Building and Querying Large Modelbases

Alexander Tuzhilin
Leonard N. Stern School of Business
New York University
atuzhili@stern.nyu.edu

Bing Liu
Department of Computer Science
University of Illinois at Chicago
liub@cs.uic.edu

Jie Hu
Leonard N. Stern School of Business
New York University
jhu@stern.nyu.edu

Abstract

Model building is one of the most important objectives of data mining and data analysis. As many 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 built 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. This 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 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 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 to 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 is one of the key concepts of marketing [10]. Marketers traditionally 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*) [10], 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 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. Due to the combinatorial explosion, this can result in hundreds of thousands or more models in some cases [18].

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 as there are many types of models and so many ways to built models. 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 a minimal effort.

All these examples highlight the necessity to develop a model management system. Such a system would provide the following benefits:

- 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.
- 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 naïve end-users with relatively little knowledge of data mining to access models of interest in the modelbase through powerful querying tools and run the accessed models on their data without worrying about the inner workings of these models. Thus, data mining technologies would become more accessible to larger audiences (similar to the way relational databases opened database technologies to the "masses").

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. We describe how to build and populate the modelbases and also present a query language (***ModQL***) that is a dialect of SQL extended with object-relational features for querying these modelbases. Our aim is to manage (to store and to query) models using existing object-relational database systems as much as possible. Although ModQL queries can be defined in standard object-relational terms, we show in the paper that the performance of some of the queries is too slow for "real-world" problems. Therefore, we describe how to improve their performance using certain indexing techniques. This still leaves a few "exotic" ModQL queries

for which these indexing techniques are not applicable and whose performance cannot be improved using the tools of standard object-relational databases. We conclude with the discussion if it is necessary to modify existing object-relational DBMSes to process such queries efficiently.

2. Related Work

Model management has been studied in the Information Systems (IS) community in the context of decision support systems (DSS) since the mid-70's when the term "model management" was coined in [15][22]. It was argued that, as in databases, it is important to insulate users from physical details of storing and processing models [4]. This led to the approach of 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 [3]. Work was also done on model lifecycle [5]. Some query languages were proposed [17] that were highly specialized and dealt exclusively with querying modelbases. However, the IS research focused mainly on Operations Research/Management Science (OR/MS) types of models, e.g mathematical programming, distribution, and transportation models [5]. A survey of these activities can be found in [11]. There was little work done on managing and querying *data mining* models.

In the data mining community, the problem of managing large numbers of discovered rules was studied by several researchers within the context of data mining query languages. One of the early query languages is the one based on templates [8]. 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 finds the matching rules.

In [6], Han *et al* presents a data mining query language, called DMQL. DMQL allows the user to specify from what table to mine what types of rules. [13] proposes a SQL-like operator for data mining (MINE RULE). Both these approaches 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.

[20] reports a more powerful query language, called MSQL. MSQL can be used for both rule generation and rule querying. With regard to rule querying, MSQL is similar to templates but allows more complex conditions.

The rule query language, Rule-QL [19], advances the technology further by allowing querying multiple rulebases. It has rigorous theoretical foundations of a rule-based calculus based on the full set of first-order logic expressions. It was shown that different types of rules that can be found by previous techniques and query languages can all be found by issuing appropriate Rule-QL queries.

Grouping and filtering rules were also studied in such applications as personalization and bioinformatics [1][18]. However, their querying capabilities were more limited.

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.

Bernstein studied the model management problem in the database context [2]. Although it uses the same name, the concepts are quite different. In his 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 business communities.

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 data manipulation and query languages.

3.1 Defining Modelbases

Heterogeneous models can be organized in modelbases by either grouping them based on the application or on the model type. In the former approach, models belonging to the same *application* 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. Although modelbases of different types differ from each other, they all have several *common characteristics*. In particular, they are stored in object-relational *model tables* having schemas with attributes of the following types:

- *ModelID*: the key attribute uniquely identifying a model in the model table. It admits only the equality operator, e.g., ModelId1 = ModelId2.
- *TrainDataID*: 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. TrainDataID attribute admits the equality operator, e.g., MB2.TrainDataID = MB1.TrainDataID, and also a set of methods for retrieving the properties of the dataset defined by TrainDataID. For example, TrainDataID.RecNo() is a method returning the number of records in the dataset, and TrainDataID.Attributes() is the method returning the list of attributes of the dataset pointed to by the TrainDataID field.
- *TestDataID*: the same as a TrainDataID, but pointing to a data file used in testing a model for accuracy. This

field is optional because some models (a) do not require test data (e.g., association rules), or (b) use cross-validation testing on the TrainDataID field.

- *Model Attribute*: the attribute that actually stores a model as an "object." For example, a decision tree is stored as a DecisionTree object. 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, and Accuracy() specifying the accuracy of the decision tree.

  Since each model table is of a particular type, this means that each table has its own set of methods associated with this model type.
- *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 and storing the 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, and store it as a model property attribute. The model property attributes are optional and vary from one type of a model table to another.

A model table can be implemented as a relational table with two caveats. First, as stated before, the TrainDataID (or TestDataID) field needs to be a pointer to a data file that can be either physical or virtual. The virtual file can be implemented as a *database view* [14] by formulating SQL queries. All these SQL views can be indexed and accessed via the TrainDataID 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 [14], together with the methods defined on it (such as the number of nodes in the decision tree).

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 can result in *different* schemas. However, even for the *same* model type, there can be *more than one* schema as the following examples demonstrate.

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

```
ModelID       Integer
TrainDataID   Integer
Model         CLOB      /*decision tree object
Attr_1        Boolean   /* TRUE if Attr_1 variable appears
                          as a node of the tree
……            ……        ……
Attr_k        Boolean   /* same as with Attr_1
Class         Character /* name of the class attribute
TestDataID    Integer
TestAccuracy  Float     /* model accuracy on test data
Accuracy      Float     /* cross-validation accuracy
TreeSize      Integer   /* Size of the decision tree
NoLeaves      Integer   /* Number of leaves in the tree
```

where the presence of each attribute in the 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, model property attributes Attr_i can represent *any* property of the model, and not necessarily the attributes of the dataset used for building the model.

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

DT(ModelID, TrainDataID, TestDataID, Model)

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

```
Nodes()                 /* returns set of nodes in the tree
NumberOfNodes()         /* returns number of nodes in the tree
Class()                 /* names of attributes used as classes.
Accuracy(TestDataID)    /* model accuracy on test data
```

Note that the last three methods correspond to the model property attributes TreeSize, Class and TestAccuracy in the alternative schema definition above.

The tradeoff between these two alternative schema definitions of decision trees is that the latter requires less storage but 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.

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

```
ModelID       Integer
TrainDataID   Integer
Attr_1        Boolean   /* TRUE if Attr_1 variable appears
                          in the regression
Beta_1        Float     /* Beta coefficient 1
Attr_2        Boolean   /* same as with Attr_1
Beta_2        Float     /* Beta coefficient 2
……            ……        ……
Attr_k        Boolean   /* same as with Attr_1
Beta_k        Float     /* Beta coefficient k
DependVar     Character /* name of dependent variable
TestDataID    Integer
TestAccuracy  Float     /* accuracy on the test data
Accuracy      Float     /* cross-validation accuracy.
```

Note that in this example, 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, system designers need to decide if it is necessary to keep Model in the schema and when it can be replaced with a set of model property attributes, as in this example.

The schema for association rules can be designed similarly. Each association rule is represented just like a logistic regression model. However, we may need only one data set (DataID) from which to generate the rules.

Several types of models, each model type having its own model table, collectively form a *modelbase.*

3.2 Building Modelbases

Once the modelbase schemas are 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 a semi-automated method. The user can *iteratively* and *interactively* formulate requests of the form:

For dataset X build the models of type Y and of the form Z

where

- *dataset X* is defined either by the TrainDataID identifier or by a 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 or regression.
- *form Z*: this is an expression specifying a *template* defining the type of model to be built and stored. For example, we can build all the decision trees having "Purchase_Decision" as a class attribute and having Income as the root node. In general, the form can be expressed as constraints in the WHERE clause of the ModQL language presented in Section 4.

There are two approaches to handle the Z-form constraints:

1. Filtering: models are first generated, and then those that do not satisfy the constraints are filtered or deleted.
2. Constrained model generation: constraints are pushed into the model generation process so that only those models that satisfy the constraints are generated.

Which approaches to use depends on the available algorithms. For example, in association rule mining, constrained rule mining techniques may be used to generate the required rules for certain types of constraints.

Each model generation request generates *multiple* models. The user can grow the modelbase in a controlled manner by *iteratively* issuing new requests, examining their results, inserting *only* the useful ones into the modelbase and formulating new requests based on the previously generated results. To make this whole approach user-friendly for a non-technical end-user, these requests can be generated via a front-end GUI, as is often done in databases when the end-user specifies database commands using a GUI-based front-end rather than directly in SQL.

3.3 Case Study

To show how to build large modelbases, we applied the method in Section 3.2 and built a modelbase consisting of decision tree, logistic regression and association rule tables.

Our study is based on a on-line customer purchases database that includes such information as 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. We segmented the customers based on some demographic characteristics and split the entire set of customer purchasing transactions into separate (virtual) datasets $SEGMENT_i$. For each dataset $SEGMENT_i$, we generated several database views using SQL statements:

SELECT <Fields > FROM $SEGMENT_i$

where <Fields> are combinations of various purchasing variables and the remaining demographic variables that were not used in generating $SEGMENT_i$ 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 TrainDataID field forming the key for that record.

These individual datasets (defined by SQL queries) were subsequently used for building data mining models by iteratively feeding them into WEKA system [21]. 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. As noted earlier, 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.

Finally, we note that all the modelbases were generated using SQL queries and Perl scripts rather than a special-purpose tool for this task. This was sufficient for our proof-of-concept purposes. However, it is necessary to develop a model-building interactive tool for the industrial-strength applications in the future to improve model building tasks.

4. Modelbase Query Language *ModQL*

As was mentioned in Section 2, previous approaches to querying model- or rule-bases were based on specially designed query languages. In contrast to this, we chose to deploy standard object-relational query language SQL99 [16] for querying modelbases. In particular, we selected a certain dialect of SQL99 suitable for querying modelbases, as will be described in the rest of this section.

As explained in Section 3, each model is defined with a particular schema that includes ModelID, the model itself stored as a CLOB object, a set of methods retrieving model properties from the CLOB, the training (and optionally testing) data set, and a set of model property attributes. For example, two decision tree schemas are presented in Example 1 and a logistic regression schema in Example 2.

ModQL is essentially SQL99 specified on the model schemas of the types described above. It explicitly deals with CLOB/BLOB objects, methods defined on these objects, and training and testing datasets. In addition, ModQL supports macros, i.e., non-SQL expressions that can be mapped into standard SQL expressions.

We next present some examples of ModQL queries. We assume in these examples that all the logistic regressions are stored in the LR model table, all the decision trees in the DT model table, and all the association rules in the AR model table. We also assume that the DT and the LR tables have the schema structure as described in Section 3.1.

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 method 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:

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

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

```
SELECT   ModelID
FROM     DT
WHERE    DT.NumberOfNodes() < 10
```

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.*

```
SELECT   ModelID
FROM     LT
WHERE    MAX(LT.Beta()) > 1
```

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

The next query demonstrates how the best-performing models are selected from a model table.

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

```
SELECT   ModelID
FROM     DT
WHERE NOT EXITSTS (SELECT R'.*
                   FROM DT
                   WHERE R'.Accuracy() > R.Accuracy())
```

where x.Accuracy() specifies the accuracy of a decision tree based on cross-validation.

The next example shows how queries are asked about models and the data from which they are built. This ability to ask questions about both modelbases and databases is an important and distinguishing property of ModQL.

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

```
SELECT   ModelID
FROM     DT
WHERE    DT.TrainDataID.NoRecords() > 10,000
         AND DT.Class() = "Purchase_Decision"
```

The next query demonstrates a self-join operation between two model tables of the same type and the use of macros.

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.*

```
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 the expression "LHS(R') CONTAINED_EQ_IN LHS(R)" that is not a part of SQL (CONTAINED_EQ_IN means "subset"). 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 on 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 on 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 based on a complex joining criteria.

Query 7: *Find decision tree models having 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 accuracy.*

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

In this query, we assumed that the list of independent

variables in logistic regressions and the nodes in decision trees are retrieved using methods IndepVar() and Nodes() respectively. Therefore, the join operator EQUAL 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 6. x.Accuracy() gives the accuracy of the model x based on cross-validation.

Next, we provide an example of a join between two model tables having a complex join criterion.

Query 8: *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. Moreover, DT and LR tables are joined using a complex joining criteria involving two sets of variables. This query is evaluated using the nested join method [14] 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. Clearly, this is a very slow and inefficient evaluation method, as shown in Section 5.

Alternatively, this query can be implemented as a SQL macro assuming the DTSchemaPlus and the LR schema from Example 2. In this case, the WHERE clause of the above query is a SQL macro that can be expanded in standard SQL as

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

This SQL-based version of the query is evaluated using standard and reasonably efficient SQL methods.

In summary, all these examples show the power of ModQL by demonstrating various useful and non-trivial queries about data mining models that can be expressed in the language. Moreover, all these queries can be expressed in a dialect of SQL99. Therefore there is no need to develop new software systems to support these queries. The well-established database technologies can be used instead. This is in contrast to the previous proposals for the development of *new* languages specifically designed for querying model- and rule-bases described in Section 2.

To see how well ModQL works in practice, we conducted empirical studies described in the next section.

5. Experiments

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, special care should be taken when formulating 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 relational SQL. For example, the second version of ModQL Query 1 (evaluated on the *DTSchemaPlus* schema) belongs to this category.
2. Those that can be expressed in SQL with macros. For example, ModQL Query 6 belongs to this category.
3. Those that cannot be evaluated in pure SQL because they require direct access to the model object using its methods. For example, the first version of Query 8 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, e.g., the first version of Query 1. To test 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 Nodes() method and the "pure SQL" version.
2. executed both versions of Query 8: the one requiring access to model objects and the one that can be expressed in SQL with macros.
3. executed Query 6, expressed in SQL with macros.

These queries were executed on the three model tables DT, LT and AR described in Section 3.3, having 220,264, 220,264 and 21,800,733 models respectively. These model tables were stored in a Microsoft's SQL Server located on the Pentium 4 server with 3GHz CPU and 1GB of RAM. The models were generated as character strings by WEKA and stored as CLOB objects. The methods accessing these objects were implemented in Perl. Finally, SQL macros used in some of the queries were decoded manually.

Figures 1, 2 and Table 1 report performance results for Queries 1 and 8 respectively. 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 object access version of Query 1 is much slower. 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. This problem can be solved by creating special indices for modelbases (see Section 6).

The performance results for Query 8 are even more dramatic. For both versions of Query 8 (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 (from 220,264 models).

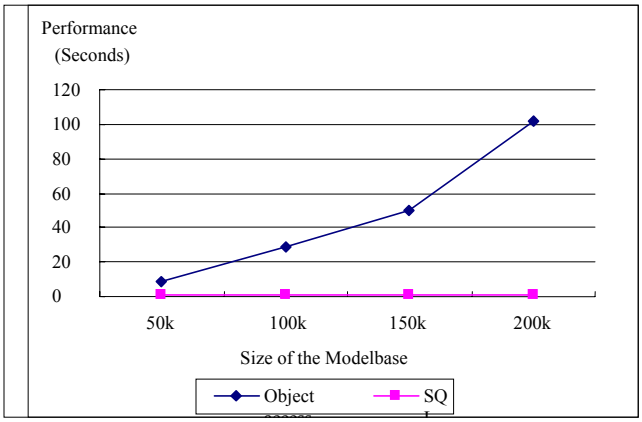

Figure 1: Performance comparison of two versions of Query 1: pure SQL vs. trees access with Model.Nodes() method.

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

Table 1: Performance results for 2 versions of Query 4.

This example demonstrates that ModQL suffers from the *query from hell* phenomenon, when some of the queries are so slow that they would run "forever." Another example of such query is Query 6 that was launched on the whole table of association rules containing 21,800,733 rules. This query was implemented as a SQL macro. It was executed on the aforementioned SQL Server for more than 72 hours and would not finish (we had to terminate it).

6. Discussions

From the results reported in Sections 3 - 5, we can conclude that object-relational databases can provide a basic platform for model management and that no special-purpose languages are required for querying modelbases. However, as we also showed, effectiveness of object-relational databases varies significantly across different model schemas, queries and other conditions. In particular, we can make the following conclusions from our work:

1. Generating and storing large numbers of different types of models is a manageable task, as demonstrated in Section 3. This capability cannot be directly supported within existing DBMSes, and the development of a new interactive model-building tool is required for industrial-strength applications.

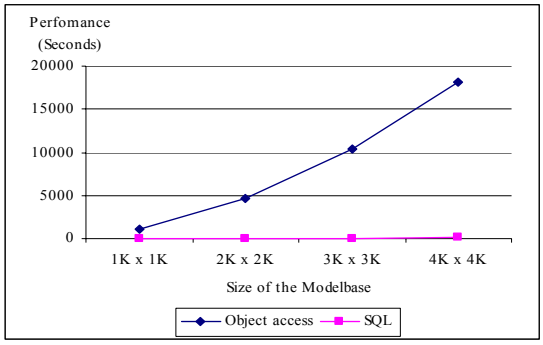

Figure 2: Performance results for 2 versions of Query 4 (graphical representation of results from Table 1).

2. While ModQL performs well for some of the queries, it has performance problems for others, especially when they are evaluated in a brute force manner. These performance problems are often attributed to the queries that cannot be expressed in pure SQL and require macros and methods.
3. There is a need to develop efficient query processing strategies to avoid the "query-from-hell" problems. We describe possible solutions to this problem below.

One way of dealing with the queries-from-hell problem is to use *indexes on methods*. For example, one can build an index on the method DT.NumberOfNodes() returning the number of nodes in a decision tree. This index can be implemented as a B+-tree. Then Query 2 can be evaluated by accessing this B+-tree index rather than sequentially scanning model table DT and accessing each method DT.NumberOfNodes().

In fact, some DBMS vendors already provide such capabilities. For example, Oracle 11i supports *indextype*, extensible indexing methods for user-defined operators (such as NumberOfNodes() in the example above) (www.lc.leidenuniv.nl/awcourse/oracle). Therefore, the indexing methods described above can be implemented in Oracle, which can significantly improve the performance of Query 2 and others from Section 4.

Although such indexing methods can solve some of the query-from-hell problems, it does not solve *all* of them because such indexes can be created only on the methods returning "indexable" results, such as NumberOfNodes(). In contrast, methods returning "complex objects," such as subtrees or lists, cannot be easily indexed using the aforementioned indextype methods. For example, it is unclear how to create an index for the first version of Query 8 where the join condition involves sets of decision tree nodes and sets of logistic regression variables. Unlike the NumberOfNodes() method, these complex object types cannot be easily indexed. Therefore, indexing methods cannot solve all of the query-from-hell problems.

To deal with this problem, it is necessary to provide extensions to the DBMS internals in order to support such indexes. However, it is not clear if making changes to the

DBMS internals are warranted to support a few "exotic" queries having methods returning complex objects. An alternative solution would be for the query preprocessor to identify such queries, flag them as "queries-from-hell" and warn the end-user about it. Finally, if such a query is really important for the application, then the method can be materialized and stored as a model property attribute, thus avoiding the query-from-hell problem.

Given the indexing methods described above, the user has three choices when designing a model table:

1. Use only the methods associated with the model without using any model property attributes.
2. Materialize some of the methods by defining and computing model property attributes.
3. Use indexes, instead of the model property attributes, for some of the methods.

Each of these choices has its strength and weaknesses. In particular, model property attributes provide for fast execution of modelbase queries, as shown in Section 4. However, they require extra space, and maintenance. In contrast to this, methods do not require any extra storage and maintenance, but can slow query processing very significantly, as was shown in Section 5.

Therefore, indexes on methods provide a good compromise between these two solutions, as discussed above, but also not requiring much more extra storage and causing fewer maintenance problems.

We can conclude that most querying capabilities of ModQL can be directly supported by DBMS vendors. The remaining functionalities are too specialized for the DBMS vendors to modify their query processors to speed up such queries. We thus recommend that DBMS vendors develop only interactive model management tools discussed in Section 3 in order to provide for the creation and maintenance of large collections of data mining models.

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 with the query language ModQL. ModQL is an object-relational dialect of SQL99 with certain features added to it to incorporate model management capabilities.

We demonstrated that modelbase querying can be done within the object-relational framework and, therefore, it can rely on proven database technologies and does not require any special-purpose query languages and systems for modelbases, as was advocated before.

We also tested some of the queries expressed in ModQL on a large modelbase. While some simple queries can be processed very quickly in ModQL, others run very slowly because they require access to the internals of the models. We then explained how this problem can be solved by creating indexes on methods, as for example, Oracle 11i does it with its indextype extensible indexing approach. However, the methods returning complex objects cannot be easily indexed and require further studies.

8. References

[1] Adomavicius, G. and Tuzhilin, A. "User profiling in personalization applications through rule discovery and validation." *KDD-99*, 1999.

[2] Bernstein, P.A. "Applying Model Management to Classical Meta Data Problems," *Proc. CIDR 2003*, pp. 209-220.

[3] Blanning, R.W., A Relational Theory of Model Management. In C. Holsapple and A. Whinston (eds.). Decision Support Systems: Theory and Applications, 1987.

[4] Dolk, D.K., and B. R. Konsynski, "Knowledge Representation for Model Management Systems." *IEEE Transactions on Software Engineering*, SE-10(6), 1984.

[5] Geoffrion, A.M., "An Introduction to Structured Modeling," *Management Science*, pp. 547-588, 1987.

[6] Han, J., Fu, Y., Wang, W., Koperski, K. and Zaiane, O. "DMQL: a data mining query language for relational databases." *SIGMOD Workshop on DMKD*, 1996.

[7] Imielinski, T., and Mannila, H. "A database perspective on knowledge discovery." *CACM*, 39(11), 58-64, 1996.

[8] Klemetinen, M., Mannila, H., Ronkainen, P., Toivonen, H., and Verkamo, A.I. "Finding interesting rules from large sets of discovered association rules." *CIKM-1994*, 1994.

[9] Konsynski, B. R., "On the structure of a generalized model management system," *Proc. of 14th Hawaii Int. Conf. on the System Sciences*, Vol. 1, pp. 630-638, January 1980.

[10] Kotler, P. Marketing Management, Prentice Hall. 2002.

[11] Krishnan, R. and Chari, K. Model Management: Survey, Future Directions and a Bibliography, *Interactive Transactions of OR/MS*, 3(1), 2000.

[12] Liu, B., Hsu, W., Mun, L., & Lee, H. "Finding interesting patterns using user Expectations," *IEEE Transactions on Knowledge and Data Engineering*, 11(6), p.817-832, 1999.

[13] Meo, R. Psaila, G., and Ceri, S. "A new SQL-like operator for mining association rules," *VLDB-96*, 1996.

[14] Ramakrishnan, and R. Gehrke, J. *Database Management Systems*. McGraw-Hill, 2000.

[15] Sprague, R. H. and H. J. Watson, ``Model Management in MIS", *Proceedings of 17th National AIDS*, 1975, 213-215.

[16] SQL99 standard reference. INCITS/ISO/IEC9075-1 and 9075-2 (2 volumes), January, 1999.

[17] Tsai, Y. Structured modeling query language, Ph.D. Thesis, Andersen Graduate School of Management, UCLA, 1991.

[18] Tuzhilin, A. and Adomavicius, G. "Handling Very Large Numbers of Association Rules in the Analysis of Microarray Data," *KDD-02*, 2002.

[19] Tuzhilin, A., and Liu, B. "Querying multiple sets of discovered rules." *KDD-2002*, 2002

[20] Virmani A., Imielinski, T. "M-SQL: A query language for database mining." *Journal of DMKD*, 1999.

[21] Witten, I.H. and E. Frank, *Data mining: practical machine learning tools and techniques with Java implementations*. 2000, San Francisco: Morgan Kaufmann.

[22] Will, H. J. ``Model Management Systems" in *Information Systems and Organization Structure*, ed. by Edwin Grochia and Norbert Szyperski, 1975 pp. 468-482.