DSL FOR DATABASE CONCATENATION AND MIGRATION

Author: Sergiu ERNU
Scientific advisor: Dumitru CIORBĂ

Technical University of Moldova, Faculty of Computers, Informatics and Microelectronics

DSL or domain-specific language is a programming language to a particular domain or a particular solution technique. Sometimes can occur the need to concatenate two or more databases and after to migrate the data. The article will not study the causes of concatenation but the solutions for achieving it. Though it sounds simple the process in a real world scenario is a complex one with lots of dependencies. It is analyzed some examples on which typically solutions are proposed. The DSL for solving such tasks is not described, instead its requirements and its goals are.

dsl, sql, concatenation, migration

I What is a DSL.

In software development and domain engineering, a domain-specific language (DSL) is a programming language or specification language dedicated to a particular problem domain, a particular problem representation technique, and/or a particular solution technique. The concept isn't new—special-purpose programming languages and all kinds of modeling/specification languages have always existed, but the term has become more popular due to the rise of domain-specific modeling.

Examples of domain-specific languages include HTML, spreadsheet formulas and macros, regular expressions for specifying lexers, software packages used for graph layout and graph rewriting.

Creating a domain-specific language (with software to support it) can be worthwhile if the language allows a particular type of problem or solution to be expressed more clearly than an existing languages would allow and the type of problem in question reappears sufficiently often. Language-Oriented Programming considers the creation of special-purpose languages for expressing problems a standard part of the problem solving process.

A domain-specific language is created specifically to solve problems in a particular domain and is not intended to be able to solve problems outside it (although that may be technically possible). The DSL is somewhere between a tiny programming language and a scripting language, and is often used in a way analogous to a programming library. The boundaries between these concepts are quite blurry, much like the boundary between scripting languages and general-purpose languages.

II Database concatenation problem.

For developing a functional and efficient system must be taken into account the ability to add some new components later, but still situations where such things can not be done or the amount of time invested will not be efficient comparing to build something from scratch can occur. This article will not study the causes of these situations but will try to study the solution. Cases where is a need to concatenate two or more databases can happen, the reasons and causes can be diverse and we are not interested in. Apparently this might look simple, but if the task is analyzed more carefully it seems that the problem is much more complex. At this point exists two solutions:

1. To dump all the tables from each database into the new one.
2. To try to analyze carefully all databases and to extract the needed tables and/or to merge some tables from all databases into new ones with their dependencies.

The first solutions will generate a lot of garbage into the new created database and this database will be not efficient at performance or memory usage. Trying to achieve the second one much more work must be done, but instead it gives a lot of benefits: the new created database will combine all databases with less tables then in case of the first solution, with a greater performance and with a less usage of phisical memory. Further will be discused how the second solution can be achieved.

The first thing to do is to optimize the resultant database by excluding the unneeded tables from each database we must concatenate, if such cases exists; or if the databases have a less amount of tables may be

more reasonable to do vice versa to chose only the needed tables, anyway in both cases the result will be the same. The second thing to do is to check wich tables can be merged as to free some other memory. This case will be the most important as it will give us a very high level of performance. The only way to do that is to analyze the content of each table from each database we need to concatenate as to find out the pairs of the tables which will satistfy this demand. However here appears other demands as well. As to be more clear i will give some real examples. Suppose we have two databases we must concatenate: database A see figure.1 and database B see figure2.

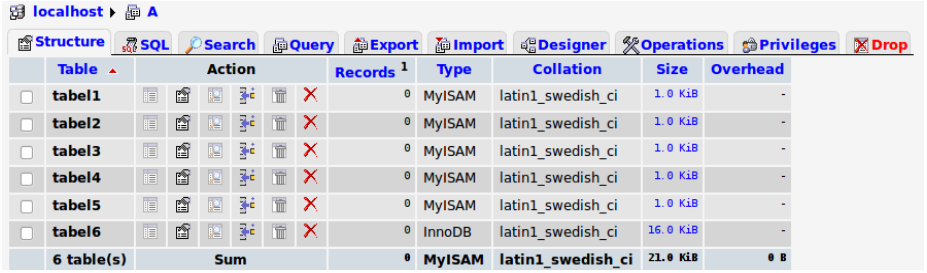

Figure 1. Database A with all its tables.

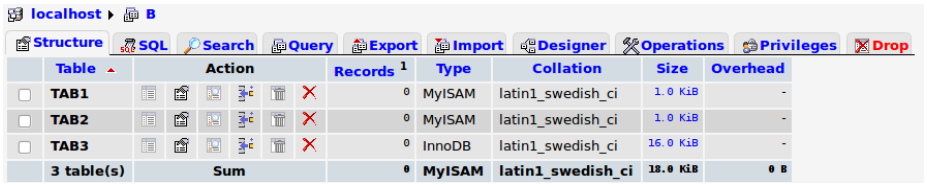

Figure 2. Database B with all its tables.

I will suppose that out of these two databases there exists two tables which can be merged together in a new database. Let suppose that from database A the table will be “table6” and from database B the coresponding table is “TAB3”.

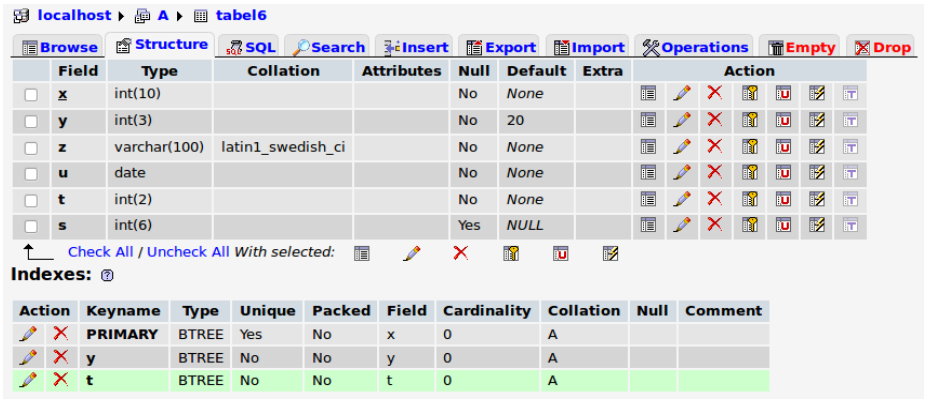

Figure 3. Database A, Table “table6” with the columns and keys.

Table “table6” from database A has the following columns and keys as can be seen in figure 3. (see above):
- Field `x` of type `int` with a length of 10 which value can not be null;
- Field `y` of type `int` with a length of 3 with default value `20` which value can not be null;
- Field `z` of type `varchar` with a length of 100 which value can not be null;
- Field `u` of type `date` which value can not be null;
- Field `t` of type `int` with a length of 2 which value can not be null;
- Field `s` of type `int` with a length of 6 with the default value null which value can be null;
- Primary key `PRIMARY` on field `x`;
- Foreign key `y` on field `y`;
- Foreign key `t` on field `t`.

Table “TAB3” from database B has the following columns and keys as can be seen in figure 4. (see below):
- Field `x` of type `int` with a length of 10 which value can not be null;
- Field `y` of type `int` with a length of 3 with default value `10` which value can not be null;

- Field `z` of type `varchar` with a length of 20 which value can not be null;
- Field `u` of type `datetime` which value can not be null;
- Primary key `PRIMARY` on field `x`;
- Foreign key `z` on field `z`;

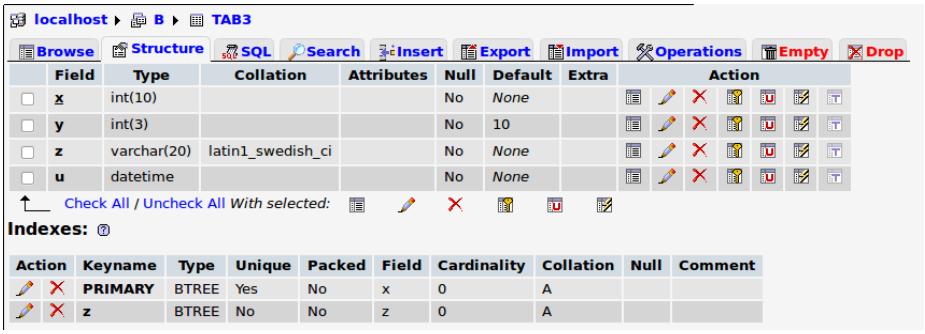

Figure 4. Database B, Table “TAB3” with all the columns and keys.

From the above examples figure 3. and figure 4. can be observed that the following two tables can be merged into one by doing some transformations. First of all the fields of the table “tabel6” from database A `t` and `s` will be copied without any analysis as they are not in the table from database B. Field `x` from both tables will be merged as well into one as all the requirements are satisfied. Field `y` in both tables is of the same type and length, but the default value is different, in this case depends on us which default value to chose, anyway should be chosen such a value which will be more proper to the situations and is not mandatory to be one out two we already have. As for field `z` the solutions requires more attention, on both tables this field is of the same type, can not be null, but the length differs – on table “table6” the length is 100 but on table “TAB3” the length is 20. At this point can be found two solution the easiest one and the most correct one is to pick the maximal length out of those two for the resultant table, or another solution is to see the ideal length this field could have by analyzing the maximal length the rows of these tables have. May be the second solution will give us some less memory usage, but if we are not the creator of the initial two tables who knows why such length were chosen, could happen that in the future will appear some values to insert bigger then the length we chose and this will lead to data loss. The last pair of field which should be merged is the field `u` from both tables. On both tables this field can not be null, but on table “tabel6” is of type `date` while on table “TAB3” is `datetime`. Here also should be chosen the most optimal value for this field, value which will satisfy the demands of the resultant table; still the format `datetime` is “yyyy-mm-dd hh:ii:ss” while the format for `date` is “yyyy-mm-dd”. In case we will chose the resultant type `date` would be perfect as we will exclude the hours, minutes, seconds from the “table6”, but what will happen if we need to know the hour , minutes or even seconds? By choosing the resultant type `datetime` for values of the table “TAB3” we will have a bunch of zeros instead of hours, minutes, seconds and unfortunately this is the price we must pay in this case.

The keys are another thing which we should take care of. Of course we can rebuild them after, but why we should lose extra time when we can do this at the beginning. The primary key will be easily recreated as it has no dependencies. However the foreign key may or may not have some dependencies and it is up to us to verify these dependencies and may be we will save other resources as well by doing that. Let's suppose we did not wanted to copy the tables from both databases the foreign key `z` is mapped to, in this case the entire field `z` from both tables could be not useful to us, so it would be a good solution to exclude this field from the resultant table.

III Real world scenario and the DSL for database concatenation and migration.

Somebody could say that all this analysis is just a waste of time and all this could be done quite simple - may be... Usually in a real world scenario we do not have databases with 6 or 3 tables, we could have hundreds or even thousands of tables, depending of the needs and the domain the database was designed. Also the tables can have up to hundreds of fields. In this case we could lose weeks or even months till we will sort out and we will achieve this goal. In such cases a DSL will be a proper solution to handle the toughest part of the problem.

In fact to migrate a database is a very easy task. But in case after we will concatenate some databases into one the migration will be also a hard process, as will appear new demands. Basically the most hardest part of this are the fields of the primary keys, which be definition a primary key can not be null and has to have unique records. So seeing the example in the second part, the solution is to add a new field for the tables that were merged, the primary key will be set to this newly added field and on the old field will be set up a foreign key instead of primary.

The main goal of such a DSL is to easier the life of people who will face with problems like that. Though they will have to write some code and to make some analysis on the databases still the DSL will reduce considerable the amount of work and the time spent for achieving the final product.

IV How a DSL comes in handy.

In the previous chapters had been discussed theoretically the issues of the problem and was proved the need of an DSL to easier the work.

dsl_declare.php

```
1  <?php
2  database $db_resultant;
3  database $db_A;
4  database $db_B;
5  $db_A = import_database("ip","user","password","A");
6  $db_B = import_database("ip","user","password","B");
7  $db_A.import_all_tables();
8  $db_B.import_all_tables();
9  $db_resultant.add_table("table1");
10 $db_resultant.table1.declare_merge(db_A.tabel6 + db_B.TAB3);
11 ?>
```

Figure 5. DSL in php – declaration of variables.

From the above example figure 5. were declared three database variables “$db_resultant” , “$db_A”, “$db_B”, all with the same names as the examples in the previous chapters. Was added a new table for resultant database “table1” which is a merge between “tabel6” from database “A” and “TAB3” from database “B”.

dsl_declare.php | dsl_sol1.php

```
1  <?php
2  $db_resultant.table1.add_rows("row1","row2","row3","row4","row5","row6");
3  $db_resultant.table1.row1=db_A.tabel6.t;
4  $db_resultant.table1.row2=db_A.tabel6.s;
5  $db_resultant.table1.row3=db_A.tabel6.x+db_B.TAB3.x;
6  $db_resultant.table1.row4=<default_value_solve>(db_A.tabel6.y+db_B.TAB3.y);
7  $db_resultant.table1.row5=<length_solve_ideal>(db_A.tabel6.z+db_B.TAB3.z);
8  $db_resultant.table1.row5=<length_solve_optimized>(db_A.tabel6.z+db_B.TAB3.z);
9  $db_resultant.table1.row6=<type_solve>(db_A.tabel6.u+db_B.TAB3.u);
10 ?>
```

Figure 6. DSL in php – a particular solution of the problem.

In the figure 6. (see above) is a solution of the problem discussed. Were created six rows for the resultant table and each row is mapped with the corresponding rows from other databases or to the pair of rows which should be merged (see 2nd line of code from figure 6.). In those cases for which had been proposed theoretical issues it is indicated how to solve the merge, for example in the 6th line of code of the figure 6. is solved the default value of the row. The next two lines will solve the problem of the length of the rows in two different ways in the ideal mode where will be set the maximum between the values or in an optimized way where will be set the average value of those two.

The approach discussed is a good one if we have few rows, but if we have a big amount of rows we need to write a lot of code and may be this is not a good idea. A more general method is to generate all the rows of the resultant table in accordance of the declared merge as can be seen in figure 7. (see below). Then will be check if exists some exceptions like unmet dependencies that had been discussed in the previous two chapters. If there exists in a for-each statement will be checked the type of the unmet dependence and will be solved the problem accordingly.

Each method has its own benefits, the first one gives us more flexibility but the second one shorter the code and the time spent. We should chose the one which is more appropriate for the situation that must be solved.

dsl_declare.php | dsl_sol1.php | dsl_sol2.php

```php
<?php
$db_resultant.table1.generate_rows_from_merge();
if ($db_resultant.table1.unmet_rows_dependencies()){
    foreach ($dependence as $db_resultant.table1.unmet_dependence){
      switch ($dependence.type){
        case "value-mismatch":
            $dependence.resolve("default_value_solve");
            break;
        case "length-mismatch":
            $dependence.resolve("length_solve_ideal");
            break;
        case "type-mismatch":
            $dependence.resolve("type_solve");
            break;
      }
    }
}
?>
```

Figure 7. DSL in php – a more general solution of the problem.

V Bibliography.

1. *MySql 5.1 Reference Manual*, http://dev.mysql.com/doc/refman/5.1/en/index.html
2. Marjan Mernik, Jan Heering, and Anthony M. Sloane. *When and how to develop domain-specific languages* . ACM Computing Surveys, 37(4):316–344, 2005.
3. Diomidis Spinellis. *Notable design patterns for domain specific languages*. Journal of Systems and Software, 56(1):91–99, February 2001.