Building and Querying Large Collections of Data Mining Models

Alexander Tuzhilin  
IOMS Department  
Leonard N. Stern School of Business  
New York University  
New York, NY 10012 USA  
atuzhili@stern.nyu.edu

Bing Liu  
Department of Computer Science  
University of Illinois at Chicago  
851 South Morgan Street,  
Chicago, IL 60607-7053  
liub@cs.uic.edu

Jie Hu  
IOMS Department  
Leonard N. Stern School of Business  
New York University  
New York, NY 10012 USA  
juh@stern.nyu.edu

ABSTRACT
Model building is one of the most important objectives of data mining and data analysis. As many statistical and data mining applications, such as personalization, bioinformatics and some large enterprise-wide business applications, become increasingly complex and require a very large number of models, it is becoming progressively more difficult for data analysts to build and to manage a large number of models in these applications on their own. Therefore, development of software tools helping data analysts in these tasks is becoming a pressing issue. The paper presents a model management system supporting various types of data mining models. It describes how to build and populate large heterogeneous modelbases. It also presents a calculus and a query language for querying these modelbases and examines performance results for some of the queries.

1. INTRODUCTION
In the past, statistical and data mining applications required only a few models that were built by a data analyst. As real-world applications become more and more complex and require a larger and larger number of models, it is getting very hard for a data analyst to manually manage them. Even in applications that need only a single good model, the data analyst typically has to try and build a large number of models based on available data, insight, and domain knowledge to produce the final model. This process is labor intensive and very time consuming. Managing such large collections of models becomes a pressing issue.

For example, customer segmentation constitutes one of the key concepts in marketing [21]. Traditionally, marketers divided their customer bases into a small number of segments, such as pool-and-patio (suburban well-to-do customers who would usually own a house with a pool) and empty-nesters (middle-aged customers whose children left the house for college), and manually built statistical models describing behavior of each segment. Subsequently, they studied a more refined partitioning of customer bases into smaller and smaller segments, called micro-segments (or niche-segments) [21], such as the pool-and-patio customers living in a certain zip code. In applications with large customer bases, such as major credit card applications, there can be thousands of such micro-segments. If purchasing behavior of each segment is represented with several models describing different aspects of the customer behavior, then the total number of models for such applications can be measured in tens or even hundreds of thousands of models.

Similar situations occur in some bio-informatics applications, such as microarray applications, where dimensionality of data is very large, often measured in tens of thousands of variables. To have a good understanding of the problem, one may need to build a large number of different models on the microarray data using different subsets of variables. Because of the combinatorial explosion, this can result in generating hundreds of thousands or even millions of models in some cases [38].

Another example requiring management of a large collection of data mining models occurs when a data analyst generates a large number of tries before finding the right model. This situation often occurs in exploratory data analysis. It is seldom the case that a successful data mining application is carried out by simply extracting the relevant data from a database and then running a data mining algorithm. Usually, the process of data analysis constitutes a manual and very labor-intensive and time-consuming intellectual activity requiring understanding of the application problem, insightful knowledge and experiences of the data analyst and a large number of tries and tests in order to produce the final (useful) models. To add to the complexity, there are many types of models that one can try, e.g., decision trees, regressions, SVMs, rules, etc. Clearly, there is a need to help the data analyst manage this process and all the different models so that he/she can easily study the models, ask questions about them and test them with minimal effort.

In this paper, we propose a system that manages very large heterogeneous modelbases (VLMBs) consisting of large collections of different types of data mining models. In particular, we propose a storage method for storing models in relational databases and a query language to query the models. The resulting model management system would provide the following benefits:

- It would significantly expand cognitive limitations of data analysts and allow them to build and manage a much larger number of models and manage the model building process. Consequently, more models and better models may be built.

- A model management system would make data mining models a commonly shared resource in an enterprise similar to the way that DBMSes make data a commonly shared resource. This would allow naive end-users with relatively little knowledge of data mining to access models of interest in the modelbase through powerful model access and querying tools and run the accessed models on their data without worrying about the inner workings of these models. Consequently data mining technologies would become more...
accessible to larger audiences (similar to the way relational databases opened database technologies to the “masses”).

2. RELATED WORK

Model management has been studied in the Information Systems (IS) community in the context of managing models in decision support systems (DSS) since the mid-70’s when the term “model management” was coined in [42] and [35]. In particular, it was argued that, as in databases, it is important to insulate users from physical details of storing and processing models [11]. This led to the approach treating models as black boxes having only names, inputs and outputs and to the development of query languages and algebras for manipulating the models that had such operators as model solution, model composition and sensitivity analysis operators [8]. However, some researchers also argued that treating models as black boxes has certain limitations and that there is a need to consider the structure of a model in the context of a modeling lifecycle [14]. This lifecycle modeling work has primarily been focusing on the Operations Research/Management Science (OR/MS) types of models, such as mathematical programming, production, distribution, network, transportation and other types of OR/MS models and covered all the aspects of the lifecycle modeling ranging from the problem identification to the model maintenance stages of the modeling lifecycle [14]. In addition, modeling query languages for the structured modeling paradigm [14] have been developed, including SEDQL [36] and a graph-based language of Jones [18]. Some of the more recent surveys of model management can be found in [9, 22].

Although this work in the IS community addressed many interesting problems, most of the work focused on OR/MS models. There was little work on managing statistical and data mining models, on building very large collections of these modelbases and on querying them. In fact, [37] observes that “unfortunately, model query facilities, even the conventional ones, are a greatly underdeveloped area in today’s modeling systems.”

In the data mining community, the problem of managing very large numbers of discovered rules was studied by a number of researchers within the context of data mining query languages. One of the earliest data mining query languages is the one based on templates [19]. In this technique, the user uses a template to specify what items should be in or not in a rule, and what level of support and/or confidence are required. The system then checks each mined rule to find those matching rules.

In [15], Han et al presents a data mining query language, called DMQL. DMQL allows the user to specify from what table (and database) to mine what types of rules. Its main purpose is to select the right data to mine different types of rules. [25] proposes a SQL-like operator for data mining (MINE RULE). Also, [32] reports a metaquery language for data mining. Both these approaches are similar to DMQL. They are not designed for querying the mined rules, but enabling the user to specify what data mining task to perform and what its required data is.

[17][40] report a more powerful data mining query language, called MSQl. MSQl can be used not only for rule generation, but also for querying the discovered rules. With regard to rule querying, MSQl is similar to templates but allows more complex conditions. Like templates, MSQl’s query conditions can be checked using the information contained in each rule itself, e.g., support, confidence, rule length, items on the left-hand side or the right-hand side of the rule. [27] proposes an index for retrieval of association rules stored in a relational database.

The rule query language, Rule-QL [39], advances the technology further by allowing querying multiple rulebases. It has rigorous theoretical foundations of a rule-based calculus, which is based on the full set of first-order logic expressions. It was shown that different types of interesting rules that can be found by previous techniques and query languages can all be found by issuing appropriate Rule-QL queries.

The problem of managing large numbers of discovered rules was also studied in the context of personalization and bioinformatics applications [2][38]. In [2] several rule validation operators, including rule grouping, filtering, browsing, and data inspection operators, were proposed to allow domain experts to validate large numbers of discovered rules. This work focused on the validation problem and, therefore, the rule querying components of the language in [2] were more limited than in Rule-QL or in the query language described in this paper.

The idea of managing large collections of data mining models, beyond querying large numbers of association rules, has been expressed recently in the data mining community. For example, Usama Fayyad stated it as one of the top 10 important data mining problems in his invited talk at the IEEE ICDM Conference in November 2003.

At the industrial front, the latest Microsoft SQL Server 2005 [26] provides model building tools such as decision trees, neural networks, association rules, etc. Model building and testing can be done using its data mining query language similar to SQL. However, this query language is still limited: it is mainly for testing a model using data to get the model accuracy. There are only limited functionalities for querying rules or models, much fewer than those in the aforementioned language Rule-QL [39]. For example, it cannot query multiple modelbases and it does not have sophisticated containment operations as in Rule-QL, which are essential for rule or pattern querying. In SQL Server 2005, the data mining server and the database server are separate entities. The advantage of the separation is that the data mining server can access the data from any type of database systems, such as SQL Server, Oracle and DB2.

Oracle Data Mining [28] provides a different approach to data mining. Its data mining techniques are embedded in its database engine, which basically constitutes the inductive database approach [16]. The advantage is that there is no need to pull the data out from a database system to perform mining externally. The disadvantage is that the user must use only the Oracle database system. Also, Oracle still does not have rule or model querying capabilities. Our proposed model management system does not depend on either the Microsoft’s approach or the Oracle’s approach. It can be implemented in either architecture.

Bernstein and his colleagues studied the model management problem in the database context [7, 24]. Although it uses the same name, the concepts are quite different. In their work, models mainly refer to schemas and meta-data of relational database systems. Note that the reason that we use the term “model management” is because it is a standard term in data analysis and
The approach presented in this paper is applicable to a broad range of data mining models, including decision trees, regression models, SVMs, rules, and other models. However, although modelbases of different types necessarily differ from each other, they all have several common characteristics. In particular, they are stored in object-relational model tables that work in the same manner for all the different models in various modelbases.

3. DEFINING AND BUILDING MODELBASES

In this section, we define a modelbase – a large collection of related models that are stored together in the same model repository and are manipulated and retrieved using standard data manipulation and query languages. We also describe methods for populating such large modelbases in this section.

3.1 Defining Modelbases

Heterogeneous models can be organized in modelbases in several possible ways, including grouping them based on the application and the model type. In the former approach, models belonging to the same application (e.g., models of all the customers of the European Division of the company) are stored in the same table. In the latter approach, models of the same type are stored in the same table, for example, all the decision trees are stored in a separate table, all the logistic regressions in another table, etc. Although each method has its advantages, in this paper, we adopt the latter approach and assume that models are grouped together based on the same type.

The approach presented in this paper is applicable to a broad range of data mining models, including decision trees, regression models, SVMs, rules, and other models. However, although modelbases of different types necessarily differ from each other, they all have several common characteristics. In particular, they are stored in object-relational model tables having schemas containing attributes of the following types:

- **ModelId**: the key attribute uniquely identifying a model in the model table. It admits only the equality operator, i.e., it is possible only to compare two models for equality, such as ModelId1 = ModelId2.
- **DataId**: a pointer to the data file used for building the model, such as a decision tree. This data file can be a real physical or a virtual file. In the latter case, we define a database view on the real file as described below. DataId attribute admits the equality operator, e.g., MB2.DataId = MB1.DataId, and also a set of methods for retrieving the properties of the dataset defined by DataId field. For example, DataId.RecNo() is a method returning the number of records in the dataset, and DataId.Attributes() is the method returning the list of attributes of the dataset pointed to by the DataId field. The methods associated with DataId refer only to the underlying data and are model-independent: they work in the same manner for all the different models in various modelbases.
- **Model Attribute**: the attribute that actually stores a model as an “object.” For example, a decision tree is stored as a DecisionTree object, a logistic regression model as a LogisticRegression object, and so on. Each model table has only one model object attribute, and it has its own set of methods defined for it. For example, if the model attribute type is “DecisionTree”, then some of the methods for this type include NumberOfNodes(), specifying the number of nodes in the decision tree, Accuracy() specifying accuracy of the decision tree (e.g., proportion of the correctly classified instances from the testing data). This accuracy is obtained from cross-validation. The accuracy can also be obtained from different hold-out testsets. In such cases, we need to use a TestsetId to associate with its corresponding accuracy.

Since each model table is of a particular type (as discussed before, no mixed model types are allowed in a model table), this means that each table has its own set of methods associated with this model type. We would also like to point out that the model attribute is **not** required for some of the data mining models. This is the case for those models that can be defined in purely relational terms with a set of model property attributes described below. For example, association rules and logistic regressions can be defined in purely relational terms in some cases, as will be demonstrated in Example 2. In such cases, we do not have to store these models as “objects,” as in the case of decision trees.

- **Model Property Attributes**: a set of attributes defining various properties of the model. These attributes are derived from the Model Attribute by computing certain properties of the model and storing computation results as relational attributes. For example, in case of decision trees, we can compute certain statistics of the models, such as the number of nodes in the tree, the class labels and the root attribute of the tree, and store these parameters as model property attributes. Other examples of model property attributes include confidence and support statistics for association rule tables and dependent variables for logistic regression tables. These model property attributes are optional and vary from one type of a model table to another. The model property attributes are stored as regular relational fields and have standard relational types, such as integers, floats, strings, etc. As explained above, model property attributes subsume the actual model in some cases, thus making the Model Attribute field unnecessary in those cases, as demonstrated in Example 2 for logistic regressions. For other model tables, such as decision trees, the collection of model property attributes cannot replace the actual model attribute. Therefore, it is important to keep both the model and the model property attributes in those cases.

A model table can be implemented as a relational table [31] with two caveats. First, as stated before, the DataId field needs to be is a pointer into a data file that can be either physical or virtual. The virtual file can be implemented as follows. If all the data used for building all the models in the modelbase belong to one universal database, then individual data files used for building the models in the modelbase can be obtained from this central database as database views [31] by formulating SQL queries. All these views (SQL queries) can be indexed and accessed via the DataId field. One possible implementation of this data access is described in Section 3.3.

The second caveat is that the model attribute needs to be implemented as a CLOB (Character Large Object), BLOB (Binary Large Object), or large text field object (that are supported by the vendors of all the main databases) and the methods defined on the object (such as the number of nodes in the
decision tree), can be implemented as stored procedures for that object.

When defining the schema for a model table, it is necessary to define the model property attributes for the table and the methods for the model attribute. Of course, different model types result in different schemas. However, even for the same model type, various applications may require different types of model property attributes and different methods depending on the nature of the application and on the available data. The next examples present schemas for decision trees and logistic regressions.

**Example 1 (Decision tree model).** Assume that the underlying data model has \( k \) attributes. Then the decision tree table may have the following schema that we call **DTSchemaPlus**:

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelID</td>
<td>Integer</td>
</tr>
<tr>
<td>DataID</td>
<td>Integer</td>
</tr>
<tr>
<td>Model</td>
<td>CLOB</td>
</tr>
<tr>
<td>Attr_1</td>
<td>Boolean</td>
</tr>
<tr>
<td>Attr_2</td>
<td>Boolean</td>
</tr>
<tr>
<td>Attr_k</td>
<td>Boolean</td>
</tr>
<tr>
<td>Class</td>
<td>Character</td>
</tr>
<tr>
<td>Correct_Class</td>
<td>Float</td>
</tr>
<tr>
<td>Incorrect_Class</td>
<td>Float</td>
</tr>
<tr>
<td>Tree_Size</td>
<td>Integer</td>
</tr>
<tr>
<td>No_Leaves</td>
<td>Integer</td>
</tr>
</tbody>
</table>

where the presence of each attribute in the decision tree is specified with the Boolean field Attr_i. This representation is useful if the decision trees are all generated from a master data set, and each tree may be produced with a subset of the data.

In general, however, model property attributes Attr_i can represent any property of the model object, and not necessarily the attributes of the dataset used for building the model.

If the number of all the attributes \( k \) in the underlying database is too large to make it practical to dedicate a separate field for each of them in the model table, and it is still necessary to have them as model property attributes, they can be grouped together in a list of attributes.

At the other extreme, we can define the decision tree modelbase schema with only three attributes that we call **DTSchemaBasic**:

\[
\text{DT}(\text{ModelID}, \text{DataID}, \text{Model})
\]

All the other attributes in the previous example can be extracted from the attribute Model using methods, including

- **Nodes()**: /* returns the set of nodes in the decision tree
- **NumberOfNodes()**: /* returns the number of nodes in the decision tree
- **Class()**: /* returns the name of the attribute used as class attribute of the tree
- **Correct_Class()**: /* proportion of correctly classified instances of the test data

Note that the last three methods correspond to the model parameter attributes Tree_Size, Class and Correct_Class used in the alternative schema definition presented above.

The tradeoff between these two alternative schema definitions is that the latter requires less storage but significantly more computation to extract all the necessary information from the attribute Model when needed.

As this example demonstrates, each model type can have several alternative schemas for the model tables, and it is necessary to decide which model property attributes to use in the schema definition according to applications.

The next example presents the schema for a logistic regression.

**Example 2 (Logistic Regression Model).** A schema of the logistic regression model table built using the database with \( k \) attributes is defined as

<table>
<thead>
<tr>
<th>Field</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>ModelID</td>
<td>Integer</td>
</tr>
<tr>
<td>DataID</td>
<td>Integer</td>
</tr>
<tr>
<td>Attr_1</td>
<td>Boolean</td>
</tr>
<tr>
<td>Beta_1</td>
<td>Float</td>
</tr>
<tr>
<td>Beta_2</td>
<td>Float</td>
</tr>
<tr>
<td>Attr_k</td>
<td>Boolean</td>
</tr>
<tr>
<td>Beta_k</td>
<td>Float</td>
</tr>
<tr>
<td>DependentVar</td>
<td>Character</td>
</tr>
<tr>
<td>Correct_Class</td>
<td>Float</td>
</tr>
<tr>
<td>Incorrect_Class</td>
<td>Float</td>
</tr>
</tbody>
</table>

where the presence of each attribute in the logistic regression is specified with the Boolean field Attr_i. The representation is useful since the decision trees are all generated from a master data set, and each tree may be produced with a subset of the data.

As this example shows, we have removed the Model object from the schema and defined the logistic regression model in purely relational terms. This also implies that we do not have to define methods for logistic regressions. In general, the system designer needs to decide whether it is necessary to keep Model in the schema and when it can be replaced with a set of model property attributes, as in the previous example.

Several types of models, each model type having its own model table, collectively form a modelbase. Once we defined the schemas of the modelbases, we need to populate them with models. This process is described in the next section.

### 3.2 Building Modelbases

Once the modelbase schema is designed, the modelbase needs to be populated with models. Insertion of individual models by the end-users of the modelbase works only for small problems and is not scalable to very large modelbases. A more scalable approach would be for the end-user to populate the modelbase in a semi-automated manner by iteratively and interactively formulating requests of the form:

\[ \text{For dataset(s) } \text{X build the models of type } \text{Y and of the form } \text{Z} \]

where

- *dataset X* is defined either by a single or a group of DataID identifier(s) or by an SQL query selecting the dataset.
- *model of type Y*: the type of model corresponding to the model table to be filled, e.g., decision tree, regression, SVM.
• **form Z:** this is an expression specifying a *template* defining the type of model to be built. For example, we can build all the decision trees having “Purchase Decision” as a class attribute and having “Shopper Age” as one of the nodes in the decision tree. These queries can be expressed in the ModQL language presented in Section 4.

Each such request generates *multiple* models that are inserted into the modelbase. The user can grow the modelbase by repeating such requests, examining their results and formulating new requests based on the previously generated results.

### 3.3 Case Study

To show how to build large modelbases, we applied the previously described method and built a modelbase consisting of decision tree, logistic regression and association rule model tables.

We started by identifying datasets used for building the models. All these datasets are derived from an universal database of customer on-line purchases that includes such information as various demographic characteristics of the customers and such purchasing characteristics as the day of the week, category of the website, product category, the purchasing price, etc. Then we segmented the set of customers based on some of their demographic characteristics and split the entire set of customer purchasing transactions into separate (virtual) datasets SEGMENT<sub>i</sub>, based on this segmentation. For each dataset SEGMENT<sub>i</sub>, we generated several database views using SQL statements:

```
SELECT <Fields> FROM SEGMENT<sub>i</sub>
```

where `<Fields>` constitute combinations of various purchasing variables and the remaining demographic variables that were not used in generating SEGMENT<sub>i</sub> files.

Altogether, we generated 220,264 virtual datasets *defined* by these SQL queries. We stored these 220,264 SQL queries in a separate database having each query explicitly identified with the unique DataID field forming the key for that record.

These individual datasets (defined by SQL queries) were subsequently used for building data mining models using WEKA [41] by iteratively feeding them into WEKA from within Perl programs using WEKA API. As a result, we generated 220,264 Decision Tree models, 220,264 Logistic Regression models and 21,800,733 association rules that we stored in three separate tables.

For the Decision Tree model table, we used the DTSchemaPlus schema described in Example 1. For the Logistic Regression model table, we used the schema from Example 2. We did not describe the structure of the association rules because of the space limitation and because their representation is somewhat similar to the representation of logistic regressions.

### 4. MODELBASE CALCULUS MC

In this section, we introduce a *model-based calculus MC* that provides a theoretical basis for the modelbase query language ModQL presented in Section 5. The vocabulary of calculus *MC* is based on many-sorted logic [13] and is defined over

- Different types of modelbases *R<sub>i</sub>, R<sub>j</sub>, ..., R<sub>n</sub>* each modelbase being of a certain sort (e.g., decision tree, logistic regression, rule, etc.).
- A set of constants, each sort having its own (possibly empty) set of constants.
- Modelbase (or model) variables *r<sub>i</sub>, r<sub>j</sub>*, ..., *r<sub>n</sub>* defined over modelbases *R<sub>i</sub>, R<sub>j</sub>, ..., R<sub>n</sub>* and having the sort of its modelbase.
- Functions for each type of modelbase, that are classified into:
  - Those that return values of individual model property attributes, such as ModelID, DataID and Confidence. These functions are denoted as r.funct(), e.g., r.ModelID.
  - Method functions associated with the ModelAttribute, such as NumberOfNodes() and Beta(), returning the number of nodes in the decision tree and the list of beta-coefficients in a logistic regression. These functions are denoted as r.funct(), e.g., r.NumberOfNodes().
- Predicates. In addition to the modelbase predicates *R<sub>i</sub>, R<sub>j</sub>, ..., R<sub>n</sub>* described above, *MC* also supports binary relational operators *⊂*, *⊆*, *⊃*, *⊇*, *∈*, *∉*, *≤*, *≥*, *≠*, defined on sets and relational operators *≥*, *≤*, *>, <*, = defined on ordinals in a standard manner. Since *MC* is a typed calculus, each of these relational operators is of an appropriate type.

An atomic formula of *MC* has one of the following forms:

- *R(r)*, where *R* is a modelbase and *r* a model variable of sort *R*.
- *α op β*, or *α op const*, where *α* and *β* are one of the set-theoretic functions defined above, *const* is a set of objects (i.e., items for association rules, nodes for decision trees, variables for logistic regressions, etc.) and *op* is one of the operators *⊂*, *⊆*, *⊃*, *⊇*, *∈*, *∉*, *≤*, *≥*, *≠*, or *=*
- *γ op δ* or *γ op const*, where *γ* and *δ* are numeric functions, *const* is a constant, and *op* is one of the standard relational operators *≤*, *≥*, *<*, *=*.

A set of well-formed formulae in *MC* is obtained from atomic formulae in a standard way as it is done in many-sorted first-order logic [13] by taking the closure of conjunction, disjunction, negation and universal and existential quantification operators. Safety of *MC* formulae is defined similarly to the safety of relational calculus formulae [1], and we skip its formal definition for this reason and because of the space limitation.

We next present some examples of *MC* queries. We assume in these examples that all the logistic regressions are stored in the LR table, all the decision trees in the DT table, and all the association rules in the AR table. We also assume that the DT and the LR tables have the schema structure described in Section 3.1. However, we decided not to use any of the model parameter attributes but only the methods to access and retrieve appropriate information directly from the models. For example, to access the nodes of a decision tree model, we use the method x.Nodes() that retrieves the names of all the nodes of the tree. In other words, each model table DT, LR and AR only has attributes ModelID, DataID and Model. Also Model assumes several methods associated with it that differ across tables DT, LR and AR. The specific methods for DT, LR and AR tables will be described in
the examples below.

We start the description of language **MC** by, first, presenting two simple queries that show how models of a certain type or structure can be selected from modelbases. The selection criterion in the first query is on the types of variables appearing in the nodes of the decision tree, while in the second query is on the size of the decision tree.

**Query 1:** Find decision trees having Income variable among the nodes of the tree.

\{x.ModelID | DT(x) and “Income” ∈ x.Nodes()\}

**Query 2:** Find decision trees having less than 10 nodes.

\{x.ModelID | DT(x) and x.NumberOfNodes() < 10\}

The selection criteria in both queries are specified using the method functions (x.Nodes() and x.NumberOfNodes()) defined for the Decision Tree model type. However, if we use an alternative representation of the Decision Tree modelbase having the model property attribute NumberOfNodes, then we could simply use x.NumberOfNodes as such attribute in Query 2.

The next query demonstrates how selection criteria are applicable to logistic regressions.

**Query 3:** Find logistic regressions having at least one beta-coefficient greater than 1.

\{x | LR(x) and MAX(x.Beta()) > 1\}

In this query, method Beta() returns the list of beta-coefficients of a logistic regression (specified by variable x), and function MAX selects the largest element from the list.

The next query demonstrates how quantification operators are used over the models from the modelbase.

**Query 4:** Find the best decision tree model in terms of its accuracy rates.

\{x.ModelID | DT(x) and (∀y) (DT(y) ⇒ x.Correct_Class() > y.Correct_Class()) \}

where x.Correct_Class() specifies the accuracy of a decision tree, e.g., the proportion of correctly classified instances on the test data.

The next example demonstrates how queries are asked about models and the data from which they have been built. This ability to ask questions simultaneously about modelbases and databases constitutes an important and distinguishing property of **MC**.

**Query 5:** Find the decision tree models that have been learned from datasets with more than 10,000 records and that have “Purchase_Decision” as the class attribute.

\{x.ModelID | DT(x) and x.DataID.NoRecords() > 10,000 and x.Class() = “Purchase_Decision”\}

The next query demonstrated a join operation between two modelbases of the same type. In this case, this is a self-join on a modelbase of association rules.

**Query 6:** Find minimal association rules, i.e., association rules whose LHS and RHS do not contain the LHS and RHS of any other rule respectively.

\{r | AR(r) and ¬ (\exists r′)(AR(r′) and r ≠ r′ and r.DataID = r′.DataID and r′.LHS() ⊆ r.LHR() and r′.RHS() ⊆ r.RHS())\}

The next query presents a join between two different types of modelbases.

**Query 7:** Find association rules with confidence higher than 90% such that the variables in their LHS appear among the nodes of all the DT models generated from the same dataset.

\{x.ModelID | AR(x) and x.Confidence = 90% and (∀y) ((DT(y) and x.DataID = y.DataID) ⇒ (x.LHS() ⊆ y.Nodes()))\}

Next, we provide an example of a more complex query. It also demonstrates how to do a more complex join between modelbases of different types.

**Query 8:** Find decision tree models that have the same class attribute and the same set of nodes as the dependent and independent variables in some logistic regression model and that outperform the logistic regression in terms of accuracy on the same data.

\{x.ModelID | DT(x) and (∃y) (LR(y) and x.DataID = y.DataID and x.IndepVar() = y.Nodes() and x.DepVar() = y.Class() and x.Correct_Class() > y.Correct_Class())\}

In this example, we assume that the methods IndepVar() and Nodes() return the sets of independent variables and the internal nodes in the logistic regression and decision tree models respectively.

The next example demonstrates how to query models learned from different datasets.

**Query 9:** Find every logistic regression model such that its predictive accuracy on one dataset is the best but on another dataset it is the worst (in comparison to any other logistic regression model in the modelbase).

\{x.IndepVars, x.DepVar | LR(x) and (∃y)(LR(y) and x.IndepVars() = y.IndepVars() and x.DepVar() = y.DepVar() and x.DataID ≠ y.DataID and (∀z) (LR(z) ⇒ x.Correct_Class() ≤ z.Correct_Class() ≤ y.Correct_Class())\}

In this example, logistic regression models x and y have the same structure (dependent and independent variables), but their parameters are estimated on different data sets (x.DataID ≠ y.DataID).

5. **MODELBASE QUERY LANGUAGE**

**ModQL**

ModQL is an extension of SQL defined on modelbases of different types. It has the following idiosyncratic features pertaining to modelbases:

1. The actual models are implemented as CLOB, BLOB or “large field” “objects,” depending on the particular database.
2. Methods of **MC** calculus are implemented as stored procedures associated with the objects described in Point 1.
3. Some of the expressions in ModQL corresponding to the atomic formulas in **MC** are implemented as macros in
ModQL, i.e., these expressions are not part of the standard SQL but are mapped into (often awkward) expressions of the standard SQL. To avoid repetition of the queries from Section 3, we present only a few examples of ModQL queries. The first example demonstrates that the ModQL query critically depends on the specific schema used for model representation.

Query 1: Find decision trees having Income variable among the nodes of the tree.

If DTSchemaBasic schema from Example 1 is used, then this query is expressed as

```
SELECT ModelID
FROM DT
WHERE “Income” IN Model.Nodes()
```

where Model.Nodes() is a stored procedure returning the list of nodes of the decision tree.

If DTSchemaPlus schema is used instead, then this query is expressed as a standard SQL statement checking if the model parameter attribute INCOME is set “on”:

```
SELECT ModelID
FROM DT
WHERE DT.Income = 1
```

The next example demonstrates more complex queries involving (self-)joins, nested queries and the use of macros in ModQL.

Query 2: Find minimal association rules, i.e., association rules whose LHS and RHS do not contain the LHS and RHS of any other rule respectively.

```
SELECT R.*
FROM AR R
WHERE NOT EXISTS
  (SELECT R’.*
   FROM AR R’
   WHERE R.ModelID ≠ R’.ModelID AND
   LHS(R’) CONTAINED_EQ_IN LHS(R) AND
   RHS(R’) CONTAINED_EQ_IN RHS(R))
```

This query contains expression “LHS(R’) CONTAINED_EQ_IN LHS(R)” that is not a part of SQL. However, if the schema of AR model table contains all the items (attributes) of the underlying dataset, then this expression is really a macro that can be formulated in standard SQL as

```
R’.Item_1 = L ⇒ R.Item_1 = L AND R’.Item_2 = L ⇒ R.Item_2 = L AND ... AND R’.Item_k = L ⇒ R.Item_k = L
```

where Item_1, Item_2, ... and Item_k are all the items (attributes) of the underlying database, and “L” stands for the fact that they appear in the left-hand sides of the association rules R and R’ respectively. In other words, this expression says that, for any Item_i, if Item_i appears in the LHS of rule R’, then it should also appear in the LHS of rule R. Also, expression “RHS(R’) CONTAINED_EQ_IN RHS(R)” can be specified in SQL in a very similar manner as the LHS expression above.

The next example demonstrates joins between two different model tables and the use of stored procedures in ModQL.

Query 3: Find decision tree models that have the same class attribute and the same set of nodes as the dependent and independent variables in some logistic regression model, that are also generated from the same data as the logistic regression model and that outperform the logistic regression model in terms of the accuracy.

```
SELECT DT.ModelID
FROM LR, DT
WHERE LR.DataID = DT.DataID AND LR.IndepVar() EQUAL DT.Nodes() AND LR.DepVar() EQUAL DT.Class() AND DT.Correct_Class() > LR.Correct_Class()
```

In this query, we assumed that the list of independent variables in logistic regressions and the nodes in decision trees are retrieved using stored procedures IndepVar() and Nodes() respectively. Therefore, EQUAL operator equates two sets of variables. However, if we define the schemas of DT and LR tables so that the nodes in DT and independent variables in LR appear among the model property attributes, then we can express “LR.IndepVar() EQUAL DT.Nodes()” as a macro in SQL using methods similar to those used in Query 2.

Query 4: Find all pairs of decision tree and logistic regression models that have at least one variable in common among the independent variables of the logistic regression and the nodes of the decision tree.

```
SELECT DT.ModelID, LR.ModelID
FROM LR, DT
WHERE LR.Model.IndepVar() ∩ DT.Model.Nodes() ≠ Ø
```

This query is specified using the basic schema (DTSchemaBasic) that requires access to the actual decision tree and logistic regression models. Alternatively, this query can be implemented in SQL assuming the DTSchemaPlus and the LR schema from Example 2 as

```
SELECT DT.ModelID, LR.ModelID
FROM LR, DT
WHERE (DT.Attr_1 = 1 AND DT.Attr_1 = 1) OR
  (DT.Attr_2 = 1 AND DT.Attr_2 = 1) OR
  ................
  (DT.Attr_k = 1 AND DT.Attr_k = 1)
```

This version of the query is expressed in SQL and is evaluated using standard SQL methods. The first version of the query, however, is processed by joining LR and DT tables in a record-by-record basis. In other words, it is done by considering all the |LR| × |DT| combinations of the logistic regression and decision tree models from LR and DT tables, retrieving all the nodes from the decision tree model, all the independent variables from the logistic regression model and checking if their intersection is not empty.

6. EXPERIMENTS AND DISCUSSIONS

Since ModQL queries involve models and often need access to the model “object” itself (CLOB, BLOB, etc.), some of these queries can be very slow. Therefore, extreme care should be taken when formulating and processing such queries. In this section, we evaluate performance of some of the queries to gain a better understanding of the query evaluation issues.
As discussed before, ModQL queries are divided into the following categories:

1. Those that can be expressed and evaluated in pure SQL. For example, the second version of ModQL Query 1 (evaluated on the \textit{DTSchemaPlus} schema) belongs to this category.
2. Those that can be expressed in SQL but require macros. For example, ModQL Query 2 belongs to this category.
3. Those that cannot be evaluated in SQL because they require direct access to the model object using its methods. For example, the first version of Query 4 (based on the \textit{DTSchemaBasic} schema) belongs to this category.

Moreover, queries of Type 3 are divided into two sub-categories: those that require joins of two or more model tables and those that don’t (such as the first version of Query 1).

To evaluate performance of different types of queries, we

1. Executed both versions of Query 1: the one that requires access to the DT object using the \text{Nodes()} method and the “pure SQL” version.
2. Query 4 also for both types of methods: the one requiring access to model objects and the one that can be expressed in SQL with macros.
3. Query 2 expressed as a macro.

The performance results for Queries 1 and 4 are reported in Figures 1, 2 and Table 1 respectively. As Figure 1 shows, direct SQL evaluation is very fast: the whole model table of 220,264 models was processed in less than 1 second. In contrast, the second (object access) version of Query 1 is much slower, as Figure 1 demonstrates. This is the case because each decision tree object in the DT table needs to be accessed and searched for the presence of the Income variable.

![Figure 1: Performance Comparison for two evaluation methods for ModQL Query 1: one implemented in pure SQL and another requiring object access to decision trees using the Model.Nodes() method.](image)

The last problem can be solved by creating special indices for modelbases, similar to the ones used in rulebases, including inverted file indices [39]. However, this type of query processing lies outside of pure SQL, and it is important to combine it effectively with SQL query optimizers to achieve good overall performance results.

The performance results for Query 4 are even more dramatic. For both versions of Query 4 (implemented as a SQL macro and requiring access to the objects of the model), it was necessary to do the join on the DT and LR tables. However, SQL join performed reasonably well, as column 3 in Table 1 shows. In contrast, the object-access version of Query 4 was extremely slow, as Table 1 and Figure 2 demonstrate. In fact, it was so slow that we could evaluate the query on the join of DT and LT tables containing only up to 4,000 models each.

![Table 1: Performance results for 2 versions of Query 4.](image)

Number of models & Object access (seconds) & SQL (seconds) & Number of Matches \\
--- & --- & --- & --- \\
1K x 1K & 1104 & 5 & 813778 \\
2K x 2K & 4614 & 19 & 3312150 \\
3K x 3K & 10458 & 44 & 7501820 \\
4K x 4K & 18158 & 113 & 13413543 \\

This example demonstrates that ModQL suffers from the \textit{query from hell} phenomenon, when some of the queries are so slow that they would run “forever.” Another example of such query, besides Query 4, is Query 2 that was launched on the whole table of association rules containing 21,800,733 rules. This query was implemented as a SQL macro (actually, quite awkward and long when expanded). It was executed on Pentium 4 machine with 3GHz CPU and 1GB of RAM for more than 72 hours and would not finish (we had to terminate the query).

We can conclude from these experiments that, while ModQL performs well on some of the queries (especially those expressed in simple SQL), it has performance problems on other queries, especially when they are evaluated in a brute force manner, such as the first version of Query 4.

This problem can be addressed as follows. First, one needs to develop methods for detecting and, possibly, blocking “queries from hell.” Second, it is important to develop efficient methods and effective indexing schemes to process \textit{parts} of ModQL queries outside of SQL and then combine the results with the

![Figure 2: Performance results for 2 versions of Query 4 (graphical representation of results from Table 1).](image)
remaining SQL expressions. Both of these topics constitute an interesting research problem that we plan to address in the future.

7. CONCLUSIONS
As data analysis and data mining is increasingly widely used in practice, there is a need to generate, store and query very large collections of data mining models. This paper describes an approach to generating and querying large modelbases. The model calculus MC is based on the first-order logic and the query language ModQL on SQL with certain extensions to incorporate model management capabilities. We tested some of the queries expressed in ModQL on a modelbase that we have generated for this project. While some simple queries can be processed very quickly in ModQL, other queries run very slowly because they require access to the internals of the models themselves. This calls for the development of new query optimization strategies specifically targeted for the model management domain. We plan to develop such query optimization techniques in the future.

Another topic of future research constitutes application-driven methods for organizing modelbases. In this approach various heterogeneous models are stored in one physical or virtual model table containing all the models pertaining to a certain application. It is not easy to organize and query such modelbases because of several technical problems. We plan to study these problems and develop a solution in the future.

8. REFERENCES
Advances in Knowledge Discovery and Data Mining, 


