MySQL Babel

Alessandro Rosa

April 27, 2006

Abstract

This article illustrates the main features of MySQL Babel: the PHP class performing the translation from native language sentences into MySQL queries. Its implementation might open databases to easier and more comfortable interrogations because they are no longer exclusively subjected to MySQL syntax knowledge, but user-defined queries can be now input according to one's everyday native language. Instructions to implement additional third-part dictionaries are also included.

The Lord said: – "If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. Come, let us go down and confuse their language so they will not understand each other."

Genesis 11:6

Contents

1 Introduction **2**

2 The concept **3**

2.1 A background analysis 3

2.2 Disclaimer 3

3 The core process **4**

3.1 The database example 4

3.2 A first glance 5

3.3 Remarks 6

4 Analyzing the dictionaries **6**

4.1 From abstraction to concreteness 6

4.2 Concrete dictionaries: status and rules 7

4.3 How to build a concrete dictionary 8

Current version: 0.0.1

This article has been published in MySQL Developer Zone on . . . (insert date of publication).

5 An example **9**
5.1 Setting up the environment 9
5.2 Introducing the code 9
5.3 Overview 9
5.4 Code explanation 10
5.5 Prelude : the idiomatic expressions 11
5.6 Translation (I): towards the abstract dictionary 12
5.7 Translation (II): turning into MySQL 13
5.8 Translation (III): the query finalization 14

6 Side functions **15**
6.1 Retrieving the parameters 15
6.2 Safe input data 15
6.3 Errors tracking 17
6.4 Queries and dictionaries debugging 17
6.5 Output customizations 18
6.6 Advanced queries 18
6.7 Dictionary identification 18

7 The class 'Languages' **19**
7.1 Short overview 19
7.2 Dynamic linking 19
7.3 Errors management 20

8 Proposal for a development plan **21**
8.1 Disclaimer 21
8.2 Directives list manifesto 21
8.3 Remarks about the plan 22
8.4 Calling for support 23

9 Acknowledgments **23**

10 License **23**

11 Conclusions **23**

A Appendix: Examples of advanced queries **24**

B Appendix: the abstract dictionary structure **25**

1 Introduction

MySQL is one of the most used database engines all over the world, often working with a PHP programming language environment which allows, by a comfortable code interface, to handle all features of the engine itself. After coding a number of web-based applications, based upon the so-called (W./L.)A.M.P.[1] architecture, the author considered that one step

[1]Acronym for the mixed programming environment including one *operative system* (Linux or Windows), a *web server* (Apache), one *database engine* (MySQL) and the *programming language* (PHP). The author

ahead to widen the range of possibilities offered by such applications would be *to prompt the user to input one's own query on-the-fly*, besides the general implementation of a default set. This opens the application to a countless number of queries to be input, now depending on user's current needs. Practically, the user might often want to query information from the database in a way being not implemented by the developer. It is obvious to demand that code shall fit user's requirements: but a limit of a database application is often to not assist the extra-ordinary need.

2 The concept

2.1 A background analysis

Therefore what could it be more comfortable than typing one's own query and run it? By this expression '*one's own query*', the author absolutely intends that the PHP class in question should come to the needs of ANY USER'S sitting in front of the application itself, regardless of one's skill in dealing with MySQL queries syntax. Now this basic idea needs to be sharpened and enhanced. A critics, based upon these two considerations, goes in that direction:

1. *does anybody know English and even the whole set of MySQL commands (SELECT, FROM, ...) ?*
2. *does anybody know the queries syntax interrelating those commands?*

The goal of MySQL Babel is to come to comfortable solution to both points by first *knocking down* the English language barrier (actually this is not a relevant issue due to its worldwide diffusion, but it might be for users not dominating this language). Secondly, more relevant, the MySQL syntax *knowledge* is one harder question because of subjected to pre-defined rules. (My)SQL syntax is known to many developers but common final users are not privileged to exploit its features. Both points can be got through by providing a solution to apply queries written in native languages and take the entry level down. For sake of simplicity, these two screen shots are taken from an elementary example devoted to see how the class works; but it is not hard to imagine them as implemented into a web-based application: in this sense, fig. 5.4.1 looks clearer.

2.2 Disclaimer

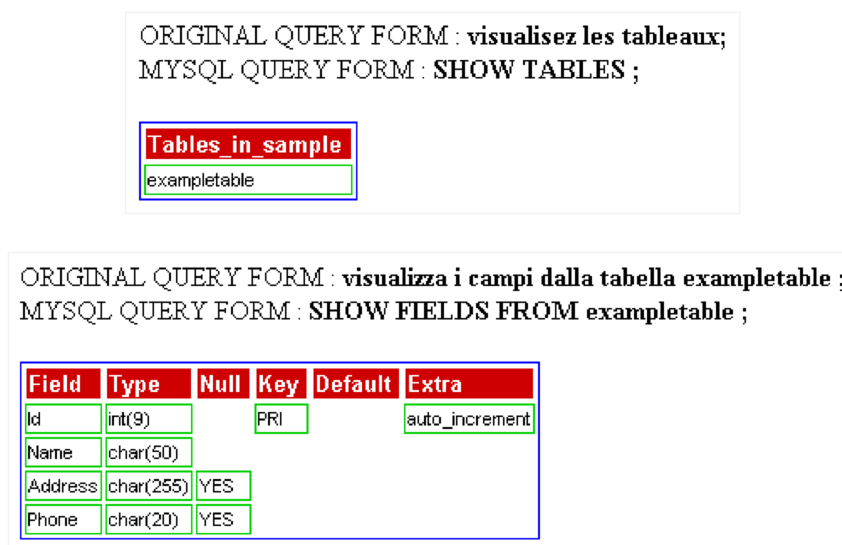
Perhaps, this section would be better entitled as sometimes we are used to say in Italian: '*here I tell it and here I deny it*'.

In fact, strictly meaning, one cannot speak of this class working as a true translation process. Native input sentences are not parsed, nor interpreted.

Henceforth, rather than '*translation*', it is honest to assume that MySQL Babel performs a '*conversion*', because sentences are just turned into a different form. No matter of grammar and sense. But let me apply the first term because, besides the class name sounding good, information is being exchanging between different languages, therefore 'translation' looked like more proper to be applied here.

This class goal does not require a complicate approach. Why? (My)SQL grammar is much simpler than of a spoken language; it does not include idiomatic forms and consists of few

applied these technologies to code the class, but any of the four elements might be obviously replaced by the favorite one.



ORIGINAL QUERY FORM : **visualisez les tableaux;**
MYSQL QUERY FORM : **SHOW TABLES ;**

Tables_in_sample
exampletable

ORIGINAL QUERY FORM : **visualizza i campi dalla tabella exampletable ;**
MYSQL QUERY FORM : **SHOW FIELDS FROM exampletable ;**

Field	Type	Null	Key	Default	Extra
Id	int(9)		PRI		auto_increment
Name	char(50)				
Address	char(255)	YES			
Phone	char(20)	YES			

Figure 2.1.1: Displaying the table and the related fields configuration. Note that, the upper original query was written in French; in the lower screen shot, it is written in Italian.

rules to build up a query: a leading command is followed by parameters like fields and table names, or by sorting criteria for example. Then one does not need huge efforts to recover a query sense for making it run properly. Otherwise, the greatest loss might be at most to see a query not working and one has to reformulate it. Transformation problems from native languages to (My)SQL queries can be solved by simple methods because one moves from a complicate to an easier grammar, so that (as we will see further) several input expressions just need to be plucked and then finally turned into the query form: definitely, this is a very simple situation. But, in general, translation becomes a 'dirty job' for spoken languages: even programs can hardly do that and rough texts may often come out.

This article is devoted to two ranks of developers: a first group possibly using this class inside applications and another one liking to join the development plan as explained in §8. The author apologizes for any grammar mistake appearing in the examples of native expressions throughout the further pages: aware of his personal inexperience with most of them, these languages application was mainly intended to offer examples of MySQL Babel features.

3 The core process

3.1 The database example

Before getting into the internals of MySQL Babel, the example environment[2] of a simplex index book will be explained by means of the class itself.

First, by two elementary queries, we show either the tables and the fields in fig. 2.1.1. So we have a given table '`exampletable`' including four fields: `Id`, `Name`, `Address` and `Phone`.

[2]Included inside the files '`index.php`', '`example.php`' of the distribution package.

3.2 A first glance

Step by step, we will enter the translation process. By now, this section will present it in general terms; throughout the reading of the next sections, one will go over it and earn a full knowledge.

choisir tout de tableau exampletable;

Figure 3.2.1: The example with a given input French sentence. The surrounding rectangle means to a unique initial entity to work with.

The idea is simple and consists of three mandatory steps. The first operation is to set up the native language to work with. Five languages are currently available: English, French, German, Italian and Mexican.

Secondly, as the input sentence is written into a given native language, it is tokenized as shown in the figure below.

choisir | *de* | *tout* | *tableau* | *exampletable*

Figure 3.2.2: The tokenization of the sentence in fig. 3.2.1. Each word is assumed as a stand-alone entity now.

Finally, each token is translated into MySQL syntax with the help of the chosen dictionary. The example in fig. 3.2.3 is the easiest one: each token is translated from French into MySQL syntax.

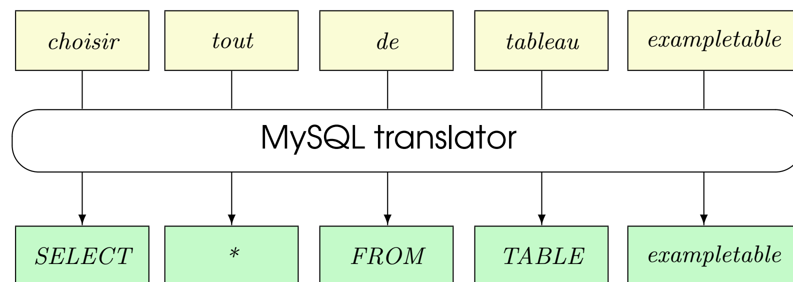


Figure 3.2.3: The second stage of the translation process from an original native sentence into MySQL syntax.

There might be more complicate cases where this correspondence does not hold because there is no such an apparently 'injective' relation from each native language word to the term in MySQL syntax; for example in fig. 3.2.4 (explaining a case in fig. 7.3.2), where the expression '*où id est majeur que 3*' (counting 6 tokens) turns into 'WHERE id $>$ 3' (counting 4 tokens): in fact the form '*est majeur que*' typically belongs to the spoken language, but it can be easily translated into the operator '$>$' (see also the analogous example in fig. 4.3.3, where the input native language is Italian).

How does it work? First one easily notices that only the word '*majeur*' is translated into MySQL syntax by means of the operator '$>$', whereas '*est*' and '*que*' are not: native language dictionaries include words which are not translated since they have no matching MySQL terms, but they are actually listed therein in order to guarantee the same full adherence and fluency like for everyday sentences.

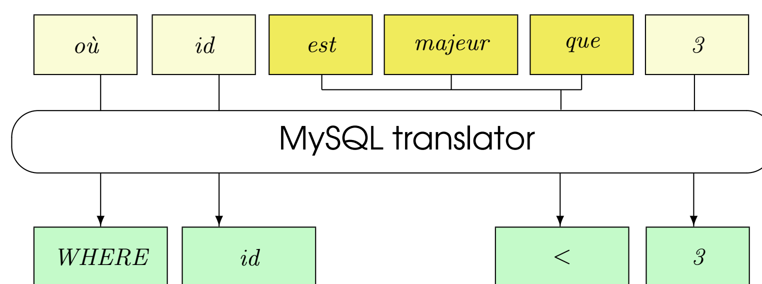


Figure 3.2.4: A more complicate case than in fig. 3.2.3. From the upper row, the darker tokens are translated into only one token of the lower row.

3.3 Remarks

It is important to understand that the kernel of the translation process relies upon a surjective function, although the previous two figures were not so much helping in this sense. While it is immediately clear from fig. 3.2.3, this might not follow from fig. 3.2.4; the goal of such figures just consist in introducing the reader to the possible casuistry at a superficial view, rather than deepening into the process where things radically change and are gathered under one fundamental concept. So fig. 3.2.4 is not misleading, but its real meaning shall be reinterpreted in the rigorous terms of fig. 5.8.3.

From now on, the reader is suggested to mostly refer to this latter figure and think in terms of single entities. *Each input token always translates into another token*, even when no final text is retrieved: this apparently looks like contradictory but, as shown in some cases, the translated token could be the empty string (red arrows in fig. 5.8.3).

4 Analyzing the dictionaries

4.1 From abstraction to concreteness

A major goal of this article is to teach developers on how to use this class and how additional native languages are supported. One initial difficulty during its early development was to choose a common referencing dictionary, used to start from some place in common and get finally to the translation. For convention, we will refer to two kinds of dictionaries from now on: '*abstract*' and '*concrete*'. This solution mainly counts on the abstract dictionary which unties all bonds from a native language.

The abstract dictionary does not exist by itself: it is nowhere found in the code files. Only concrete dictionaries exist and they are built upon a bi-dimensional array where each indexed entry either includes the native term and the associated code number.

In fact the abstract dictionary is both the set of directives for generating new concrete dictionaries and a sort of artifice for linking the latter ones; one can find the abstract dictionary as disguised in the code numbers set and during the working process. See fig. 4.1.1 and the further table. Every such number codifies the action to be performed: for example 0002 is the code to ignore the input term during the translation process; otherwise the search goes on: thus, if the matching number is found, the process picks up the related MySQL command, whereas the original word is copied into the final sentence if no such matching term is included inside the concrete dictionary.

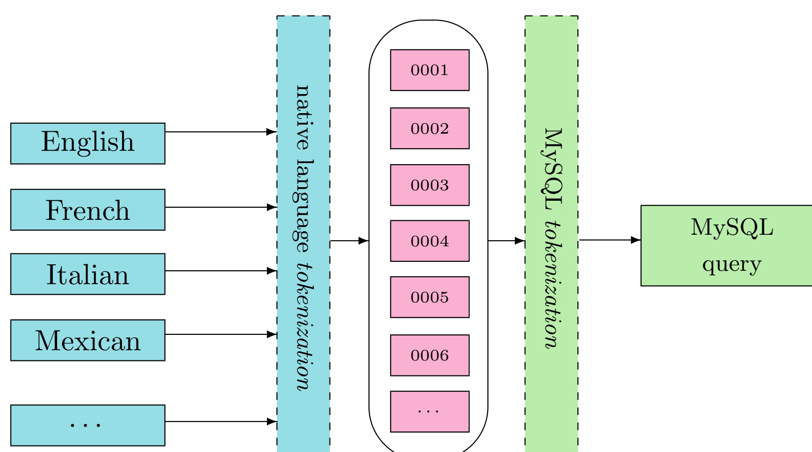


Figure 4.1.1: A different technical revisiting, explaining the general process for the examples displayed in the pictures 3.2.3 and 3.2.4. In case the native language tokens are converted into the related code numbers, later they are converted again into MySQL terms; otherwise, they are imported into the query as they originally are. The abstract dictionary (the centered oval) links the concrete ones between, that is, from the languages on the left to the MySQL syntax on the right.

The abstract dictionary lists an array of terms[3] and, for lessening the comprehension to the reader about how it works, *the meaning as well as the lexical analysis are both reported in English*, just because this is the standard language for papers diffusion: the meaning of each entry/term can be also easily understood in order to build up new concrete dictionaries. Here is the list of all entries. The entries sequence was taken from Italian dictionary (`it.php`) and thus the text analysis compiled per each entry was relatively written; in fact, although it often shows to be superfluous in English, other languages need verbs to be declined. The fourth column lists the matching English term, if existing, for pointing out to its meaning; otherwise field was left blank: in this case, the translator should look at the text analysis column for the given translation.

Let us see one example: the Italian dictionary includes compound prepositions[4] too. If the translated dictionary does not, just fill the corresponding entry with the related simple preposition (for example, entries 9 and 10). The filling sequence of terms in the abstract dictionary does not follow any rule: the list[5] was generated and expanded as the dictionary was tested with examples of growing complexity.

4.2 Concrete dictionaries: status and rules

All the already implemented dictionaries are updated to present. During the compilation of all entries, one takes care of including each one only once. One warmly encourages to add new synonyms for an higher and higher translation *performance* and *versatility*. For further expansions, every existing dictionary shall be updated with the same number of new terms.

If the proposal outlined in §8 would be welcomed by the developers community, one should discuss about the guidelines to drive the dictionaries maintenance.

[3]The developer of new dictionaries shall absolutely take care that every entry shall appear *once and only once* !

[4]Being expressed in one only word.

[5]Any resort might be however achieved if required for a better performance.

ORIGINAL QUERY FORM : **choisissez tout de la tableau exampletable;**
MYSQL QUERY FORM : **SELECT * FROM exampletable ;**

Id	Name	Address	Phone
1	Mario Rossi	Via le mani dal naso, 23	3-222-11442389
2	Valerio Bianchi	Piazza LaTenda, 11	33-455553434
3	Alessandro Rosa	Via Vai	333-2221221
4	Gino Lino	Via col vento	344-21232123
5	Guido LaVespa	Largo che ci passo	380-22300987
6	Armando Lanave	Corte Dichiarata	443-2323231
7	Gustavo Latorta	Strada Diritta	6-652-344232
8	Laura Diddio	Via del Cielo	1-111-3338764
9	Bruno Biondi	Vico Fico	3-233-2116783
10	Salvo Lapelle	Via da qua	5-555-2874102

Figure 4.1.2: Another example with French.

4.3 How to build a concrete dictionary

These are the instructions to be followed for building up a new dictionary:

1. *translate* all n entries in the same order as in the table at §11 so that they appear inside a bi-dimensional array of strings, where each element $[n][0]$ includes the term and $[n][1]$ includes the related code number;

NOTE : keep in mind that no duplicates shall occur. Each entry shall be translated into a distinct term. For example, each verb points to a distinct MySQL action thus it does need to be translated into a distinct term. Hence '*to show*' and '*to select*' shall be necessarily translated differently; it might happen, like in German, to erroneously turn both verbs here into '*zeigen*'.

2. *add* synonyms of entries, if any, at the end of the resulting array and check that synonyms keep the same code number: they add fluency to the parser and freedom of entering the sentences/queries.

3. *rename* the file according to the ISO 3166-1 country (bi-literal) codes of the language: for example, '`es.php`' for Spanish, or '`de.php`' for German, '`fi.php`' for Finnish, '`nl.php`' for Dutch or '`pt.php`' for Portuguese.

4. *release* the new dictionary under GPL v2 license (see §10).

ORIGINAL QUERY FORM : **seleziona id, name, address e phone dalla tabella exampletable ed ordina per name;**
MYSQL QUERY FORM : **SELECT id , name , address , phone FROM exampletable ORDER BY name ;**

id	name	address	phone
1	Company 01	Address 01	++39-123-45679876
2	Company 03	Address 03	Phone 03
3	Company 04	Address 04	Phone 04
4	Company 05	Address 05	Phone 05
5	Company 06	Address 06	Phone 06
6	Company 07	Address 07	Phone 07
7	Company 08	Address 08	Phone 08
8	Company 09	Address 09	Phone 09
9	Company 10	Address 10	Phone 10

Figure 4.3.3: One more complex example in Italian.

5 An example

5.1 Setting up the environment

A basic example, including a table filled with some data, was arranged inside the class package. Follow these instructions to set it up:

- *switch on* your web server and check if PHP is installed;
- *switch on* the MySQL server ;
- *open* the MySQL commands shell ;
- *load* the file **DBEXAMPLE.SQL** from the MySQL shell via

(fill the path field, delimited by %, according to your OS syntax and file location):

SOURCE %PATH%DBEXAMPLE.SQL

- *close* the MySQL shell and *open* your browser;
- *load* the example files `index.php` or `example.php`

5.2 Introducing the code

In the next section, we see the code example `index.php` on how to implement MySQL Babel. For sake of clarity, the reader should point out that the variables `$host, $dbname, $username, $psw` refer to host address, the database name, the user name and the related password respectively. All parameters shall be immediately entered because the class was planned to automatically open and close the database connection.

5.3 Overview

MySQL Babel works together with the supporting class Languages, handling the proper dictionary selection for the translation. The code of both classes has been included inside the same source file `mysql_trans.php` and the following example refers to `index.php`.

```
require_once('mysql_trans.php');

$tr = new mysql_babel( $host, $dbname, $username, $psw );

$lang = new languages($language);

if ( !( $lang->isClassOperative() ) ) {
    echo $lang->getErrMsg() ;
    return ;
}

$tr->insert_language_code( $language );

$tr->native_dictionary_array = $lang->get_native_dictionary() ;
$tr->mysql_dictionary_array = $lang->get_mysql_dictionary() ;
$tr->errors_messages_array = $lang->get_errors_array() ;

$tr->insert_mysql_source_cmdline( $query );

$tr->translate();

$sqlQRY = $tr->get_mysql_translated_cmdline() ;

$tr->run_query( true );
```

The most important passages, for the classes MySQL Babel and Languages, have been investigated in the next subsection, where one will see how this code works step by step.

5.4 Code explanation

As both classes are instantiated and initialized after the respective number of parameters are entered (see §4.3 for setting the string variable `$language`), the class Languages is tested for operativity by its member function `isClassOperative()`, that is, if the chosen dictionaries and error messages are available. If so, this class picks up the three arrays, required for translation; otherwise the running code is halted by displaying an error message, which is associated to a number[6]. Then, these three lines

```
$tr->native_dictionary_array = $lang->get_native_dictionary() ;
$tr->mysql_dictionary_array = $lang->get_mysql_dictionary() ;
$tr->errors_messages_array = $lang->get_errors_array() ;
```

link those three arrays to the working arrays of MySQL Babel, so that this latter class is now ready to work. The first array includes all terms of the selected native language to work with, the second one includes the list of the currently supported MySQL commands to be translated and finally the third array includes the error messages which have been dynamically implemented into a function loading the native language texts. Let `$query` be a variable (whose contents come from an edit box, for example) including the user input of the database query, written in the native language. So it is time to insert this string into MySQL Babel:

```
$tr->insert_mysql_source_cmdline( $query );
```

Now this class is set up for translation by calling the following function:

```
$tr->translate();
```

which also performs a bunch of subroutines to check both input and output data. Optionally one might want to retrieve the output translated query by the member function

```
$sqlQRY = $tr->get_mysql_translated_cmdline();
```

Notice the separator character '/' which sometimes appears twice in succession: this is because the translation process encountered the code '0002', which points to no action. Finally the following code:

```
$tr->run_query( true );
```

displays the resulting table[7], as for example shown in picture 4.1.2. Here below, the example[8] code to display fields name inside a table row is reported:

```
echo "<tr class=\"example1header\">\n";

while ( $cols = mysql_fetch_field( $h_qry ) )
{
```

[6] Three values were set: -1, 0, 1 which refer to no errors, to incorrect and to nonexistent language code respectively.

[7] The output style can be customized by modifying the code inside this same member function.

[8] For a performant code, Cascading Style Sheets were applied.

```
        echo "<td class=\"example1header\">";

        if ( $cols ) echo "$cols->name ";

        echo "</td>\n";
    }

    echo "</tr>\n";
```

The code to display the fields contents per each row runs analogously and also relies on a `while` loop but the function `mysql_fetch_row` is used instead.

MySQL Translator example

Insert below the query and choose the language

Mexican Submit

NATIVE LANGUAGE FILE : **mx.php**
ORIGINAL QUERY FORM : **selecciona todos los campos desde la tabla exampletable ;**
MYSQL QUERY FORM : **SELECT * FROM exampletable ;**

Id	Name	Address	Phone
1	Mario Rossi	Via le mani dal naso, 23	3-222-11442389
2	Valerio Bianchi	Piazza LaTenda, 11	33-455553434
3	Alessandro Rosa	Via Vai	333-2221221
4	Gino Lino	Via col vento	344-21232123
5	Guido LaVespa	Largo che ci passo	380-22300987
6	Armando Lanave	Corte Dichiarata	443-2323231
7	Gustavo Latorta	Strada Diritta	6-652-344232
8	Laura Diddio	Via del Cielo	1-111-3338764
9	Bruno Biondi	Vico Fico	3-233-2116783
10	Salvo Lapelle	Via da qua	5-555-2874102

Figure 5.4.1: Another example (included in the file '`example.php`'), where an input form was implemented. The query is in Mexican. By the combo on the right, user switches the native language to write queries.

5.5 Prelude : the idiomatic expressions

According to the main goal of MySQL Babel (entering queries as close as possible to everyday language), a native sentence shall be pre-processed, before the translation, to look for idiomatic expressions which might *be turned into known single tokens*. This task is achieved by the global function `pre_idiomatic`, declared in each dictionary file and dynamically loaded into memory (see §7.2). An example in Mexican is reported below:

```
function pre_idiomatic( $strIn ) {
    $strOut = str_replace( "todos campos", "todo", $strIn ) ;
    $strOut = str_replace( "todos los campos", "todo", $strOut ) ;
    $strOut = str_replace( "cada campo", "todo", $strOut ) ;

    return $strOut ;
}
```

This is a many-to-one function: a bunch of terms, often belonging to a given idiomatic form, turns into one. Besides improving the fluency of the input sentences, this member function fits the input requirements of the translation kernel – single tokens to be turned into single MySQL terms (refer to §3.3). An example follows, listing a same sentence written in the currently available native dictionaries:

FRENCH : choisir *chaque champ* de le tableau exampletable;
GERMAN : wählen *alle Felder* von das Tisch exampletable;
ENGLISH : select *each field* from the table exampletable;
ITALIAN : seleziona *tutti i campi* dalla tabella exampletable;
MEXICAN : seleccionar *todos los campos* desde la tabla exampletable;

The idiomatic expression in italic turns into the asterisk (*), so all above inputs turn into this simple query:

MySQL : **SELECT * FROM exampletable** ;

The function `pre_idiomatic` current status is still rough and its development might branch to different and independent paths, because idiomatic expressions relate to each language. Therefore any further improvement of this function should be achieved by native speakers autonomously.

5.6 Translation (I): towards the abstract dictionary

With regard to fig. 5.8.3, the core of the whole translation works throughout three passages. From §5.6 to 5.8, the internals of function `translate()` are described.

The preamble: it is really useful to arrange a working environment thus one checks either the existence of the input native language sentence, stored inside the variable `$work_str`, and its syntax by means of the member function `syntax()` which checks parentheses, commas and double quotes, then the tokenization is achieved by the string handling function `explode()`: here the separator between tokens is the blank space. This function returns an array of strings. Now the class gets ready for a first translation.

```
$tokenized_array = explode( " ", $work_str );
$coded_cmdline = "" ;

   foreach ( $tokenized_array as $token )
   {
       $bFound = false ;

       foreach( $this->native_dictionary_array as $entry )
       {
           $word = $entry[0] ;
           $code = $entry[1] ;

           if ( strcmp( $token, $word ) == 0 )
           {
               $coded_cmdline .= "$code@@" ;
               $bFound = true ;
           }
       }

       if ( !$bFound ) $coded_cmdline .= "$token@@" ;
   }
```

The two and doubly-nested `foreach` loops get each token (external loop) and then scans it (internal loop) inside the concrete dictionary, the member array `$this->original_array`. So `$coded_cmdline` finally consists of a unique string where those tokens, found inside the concrete dictionary, are represented by the related code number; otherwise (when `$bFound` is false) they are copied into the string as they are:

```
if ( !$bFound ) $coded_cmdline .= "$token@@" ;
```

As one notices on the right, the tokens separator was set to the slash '/' for completely differing such string contents from the original ones: see also footnote at p. 17 for more related information.

5.7 Translation (II): turning into MySQL

Now the goal is to move from the abstract dictionary to the MySQL form. The method is the same doubly-nested loop as shown before.

Here we start from what we left, the `$coded_cmdline` variable, and then we initialize the string `$this->mysql_query_string` for progressively store, as the internal loop runs, the resulting MySQL query:

```
$tokenized_array = explode( "/", $coded_cmdline );
$this->mysql_query_string = "" ;

  foreach( $tokenized_array as $token )
  {
      $bFound = false ;

      foreach( $this->mysql_dictionary_array as $entry )
      {
         $word = $entry[0];
         $code = $entry[1];

         if ( strcmp( $token, $code ) == 0 )
         {
            $bFound = true ;

            if ( strcmp( $code, "0002" ) != 0 )
            $this->mysql_query_string .= "$word " ;
         }
      }

      if ( !$bFound ) $this->mysql_query_string .= "$token " ;
  }
```

The variable `$tokenized_array` points to a uni-dimensional array including all tokens coming from the variable `$coded_cmdline`; each element is a token whose related code number shall be compared with the codes inside the MySQL dictionary. The `$mysql_dictionary_array` indicates the MySQL commands dictionary, a bi-dimensional array where each element stores two strings: the term and the related number code. If so, the token is a MySQL command and then the translated, otherwise the token content is copied as it is into the output variable `$this->mysql_query_string`. Notice, at the bottom of the previous code, that the translation skips all tokens whose related number is 0002.

Figure 5.7.2: Another shot from '`example.php`': the input query is in Italian.

5.8 Translation (III): the query finalization

Although the query came to light in MySQL form, the process is not definitely over. Since it is not a true translation absolutely and the goal of the class code does nor need an high algorithmic complexity – as we remarked in §2.2, the resulting expression might still include formal errors, that is, a flow of terms which do not obey to the (even strict) MySQL grammar and syntax: the sentence in question cannot be assumed as a correct query and will not run yet. So it was found useful to add another stage, the last one, defined as '*query finalization*' because consists of the conclusive correction and refinement which closes the whole process. It is codified into a member function, `finalize`, whose purpose is to fix definitely all flaws and it automatically runs inside the function `translate()`. It comes natural that this function performance could increase only as the number of discovered errors grows: thus several queries of different kind need to be tested before reaching to a satisfactory level. Honestly, the author did not look so much for such possible errors and just begun this quest from this current version of the function, where few flaws[9] were detected and recovered:

```
function finalize( $strIn )
{
  $strOut = str_replace( "SELECT TABLE", "SELECT", $strIn ) ;
  $strOut = str_replace( "SELECT FIELDS", "SELECT", $strOut ) ;
  $strOut = str_replace( "SELECT FIELD", "SELECT", $strOut ) ;
  $strOut = str_replace( "FROM TABLE", "FROM", $strOut ) ;
  $strOut = str_replace( "SUM (", "SUM(", $strOut ) ;
  $strOut = str_replace( "AVG (", "AVG(", $strOut ) ;
  $strOut = str_replace( "MAX (", "MAX(", $strOut ) ;
  $strOut = str_replace( "MIN (", "MIN(", $strOut ) ;
  $strOut = str_replace( "< , =", "<=", $strOut ) ;
  $strOut = str_replace( "> , =", ">=", $strOut ) ;
  $strOut = str_replace( "\'", "'", $strOut ) ;
  $strOut = str_replace( "\\\"", "\"", $strOut ) ;
  $strOut = str_replace( ", ", ",", $strOut ) ;
  $strOut = str_replace( " ,", ",", $strOut ) ;
```

[9]One should also find the way to input strings with accented characters and keep accents inside the database.

```
  $strOut = str_replace( " , ", ",", $strOut ) ;

  return $strOut ;
}
```

This function is subjected to continuous updates so the current version might be not the same as shown here in this article.

6 Side functions

A number of functions was designed to give the developer the chance to track down errors, to debug the translation process and to customize the output errors messaging.

6.1 Retrieving the parameters

Despite of other high level languages like C++ or Java for example, PHP cannot allow the definition of multiple constructors, so that it was preferred to code one accepting all parameters for MySQL database connection,

```
mysql_babel( $server, $dbname, $user, $psw ) ;
```

rather than having a basic one so that the parameters should have been inserted later by proper member functions. The adopted solution thus wants to lessen the code somehow. The functions aimed to retrieve such parameters are listed below and their work can be easily understood directly from the nomenclature:

- `get_mysql_server`
- `get_mysql_dbname`
- `get_mysql_user`
- `get_mysql_psw`
- `get_language_code`
- `get_mysql_source_cmdline`
- `get_mysql_translated_cmdline`
- `get_mysql_dbname`
- `get_mysql_user`
- `get_mysql_psw`
- `get_language_code`
- `get_mysql_source_cmdline`

The functions on the left have the input version too, in case developers need to replace the database connection parameters during the execution of the code:

- `insert_mysql_server`
- `insert_mysql_dbname`
- `insert_mysql_user`
- `insert_mysql_psw`
- `insert_language_code`
- `insert_mysql_source_cmdline`

6.2 Safe input data

There is the actual possibility that the native sentence tokenization might include, for sentences devoted[10] to SELECT, to INSERT or to UPDATE, the same words as inside the concrete dictionary, so that this would cause such strings to be 'corrupted' by the translation process. The member function `safe_data` was created to avoid this corruption and it is called twice in the function `translate`: before the translation, strings protection is first locked on, and

[10]With strings between quotes ([‘] and [’]) or doublequotes ([“] and [”]).

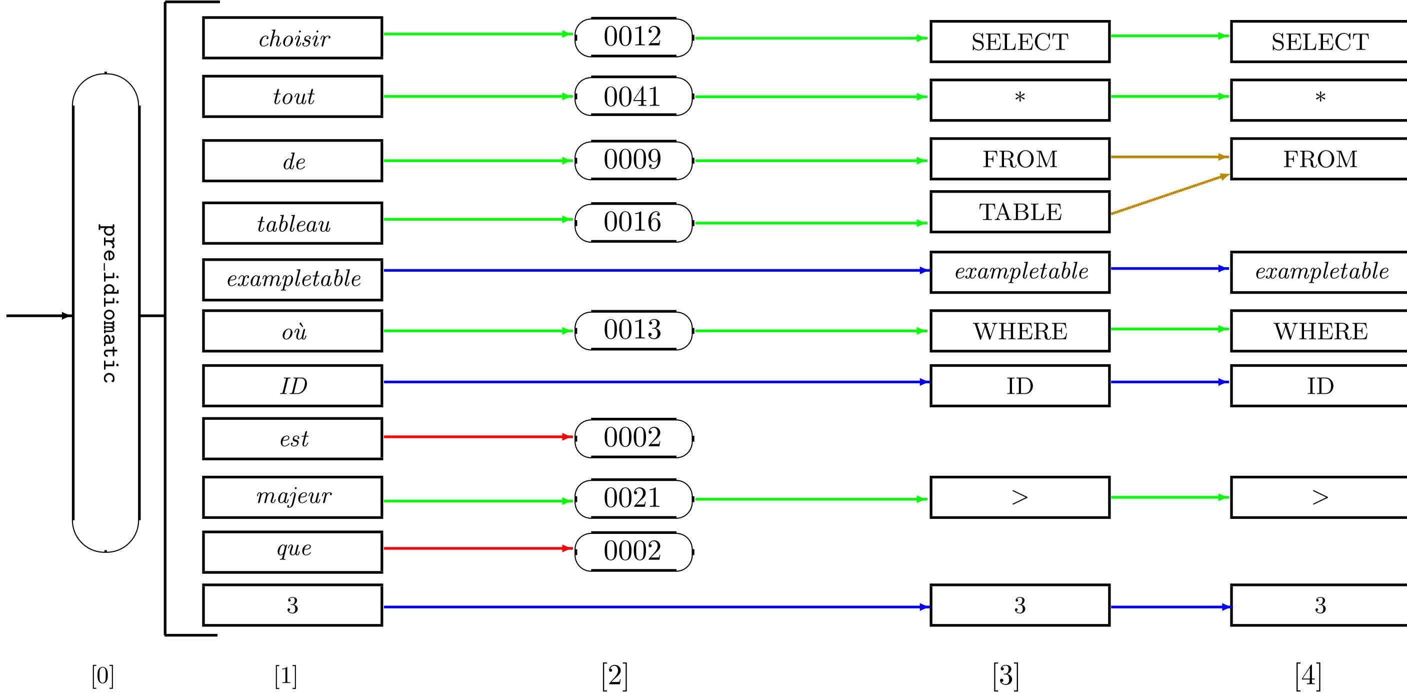


Figure 5.8.3: The example in §3.2 is investigated again, but pointing out either to the abstract dictionary role and to the finalization. The whole process consists of five stages. [0] This does not properly belong to the translation process, rather it is a preparation stage to it: the input sentence is scanned and idiomatic form are transformed so that the tokens from resulting string could be later related to MySQL terms by a surjective relation. [1] The native language query was input and then tokenized. [2] Each token is scanned: if found in the abstract dictionary, it is associated to a code number leading to the translation (green arrow); but, in case of no action code, the token itself is not carried on through the next stages (red arrow). If no code is found, it is just copied (blue arrow). [3] All resulting tokens are output into the rough version of the MySQL query. [4] This query is finalized (brown arrow) and it is ready to interrogate the database.

again, as the process is over, to lock it off. The function gets two parameters, the input query and the locking action boolean status:

```
safe_data( $qry_input, $bLock )
```

In this example, let the input data be written in Italian as follows:

```
inserisci nella tabella exampletable i valori (24,‘Jackob’,
‘Contrada per caso’,‘344-4543876’ ) ;
```

the function `safe_data` protects the quoted strings by packing them with the double character ‘@@’:

```
inserisci nella tabella exampletable i valori (24,‘Jackob’,
‘Contrada@@per@@caso’,‘344-4543876’ ) ;
```

So translation is unable to affect the quoted strings, especially those with blank spaces which are more susceptible to ‘corruptions’, and thus the above sentence finally turns into:

```
INSERT INTO exampletable VALUES (24,‘Jackob’,‘Contrada@@per@@caso’,
‘344-4543876’ ) ;
```

Calling `safe_data` again removes all ‘@@’ and resumes the original strings form; finally this is the correct MySQL query:

```
INSERT INTO exampletable VALUES (24,‘Jackob’,‘Contrada per
caso’,‘344-4543876’ ) ;
```

Otherwise, without any such input data protection, one would find this corrupted form:

```
INSERT INTO exampletable VALUES (24,‘Jackob’,‘Contrada BY
caso’,‘344-4543876’ ) ;
```

That is, quoted strings would be also affected by tokenization and the term ‘per’ is associated, according to the Italian concrete dictionary, to the translated MySQL term ‘BY’. Check this latter dictionary to deepen the reasons why this one would have been the resulting query.

6.3 Errors tracking

The functions `getErrNo` and `getErrMsg` are devoted to track down errors and display the related message respectively. The global variable `$errNo` was declared in order to store the number when an error occurs during the class code execution. It currently ranges from 0 to 9, where 0 means that no errors were detected; the values 1-9 point to errors coming out from database connection attempts, from syntax or from query results.

`$errNo` is read by `getErrMsg`: any call to this function retrieves the error message from the internal array, fulfilled by a set of strings which are dynamically loaded into the class by the function `load_errors()`, according to the input native language code.

6.4 Queries and dictionaries debugging

The developer might also find useful to retrieve the command line written into the abstract dictionary codes, in order to *debug* the generation of the new dictionary. The member function `get_debug_abstract_cmdline()` makes it possible. For example, let the source Italian command line be:

seleziona id, name, address e phone dalla tabella exampletable ed ordina per name;

which finally turns into the MySQL query:

SELECT id, name, address, phone FROM exampletable ORDER BY name;

What follows is the related debug command line:

0012/id/,//name/,//address/0001/ (*... continues into the next line*)
phone/0009/0016/exampletable/0002/0014/0039/name/;/

A look at the last command line may help to debug the conversion from the dictionary to another one, assuring for example that the code numbers for the entries have been really compiled.

6.5 Output customizations

The developer can customize the screen output in case of errors by the member function `general_error_output()`. This function is called inside the member function `run_query`, explained here at page 10.

One could also customize the output style of resulting tables: this does not regard MySQL Babel development up to present, but it might be something to implement later. Both examples inside the package apply CSS but, for example, one could code a bunch of member functions to handle the tables graphics preferences[11].

6.6 Advanced queries

In the table at p. 24, some advanced queries from different native languages are presented and translated in order to show the efficiency of the class for different tasks with MySQL queries. Look at the first and second examples: here the command 'WHERE' a double condition.

6.7 Dictionary identification

One might want to retrieve the data related to each dictionary identification for any further use inside one's code; see fig. 5.7.2. This is the current list of related global functions (thus, open to new entries), coded inside each dictionary file:

- `get_dict_original_name()`
  retrieving the original name in the native language;
- `get_dict_english_name()`
  retrieving the English name of the dictionary;
- `get_dict_version()`
  retrieving the version number;
- `get_dict_ISO_code()`
  retrieving the ISO 3166-1 code number;
- `get_dict_file_name()`
  retrieving the full file name;
- `get_dict_author()`
  retrieving the name of the author compiling the dictionary;
- `get_dict_last_modified()`
  retrieving the date of last modification;

[11] To be included into point 3 at p. 21

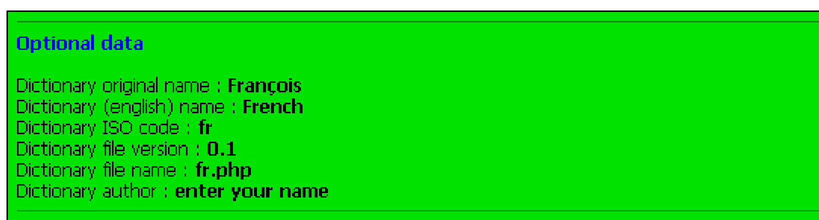


Figure 6.7.1: A screen shot listing all data retrieved by dictionary identification functions.

7 The class Languages

7.1 Short overview

As in §5.4, the class Languages was implemented in order to separate the task of 'recruiting' the required dictionaries from translation into two distinct blocks of code. All this allows a tidy code for the developer.

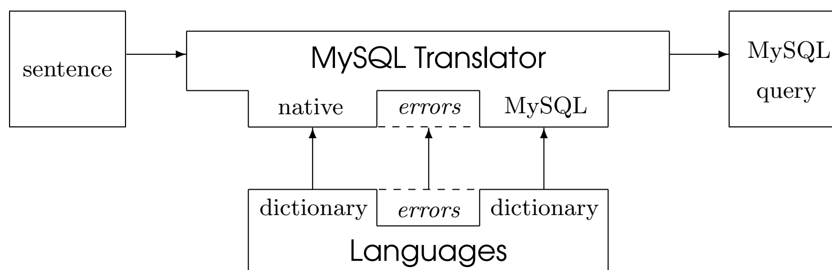


Figure 7.1.1: The two classes cooperation.

Alike for MySQL Babel, there is a parameterized constructor in one variable accepting the ISO country code of the language (see §4.3). As the constructor is called, the class automatically checks whether the country code fits the requirements; if so, then it loads the MySQL dictionary and, if the file exists, the native language dictionary into two different bi-dimensional arrays for passing them to MySQL Babel. Both arrays can be retrieved by:

`get_native_dictionary()` and `get_mysql_dictionary()`

The class is operational for dictionaries linking with the translator, when the member function `isClassOperative` returns `true`. Errors list are passed to the translator by `get_errors_array()`.

Any sort of modification of MySQL dictionary shall be achieved inside the member function `loadMySql`.

7.2 Dynamic linking

As shown in the previous sections, native dictionaries have been distributed inside different files, each one including only one list of words. Each file either includes two arrays: one for the dictionary, the other for the error messages. What is the class Language management of both arrays? The solution relies on a sort of '*dynamical library linking*' (very distantly borrowed from C++) between the main class file and those ones containing the dictionaries, which are loaded by the following two global functions `load_language()` and `load_errors()`, implemented inside each dictionary file:

```
function languages( $lng_code )
{
   $this->loadMySql() ;

   if ( strlen( $lng_code ) != 2 ) $this->errNo = 0 ;
   else
   {
        $this->language_code = $lng_code;

        $lng_code = strtolower($lng_code);
        $this->dynamic_file_path = "$lng_code.php" ;

        if ( file_exists( $this->dynamic_file_path ) )
        {
           require_once ( $this->dynamic_file_path );

           $this->native_language_array = load_language() ;
           $this->errors_messages_array = load_errors() ;
           $this->errNo = -1 ;
        }
        else $this->errNo = 1 ;
    }
}
```

The latter two functions retrieve two arrays which will be stored in the two member arrays of the class Language and whose values will be later copied inside the arrays of MySQL Babel as explained at page 10. The core of the dynamical linking stands inside these lines:

```
        $this->language_code = $lng_code;

        $lng_code = strtolower($lng_code);
        $this->dynamic_file_path = "$lng_code.php" ;

        require_once ( $this->dynamic_file_path );
```

It is an easy job: the bi-literal country code input is caught and then turned into a file name (with .PHP extension) which will be taken into the whole code machinery to call the two globally declared functions; obviously this method demands that only one file can be required at a code run, otherwise the naming conflict would arise. Last but not least, the member function `loadMySql()` loads the MySQL dictionary into another third array, to be passed later in the above mentioned lines of code.

7.3 Errors management

This class keeps an extremely simple error tracking management. Since its job is few, also possible errors are. In fact, only three cases are listed and coded by -1, 0, 1: they point to no errors, to missing and unsupported language code respectively. While the second case refers to a nonexistent file, the last one means that a country code with wrong syntax was entered.

ORIGINAL QUERY FORM : **choisissez id et phone de la tableau exampletable où id est majeur que 3;**
MYSQL QUERY FORM : **SELECT id,phone FROM exampletable WHERE id > 3 ;**

id	phone
4	344-21232123
5	380-22300987
6	443-2323231
7	6-652-344232
8	1-111-3338764
9	3-233-2116783
10	5-555-2874102

Figure 7.3.2: An example on how the class works with French.

8 Proposal for a development plan

8.1 Disclaimer

The status of this class is very primitive at present. The consistence of the same dictionaries is even rough: the author has honestly few competence in linguistics and even none for a couple of existing dictionaries (say, German). But they have been included in this version in order to simply set the ground and create examples of how MySQL Babel works with different languages: that is, by now the main goal is just to offer the work contexts for the class. The dictionaries themselves do not pretend to be absolutely correct and functional at present.

In author's opinion, even an acrobatic context like this is more productive than keeping something hidden to any interested audience due to a lack of valid support, in this case for fixing the stability of dictionaries. Here open problems can be discussed, attacked and, hopefully, solved.

For example, another topic of discussion would be whether MySQL Babel might actually work with native languages whose syntax does not follow the sequence

action/verb — object — complements.

The original implementation of MySQL Babel was achieved with languages based upon such a syntax (Italian, French, English and Mexican). In general, this would be a restriction and probably the code should be even revisited in case the future of this project would see it extending to languages with different syntactical rules. *Therefore any further improvement does require the expertise of native speakers for each dictionary.*

8.2 Directives list manifesto

In author's mind, there would be still much work for one person: the full development of this class can be actually intended as a larger project, consisting of seven main directives:

1. Keep on *fulfilling* the existing dictionaries by translating the still missing MySQL commands;

2. *Create* new same-sized dictionaries as at point 1;

3. *Improve* the running performance of MySQL Babel class code, if required;

4. *Increase* the query finalization quality performance;

5. *Write* the manual[12] for each native language;

6. *Extend* MySQL Babel class performance to other SQL engines;

7. *Re-code* the MySQL Babel class into other programming languages which support the linking interface to other database technologies, according to point 4;

NOTE: MySQL Babel *cannot be distributed unless the available dictionaries are not provided by companion manuals: it would make no sense to spread this package without related instructions. This is one reason why the author calls for supporters in this direction too.*

8.3 Remarks about the plan

It is clear that *point 1* strongly requires the support of developers mastering native languages, like Finnish, German, Portuguese for example, thus it is hoped to gather an international task force. In this direction, the author suggests to improve a more *fluent parsing*[13] of the native language sentences

- by adding the proper entries[14] in the abstract dictionary (and, consequently, into the concrete[15] ones);

- by handling those languages with non-latin alphabet, that is, not supported by the ASCII characters table, like Greek, Polish, Russian and even Chinese, Japanese. Formally, the solution should rely upon the UNICODE table here.

The already available dictionaries show how to implement this all.

About *point 2*, the author is not so expert with the whole MySQL commands set, therefore such a completion requires an external support too. The dictionary at page 26 shall not be absolutely considered as definitive.

In this direction, *point 3* shall obviously follow point 2 because one manual[16], illustrating all features for each language, shall be written.

Points 4 and 5 open to very wide possibilities. MySQL Babel is an abstract idea of a functionality which can be implemented into different languages operating with the different DBMS. Here we just presented one implementation into PHP because this language features a programming interface dealing with MySQL engine. But one wishes that other technologies relying on the SQL standard (Firebird, Ingres, Oracle, PostgreSQL, SQLite, Sybase, . . .) could be supported. Thus MySQL Babel might represent the start-up for a wider project involving SQL technology and which may be called under the name of 'SQL Babel'.

Point 6 will be developed in case of new ideas and suggestions in that direction.

Point 7 calls for new tests on more complicate queries, in order to discover translation flaws and fix them.

[12]Such documentation shall include all available commands in the given native language, with examples (of increasing complexity) per each command; in addition, one list of contents sorted in alphabetic order. And two ending indexes shall appear: one for native language, the other for MySQL terms respectively.

[13]That is, reaching to a closer and closer similarity to everyday parsed sentences.

[14]Adverbs and propositions.

[15]See the related definition in §4.

[16]Suggested format is the Acrobat PDF.

8.4 Calling for support

For the reasons explained in the beginning of the previous section, *the author looks for a maintainer of the whole project, hopefully devoting one web-site*[17] *to it exclusively, as well as the collaboration of several partners would be required to achieve the above points, especially for the making of new dictionaries, which deserve native speakers and developers.*

The maintenance of a web site would be appreciated for class code distribution, but mostly for discussing changes and set up a coordinating board of the code development. I especially mean to major changes, i.e. involving for example the abstract dictionary, which would influence concrete ones contents; here suggestions could be advanced, discussed and, if approved, implemented and then ported to concrete dictionaries. But also for minor changes, that is, enhancing the list of idiomatic expressions (for each native language) handled by MySQL Babel: here dictionaries could be upgraded, stored and listed into an archive reporting always the latest date of the new file, in order to let developers check the current version.

9 Acknowledgments

The author is in the debt with Arjen Lentz (MySQL AB) for his interests in this code and for offering the chance of writing and publishing this article. A special thank to people contributing to generate the currently available concrete dictionaries and to anyone going to support the plan at §8.

10 License

All the code, related to the class MySQL Babel, together with all already existing dictionaries or those forthcoming ones, can be assumed as a whole code package or scattered into different files regarding the class itself or each dictionary. In both cases, it shall be intended as free software; you can redistribute this and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation; either version 2 of the License.

All the code, related to the class MySQL Babel, is distributed in the hope that it will be useful to the developers community, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

11 Conclusions

MySQL Babel can be asked to author or freely downloaded from author's web-site[18], tested, used, developed according to the directives suggested at §8.

Alessandro Rosa
zandor_zz@yahoo.it
Brindisi, Italy

[17] For example, it may include as follows: 1. a page illustrating the goal of the project as well as code examples; 2. one download page listing all available native language dictionaries (with distinct code versions) and related user guide, together with the download of the class with or without all available dictionaries; 4. one page gathering suggestions.

[18] http://www.malilla.supereva.it/Mirror/Pages/code.html

A Appendix: Examples of advanced queries

FRENCH, GERMAN
insères en exampletable (Id, Name, Address, Phone) les valeurs (12, "Ciccio", "Via da là", "+555-555555");
einlegen in das Tisch exampletable (Id, name, address, phone) die Werte (12, "Ciccio", "Via da là", "+555-555555");
INSERT INTO exampletable (Id , Name , Address , Phone) VALUES (12 , "Ciccio" , "Via da là" ,"+555-555555");

FRENCH
altère tableau exampletab et renomme exampletable;
ALTER TABLE exampletab RENAME exampletable;

FRENCH
choisissez id, name et phone de tableau exampletable où id est majeur et égal à 3 simultanément id est mineur et égal à 7 alors ordre pour name;
SELECT id , name , phone FROM exampletable WHERE id >= 3 AND id <= 7 ORDER BY name;

FRENCH, GERMAN, ITALIAN
choisissez id et address de tableau exampletable où id est distinct de 4;
zeigen id, address von das Tisch exampletable wo id ist zusammen 4;
mostra id e address dalla tabella exampletable dove id é diverso da 4;
SELECT id,address FROM exampletable WHERE id != 4;

GERMAN
zeigen alle Felder von das Tisch exampletable;
SELECT * FROM exampletable ;

ITALIAN
*calcola (5 + 12)*5/9;*
SELECT (5 + 12)*5/9;

ITALIAN, MEXICAN
seleziona tutti i campi della tabella exampletable dove id é maggiore di 5 ed insieme address include la parola "%via%" ;
selecciona todos los campos desde la tabla exampletable donde id es mayor que 5 y junto con address contiene la palabra "%via%" ;
SELECT * FROM exampletable WHERE id > 5 AND address LIKE "%via%";

MEXICAN
selecciona id, name y phone desde la tabla exampletable entonces ordena por phone juntos id;
SELECT id , name , phone FROM exampletable ORDER BY phone AND id;

MEXICAN
selecciona id y name desde la tabla exampletable como name;
SELECT id , name FROM exampletable AS name ;

B Appendix: the abstract dictionary structure

N.	CODE	LEXICAL ANALYSIS	MEANING	ACTION
0	0001	conjunction	*and*	Yes
1	0002	verb 'to be', 3rd singular	*is*	No
2	0002	preposition	*at/to*	No
3	0002	preposition	*with*	No
4	0002	conjunction	*or*	No
5	0002	definite article/masculine/singular	*the*	No
6	0002	definite article/feminine/singular	————-	No
7	0002	definite article/neutral/singular	————-	No
9	0002	definite article/masculine/plural	—————	No
8	0002	definite article/feminine/plural	—————	No
10	0002	definite article/neutral/plural	—————	No
11	0009	preposition	*from*	Yes
12	0011	verb/imperative tense/2nd singular	*to use*	Yes
13	0012	verb/imperative tense/2nd singular	*to select*	Yes
14	0015	verb/imperative tense/2nd singular	*to show*	Yes
15	0013	adverb	*where*	Yes
16	0014	verb/imperative tense/2nd singular	*to order*	Yes
17	0016	noun / singular	*table*	Yes
18	0017	noun / plural	*tables*	Yes
19	0018	noun / plural	*fields*	Yes
20	0019	verb/present tense/3nd singular	*includes*	Yes
21	0020	adjective	*equal*	Yes
22	0021	comparative	*larger,* >	Yes
23	0022	comparative	*lesser,* <	Yes
24	0023	verb/imperative tense/2nd singular	*to create*	Yes
25	0024	verb/imperative tense/2nd singular	*to drop*	Yes
26	0025	verb/imperative tense/2nd singular	*to delete*	Yes
27	0026	verb/imperative tense/2nd singular	*to update*	Yes
28	0027	verb/gerund	*to set*	Yes
29	0028	verb/imperative tense/2nd singular	*to insert*	Yes
30	0029	verb/imperative tense/2nd singular	*to replace*	Yes
31	0030	preposition	*into*	Yes
32	0002	noun/singular	*field*	Yes
33	0031	noun / plural	*values*	Yes
34	0032	adjective	*ascending*	Yes
35	0033	adjective	*descending*	Yes
36	0034	noun / singular	*instance*	Yes
37	0035	noun / plural	*instances*	Yes
38	0036	noun / singular	*option*	Yes
39	0037	noun / plural	*options*	Yes
40	0038	noun / singular	*database*	Yes
41	0003	verb/imperative tense/2nd singular	*to truncate*	Yes
42	0004	verb/imperative tense/2nd singular	*to alter*	Yes
43	0005	verb/imperative tense/2nd singular	*to add*	Yes
44	0006	noun / singular	*column*	Yes
45	0007	verb/imperative tense/2nd singular	*to rename*	Yes

N.	CODE	LEXICAL ANALYSIS	MEANING	ACTION
46	0019	adverb (inclusion of strings)	*like*	Yes
47	0010	verb/imperative tense/2nd singular	*to group*	Yes
48	0039	preposition	*by*	Yes
49	0002	preposition	*than*	No
51	0012	verb/imperative tense/2nd singular	*to make*	Yes
52	0040	noun	*average*	Yes
53	0041	noun	*all*	Yes
54	0002	adverb	*then*	No
55	0042	noun	*sum*	Yes
56	0012	verb/imperative tense/2nd singular	*to find*	Yes
57	0043	noun	*max*	Yes
58	0044	noun	*min*	Yes
59	0045	preposition	*together*	Yes
60	0046	preposition	*as*	Yes
61	0047	adverb	*distinct*	Yes
62	0012	verb/imperative tense/2nd singular	*to compute*	Yes