STUDIA INFORMATICA 2010
Volume 31 Number 2A (89)

Marcin BUDNY, Katarzyna HARĘŻLAK
Politechnika Śląska, Instytut Informatyki

COMPARISON AND SYNCHRONIZATION OF DATABASE SCHEMAS

Summary. This paper covers a problem of comparison of two relational database schemas, which may appear during work on a daily basis of both application developers and database administrators. In order to solve this problem, a set of mechanisms for application automating this task was suggested. Among those mechanisms there are transformation of database schemas to object model, data type mapping and differential DDL script generation.

Keywords: database schema comparison, object data model, DDL scripts

PORÓWNYWANIE I SYNCHRONIZACJA SCHEMATÓW BAZ DANYCH

Streszczenie. W artykule zaproponowano zbiór mechanizmów pozwalających na automatyzację procesu porównania i synchronizacji schematów baz danych. Mechanizmy te, obejmujące transformację schematów baz danych do opracowanego uniwersalnego modelu danych, mapowanie typów danych oraz generacje skryptów DDL przetestowano w przykładowej aplikacji.

Słowa kluczowe: synchronizacja schematów baz danych, obiektowy model danych, skrypty DDL

1. Introduction

When participating in a team project, a developer very often uses code repositories like CVS, SVN or Visual SourceSafe. This introduces a defined work cycle in which some repetitive phases can be distinguished (Fig. 1). A developer, who implements functionality in an application, begins his work with getting the latest version of the source code. From this moment, he works with its local copy, so that his changes do not influence other people's work. The same rule applies to application's database. Each developer should work with his local copy of the database, so that his changes do not have a negative effect on other team member's work.

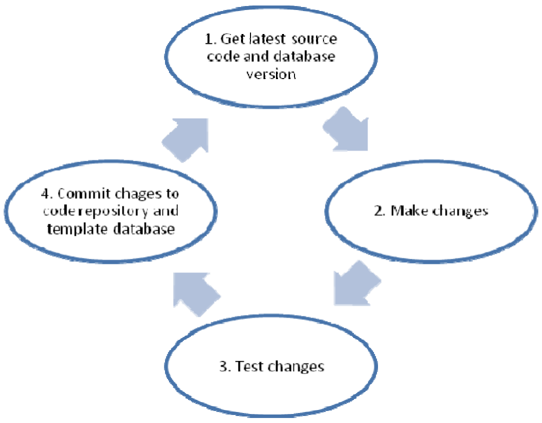
1. Get latest source code and database version
2. Make changes
3. Test changes
4. Commit chages to code repository and template database

Fig. 1. The work cycle in development team
Rys. 1. Cykl pracy programisty

There should also exist a template copy of an application's database containing all the objects required to run an application compiled from the latest repository source code. After making the changes, the application is initially tested by a developer (also in an automated way). When the new functionality is ready, a developer commits his changes to the code repository and applies database modifications to the template database copy.

During the last phase, some problems may emerge, since it is a common scenario for two developers t edit the same source code file simultaneously. The conflicts that can be a result of this situation are handled by tools integrated with source code repository. It is much more complicated when similar problems involve the database. It requires manual creation of DDL scripts, which will introduce changes a developer was working on locally to the template database. This creates a possibility of an error.

This paper discusses the problem of automated comparison and integration of database schemas maintained by the same or different Relational Database Management Systems (RDBMS). This activity is part of both developer and database administrator's daily work. Its automation will allow avoiding many errors and increase productivity.

1.1. Example of database schema synchronization

To understand the problem better, an example situation in database application development team will be analyzed (Fig. 2). In the beginning the version of the template database is 2.01. Developer C has not made any changes to the database, so his local copy of database is the same as template. Developers A and B made different sets of changes in their local database copies and as a result, versions 2.01.a and 2.01.b were created. During the change committing phase both developers will synchronize their local databases with the template and version 2.02 will be created. At this point, the need for automated synchronization process becomes apparent. Developer A would use a tool application to generate DDL script, execution of which would result in the template database having same structure as version 2.01a database. Developer B would follow the same path. Ability to easily browse a list of changes and decide which of them should be applied is also of significant importance.

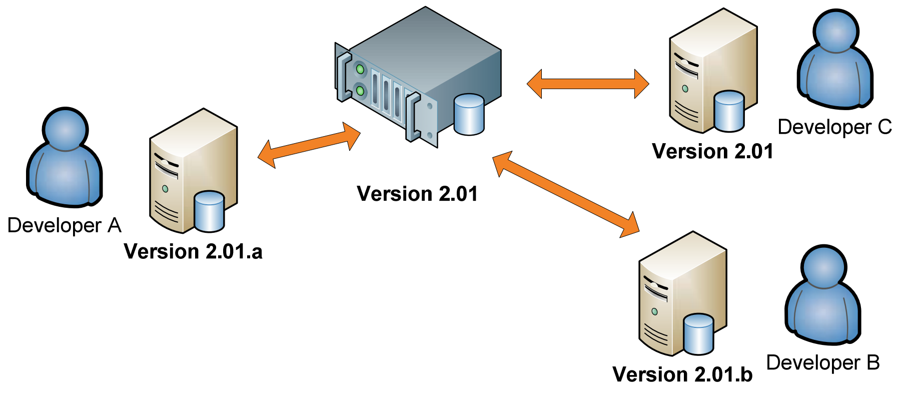

Fig. 2. A situation in development team
Rys. 2. Obraz sytuacji w zespole programistów

1.2. Other usage scenarios

Another potential usage of the application is migration of database schema to different RDBMS. It is a common requirement for database application to run on two or more RDBMS. In this situation, the development team has to create DDL scripts for each RDBMS by hand. Suggested approach can remedy this situation by automating that process. Migrating a schema from RDBMS Oracle to RDBMS SQL Server will be performed by comparing Oracle schema containing objects with empty SQL Server database. The application will generate scripts creating all objects in SQL Server database.

Some other usage is to facilitate the database administrator's work. In the world of enterprise applications it is common to have two instances of database, one of which is testing environment and the other – production environment. IT department employees, who are responsible for installing business application updates, often do this in the test environment first. When no problems are found, the update is installed in production environment. While performing those tasks, the database administrator often wants to quickly check database schema changes between two application versions. This task can be facilitated by suggested solution, which will present the list of changes in an easy to understand, graphical fashion.

There are some tools on the market that perform tasks similar to those described earlier, but they have some limitations. For example the SQL Compare tool by RedGate [9] company as well as Microsoft Visual Studio Team System for Database Professionals [8] support only RDBMS SQL Server. The DBCompare application by Automated Office Systems [1] on the other hand supports different RDBMS thanks to usage of ODBC libraries, but it does not preserve RDBMS specific features of objects and cannot generate differential scripts.

2. Universal object model

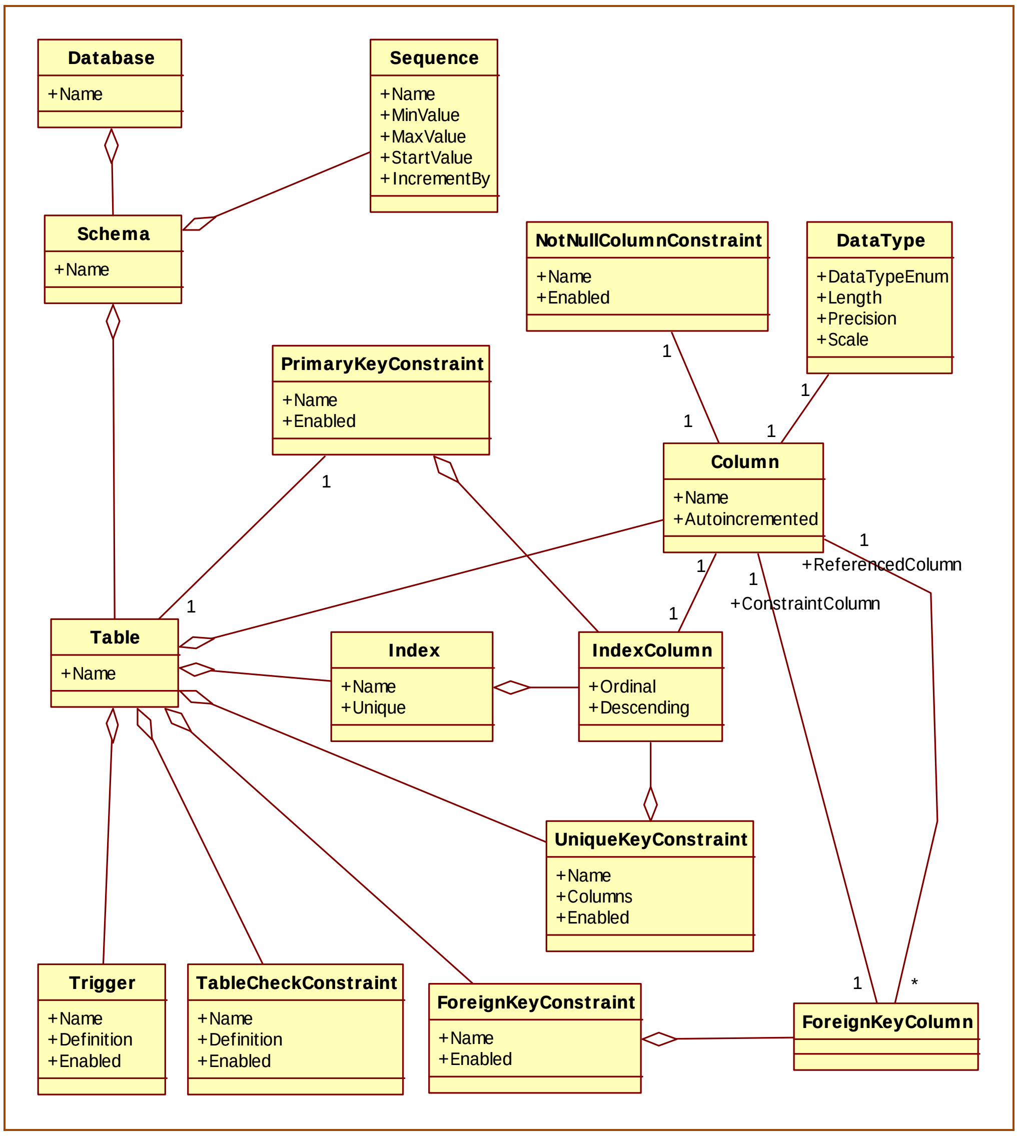

Fig. 3. Universal object model of database schema
Rys. 3. Uniwersalny model obiektowy schematu bazy danych

To achieve the goal, information on database structure has to be gathered and placed in data structures enabling easy comparison.

Information gathering is performed by querying certain system elements of selected database servers. Definitions of most database objects are usually stored by RDBMS in relational structures. The information is contained in several tables and related with foreign keys. Each object feature is stored in a separate column. This applies to objects like: tables, columns, indexes, keys, synonyms, users, roles or sequences. Other objects like: views, triggers, procedures and functions are stored in form of text definition in language being a SQL variation specific to given RDBMS. For SQL Server system it is T-SQL [6] and for Oracle system – PL/SQL [7]. Analysis of those objects is considerably more difficult and requires creation of interpreters for RDMBS specific languages.

The other task is to prepare data structures to store information loaded from system tables. It is more complicated since it involves solving the problem of RDBMS servers not being fully compatible. Some mechanisms may not be implemented or may be built differently, example of which are autoincrementing columns.

To perform this task, universal object model of data structures was proposed. All database objects loaded from RDBMS are transformed to this model. This way it is possible to compare objects regardless of their source system. Figure 3 shows simplified data object model. It represents objects, definitions of which can be obtained by querying metadata stored in relational structures [3].

Objects of the model and their relations are forming a tree (except for foreign keys). The *Database* class representing a database is the root of this tree. This class contains a collection of *Schema* class instances, which represent database schemas. This class in turn contains collections of *Sequence* and *Table* class instances. In a similar way other database objects are represented. Detailed description of this model is available in [3].

3. Application architecture

One of the requirements for application integrating database schemas of different RDBMS is a modular structure. This requirement is important if this tool is to be universally and easily extensible with support of different RDBMS. It is assumed, that extending the application does not require modifications to existing application code.

3.1. Plugin mechanism

The realization of architecture requirements of the application called from now on *CompDb* will require implementation of plugin mechanism. Application plugin is a library containing some functionality, which extends software’s capabilities. In this paper, a plugin will be responsible for handling a certain RDBMS.

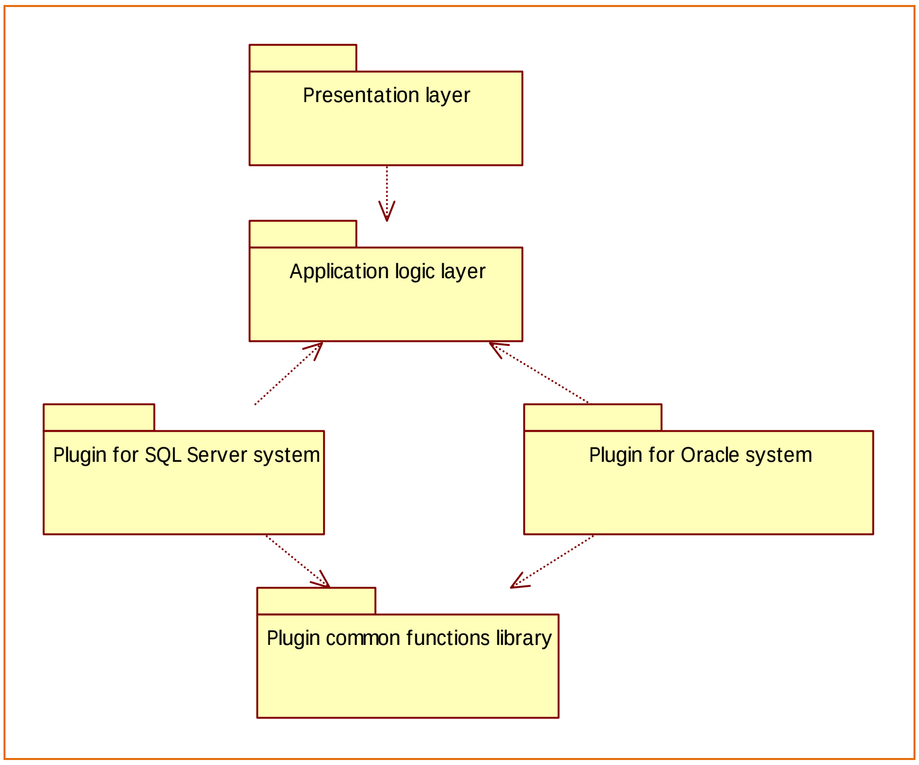

Fig. 4. General architecture diagram of CompDb application
Rys. 4. Ogólny diagram modelu architektury aplikacji CompDb

Installation process of the plugins should be easy and require only copying the plugin's library to application's directory. The application should automatically detect which RDBMS are supported by scanning provided plugins.

During the research, plugins for SQL Server and Oracle systems were created. Diagram in the figure 4 presents a general view of the CompDb application's architecture in UML notation [4435]. According to application layering rules, the presentation and logic layers were separated [2]. The presentation layer, which is user graphical interface, performs all database comparison operations according to user's commands by using logic layer functions. Inside the logic layer, the universal object model was defined. This model also contains functions required to compare database objects.

Each application plugin is placed in a separate library. It should be noted, that dependency between plugin and application logic is inverted. This removes the need for application to know the list of all plugins during compilation phase and allows them to be copied to application's directory subsequently. The application also contains a mechanism automatically detecting plugins on a startup. There is also a library, common to all plugins, which makes creation of new plugins easier without the necessity to duplicate the code.

3.2. Packages composing application layers

Presentation layer – **CompDb** – Highest level package in which the user interface is defined. Classes contained in this package are responsible for displaying information and reacting to user generated events.

Application logic layer – CompDb.Core – Universal object model is defined in this package. All calls to application plugins from *CompDB* and *CompDb.Core* packages are performed through interfaces. This way, the mentioned packages are not directly depending on particular plugins and, therefore, dynamic plugin loading is possible.

SQL Server system plugin – CompDb.Provider.SqlServer – A sample plugin for RDBMS SQL Server. For the Oracle system, another package named *CompDb.Provider.Oracle* was prepared. The plugin package contains classes responsible for loading database object definitions from RDBMS and transforming them to universal database model. It also contains classes able to generate differential scripts suitable for given RDBMS, basing on differences detected during two databases comparison phase. Additionally, the plugin defines user interface for inputting RDBMS connection parameters.

Common plugin library – CompDb.ProviderUtils – This package contains helper classes for plugins, which make their implementation easy and reduce the necessity to duplicate the code.

4. Database data types

One of the most difficult elements to compare in databases maintained by different systems are data types. Each RDBMS has its own data type set used during creation of tables and stored procedures. Often a certain data type defined by SQL 2003 standard in given RDBMS is only an alias to another data type. This results in different behaviour of equally named data types on different systems. What is more, equally named data types often have a completely different meaning in different RDBMS. An example of this situation is the *TIMESTAMP* data type, which in both Oracle system and SQL 2003 standard [7, 10] represents a point in time, but in SQL Server it is used for versioning rows in a table and is an alias for *ROWVERSION* data type [6]. A point in time in this system is represented by the *DATETIME* type.

To compare data types from different RDBMS possible, a concept of universal data types was introduced. Those types are based mostly on the SQL 2003 standard, but there are also types, which are not in the standard, yet are often implemented by RDBMS.

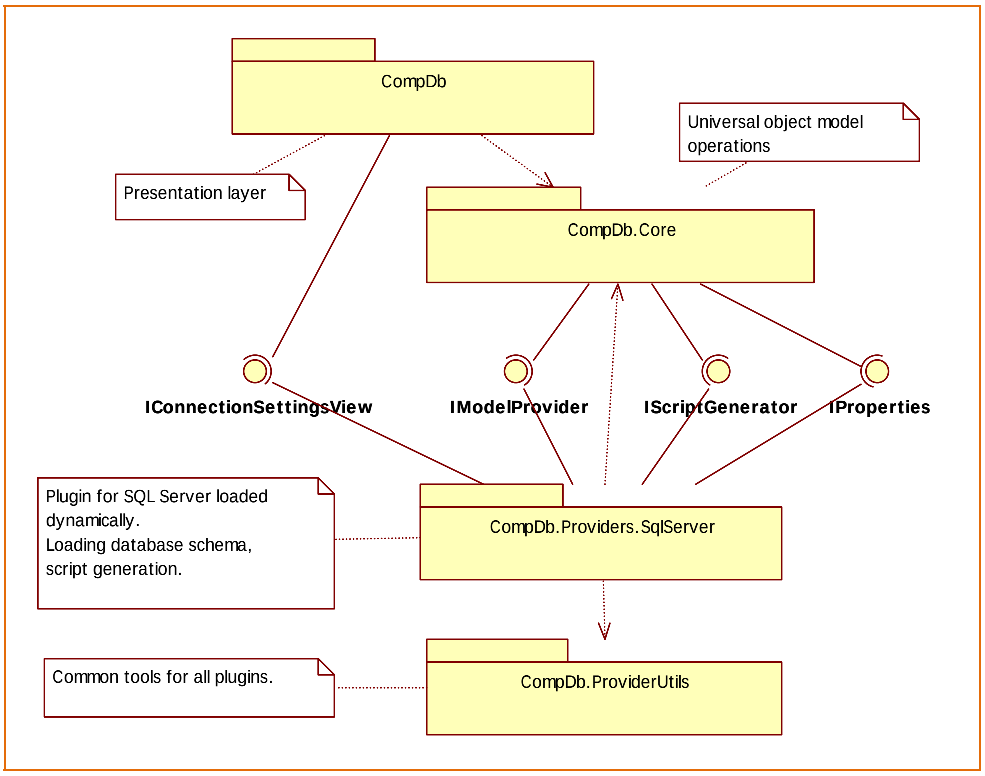
CompDb
Universal object model operations
Presentation layer
CompDb.Core
IConnectionSettingsView
IModelProvider
IScriptGenerator
IProperties
Plugin for SQL Server loaded dynamically.
Loading database schema, script generation.
CompDb.Providers.SqlServer
Common tools for all plugins.
CompDb.ProviderUtils

Fig. 5. Architecture model
Rys. 5. Model architektury

A plugin supporting certain RDBMS is responsible for transforming data types specific to this RDBMS to universal data type. Additionally, information about length, precision and scale is stored. During the database comparison phase, universal data types are compared. This way it is possible to compare two databases maintained by different RDBMS while keeping the application plugins unaware of each other's existence. During the differential script generation phase, universal data types are transformed to data types specific to a given RDBMS.

In order to transform database data types to universal data types, it is necessary to define a transformation map, called from now on the data type mappings. They are defined by each plugin separately. In order to make modifications to this mappings as easy as possible, they are stored in a XML file containing following information:

- for each database data type, corresponding universal data type is defined,
- for each universal data type, corresponding database data type is defined,
- for specific database data types, which cannot be directly derived from universal data type, a format is defined.

The last piece of information requires additional explanation. It is a common situation when a RDBMS has a specific data type, which is not present in SQL 2003 standard. The *MONEY* data type from SQL Server system representing an amount of money can serve as an example here. Its transformation to universal data type results in *Numeric* type. This way some information is lost. During creation of a column of this data type in destination database maintained by SQL Server system, a different data type will be used than this of corresponding column in source database (*NUMERIC(19, 4)* instead of *MONEY*). To avoid this effect, the application stores column's original data type in addition to the universal one. If compared databases are maintained by the same RDBMS, this additional information is used to create a column of data type identical with this in source database.

A sample mapping XML file fragment for SQL Server system is presented on below listing. Inside the *DatabaseToUniversalMapping* tag, there is information about transformations from database data type to universal data type. The *Mapping* tag contains information about a single data type. Inside the *UniversalToDatabaseMappings* tag, there are transformations from universal data type to database data type described. The *SpecificTypeFromat* contains information about formats of data types specific to the RDBMS.

```
<?xml version='1.0' encoding='utf-8' ?>
<Mappings xmlns='http://marcin.budny/MappingsSchema'>

  <DatabaseToUniversalMappings>
    <Mapping>
      <From DatabaseDataType='VARCHAR' />
      <From DatabaseDataType='CHARACTER VARYING' />
      <To UniversalDataType='CharacterVarying' />
    </Mapping>
    …
  </DatabaseToUniversalMappings>

  <UniversalToDatabaseMappings>
    <Mapping>
      <From UniversalDataType='CharacterVarying' />
      <To DatabaseDataType='VARCHAR(@L)' />
    </Mapping>
    …
  </UniversalToDatabaseMappings>
  <SpecificTypeFormats>
    <Format DataType='MONEY' As='MONEY'/>
    …
  </SpecificTypeFormats>

</Mappings>
```

It should be noted, that database data types mapped to universal data types can have many-to-one relation, while the inverse mappings can only be in one-to-one relation. The XML schema used to validate the XML mapping files was also created during the research.

Pseudo-code transforming data types specific to this RDBMS to universal data type is presented in the listing below.

```
public GetUniversalDataType(){
search in mapping XML file for database type;
if (found > 1)
    return "error found in XML file";
    if (found = 0)
        mark database type as unknown;
    else
        get the name of the universal data type;
    create and return the object describing data type;
}
```

Table 1 shows some of universal data types created in order to perform SQL Server and Oracle systems data type mappings.

Table 1
SQL Server 2005 and Oracle data types with corresponding universal data types

Universal data type	SQL Server 2005	Oracle 10.2g
Character(L)	CHAR(L)	CHAR(L)
Character Varying(L)	VARCHAR(L)	VARCHAR2(L)
Character Large Object	VARCHAR(MAX)	CLOB
National Character Large Object	NVARCHAR(MAX)	NCLOB
Binary Large Object	VARBINARY(MAX)	BLOB
Numeric(P, S)	NUMERIC(P, S)	NUMBER(P, S)
Decimal(P, S)	DECIMAL(P, S)	NUMBER(P, S)
Smallint	SMALLINT	SMALLINT
Integer	INT	INT
Bigint	BIGINT	NUMBER(19, 0)
Float(P)	FLOAT(P)	FLOAT(P)
Real	REAL	REAL
Double Precision	DOUBLE PRECISON	DOUBLE PRECISION
Boolean	BIT	(not available)
Date	DATETIME	DATE
Time	DATETIME	TIMESTAMP
Timestamp	DATETIME	TIMESTAMP
Row Version	ROWVERSION	UROWID
Unique Identifier	UNIQUEIDENTIFIER	(not available)
Xml	XML	XMLType
Date	DATETIME	DATE

5. Differential script generation

Generation of differential DDL scripts, basing on differences detected during two database schema comparison, is another important task of an application plugin including classes responsible for DDL scripts generation. Every class knows how to modify one specific type of an object. This is achieved by passing universal object model element to static methods of this class, which perform given operation (creation, deletion or updating). If given object has substandard ones (for example tables have collection of columns), creating or updating method is responsible for calling methods performing appropriate actions on subordinate objects. How this action should by executed is described by *ComparisonInfo.ComparisonResult* property, which is possessed by every object of the universal object model. Its value is set during database schema comparison.

Classes generating differential scripts call methods recursively, going deep into universal object model hierarchy. Each method returns string containing a part of the differential script. Method of root object returns merged string from all levels of the hierarchy. The string is presented to the user of the application.

During differential scripts generation there is often a necessity to take into consideration dependency between database objects. Following examples explain such situations.

There is a foreign key FK_AB joining two tables A and B. In case, when differential script includes creation operations of tables A and B, it should be taken into account that foreign key FK_AB can be created only when the tables A and B already exist. Similarly during deletion of those tables, foreign key FK_AB must be removed first.

There is a primary key PK_A built on column KolA in table A. During data type changing for column KolA in SQL Server System, it is necessary to delete primary key and to recreate it after finishing this operation. To meet these requirements, statements of constraints creation are placed at the end of the differential script, while statement of deletion of such object is located at the beginning of the script.

6. Application's operation algorithm

The application implementing mechanisms described earlier performs its tasks in four steps.

Step 1: Load all plugins from DLL files residing in application's directory.

```
public LoadPlugins(string path){
   initiate empty list of plugins;
   for each *.DLL file in path{
      load plugin from file;
      if (plugin != null)
         add plugin to the plugin list;
   }
   return pluginList;
}
```

Step 2: Load database objects from both source and destination database and transform them to universal object model. Data type mapping is also performed in this step. A plugin supporting the RDBMS on which a database resides is responsible for carrying out those tasks. Pseudo-code presented below, uses loading columns as an example.

```
private void LoadColumns(){
 initiate columns dictionary;
 load columns definition from the system view;
 for each returned column{
   if (table, which a column belongs to, exists){
      initiate new column;
      get name;
      if (NOT NULL constraint exists for the column){
         create NOT NULL constraint object;
         get the universal data type for the column;
            assign the column to the table;
            add to the columns dictionary;
      }
   }
}
add autoincrement feature for appropriate columns;
}
```

Step 3: Compare object models. This task is performed recursively, beginning at the root of the object tree and moving towards the leaves. Since two databases are compared, there are two object trees in operation memory, one of which represents source database and the other – destination database. Therefore the comparison algorithm has to process both trees simultaneously. Objects are selected for comparison on given tree level by name. In RDBMS it assumed that objects of given type have unique names in scope of one database schema. The comparison and storage of found differences is performed by CompDb's application logic, which is not aware of objects' source RDBMS. Detailed description of the algorithm of two database schemas comparison was include in [3], while general algorithm is presented in pseudo-code below.

```
public void CompareCollections(){
   initiate empty lists of objects;
   for each object in current collection{
      if (object in collection){
         add the object to the existing objects list;
         mark that no action is required for the object;
      }
      else{
         add the object to the list of objects for creation;
         mark the object „to create";
      }
   }
   for each object in currently compared list{
      if (object does not exist in current collection){
         add the object to list of objects for dropping;
         mark the object „to delete";
      }
   }
}
```

Step 4: Pass the list of found differences to plugin supporting destination RDBMS. The plugin is responsible for generation of differential DDL script in SQL dialect specific to given RDBMS. The following pseudo-code presents the example of column creation.

```
public static string Create(Column column){
   change column universal data type to database data type;
format statement for column creation;
   statement = CreateColumnStatement, Name, dataType, column.Nullable ? "NULL"
               : "NOT NULL");
   if (column is autoincremented)
      statement += IdentityStatement, StartValue, IncrementBy);
   return statement;
}
```

7. Summary

This paper covers a problem of two relational database schemas comparison, which may appear during work on a daily basis of both application developers and database administrators. In order to solve this problem, a set of mechanisms was suggested for application automating this task. Among those mechanisms there are transformation of database schemas to object model, data type mapping and differential DDL script generation.

This functionality was implemented in CompDb application, which allows comparing databases maintained by SQL Server and Oracle systems. This application has a module-based architecture, which allows extending the functionality with support for additional RDBMS. That ability was achieved with mechanism of autonomous application plugins. Thanks to object model of database objects, it is possible for objects originating from SQL Server system to be used later for the generation of DDL scripts with plugin supporting the Oracle system. The application's operation effects are all satisfactory.

BIBLIOGRAPHY

1. Automated Office Systems. DBCompare. http://www.automatedofficesystems.com/products/dbcompare/, 2007.
2. Boodhoo J.P.: Design Patterns: Model View Presenter. http://msdn.microsoft.com/en-us/magazine/cc188690.apsx, 2008.
3. Budny M., Harężlak K.: Usage of the universal object model in database schemas comparison and integration, International Conference On Man-Machine Interactions, http://icmmi.polsl.pl, 2009.
4. Comparison of different SQL implementations. http://troels.arvin.dk/db/rdbms/, 2007
5. Fowler, Martin. UML Distilled: A Brief Guide to the Standard Object Modeling Language. Addison-Wesley, 2003..
6. Microsoft. SQL Server Books Online. http://msdn.microsoft.com/en-us/library/ms130214.aspx, 2008.
7. Oracle. Oracle Database Online Documentation 10g Release 2. http://www.oracle.com/pls-/db102/homepage, 2008.
8. Randell, Brian A. Introducing Visual Studio 2005 Team Edition for Database Professionals. http://msdn.microsoft.com/en-us/magazine/cc163472.aspx, 2007.
9. Red Gate. SQL Compare.http://www.red-gate.com/products/SQL_Compare/index.htm, 2007.
10. SQL 2003 Standard. http://www.wiscorp.com/sql_2003_standard.zip, 2007

Recenzenci: Dr inż. Bożena Małysiak-Mrozek
Prof. dr hab. inż. Henryk Rybiński

Wpłynęło do Redakcji 14 stycznia 2010 r.

Omówienie

W artykule omówiono problem porównania i synchronizacji schematów baz danych – utrzymywanych przez te same lub różne Systemy Zarządzania Bazą Danych (SZBD). Przed problemem tym, często w swojej codziennej pracy, stają zaawansowani użytkownicy – programiści w ich grupowej pracy, jak również administratorzy baz danych.

W celu rozwiązania tego problemu zaproponowano szereg mechanizmów, które zaimplementowano w przykładowej aplikacji. Aplikacja ta wyposażona została w mechanizm wtyczek, dzięki czemu uzyskano łatwą jej rozszerzalność o obsługę kolejnych serwerów baz danych. Do zadań wtyczki należy pobranie z serwera definicji obiektów i przekształcanie ich na opracowany w tym celu uniwersalny model obiektowy. Szczególnym aspektem tego przekształcenia jest obsługa typów danych. Niemal każdy SZBD posiada typy danych, które nie mają odpowiednika zarówno w standardzie SQL 2003, jak i w innych SZBD, dlatego wtyczka musi zawierać odpowiednią mapę przekształceń typów danych. Mapa ta zawarta została w pliku XML.

Dwie bazy danych porównywane przez aplikację wczytywane są do pamięci w postaci uniwersalnego modelu obiektowego. Logika aplikacji zajmuje się ich porównywaniem i zapamiętywaniem różnic. Na podstawie tak znalezionych różnic generowane są skrypty różnicowe DDL. Wtyczka aplikacji musi znać składnię DDL właściwą dla obsługiwanego przez nią SZBD, jak również musi uwzględniać wszelkie zależności pomiędzy wykonywanymi w skrypcie operacjami.

Addresses

Marcin BUDNY: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, marcin.budny@gmail.com .
Katarzyna HARĘŻLAK: Politechnika Śląska, Instytut Informatyki, ul. Akademicka 16, 44-100 Gliwice, Polska, katarzyna.harezlak@polsl.pl .