Metadata Driven Data Transformation

Petr Aubrecht
and
Zdenek Kouba
{aubrech,kouba}@labe.felk.cvut.cz
Czech Technical University
Technická 2
Prague, 166 27, Czech Republic

Abstract

The bottleneck of a data warehouse implementation is the ETL (extraction, transformation, and load) process, which carries out the initial population of the data warehouse and its further (usually periodical) updates. There is a number of software products supporting the OLAP analysis. However, the ETL process implementation is not repeatable in a significant way.

This paper reports on a research of a model-based data transformation applicable to data warehouse population and updates. The ETL process is based on a metadata repository, which contains data models of the data sources, the target data warehouse model, and the correspondence among them.

Keywords: ETL, Data Transformation, Metadata, Data Warehouse, Scripting Language, Data Pump, Data Mining

1 System overview

State of the art in metadata development is focused mainly on declarative metadata. The nature of desired system behavior requires to have both declarative and procedural metadata.

The metadata repository is a core of the whole system. It has a form of a tree, in which all metadata is stored. The two branches leading from the root represent declarative and procedural information, as described in the section 2.

The procedural part of metadata is supported by proprietary scripting language called Sumatra, developed specially for this purpose. Sumatra is a Java-like language capable to deal with the metadata repository and to access data sources. The data transformation system uses Sumatra's interpreter, assembling parts of a source code (possibly) just before the desired transformation. For details see the section 3.

2 Metadata

All information in the system is stored in the metadata repository. Its tree structure allows to separate declarative information from the procedural one.

The declarative part of metadata describes the data models of all data sources. Data sources are determined by their name, type, and a list of fields (columns, attributes). All data sources are listed in a branch called **Datasources**.

The procedural part of metadata is intended to store transformation procedures. They are stored in the branch **Transformations**. The information stored in this branch is used to prepare the transformation task, i.e. before the transformation starts.

Metadata itself can be stored in the standard data interchange format XML. If so, the information can be shared with other data transformation/preparation tools.

2.1 Data Sources

For the purposes of the presented system, the data source is represented by a table with a fixed number of fields (columns, attributes).

All data sources have three attributes in common—name, type, and a list of fields.

`Name` represents an identifier, which is used by transformation scripts. This concept is similar to aliases in database access (ODBC, BDE).

`Type` determines access objects. For each registered type there exists an object in the system, which provides the access to the data source. This object is responsible for all operations—getting a record, opening, and closing the data source, navigating in the dataset.

Various data source families have been taken into account. At the moment, the SQL data sources, files with the comma-delimited text format, and files with fixed-sized records are supported. Further data source types (e.g. XML or simple Prolog facts) can be added easily. The system provides internal virtual data source, which has to be inherited (it is an C++ class). The virtual class provides basic behavior and

its simple descendant is able only to obtain a text line representing a record.

`List of fields` represent the fields (columns) of the data source. The fields are typed. There exist standardized data types. The standard field types are Integer, Number, String, and DateTime. New types can be introduced by data source objects. A set of attributes can be attached to each particular type.

For example DateTime has a Format attribute, which is a string containing detailed format of the date and time. System offers a set of functions processing standard attributes to simplify introducing new types of data sources with full formatting capabilities.

Attributes of fields are very important for special cases. National formats of date and time or of writing real numbers (using comma instead of point) etc. make troubles in most products or (for example—ODBC and BDE) database drivers, which may not be able to process them. Our data sources use all the offered functions and can be simply customized by making use of these attributes.

2.2 Transformations

Transformations represent the procedural part of metadata, as was mentioned above. The basic principal is similar to data sources. The corresponding branch in the metadata tree is called Transformations, under which all transformations are listed.

Transformations have two obligatory attributes—the name and the template.

Name acts as an internal identifier.

Template is either a name of a script or a name of a template. Both are scripts in written in the Sumatra language, possibly enriched with special macros. Before the script is started, the macros are expanded into regular Sumatra code.

Optional attribute **desc** contains a description of the transformation in a natural language for better readability.

Parameters to templates are stored in a vector attribute denoted **Params**. All parameters necessary to customize the template are stored in it.

Templates and their language are described in the next section in detail.

3 Sumatra

Sumatra is a language intended for procedural tasks in the described data transformation system. It is proceeded by the built-in interpreter equipped with a preprocessor.

The preprocessing part of Sumatra looks for all known macros and replaces them by their expansion. This process is done in a fixed order given by the system. It allows making use of macros as arguments of others without the need to parse the text in a more complicated way. The replacement is done purely textually, independently on the location of the macro. Thus, macros are expanded in comments. It allows for example to concatenate a customizable code into a legal Sumatra identifier.

The grammar part of Sumatra analyzes the pure Sumatra code (after preprocessing) and constructs execution structures. The execution of these structures is fast enough. Overwhelming majority of time is spent in data source access methods. As tests have shown, time consumed by executing Sumatra code is under 1% of the total time of a transformation task.

3.1 Language

Sumatra language has been developed as a scripting language, with emphasis given to a simple usage from programmers point of view, simple parsing and easy integration into a data transformation system.

Sumatra meets all these requirements. It has been motivated by C and Java syntaxe. It has all commonly known programming structures as assignment, all three types of cycles, and the conditional statement. It supports very simple form of procedures for repeatedly used parts of code.

Sumatra language has a built-in set of objects and functions. They are divided into two parts.

First group of objects and functions is intended to access the data sources. These objects can fully control all methods and properties of internal objects. They support also the independence of the data access on the particular data source type. All data sources are accessed via a standardized interface and all of them have the same behavior. From Sumatra point of view all the data sources seem to be equivalent.

Second group of objects and functions support formatting, displaying messages etc.

A short simplified example of a Sumatra code follows:

```
string report="";
ASTDataObjectPtr source;
source.Open();
int cont = source.FirstRecord();
while(cont)
{
  report = report +
    source.GetFieldByName("description")
      .AsString+"\r\n";
  cont = source.NextRecord();
}
source.Close();
Message(report);
```

3.2 Templates

One of the most important features introduced in this system is making use of templates. Templates are skeletons of Sumatra scripts. These scripts are written for specific purposes.

Templates are intended to be reusable. As soon as a template is ready, it should be general enough to cover as wide area as possible customizing only a specific behavior.

The customization of templates is usually done by parameters. Parameters are properties of a particular transformation and they are referenced from templates.

3.3 Parameters

As templates provide only a basic structure of the final script, the additional information is stored in transformation's parameters. Any occurence of the `##param(name)` macro in the template will be replaced by the value of the parameter. Let us have for example a part of a script `int i=##param(From);` and a parameter From="0". This part of the script will be expanded into `int i=0;` This is the way templates utilize parameters. The same templates used in different transformations with different parameter values will lead to different behavior of the final code.

So far we presented only the basic way of making use of templates. A lot of parameters is often needed when developing a template. However, they have usually the same value in most occurences. Using default values allows to change the default behavior only when necessary. The example from the previous paragraph can be changed to `int i=##param(From,0);` If transformation parameters don't contain the attribute From, 0 will be used in the resulting code. If present, the given value will override the default one.

Very often bits of Sumatra code are used as parameters. However, it is inconvenient to store large texts in metadata. They are usually saved in external files. For these reasons parameters have the ability to be stored outside metadata repository. It is done by means of concatenating the name of the particular parameter with the string "File". This means that the particular occurence of the parameter will be replaced by the contents of the file specified by the parameter value rather than by the value itself. Very often we can meet the following example in real life data transformation templates. Let there is an optional parameter `code`, referenced by the `##param(Code,)` macro in the template. Providing the parameter CodeFile="proc.sum" will cause that the macro will be replaced by the content of the file proc.sum rather than by the proc.sum string itself.

Last presented feature simplifies calculating characteristics like minimum, maximum, count, sum, local minimum, etc. Combining the AnalyzeTable template and Calculate parameter will automatically generate calculations into the code. These calculations have a special branch from the root of metadata tree. It allows to share prepared calculations by various tasks. Prepared calculations are written in a special manner. After some processing they are put into the final script.

Calculations are listed in the vector parameter Calculate. It contains a set of vector nodes. Their names determine calculations—the system looks for a definition with the same name. A content of these nodes is a list of (usually) fields, which have to undergo the calculation. In this way a large set of calculations can be done in a simple way—only by listing them in a chosen node.

3.4 Macros

Macros play two roles in this system. All macros start with `##`, followed by a name—only letters. A list of parameters is in parenthesis.

The aim of the first group of macros is to access transformation parameters.

The most important macro is `##param(name)`. It takes the parameter with the given name and replaces itself by it. All parameters are processed as string.

Another macro is `##Calculate(ds,part)` for processing calculations.

The second, larger group of macros is intended to access the data source information. These macros usually create a list of fields and carry out some operation on it.

As a representative of the second group the `##DSCopyMatchingFields` macro can be chosen. It does assignments for all pairs of fields with compatible data types, where the fields may belong to different data sources.

Macros for accessing data sources also provide a parameter for history handling. Specifying the history with a value greater than zero leads to storing given levels of records in memory. These records can be retrieved easily. The history can be used e.g. for determining the differences between subsequent records.

3.5 Basic Templates

There is a basic set of templates already developed. It covers most of common transformation scenarios.

Most of transformations is done by reading a record from one data source, making some calculations and saving results into another data source repeatedly until the end of the first data source has been reached. This is exactly what the **TransformTable.sum** template does. Its most important parameters are From and To determining chosen data sources and Code specifying the calculations.

In an older version, there was the **CopyTable** template, but it has been removed and it has been replaced by the TransformTable one with Code containing the text `##DSCopyMatchingFields( ##param(From), ##param(To))`.

Sometimes, it is needed to process several input records and then to process the output or to generate more output records from one input record. The **TransformTable1toN.sum** template allows it.

For easy analysis of processing and saving there is a template **AnalyseTable.sum**. It is similar to the TransformTable one (with save part in a procedure) extended with macros making the analysis. It is prepared for simple calculation of characteristics of a large number of fields.

3.6 Running Script

Sumatra script (transformation) can be run in three ways.

The usual way is the most complex one. First of all, metadata is read from the repository, then a transformation is selected, a template file name is determined and the file is loaded and preprocessed. A complete script is generated. (At this step the script can be saved for the later usage.) At the end the script is executed by Sumatra interpreter.

When the transformation is prepared without use of templates or it has been saved after instantiating the template (as in the previous paragraph), there is no need for template processing. Because templates usually depend on metadata, they require to load the repository before processing.

The simplest processing can be done with raw scripts. This kind of scripts can run without large demends on the system. The script itself can handle the repository freely. It can be successfully used to handle the metadata itself, for example to change the information on the data source if monthly acquired file names change according to the date.

4 Accompanying Tools

In the beginning of the system implementation all the metadata and scripts have been manually typed into raw text files. This is the fastest way for experts, but unacceptable for common users. For non-experts a set of supporting tool has been developed.

4.1 GUI

A graphical user interface has been designed and implemented to simplify the preparation of transformations. The program allows to arrange simply the transformations by selecting data types in a list box, dragging lines representing data flows between desired data sources and supplying additional information.

This environment is written in Java. Thus, transformations can be prepared on any platform. The whole system has been designed as portable between platforms. However, so far it has been exhaustively tested under Windows NT/98/2000. Its portability to Linux platform will be tested later.

4.2 DBreport

DBreport is an importer of data sources. It navigates the user through the process of creating the data source description, automatically recognizing data types. It has one part for SQL, simply getting information relational databases.

The second part is indented for analyzing of text files. It is able to recognize delimiting char and types of fields. It also supports attributes—usually name and type.

4.3 RepEdit

The last tool is a low-level tool allowing to edit metadata in a graphical way (in a tree representation). On the other hand it can be used for demonstration of metadata structure.

5 Most Important Advantages

The whole system has many important features, which are difficult to be explained in the retsricted extend of this paper. For this reason a list of most interesting advantages for regular users is provided.

5.1 Graphical GUI

Graphical GUI is an intuitive tool guiding the user through the data transformation design. It is the usual way provided by most of current data base administrating tools. In contrary to most of such tools, capabilities of which finish at this level, capabilities of the presented system have to start here, otherwise the full power of the system could not be exploited.

5.2 Data Source Aliases

The user doesn't care about the data source type. The processing runs in the same way both for SQL databases and delimited text files. The only difference can be the speed, as an optimalized reading of text file will be much faster than accessing JDBC source through TPC/IP network protocol.

5.3 Templates

Although there exist only three basic templates, it is not needed to increase their number at the moment.

The reason is that they provide enough power for most of transformations.

If there a need arises to create a new one, there is no problem to do it. A new template can be simply copied into the template directory and then it can be referenced by transformation metadata and freely reused.

5.4 Reusability

Also small parts of code can be reused. A good example of this need is the preparation of a training and testing datasets in data mining tasks. After debugging a script processing the training dataset, the same script (saved in a file out of the metadata repository) can be used for processing the testing dataset. Both transformations for both training and evaluate datasets are the same except used data sources.

5.5 Extensibility

A lot of prepared ways of extensions is provided by the system core: a virtual data source for implementing new data sources, a uniform work with macros for creatimg new macros, Sumatra language is also prepared for including new classes and functions (continuously extended according to the needs). All these ways increase utilizability of the system into any desired level.

However, it is possible to extend basic ideas even without accessing the source code and compiling it. For example there can be systems accessing XML based metadata files and generating Sumatra scripts or fragments of scripts. A relevant study has been already started with promising intermediate conclusions.

6 Conclusion

The system has been successfully tested and used in several real life applications for both data transformation and data analysis tasks. It allows a quick and simple preparation of data transformations thanks to a wide range of formatting capabilities and to the power of Sumatra language and its macros.

The well designed structure allows to develop other tools with additional features. These tools can for example create parts of Sumatra code, which will be included into full transformation code afterwards.

Further development will affect also the set of available data sources and the set of macros including external control in both cases.

Acknowledgement

The work presented in this paper has been supported partially by the Ministry of Education of the Czech Republic under Project No. LN00B096 and partially by the INCO COPERNICUS 977091 Geographical Information On-line Analysis (GOAL) research project.

References

[1] Petr Aubrecht. Sumatra Basics. Technical report GL–121/00 1, Czech Technical University, Department of Cybernetics, Technická 2, 166 27 Prague 6, December 2000.

[2] Z. Kouba, K. Matoušek, P. Mikšovský, and O. Štěpánková. On Updating the Data Warehouse from Multiple Data Sources. In *DEXA '98*. Vienna, Springer-Verlag, Heidelberg, 1998.