Enabling End-User Datawarehouse Mining
Contract No. IST-1999-11993
Deliverable No. D7b

Description of the Metadata-Compiler using the M4-Relational Metadata-Schema

Regina Zücker[1]

Swiss Life, CC/ITRD
Information Technology Research & Development
CH-8022 Zürich, Switzerland
`regina.zuecker@swisslife.ch`
`http://research.swisslife.ch`

June 22, 2001

Abstract

In this part b of the deliverable, the metadata compiler, short MD-Compiler, is described. It is responsible for generating executable SQL-code from the information stored within the M4-Relational Metadata-Schema. The paper is structured into two parts.

The first part describes the actual state of the compiler. It explains the general functionality of the MD-Compiler, the basic ideas of the underlying concept and how much is realized up to now. In this version the compiler can 'only' generate one manual preprocessing operator at a time. It is still an open question how the process will look like for handling a whole preprocessing chain and which part of the system will have the control over that.

The second part gives some ideas on compiler-optimization and how a preprocessing chain could be handled. These topics have to be solved in the continuing work package WP7+. Also, some open questions are described. Even if this part specifies future work, it's important to embed the current state into a complete picture of the compiler functionality.

Chapter 1

Actual state of the MD-Compiler

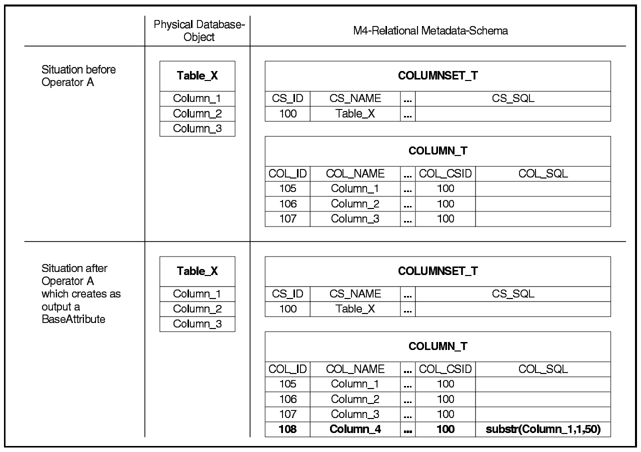
1.1 General functionality

Today the MD-Compiler is responsible for handling manual preprocessing operators. It first reads the metadata for an operator from the database, creates executable SQL-code from that, compiles the SQL-code and writes the output-results as metadata back into the M4-Relational Metadata-Schema. The MD-Compiler works with the implementational level of the M4-MetaModel. Because the parameters of an operator are defined with objects of the conceptual level (the case designer is doing this) the MD-Compiler must convert these objects to the implementational level. The results of an operator, which are written back, belong only to the implementational level.

Considering the list of manual preprocessing operators, which shall be available within the Mining Mart system, they can be divided into two groups: (1) operators which have as output a BaseAttribute and (2) operators which have as output a Concept.

1.1.1 Operators with a BaseAttribute as output

For manual preprocessing operators, which have as output a BaseAttribute, this BaseAttribute belongs to the input concept of the operator. So the MD-Compiler must 'create' a new column for an existing column set. In case the column set is a database table, this would mean to physically add a new column to the table. Changing such an operator respectively the output BaseAttribute of that operator would mean to copy the table physically into a temporary table, delete the original table and create a new table with the changed column. Considering database performance, this process is not acceptable. Therefore the compiler creates always a 'virtual column' by only generating a sql-string describing this new column. This sql-string is stored



	Physical Database-Object	M4-Relational Metadata-Schema
Situation before Operator A	Table_X: Column_1, Column_2, Column_3	COLUMNSET_T; COLUMN_T
Situation after Operator A which creates as output a BaseAttribute	Table_X: Column_1, Column_2, Column_3	COLUMNSET_T; COLUMN_T

Situation before Operator A:

Table_X

Column_1
Column_2
Column_3

COLUMNSET_T

CS_ID	CS_NAME	...	CS_SQL
100	Table_X	...	

COLUMN_T

COL_ID	COL_NAME	...	COL_CSID	COL_SQL
105	Column_1	...	100	
106	Column_2	...	100	
107	Column_3	...	100	

Situation after Operator A which creates as output a BaseAttribute:

Table_X

Column_1
Column_2
Column_3

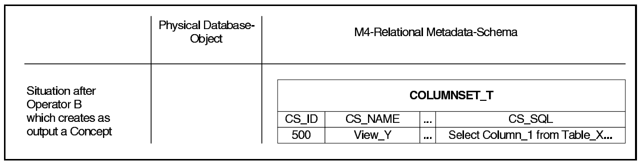
COLUMNSET_T

CS_ID	CS_NAME	...	CS_SQL
100	Table_X	...	

COLUMN_T

COL_ID	COL_NAME	...	COL_CSID	COL_SQL
105	Column_1	...	100	
106	Column_2	...	100	
107	Column_3	...	100	
108	**Column_4**	**...**	**100**	**substr(Column_1,1,50)**

Figure 1.1: New 'virtual column'



	Physical Database-Object	M4-Relational Metadata-Schema
Situation after Operator B which creates as output a Concept		COLUMNSET_T

COLUMNSET_T

CS_ID	CS_NAME	...	CS_SQL
500	View_Y	...	Select Column_1 from Table_X...

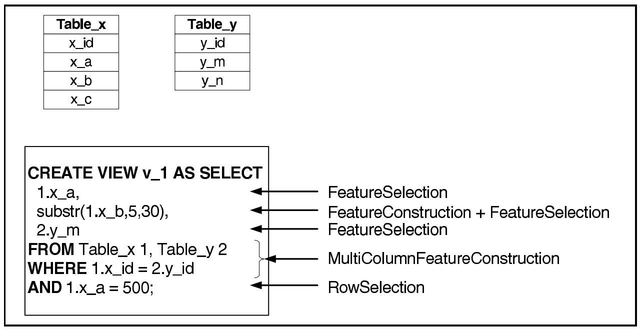
Figure 1.2: New 'virtual view'

in the M4-Relational Metadata-Schema. The existing database table is not changed. Figure 1.1 illustrates this process.

1.1.2 Operators with a Concept as output

Operators which have as output a concept are faced with the same problem of how to handle physical database objects. Imagine the case designer defines a feature-selection operator with 5 base attributes. After executing this operator he wants to re-execute this operator, but now only choosing 4 base attributes. If the output concept would be a physical database table, it would have to be deleted and re-created. As solution for this problem, the MD-Compiler always creates a view-object. The database view-object does not have own data so changing attributes of a view is no problem. The MD-Compiler stores the view-definition as sql-string in the M4-Relational Metadata-Schema. We call this a 'virtual view'. Figure 1.2 illustrates this process. As far as we know today a physical view-object is still necessary for some reasons which are explained in section 2.1.2.

We are aware of the problem that some manual preprocessing operators need a physical database table, i.e. RANDOM_SAMPLING. Because this operator is not realized yet, we cannot present THE solution. But for such cases the MD-Compiler will probably create a physical database table in the



```
CREATE VIEW v_1 AS SELECT
  1.x_a,
  substr(1.x_b,5,30),
  2.y_m
FROM Table_x 1, Table_y 2
WHERE 1.x_id = 2.y_id
AND 1.x_a = 500;
```

Figure 1.3: Example illustrating operator granularity

background and use it. The case designer is not aware of this process.

The general idea of the MD-Compiler is to create a 'virtual view' whenever possible and the case designer has not defined something else. It seems that using virtual views create the least problems during re-creation with changing attributes.

Using SQL-definitions instead of physical database view-objects results in better performance on the database server during execution of hierarchical views. Details are explained in section 2.1.

1.2 Operator granularity

As described in the M4-MetaModel of Deliverable 8&9, a manual preprocessing operator is "very elementary". Compared to the power of the SQL-engine, one preprocessing operator only needs very little functionality of it, however several operators are necessary one after another. We want to illustrate this circumstance on a small example.

Figure 1.3 shows two tables and a sql-statement, which should be modeled with the M4-MetaModel. To do this, several preprocessing operators are necessary.

- 1. RowSelection
  CREATE VIEW V_01 (x_id, x_a, x_b, x_c) AS
  SELECT x_id, x_a, x_b, x_c FROM Table_x WHERE x_a = 500;

- 2. MultiColumnFeatureConstruction (needs a relation)
  SELECT x_a new_a FROM V_01, Table_y WHERE x_id = y_id;

- 3. MultiColumnFeatureConstruction (needs a relation)
  SELECT x_b new_bFROM V_01, Table_y WHERE x_id = y_id;

- 4. MultiColumnFeatureConstruction (needs a relation)
  SELECT y_m new_m FROM V_01, Table_y WHERE x_id = y_id;

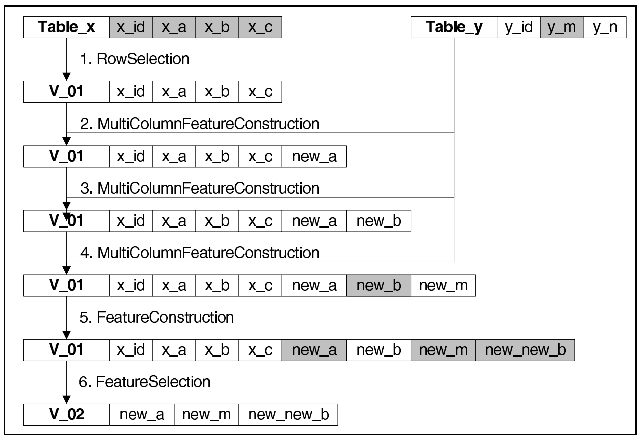


Figure 1.4: Operator-outputs for the example

- 5. FeatureConstruction
  SELECT substr(new_b, 5, 30) new_new_b FROM V_01;

- 6. FeatureSelection
  CREATE VIEW V2 (new_a, new_new_b, new_m) AS
  SELECT new_a, new_new_b, new_m FROM V_01;

Figure 1.4 shows as results the virtual columns and virtual views which are generated by the different preprocessing operators. The greyed columns are used as input parameter for one of the next operators. For example the column 'new_b' is used to create the new column 'new_new_b' of operator number 5.

Operator 1 creates a new virtual view 'V_01', operator 2, 3, 4 and 5 create a new virtual column 'new_a', 'new_b', 'new_m' and 'new_new_b'. Operator 2 again creates a new virtual view 'V_02'. This example demonstrates that 6 elementary preprocessing operators and 2 view-definitions are necessary to model the relative simple select-statement of figure 1.3.

The experience of Deliverable D6.2, including the prototype version of the implemented compiler, have shown that in reality select-statements will be much more complicated than the one of the example. Therefore optimization is a very important topic when specifying the compiler. Section 2.1 gives details on the optimization part.

1.3 What is realized?

The MD-Compiler is realized with the programming language JAVA. Two versions are available, one runs outside the database and connects via a JDBC-ODBC-Bridge, one runs inside the database server with a PL/SQL-wrapper around every java-class.

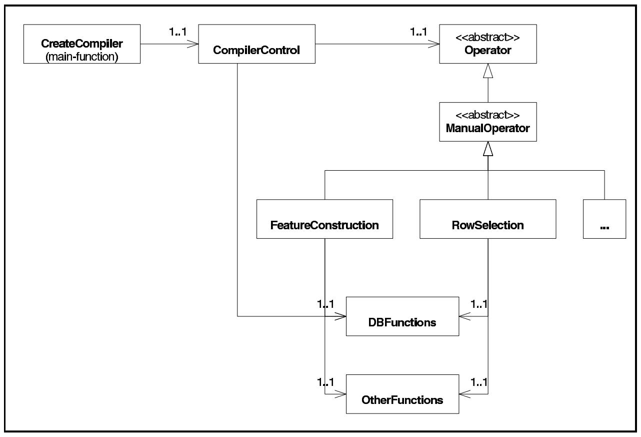


Figure 1.5: Simplified UML-class-model

1.3.1 Class model

The inheritance of the M4-MetaModel is adopted. So an abstract class for 'Operator' and 'ManualOperator' exists. The 'ManualOperator'-class defines the necessary process for every manual preprocessing operator. Every concrete manual preprocessing operator is implemented as own class which inherits from 'ManualOperator'. Right now the operators 'RowSelection' and 'FeatureConstruction' are implemented. For feature construction a database function must exist which makes the actual data transformation. Figure 1.5 shows the simplified UML-class-model.

The class 'CreateCompiler' is only necessary when using the database-outside running version of the MD-Compiler. This class makes the connection to the database.

The class 'CompilerControl' checks, if the operator is of type manual, creates an operator instance according to the operator name and then starts the execution process.

The class 'DBFunctions' handles all actions with the database, the class 'OtherFunctions' generates all sql-statements.

1.3.2 Execution

To start the MD-Compiler for an operator, only the unique operator-id and a manual-operator-flag are necessary (API-Call to start outside-database running compiler : **"java CreateCompiler database-name user-name password operator-id true-flag"**) . According to the information read from the database, the proper operator instance is created.

All parameters for an operator are loaded and stored as class attributes; conceptual as well as implementational information. When generating the sql-statement, all information is available within the class. No further load from the database is necessary.

After the sql-statement is generated, a syntax check is done within the database.

Last, the results, according to the output parameter of the operator, need to be written into the M4-Relational Metadata-Schema. For an output-BaseAttribute an insert in table COLUMN_T is necessary, for an output-Concept one insert in table COLUMNSET_T and an insert in table COLUMN_T for every column of that column set are required. Also the references to the conceptual level have to be set.

Chapter 2

Outlook for WP7+

2.1 Optimization

Optimization is THE central topic if the compiler shall run on real datasets like data warehouses. The example of figure 1.3 have shown that many elementary preprocessing operators are necessary to model a complex select-statement. So the challenge for implementing the compiler is to transform the preprocessing operators into optimized sql-code which results in good database performance and execution time.

2.1.1 View Hierarchy

As explained before, the MD-Compiler normally generates views respectively virtual views. Therefore applying several operators means creating a view hierarchy.

Figure 2.1 shows such a view hierarchy. Imagine, Table_x, Table_y and Table_z are physical database tables which content real data to be mined, e.g tables from a data warehouse. We call these base tables. All generated views get data from these base tables ('vc_' means in this picture virtual column). We expect that for a complete preprocessing chain much more than 6 views are necessary until the last view contents the mining data.

Discussions with experts from Oracle about the view hierarchy and performance aspects got following result:

It should be much better to use sql-statements for view-definitions, so called inline-views, instead of physical view-objects. Apparently the Oracle optimizer is able to re-write sql-statements and optimize it when inline-views are used. All inline-views only reference to the base tables, so the optimizer can generate one complex sql-statement from it. For actual existing view objects this is not possible.

How do the inline-view-definitions look like for the example of figure 2.1?

- V_01

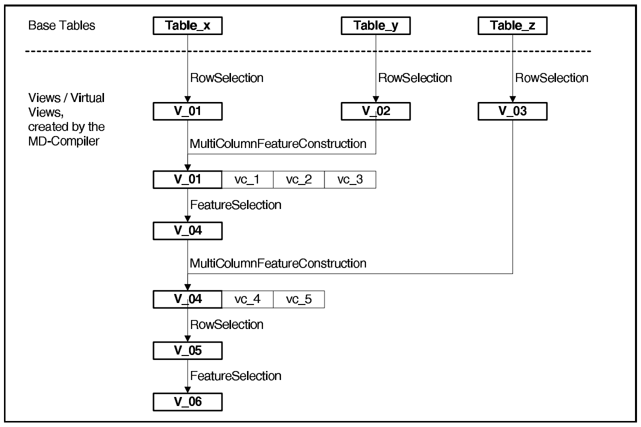


Figure 2.1: View Hierarchy

SELECT ... FROM Table_x

- V_02
  SELECT ... FROM Table_y

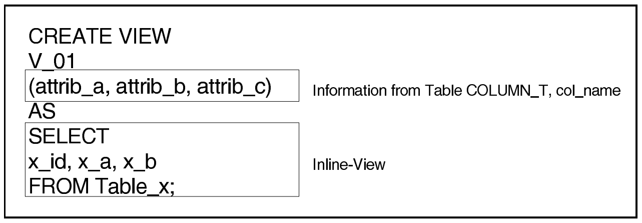
- V_03
  SELECT ... FROM Table_z

- V_04
  SELECT ... FROM (SELECT ... FROM Table_x)
  instead of
  SELECT ... FROM V_01

- V_05
  SELECT ... FROM (SELECT ... FROM (SELECT ... FROM Table_x))
  instead of
  SELECT ... FROM V_04

- V_06
  SELECT ... FROM (SELECT ... FROM (SELECT ... FROM (SELECT ... FROM Table_x)))
  instead of
  SELECT ... FROM V_05

Unfortunately this concept is not tested on large data sets yet. So we cannot proof that it is really faster than using physical view-objects. But as far as we understood the Oracle optimizer it should gain database performance.



```
CREATE VIEW
V_01
(attrib_a, attrib_b, attrib_c)
AS
SELECT
x_id, x_a, x_b
FROM Table_x;
```

Figure 2.2: View and inline-view

2.1.2 Physical database views

In section 1.1.2 we stated that the MD-Compiler generates virtual views which are inline-views and additionally creates physical view-objects on the database. This seems redundant. But inline-views do not carry all necessary information for reading one specific column of a view. Missing is the reference from a BaseAttribute to a Column, which is necessary when:

- the case designer wants to see data contents of one specific BaseAttribute
- statistics are calculated

Inline-views exist only as definitions for all columns of a view. The assignment of one specific column-name to the select-part of the view, which is the inline-view, is done in the create-view-statement. Figure 2.2 explains this circumstance.

The inline-view is only the select-statement to load all data. The reference to a specific column is given by the order of the view-column-name and the select-column-name, e.g. attrib_a gets data from the select of x_id, attrib_b from the select of x_a and so on. Attrib_a is the specified column-name from the M4-Relational Metadata-Schema. So only through a physical view-object data contents for a specific column can be loaded.

2.1.3 Materialized views

Materialized views exist since version Oracle 8i. In Mining Mart we want to use them to gain additional database performance.

The main functionality of materialized views is like former snapshots. They store data physically in the database with an automatic refresh modus. The big advantage is, that the Oracle optimizer recognize the existence of a materialized view in the background and uses it when it is advantageous for executing a special sql-statement. Nowhere in the sql-statement the name of the materialized view has to be defined.

In general it is most efficient to read data directly from a physical database table. But the disadvantages of physical database tables within

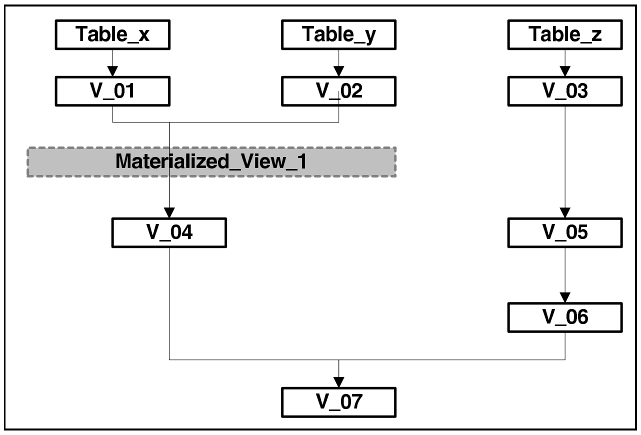


Figure 2.3: Materialized View

Mining Mart we have already explained. So materialized views seem to be a good solution. They store data on an intermediate level but do not appear in the M4-Relational Metadata-Schema. They are totally in the background and so it's no big problem if columns are changing. Figure 2.3 shows an example. The view-definition and inline-view-definition for V_04 stay unchanged. But when selecting data from V_04 the Oracle optimizer recognizes automatically that it can use the Materialized_View_1 instead of Table_x and Table_y, which should result in better execution time.

The MD-Compiler should generate materialized views, also in the background. Unsolved is the question of the best time and best data content for creating materialized views as well as some functionality for deleting old materialized views.

2.2 Executing a complete preprocessing chain

Our idea is that before a complete preprocessing chain is executed, it should be recompiled completely. That means every view and inline-view should be re-generated to ensure data actuality.

Learning operators are also allowed as preprocessing operators. Often they cannot handle large data sets so they should be started with a data sample, which could be automatically created by the MD-Compiler.

The learning operator itself must write it's results as metadata into the M4-Relational Metadata-Schema. So after executing a learning operator the MD-Compiler must be able to generate a corresponding manual operator using these results. The corresponding manual operator is necessary for handling the complete large data set.

When the case user starts executing a preprocessing chain, no further interaction should be necessary, except an error occurs.

It's not clear yet which part of the Mining Mart system has the control over this process.

2.3 Open questions

- Optimization
  The already implemented part of the MD-Compiler is not tested on large data sets yet. Also all ideas described in section 2.1 are not tested in reality. So we don't know how easy it can be implemented and if it really brings the expected performance.

- Embedding learning operators
  It's not totally clear how a learning operator can be embedded in the process of automatically executing a complete preprocessing chain as well as how and which metadata have to be written back as results into the M4-Relational Metadata-Schema so that the MD-Compiler is able to generate a manual preprocessing operator for that.

At the 9th/10th of July, 2001 an internal Mining Mart workshop will take place. These open questions will be discussed there with all project partners and maybe we can find already a solution.