MIPMap

General Description and Functionalities

MIPMap is a schema mapping and data exchange tool. It is based on the ++Spicy project[1]; it has extended most of its features while replacing some of its functionalities to fit the specifications of the Human Brain project.

The tool offers an easy to use graphical interface where the user, given a source and a target schema, can define the correspondences between them, as well as the join conditions, constraints and functional transformations by simply drawing arrow lines between the elements of two tree-form representations of the schemata.

It has been extended to take as input csv files that correspond to the tables of a database, additionally to relational input (connection to a database). Additional instances from csv files can be loaded to either of the schemata. Moreover, a new feature that supports the preprocessing of a table has been added. More specifically, given a table as csv file input, the user is given the option to choose some of its columns to un-pivot.

The tool supports simple 1:1 correspondences between elements, as well as n:1 correspondences with complex transformation functions and the assignment of constant values or function generated values. Selection conditions can also be applied.

As far as the data exchange process is concerned, core solutions, i.e. “optimal” ones, are generated, taking into consideration source and target constraints (either in the form of foreign or primary keys) and functional dependencies between elements that the user has provided.

The main difference between ++Spicy and MIPMap is in the approach of translating instances, even for large scale data. MIPMap is equally focused on data exchange as well as schema mapping. Its predecessor offered high performance during the data exchange process when the source schema had a few instances, however in cases when the data was increased to a class of some thousand rows per table the time and memory requirements grew exponentially. Thus, it was considered not suitable for such operations. As an alternative the ability to run the generated scripts on a local Postgres database was integrated into MIPMap. Moving the task to a local database eliminates the need for immense memory size while at the same time offers many potential future extensions of the tool taking advantage of the utilities that Postgres provides.

In order to achieve this, a user-defined database is created –if it doesn’t exist already- upon startup. The appropriate schemata on the local database are created each time a new mapping task is loaded and are deleted when no longer needed. The only requirement is that the user has installed Postgres on the machine MIPMap is running and has provided his credentials on the connection properties file.

Finally, MIPMap offers users the ability to export the translated instances to a new csv or json file or append them to an existing one. Primary key constraints that will likely be violated during the data exchange process are also identified and the instances that could not be loaded to the target database are exported separately.

Basic structure – Design patterns

MIPMap is a desktop application that has been developed in Java (Java 8 release). The project has been created as a module suite with its components and external libraries each being a separate module. The modules are related to each other and all of them are built together to form the whole project.

The application follows the model-view-controller (MVC) architectural pattern, isolating the application logic from the user interface layer.

The core of MIPMap is the 'mipmapEngine' module, which contains the classes that generate the rules responsible for the mapping between the source and target schema and produce the scripts for the data exchange process. It also implements the connection to the database layer. It will be described in detail subsequently.

The Netbeans platform has been used as a basis for the tool's graphical user interface with the main actions being implemented as Callable System Actions. The modules that constitute the interface and handle the user input and responses are the 'mipmapGUI' and 'mipmapGUIcommons' modules, the former of which is the second most important component of the project (after its engine module).

The 'mipmapGUIcommons' module contains some basic classes for the interface functionality, such as the last action performed by the user, and the abstract classes representing a Scenario, where each Scenario has a corresponding Mapping task, a state and an id. The 'mipmapGUI' module is responsible for the menus, wizards and widgets including both their graphical representation as well as their behavior to user interaction.

In these two modules the Observer design pattern is applied, so that, after a change in the state of the scenario or the last action performed by the user has occurred, the classes representing the application's main actions are notified and updated properly.

MIPMap can take as input either a set of csv files forming a database schema with each one corresponding to a table, or a relational database (MySQL/Postgres/Apache Derby) itself. For csv input the OpenCsv[2] external library is used and for relational input the Jdbc API, respective to each database. Jdbc implements the protocol for transferring the query and result between the application and the database. Both libraries have been added to the project as modules. When the input data are loaded they are stored in two ways; a) a corresponding schema is created in a local Postgres database and the tables are populated with their matching instances –if any- and b) a set of nodes (INodes) is built, structured in a tree form, for the schema representation, while also a similar set of INodes is created corresponding to a sample of the instances for preview purposes.

The access to the data is implemented through DAO (Data Access Object) classes. This way an abstract layer is interposed between the interface calls and the persistence layer adding more flexibility as to what database backend can be used by hiding the database details from the data operations. The composition of the INode trees is also materialized in the DAO classes.

The creation of the INode and Correspondence classes and the ones that represent operations on them follows the Prototype design pattern; while behaviorally the Visitor pattern is mainly applied allowing different kind of operations to be performed on the elements based on both their visitor's and their own type.

Tuple-generating dependencies (TGDs) generation process

The generation of the TGD rules that correspond to a certain mapping task has been kept mostly the same as ++Spicy, since the rewriting process to achieve optimal solutions generated has delivered great results.

The concept behind the rewriting process can be summarized as follows: producing the "optimal" solution during the data exchange procedure translates into discarding the redundant produced tuples while at the same time merging tuples that correspond to the same instance. One way to achieve this could have been to generate the so-called canonical solution and then apply a post-processing algorithm on all produced tuples. This approach however requires very high computing time and can hardly scale to large datasets. Instead, possible redundant target tuples, called homomorphisms, are recognized already in the TGD level and since the number of TGDs is considerably smaller than the number of instances, the computing procedure is much more efficient.

Taking into account the provided correspondences, constant assignments and the constraints declared, the original algorithm of the Clio[3] system is used to generate the candidate TGDs. Next, the rewriting algorithm recognizes possible cases of homomorphism among these rules, either in the form of subsumptions or coverages. These constraints may have been expressed by the user as foreign key relationships or joins between columns of either the source or target schema. After potential generation of redundant tuples has been detected, the candidate TGDs are rewritten to take those into account and hence produce the optimal results. These computations and changes are expressed in the generated SQL executable in the form of negations and joins. Self-joins, applied by relations between duplications of the same table, can also be taken into account using the same rewriting algorithm to produce the core solution. Lastly, primary key constraints in the target schema and functional dependencies between elements, often expressed as EGDs, can be selected to be taken into consideration when computing the final solution.

Local Postgres database

Overview – Reasons for applying local database storage

A key change to MIPMap's data exchange utilities is the use of a local database opposing to its predecessor's (++Spicy) in-memory node representation of all the data instances. With the previous engine and in cases of medium sized source datasets, the vast number of simultaneous instances of the INodes, representing the source and target instances, which had to be kept in memory all during the data exchange process, was leading to a memory leak problem, whereas possible source constraints called for even higher memory requirements. The solution to the problem that MIPMap provides is the temporary storage of the data in a local Postgres database. The main advantage of a local storage of data is the low memory requirements while at the same time improving the performance of the data exchange process.

However, since the previous approach used to produce the correct results at a very fast rate only when working on small datasets, the INode representation of the instances has been kept and the former data exchange method is still applied, but just on a sample of instances (100 in number) for preview purposes.

For the local Postgres configuration, the parameters are stored at an external file in order to be easily customized by the user and are loaded each time the application is initialized. Jdbc connection pooling is also used for reasons of performance and concurrency of connections.

Detailed usage of the local database

Upon starting the application the corresponding DAO class checks if a Postgres database named after the configuration file's parameters (default database name is 'mipmaptask') already exists in the local layer. If not, it generates and executes the script for the creation of the database

During loading or creating a new mapping scenario the script for the creation of a source and a target schema in the local database is generated and executed. The names of the temporary schemata consist of the prefixes 'source' or 'target' respectively followed by the number of the corresponding mapping task. For performance reasons, data are not loaded to the database yet; however a sample of 100 instances per table is presented to the user for preview purposes.

For each communication with the database a new instance of the AccessConfiguration class is created. An AccessConfiguration object holds the necessary information like the username and password, the uri and port of the database and the JDBC driver that is going to be used for the interaction with the database. These parameters are set upon the initialization of the tool and get their global values from the parameterized connection properties file as well.

The schema creation is based on the selected type of input. In the case of csv input, a table is created for each file, with the filename denoting the table name, while column names are taken from the first row of the file. Both table names and column names are case sensitive and all fields are treated as “Text”, since csv format lacks the information about column types. Csv type recognition techniques, using samples of data and regular expressions, were avoided due to high possibility of incorrect data type issues. Instead, converting type issues are resolved in an easy way when needed, through provided transformation functions (toInteger, toDouble, toString etc) or automatic casting according to the symbols used in function input. For parsing csv files the external library OpenCsv is used with the comma character as the default separator.

When relational input is selected, an additional AccessConfiguration object is created with its parameters provided this time by the user. Schemas and data are copied to the temporary database along with primary key and foreign key constraints. The sql.Connection class of java offers access to the database metadata as well, from which these constraints can be retrieved. For each primary key, consisted of one or multiple columns, found the table in the temporary database is altered to fit the constraint and then an insertion trigger and a corresponding function, which checks for double key values during data exchange, are created.

In both situations above, a tree-form node representation is created both for the schema and for a sample of instances. The first is crucial for the generation of the TGD rules, while the latter for preview purposes of the generated solution.

The local database is also accessed during the data exchange procedure. After the engine has generated the TGD rules corresponding to the mapping of the schemata, a solution, in the form of a PostgreSQL script, is generated to perform the translation of these rules. What is more, data instances are also loaded during the first execution of this step for each mapping task. As a result, the processing time during the first execution is increased a bit. In later executions, only additional instances that have possibly been added are inserted to the local database.

The generation of the SQL executable has gone through some alterations in order to fit PostgreSQL. All DML statements are included in a single transaction and are committed at the end with their constraints being set as deferred; i.e. not being checked until transaction commit. The creation of an intermediate schema, called “work” is also included in the script. The created script takes into account relationships between columns, for example a mandatory join condition in the source schema is translated as a join between columns, and egd rules as negations in sql. The script is executed on the temporary database for the final exchange of data.

If a transformation function is applied on the source data, firstly the syntax of the function expression is evaluated by the JEP Java library. The JEP library has been extended with the addition of new functions (Length, Datetime, ToDouble, ToString, ToLowerCase etc) or expansion of existing ones; the “substring” function, for example, can take as optional input the end index as well.

Subsequently, after parsing the expression to recognize the function itself and its parameters, the output returned is converted to the appropriate SQL script by using the corresponding Postgres functions and operators along with casting the variables to the appropriate types; for example if the absolute value of an field is mapped to a target element, then the field value is casted to numeric and the “abs” Postgres function is applied on it. Casting of the parameters to the appropriate type also occurs automatically if certain operators (for example arithmetic or comparison ones) have been used right before or after them.

Using and running scripts on a database like Postgres provides additional functionalities. One of them is the treatment of Primary Key constraints on the target schema, briefly mentioned above; MIPMap creates a temporary pre-solution and adds a trigger function for the insertion of data on each the final tables. The trigger checks for primary keys and in case of a constraint violation the data that could not be inserted are stored temporarily. The user is informed on the situation and is given the choice to export these instances separately.

Both translated instances and instances that violate primary key constraints mentioned above can be exported in either csv or json format. Again, the appropriate DAO class connects to the database and the final translated target data are retrieved. For csv output the OpenCsv parser library for Java is employed, whereas for json files no API is used, since Postgres’ utility to produce Json data straight away gives the advantage of faster performance.

An additional feature that has been added to MIPMap is the pre-processing of csv files with the option of unpivoting tables. Since csv files may sometimes not correspond directly to a relational schema, the operation of unpivoting a table transforms the -selected by the user- columns into attribute-value pairs where columns become rows, so that a relational database can be formed. A DAO class, called UnpivotCSVDAO, is used for the creation and connection to the Postgres database. The DAO class first creates two new temporary schemata, ‘source0’ and ‘target0’ respectively. Two lists, one containing the columns that will be kept and another one with the columns that will be unpivoted to rows are passed to the DAO class which then generates and runs a script that creates a view on the temporary database using the “UNNEST(ARRAY)” function that Postgres supports. The contents of the view are finally exported to a new csv file and the temporary schemata are dropped since they are no longer needed.

1. B. Marnette, G. Mecca, P. Papotti, S. Raunich, D. Santoro - ++Spicy: an OpenSource Tool for Second-Generation Schema Mapping and Data Exchange. In VLDB Conference, 2011

2. http://opencsv.sourceforge.net

3. L. Popa, Y. Velegrakis, R. J. Miller, M. A. Hernandez, R. Fagin - Translating Web Data. In Proc. Of VLDB, pages 598–609, 2002.