other database systems is very straightforward. The systems comes with a smart browser which can let a user query rules by example, perform mining "around" a first guess, and a pilot suite of applications which treat rules as first class objects, and we believe should be part of core M-SQL.

Currently, the system runs on Solaris 2.5 on an UltraSparc 170E, but in the near future, we hope to run a parallelized version of this algorithm on a cluster of 6 UltraSparcs connected via a gigabit switch.

Performance

The DataMine engine can be run in either a "Memory Saver Mode" when it uses more economical data structures (at the cost of time) which conserve memory (less swapping), or a "Performance Mode" where it uses slightly more memory intensive data structures, but performs much better. The idea is that if the data is relatively small (see fig 3 below), then it makes sense to make full use of memory available. The graph below was obtained by mining different fractions of a 1.2 million record database containing seven attributes in both modes, and comparing the performance. The steep rise in the time taken by performance mode (from 50% to 100% of data size) is explained by the fact that extensive swapping began at that point.

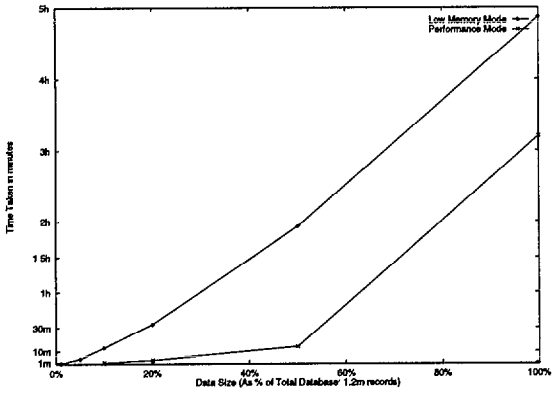

Figure 3: *Performance comparison in two modes of operation.*

Comparison with Relevant Systems

Systems which are compatible in scope include the DBMiner system (Han *et al.* 1996b) which is built on top of DMQL (Han *et al.* 1996a), another query language for rule generation.

However, our system is somewhat different in language design by providing a specific language primitive "Mine" as an extension to SQL. Besides, a query expressed in our language can work in rule generation (data-mining) mode as well as rule retrieval (rule-mining) mode.

In addition, this is, to our knowledge the first prototype of a system which provides an API to facilitate building complex data mining applications.

Conclusions

We believe that a "Query Based" approach is particularly suited for the data mining with human in the loop, where the human actively participates in the data mining process by changing and modifying the data mining requests. It makes an effective tool to provide users with precise specification of their mining interests, and also helps to deal with potentially very large results of data mining.

There is a wide variety of applications which are using knowledge discovery techniques today. Typically, each application is handled by a specialized "stand alone" system which is developed from "scratch". The objective of our API design is to provide a platform which will make the knowledge discovery application development faster and easier. SQL and relational APIs increased programmer's productivity for business applications, DataMine will offer similar advantages for knowledge discovery application developers.

References

Agrawal, R., and Srikant, R. 1994. Fast algorithms for mining association rules. In *VLDB-94.*

Agrawal, R.; Ghosh, S.; Imielinski, T.; Iyer, B.; and Swami, A. 1992. An interval classifier for database mining applications. In *VLDB-92*, 560–573.

Agrawal, R.; Imielinski, T.; and Swami, A. 1993. Mining associations rules between sets of items in large databases. In *SIGMOD-93.*

Han, J., and Fu, Y. 1995. Discovery of multiple level association rules from large databases. In *VLDB-95*, 420–431.

Han, J.; Fu, Y.; Koperski, K.; Wang, W.; and Zaiane, O. 1996a. DMQL: A data mining query language for relational databases. In *DMKD-96 (SIGMOD-96 Workshop on KDD).*

Han, J.; Fu, Y.; Wang, W.; Chiang, J.; Gong, W.; Koperski, K.; and Li, D. 1996b. DBMiner: A system for mining knowledge in large relational databases. to appear in KDD-96.

Imielinski, T., and Hirsh, H. 1993. Query based approach to knowledge discovery. Technical report, Rutgers University, NJ.

Kero, B.; Russell, L.; and Tsur, S. 1995. An overview of database mining techniques. Technical report, Argonne National Laboratory, Argonne, Illinois.

Mannila, H.; Toivonen, H.; and Verkamo, A. I. 1994. Efficient algorithms for discovering association rules. In *KDD-94.*

Piatetsky-Shapiro, G., and Frawley, W. J. 1991. *Knowledge Discovery in Databases.* AAAI/MIT Press.

Quinlan, J. R. 1986. Induction of decision trees. *Machine Learning* 1(1):81–106.

R.Agrawal; Imielinski, T.; and Swami, A. 1993. Database mining: A performance perspective. In *IEEE Transactions on Knowledge and Data Engineering, Special Issue on Learning and Discovery in Knowledge-Based Databases.*