Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
No. TR 12-04

How to implement M4 operators

Timm Euler

Dortmund, March 27, 2003

1 What this document is about

This document wants to provide explanations about the current compiler implementation for those who would like to implement a new operator for the MiningMart system. It explains the internal operator hierarchy of the Java code for the compiler and describes which methods need to be implemented for a new operator. Please also refer to the following document: MiningMart technical report TR12-02, "Compiler Constraints and Operator Parameters" (referred to as TR12-02 henceforth in this document).

2 General issues

In general, the compiler works under the assumption that all metadata of the conceptual level of M4 (concepts, features, relations) are provided before a step is compiled. Also, all metadata of the relational level of M4 (columns and columnsets) exist for the input of the step (of the operator), and the parameters for the operator are listed in the table `Parameter_T`. Compiling an operator means to create the metadata for the output of the step the operator is attached to, and write it into the database.

The compiler code is organized in two packages, `miningmart.compiler` (with subpackages) and `miningmart.compilerInterface`. The latter provides the methods to call the compiler. New operators belong into the package `mining-mart.compiler.operator` or a subpackage thereof.

All M4 objects (e.g. concepts or columns) have corresponding Java classes in the package `miningmart.compiler`. The operators work with Java objects representing the M4 database entries; writing new objects back into the database is done by special methods in the higher classes of the operator hierarchy. Usually, new operators inherit these methods and do not need to write metadata into the database.

All database access is done through the class `miningmart.compiler.DB.java`. During every compiler run, exactly one instance of it exists and is accessible by every operator using the inherited method `getM4Db()`. It provides a number of public methods to read and write SQL commands to the business or metadata schema, such as `executeM4SqlRead()` or `executeBusinessSqlWrite()`. Operators must not open their own database connections.

Every operator has a specification which is part of the M4 model; please refer to the MiningMart technical reports in WP 18, TR18-0x. These specifications must exist in the database to be able to execute an operator. Further, please note that the name of an operator as it is entered into the M4 table `Operator_T` must correspond exactly to the name of the Java class that implements this operator, respecting case (but ignoring the suffix `.java`). If you introduce a new package for your operator, the constructor of the class `Step.java` must be extended in a straightforward way, so that the new package is also searched through to find the new operator.

For output to the screen or a log file, all operators inherit the method `doPrint(int, String)` which expects a verbosity level and a String which is to print if the user-defined verbosity level for the current compiler run is at least as high as the given one. The possible levels of verbosity are provided as static fields in the class `miningmart.operator.Print.java`. The method `doPrint(Exception)` is also available for operators, it prints the message String of the given exception object using the maximum verbosity level.

You may want or need to implement additional classes which are not part of the operator hierarchy, and which therefore do not inherit the methods for printing (for example wrapper classes, see section 4). For them you can access the print object for the current compiler run using `this.getM4Db().getCompiler-AccessLogic().getCasePrintObject()`. This object provides the `doPrint()` methods mentioned before.

3 Operator hierarchy

There are two different kinds of operators, `ConceptOperators` and `Feature-Construction` operators. The first have as output on the conceptual level a `Concept`, the latter a `BaseAttribute` (see TR12-02). Correspondingly, the abstract Java class `Operator.java`, which is the superclass of all operators, has two abstract subclasses, `ConceptOperator.java` and `FeatureConstruction.java`. Both manual and machine learning operators are part of this hierarchy.

The concept operators are divided into two further abstract classes, `Single-CSOperator.java` and `MultipleCSOperator`. The latter is for operators that create more than one `Columnset` for the output concept. At the moment, only the segmentation operators do so; new operators will very likely be subclasses of `SingleCSOperator.java` or `FeatureConstruction.java`. So they create only one `Columnset` for the output concept, or only one `Column` for the output attribute.

The implementation of the creation of Java M4 objects representing the new `Column` or `Columnset` is provided by these superclasses. New operators need to implement all abstract methods that they inherit; these methods will be explained below. In essence, what an operator implementation has to provide is an SQL definition for a new `Columnset` (usually a database view definition) or a new `Column`, respectively. Some more complex operators, such as `MultiRelationalFeatureConstruction` for instance, have a more complex task and override the methods of the superclasses that create the new M4 objects. It is expected that most new operators do not need to do that, which will make their implementation easier.

Rather simple operators to take as an example implementation are `Linear-Scaling.java` for feature construction and `RowSelectionByQuery.java` for a SingleCSOperator. Please note their abstract super classes as well.

4 Special hints for machine learning operators

ML operators call external algorithms to be able to provide their results. These external algorihms must be wrapped by wrapper classes that are part of the compiler. However, the wrapper itself can be part of a different package, for example `miningmart.tools`; only the operator calling the wrapper should be part of the operator package. So for the operator itself, there is not much difference between a manual and a machine learning operator. Yet, ML operators are more complex because the wrapper for the external algorithm must be written.

The result of the external algorithm must be used in the SQL definition of the new `Columnset` or `Column`. For this to be possible, the learned model (for example a decision tree) must be converted into a function that is executable in the database, ie a PL/SQL function. The call to this function provides the SQL definition for the new `Column(set)`. The conversion to PL/SQL is one of the tasks of the wrapper.

An example implementation of an ML operator is the operator `Missing-ValuesWithRegressionSVM.java` which uses the external algorithm *mySVM* by calling (a subclass of) the wrapper class `SVM_Wrapper.java`. Also, `Segmentation-WithKMean.java` is a good example for an operator which is separated from the wrapper it calls.

5 Methods to be implemented by a new operator

5.1 Methods for all operators

It is useful to have parameter-getter methods, which should be used whenever a parameter is accessed. For example, if an operator has a parameter of type `Value` called *SampleSize*, which is expected to have an integer value, a method like `int getSampleSize()` should be implemented. This can be done as follows.

All parameters for an operator are loaded by the superclass `Operator.java`, which uses the information in the table OP_PARAMS_T to be able to do so (see MiningMart technical reports TR18.1 and TR18.2). In the subclasses, parameters are accessed by calling the inherited methods `getParameter(String name)` and `getSingleParameter(String name)`. The first is for lists of parameters (see MiningMart technical report TR12.2) while the latter is for single parameters. Special attention is needed for FeatureConstruction operators, because they are loopable and the set of parameters is different for each loop. The number of loops can be found by calling the inherited method `int getNumberOfLoops()`; in the case of loopable operators, the method `get(Single)Parameter(String name, int loopnr)` must be called, using the inherited method `int getCurrent-LoopNumber()` to find the right value for `loopnr`.

The inherited methods to get a parameter return an M4 object like `Value` etc which comes directly from the database. You can now convert this object (for example into integer) and check if it is in the right range.

Note: Several of the so-far implemented operators load their parameters themselves, in a method called `load()`; however, this is part of an older implementation. It still works, but for new operators, the new mechanism should be used because it is simpler. Example implementations of the new mechanism can be found in the operators `MultiRelationalFeatureConstruction.java` (without loops) and `LinearScaling.java` (with loops).

5.2 Methods for FeatureConstruction operators

5.2.1 String generateSQL(Column targetColumn)

This method returns the SQL definition of the new virtual `Column` as a Java String. The given `targetColumn` is the column that belongs to the target attribute of this operator (every feature construction operator has a parameter called *TheTargetAttribute*; see TR12-02). During execution, make sure that the right set of parameters is accessed with respect to loops (see section 5.1).

5.3 Methods for SingleCSOperators

5.3.1 String generateSQLDefinition(String selectPart)

This method returns the SQL definition of the new `Columnset` as a Java String. The given String `selectPart` is the String to be placed between the `SELECT`- and the `FROM`-part of the resulting definition if it is a definition for a database view. If this operator does not create a view, but a table, then the method `String generateColumns(Columnset csForOutputConcept)` in the super class `ConceptOperator.java` must be overridden. Then the String `selectPart` may be ignored by this method (as it will be `null` anyway), and the SQL definition for the `Columnset` is equal to the name of the created table. The usual case is that an operator creates a view; therefore, all that is needed here is to provide the parts after `FROM` and, optionally, `WHERE`. A simple example is `RowSelection.java`.

5.3.2 String getTypeOfNewColumnSet()

This method must return the String "V" if this operator creates a view, and "T" if it creates a table.

5.3.3 boolean mustCopyFeature(String featureName)

If the super class method `String generateColumns(Columnset csForOutputConcept)` is not overridden, it copies every column from the input concept to the output concept for which a) there is a feature in the output concept and b) this method returns `TRUE`. This method provides the option to operators to exclude this copying for certain features, using their name to decide; however, in general operators should simply return `TRUE` here, unless there are specific purposes. If `generateColumns(...)` is overridden, this method (`mustCopyFeature(...)`) is never called.

5.4 Methods for MultipleCSOperators

5.4.1 int numberOfColumnSets()

This method must return the number of `Columnsets` this operator is going to create. If this number is the result of an external algorithm, like in `SegmentationWithKMean.java`, the algorithm must be called first before this value can be returned. The number of `Columnsets` to be created may also be specified by a parameter of this operator; then this method simply returns the value that was loaded in the load method.

5.4.2 String generateSQLDefinition(String selectPart, int index)

This method is similar to the one for SingleCSOperators (section 5.3.1). The given `index` is the number of the `Columnset` that the newly created SQL definition will be used for. It runs from 0 to $numberOfColumnSets() - 1$.

5.4.3 String getTypeOfNewColumnSet(int index)

This method must return the String "V" if the new `Columnset` with this index is a view, and "T" if it is a table.

5.4.4 boolean mustCopyFeature(String featureName)

See section 5.3.3.