Case study 3

Interactive SQL interpreters

Objective: An interactive SQL interpreter is this kind of graphical interface through which one enters an SQL query and that displays the result of its execution in a text window. They are used, for instance, to learn SQL or to test and tune SQL queries that are to be integrated in application programs. There are many of them available for free on the internet or included in relational DBMS. In this study, we will build, step by step, our own SQL interpreter by implementing the functions and features we want to use, notably (but not exclusively) to train students in writing SQL queries.

Starting from a very tiny interpreter (just 7-character long!) we will build a series of more comprehensive and versatile versions, up to the last one, that will be able, not only to execute the queries submitted by the user, but also to evaluate their correctness.

All these versions are available as two ready to run applications.

Keywords: SQL interpreter, GUI, learning SQL, query evaluation, multiset, set operator, base64

3.1 Introduction

The graphical interface of the SQLfast environment is particularly fit for experimenting with the SQL language: we type a query in the main window, then we click on button **Run** and finally we examine the result in the output window. That is fairly convenient to learn the concepts of databases and SQL.

This interface is quite general but may be felt too complex or, on the contrary, too simplistic, as a support to SQL training. So, why not develop our own graphical interface that meets our specific needs? For example we would like, among others:

- to integrate the query and output text fields into a single window
- to discard all these useless buttons and menu items that clutter the interface
- to add new functions specifically devoted to SQL learning
- to customize the layout of the interface
- to display more user-friendly error messages
- to integrate tutorial and exercise modules
- and (let us have a dream) to automatically evaluate student's answers to the exercises.

The goal of this study is to explore some techniques to implement these wishes into what we will call *Interactive SQL interpreters*.

3.2 The smallest interactive interpreter in the world!

For this first try, let us work with the *Basic* interface (check the top right label of the main window). We open database ORDERS.db, we type the statements of Script 3.1 then we click on button **Run**. The first statement invites the user to type an SQL query and stores it in variable **Query**. The contents of this variable forms the second statement, which is then executed. The result appears in the standard output window (Figure 3.1).

```
ask Query;
$Query$;
```

Script 3.1 - An elementary interactive SQL interpreter

Note (useless, just for fun)

This script can even be made more compact (as low as **7 characters**, space and semi-colons excluded) by reducing the name of the variable to a single letter:

```
ask Q;
$Q$;
```

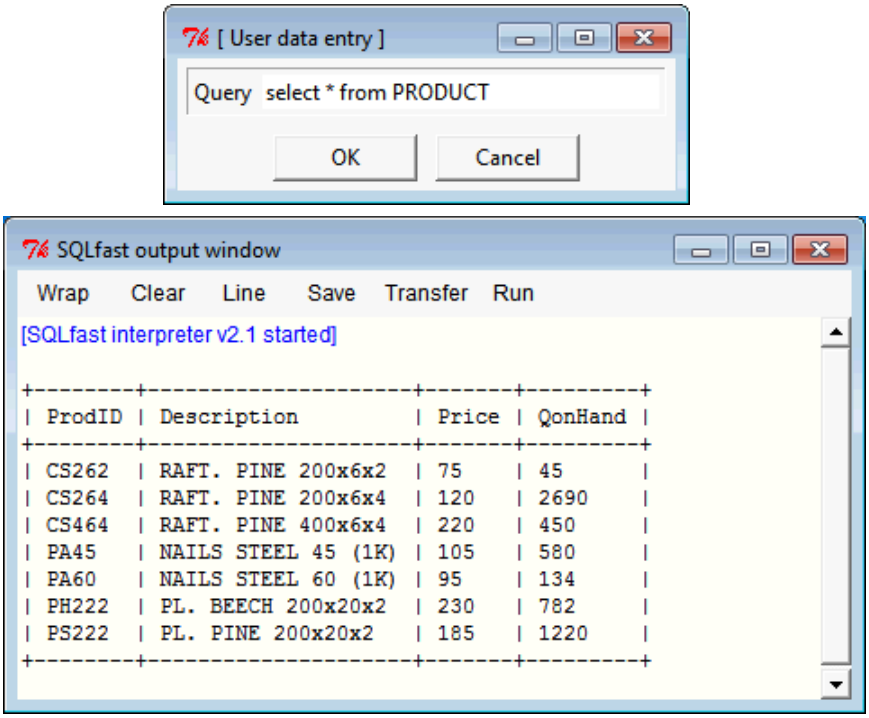

Figure 3.1 - The elementary query entry box and the output window

Extending this script by making it *iterative* is quite easy (Script 3.2). The query entry box opens repeatedly, until the user has no query to execute any more and clicks on button **Cancel**.

```
while (True);
  ask Query;
    if ('$DIALOGbutton$' = 'Cancel') exit;
  $Query$;
endwhile;
```

Script 3.2 - An elementary interactive SQL interpreter

Now we will learn to develop more sophisticated interpreters.

Though quite operational, this first trial is not particularly elegant nor easy to use for any serious experimentation.

First, queries are limited to one short line. Fine for some elementary queries from beginners, but submitting more complex queries will be awkward. Let us replace the data entry box by a **text entry box** (Figure 3.2). We just replace statement **`ask`** with **`askText`** (Script 3.3).

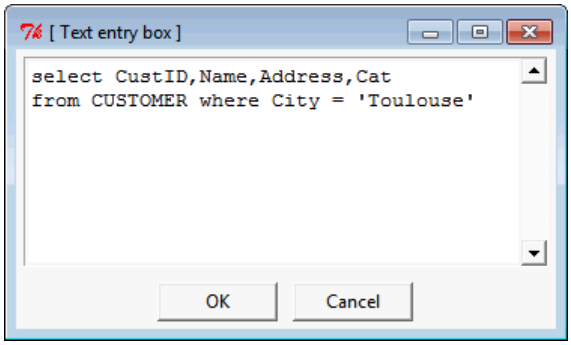

Figure 3.2 - Getting the query to submit through a text entry box

```
while (True):
  askText Query;
    if ('$DIALOGbutton$' = 'Cancel') exit;
  $Query$;
endwhile;
```

Script 3.3 - Entering larger queries in a text box

Now, let us integrate the query and output windows. To do so, we abandon the standard output window and we add a new text field in the query entry box. The result set of the query is sent in variable **result** by redirecting the output channel to this variable:

```
outputAppend result.var;
```

The contents of **result** is then shown by statement **showText**. Both text fields are assembled vertically by an **askCombo** statement:

```
askCombo [askText Query]|[showText result];
```

The code of this version is shown in Script 3.4.

```
set result = ;
outputAppend result.var;
while (True);
  askCombo [askText Query]|[showText result];
    if ('$DIALOGbutton$' = 'Cancel') exit;
  $Query$;
endwhile;
```

Script 3.4 - Integrating query and output text boxes

3.3 A simple but realistic SQL interpreter

It is time to leave these first experiments for a more professional approach.

First, we adjust the **size of the text fields**. Eight lines for the query field and twenty four lines for the result field seem fairly comfortable. In addition, **word wrapping** in the query field is what we expect (long lines are always visible) while it would spoil the presentation of wide result sets in the result field. Hence the different parameter settings for these fields:

```
[askText Query = [/w1/y8]]|[showText result = [/w0/y24]]
```

Now, let us address the question of the **contents** displayed in the text fields. We decide that the query that has been executed **remains** in the **query field**, so that the user can modify it for further execution, for example to fix errors or to run variants. We replace statement **`askText`** by **`askText-u`**, that displays the contents of variable **Query** for modification (**`-u`** for *update*).

As to the contents of the **result field**, we find it useful to **copy the query** before execution (**`write-ab`** `$Query$`). This will produce a nice report in which each query is followed by the result set of its execution.

Finally, the layout in which users enter their queries in the query field is not always elegant. So, we rewrite it both in the query and the result fields. For this, we use the *pretty print* function of the **LStr** library:

```
function Query = LStr:PrettyPrint {$Query$};
```

If this query is typed in a single line in the query field:

```
select CustID,Name,Address,Cat from CUSTOMER where City =
'Toulouse'
```

then it is rewritten in both text fields as:

```
select CustID,Name,Address,Cat
from   CUSTOMER
where  City = 'Toulouse'
```

These features are translated in Script 3.5. Figure 3.3 shows the state of the SQL interpreter when the query mentioned above has been entered and executed, then executed again for new value `'Paris'`.

3.4 Error management

If an erroneous SQL query is typed in (e.g., `select * from KUSTOMER`), as is natural from SQL learners, an error message will pop up, like that of Figure 3.4.

```
set Query,result = , ;
outputAppend result.var;
while (True);
  askCombo [askText-u Query = [/w1/y8]]
           |[showText result = [/w0/y24]];
    if ('$DIALOGbutton$' = 'Cancel') exit;
  function Query = LStr:PrettyPrint {$Query$};
  write-ab $Query$;
  $Query$;
endwhile;
```

Script 3.5 - At last, a realistic SQL interpreter

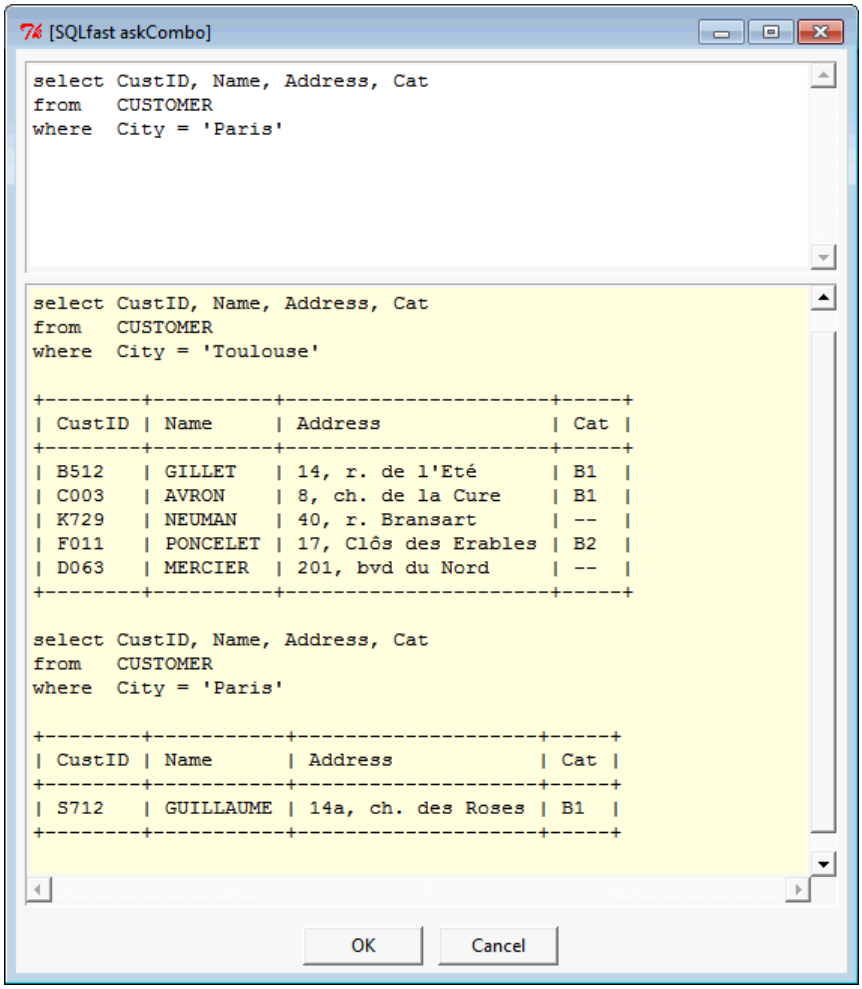

Figure 3.3 - State of the SQL interpreter of Script 3.5

This standard message is quite detailed, but it refers to the script we have executed and not to the operation the student wanted to carry out. Message "no such table: KUSTOMER" is fine, but the rest of the text will just confuse the student.

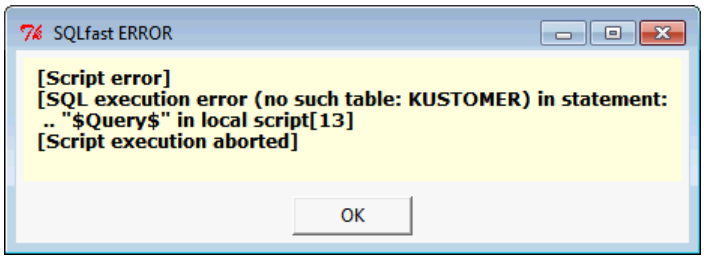

Figure 3.4 - Not exactly the kind of message a novice user would like to read!

First, we disable the standard error management, letting the script continue its execution in case of error:

```
onError continue;
```

Then, we can examine the error messages returned by the SQLfast engine through three system variables:

- **Error** is a simple indicator telling whether the last statement has been successfully executed (Error = 0) or not (Error = 1)
- **SQLdiag** provides a concise diagnostic on the execution of the last query: '**OK**' if everything went fine, '**NONE**' if no row was found, '**ID**' for a uniqueness error, etc.
- **EXTENDEDdiag** gives more detail on the error. Generally, it includes the nature of the query and the error message provided by the SQL engine.

We agree on two successful messages, namely '**OK**' and '**NONE**', all the other ones being considered as unsuccessful execution. Let us replace the simple execution statement

```
$Query$;
```

by this sequence

```
onError continue;
  $Query$;
  if ($Error$) write $SQLdiag$; $EXTENDEDdiag$;
onError stop;
```

Now the message is much sober:

```
SCHEMA; select: no such table: KUSTOMER
```

The contents of these two variables provide us with the necessary information to build a more *student-friendly* error message, as shown in Figure 3.5.

```
select *
from   KUSTOMER

*** SCHEMA error in "select" query:
***   no such table KUSTOMER.
```

Figure 3.5 - A simpler and more informative error message

To cope with the different errors that may occur in the execution of all SQL queries, it is best to create a specific procedure that identifies the error and that creates the appropriate error message. Now, the modification of the main script looks like this:

```
onError continue;
  $Query$;
  if ($Error$) execSQL SQL-ErrorProcessing.sql;
onError stop;
```

The code of script **ErrorProcessing.sql** could be that of Script 3.6.

```
compute verb = lower(item('§Query§',1,' '));
if ('§verb§' not in ('select','insert','update','delete'))
   set verb = unknown;
goto $verb$;
label select;
   compute mess = '*** $SQLdiag$ error in "$verb$"'
                  ||'" query:@n***  '
                  ||item('$EXTENDEDdiag$',2,':')
                  ||item('$EXTENDEDdiag$',3,':')||'.';
   goto Display;
label delete;
   ...
   goto Display;
...
label unknown
   set mess = *** Unknown operation "$verb$". Should be:
            @n***  "select", "insert", "update" or "delete".;
label Display;
   write $mess$;
```

Script 3.6 - The procedure processing SQL errors

3.5 Adding information fields

First time users could be puzzled by the *complexity* of the dialogue box! Let us add short informative messages to help them identify the different components of the dialogue box (Script 3.7).

```
...
askCombo [/bInteractive SQL interpreter.
            Enter an SQL query then click on OK.]
         [askText-u Query = [/w1/y8/bMy SQL query]]
        |[showText result = [/w0/y24/bThe result of my query]];
...
```

Script 3.7 - Adding information messages

3.6 Adding functions

To make the life of students easier and more comfortable, we will add some functions to the interface. First, copying the query in the result field will be made an option. Same idea for appending the result of the next query to the text already written in the result field. So, we create two buttons:

- **Show query**: if checked, the next queries will be copied in the result field before each of their result set.
- **Preserve history**: if checked, the query just executed and its result set (or diagnostic) will be *appended* to the contents of the result field (hence the term *history*). Otherwise, this material will *replace* the current contents.

It would be nice too to add a *short tutorial* to help first time students to start using the interpreter. We add a button (**Show help**) that, when checked, opens a help document the next time button OK is clicked on. If unchecked, this document does not appear.

A fourth function will allow students to ask for a *report* of their session, comprising the trace of their activities. This report is the contents of the result field and will be written in a text file on exit of the interpreter. When checked, button **Save history on exit** automatically activates function **Preserve history**.

Now, the dialogue box creation statement looks like Script 3.8. The complete script is shown in Script 3.9. It produces the interface of Figure 3.6.

Two comments on this script:

- Saving the history depends on the value of indicator **S** on exit. However, this value will be not be updated when the box is closed with button **Cancel**. There-

fore, button **Save history on exit** must be checked before some queries are submitted, which is fairly natural.[1]

– The save file is given a standard name like SQL-Save-2017-10-28_16-08-32.txt. This name is built from time registers **date** and **time**. Since the time format includes semi-colons (according the ISO format), the latter are converted into simple dashes.

```
askCombo [/bInteractive SQL ...]
 [selectMany-u S = Save history on exit]
|[askText-u Query = [/w1/y8/bMy SQL query]]
|[selectMany-u Q,P,H = Show query||Preserve history||Show help]
|[showText result = [/w0/y24/bThe result of my query]];
```

Script 3.8 - Four function check buttons are added

```
outputAppend result.var;
set Query,result,log = ,,;
set S,Q,P,H = 0,1,1,0;
while (True);
   askCombo ...;
      if ('$DIALOGbutton$' = 'Cancel') exit;
   if ($S$) set P = 1;
   function Query = LStr:PrettyPrint {$Query$};
   if (not $P$) set result = ;
   if ($Q$) write-ab $Query$;
   onError continue;
      $Query$;
      if ($Error$) execSQL _SQL-ErrorProcessing.sql;
   onError stop;
   if ($H$) displayHelp SQL-interpreter.tuto;
   if (not $H$) closeHelp;
endwhile;
if ($S$);
   compute tim = '$date$'||'_'
                 ||substr(replace('$time$',':','-'),1,8);
   outputOpen SQL-Save-$tim$.txt;
   write $result$;
endif;
```

Script 3.9 - Interactive SQL interpreter - Final version

1. In other words, checking this button just before leaving the interpreter is useless!

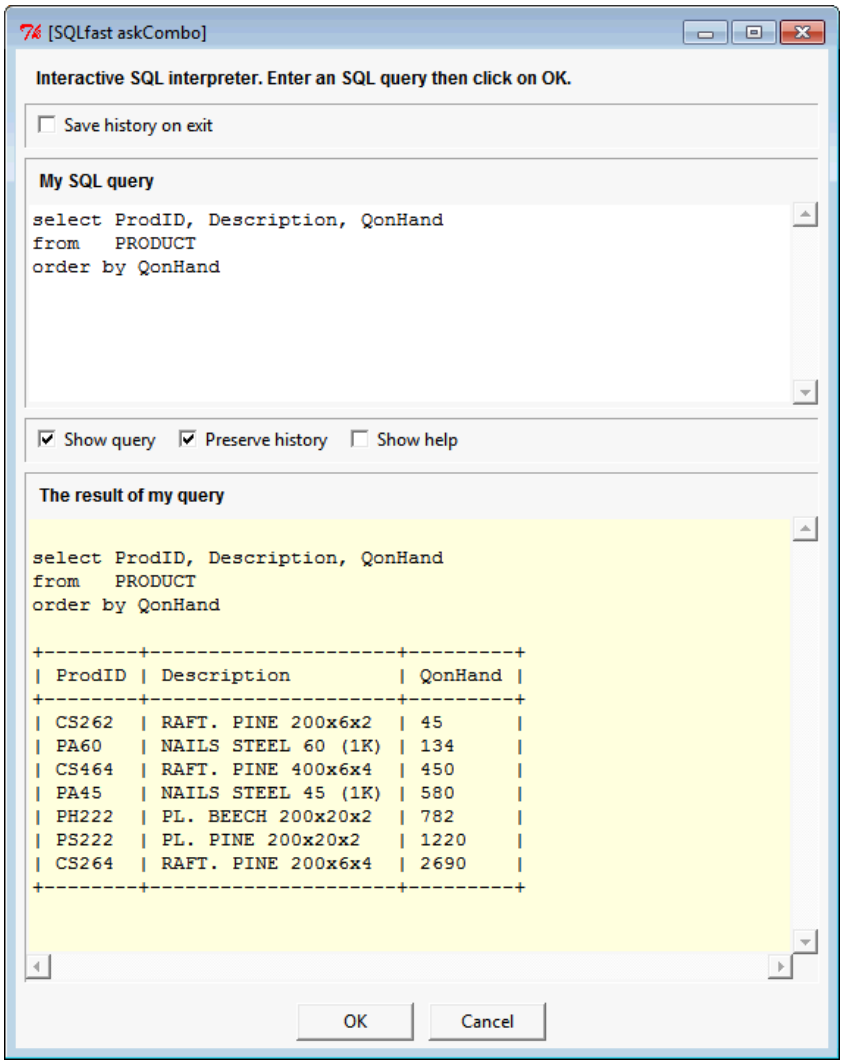

Figure 3.6 - Interface of the final version of the SQL interpreter

3.7 SQL interpreters as training support

One of the most common uses of SQL interpreters is to *support SQL learning*. The training scenario suggested by the SQL interpreter we have developed is fairly simple:

- the student reads the text of an exercise in some paper or electronic document (*pdf* or *web page* for instance), generally as a query expressed in plain language,
- the student enters the SQL query that answers it,
- the interpreter checks the syntactic validity of the SQL query,
- the interpreter executes the SQL query
- if the query is syntactically correct the interpreter displays its result set,
- otherwise it displays an error message.

If, as an answer to this exercise text:

what are the cities in which at least two customers live?

the student enters this **erroneous** SQL query:

```
select City from CUSTOMER where CustID >= '2'
```

the interpreter will find it syntactically correct and will display its result set, be it correct or not. The interpreter is happy with this answer, ... and so will be the student!

3.8 Semantic correctness of a query

The problem we put in light is the *semantic correctness* of the student's answer: does the SQL query **exactly translates** in the (formal) SQL language the *intension* of the (informal) query of the exercise?

A first solution that could come in mind would be to store in an *exercise database* both the text of each exercise and the expected SQL query that translates it. So, we could compare the query of the student with this reference query.

Unfortunately, this naive approach does not work (at all). This problem have proved very difficult to solve for two main reasons. First, natural languages are ambiguous. As every teacher has experimented, whatever the care with which one writes the text of an exercise, there will almost always be an student who will give it a plausible but unexpected interpretation. Secondly, even when we agree on the meaning of an exercise text, the richness and flexibility of the SQL language allows a wide variety of equivalent SQL queries to be written.

Let us show it through the following example, that seems quite simple and unambiguous:

which are the orders placed in 2017?

What does exactly mean expression "*which are the orders*"? Does it mean "*the value of column CustID of table CUSTORDER*"? or "*the values of all the columns of this table*"? Can we add the name, address and city of the customers who placed the orders? Can we add the total amount of each order? Can we sort the result set?

Expression "*placed in 2017*" is unambiguous but can be translated in many ways into SQL conditions:

```
DateOrd between '2017-01-01' and '2017-12-31';

DateOrd >= '2017-01-01' and DateOrd <= '2017-12-31';

DateOrd > '2016-12-31' and DateOrd < '2018-01-01';

DateOrd like '2017-%';

cast(DateOrd as char) like '2017-%';

substr(cast(DateOrd as char),1,4) = '2017';

year(DateOrd) = 2017;

extract(year from DateOrd) = 2017;
```

. . . *and some more*

The idea of storing a reference query that we can compare with the student's answer is absolutely and definitively unrealistic. So, let us drop it.

Instead of examining the query submitted by the student, we could execute it and **compare its result set** with that of the reference query. If they are identical, then the answer of the student is likely to be correct.

This is easy to check, but the conclusion of this comparison must be drawn more carefully:

- First, we should compare sets and not sequences, since row ordering, at least when not specified by the query, depends on the execution strategy chosen by the SQL engine.
- If the sets are different, we can conclude that the student's query is semantically incorrect; however, our diagnostic can be more precise to help the student modify her answer:
  - what is the size of each result set
  - how many rows does each result set contain that are not in the other one.
- If the sets are identical, the student's query may be correct, but this also may be a probabilistic accident! Indeed, two independent queries may produce the same result set though they have different meaning.
- The result sets may be the same while their sizes differ; this means that the student's result set includes duplicates (we suppose that the reference query discards duplicates).
- If both sets are empty, their comparison would be highly inconclusive.

In addition, showing the result set of their own answer and that of the reference query (of course without disclosing the latter) may help the students develop their answer or correct it.

3.9 An interactive SQL tutor

We consider the short analysis of the previous section as the set of minimal requirements for a variant of the SQL interpreter that we will call *Interactive SQL tutor*.

3.9.1 The database

For each exercise, we store its *full text* in plain language, its *reference solution* and the name of the *database* against which the query must be executed. Since the text may be rather long, a short summary, acting as a *title*, can be useful to allow the student to select the exercise in a list.

In addition, it is useful to classify exercises into *categories*. A category is a collection of exercises that share common properties, such as the structure of the solution (basic queries, sub-queries, join, grouped data, recursive queries, etc.) or the nature of the application domain (statistics, tree processing, text processing, active database, temporal database, etc.)

Finally, within each category, we could assign a *difficulty level* to each exercise.

This gives us the structure of table EXERCISE in which we will store the description of the exercises (Script 3.10).

```
create table EXERCISE(
       ExID        varchar(64) not null primary key,
       DB          varchar(128) not null,
       Category    varchar(32),
       Level       integer,
       Title       varchar(64),
       ProblemText varchar(512),
       RefQuery    varchar(512) );
```

Script 3.10 - Structure of the exercise database

Script 3.11 shows the definition of an exercise (medium difficulty: Level = 2) in the **sub-query** category (i.e., the solution of which is suggested to use a sub-query). It must be executed on database ORDERS.db. Its title, *Customers whose account is greater than the average of their city*, will be used by the student to select it.

3.9.2 The interface

Now, let us define the way students will use the SQL tutor. We call a *session*, the activity during which a student is invited to solve a set of exercises. We suggest that students proceed in two steps:

1. selecting the subset of exercises of the session; this subset is defined by one or several levels within a category (Figure 3.7),
2. trying to solve each (or some) of them (Figure 3.8).

```
insert into EXERCISE values(
   'Q02-07',
   'ORDERS.db',
   'sub-query',2
   'Customers whose account is greater than the average
    of their city',
   'Display the list of the Id of the customers whose account
    is greater than the average account of the customers
    of their city.',
   'select CustID,Name,City,Account from CUSTOMER as CUST
    where  Account > (select avg(Account) from CUSTOMER
    where City = CUST.City)'
);
```

Script 3.11 - Definition of an exercise

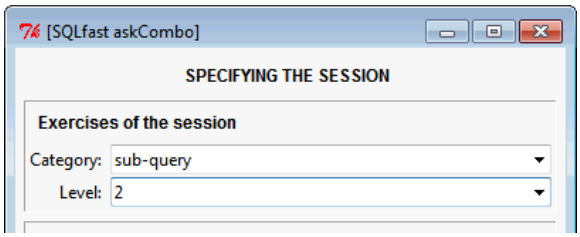

Figure 3.7 - Selecting the exercises of the session

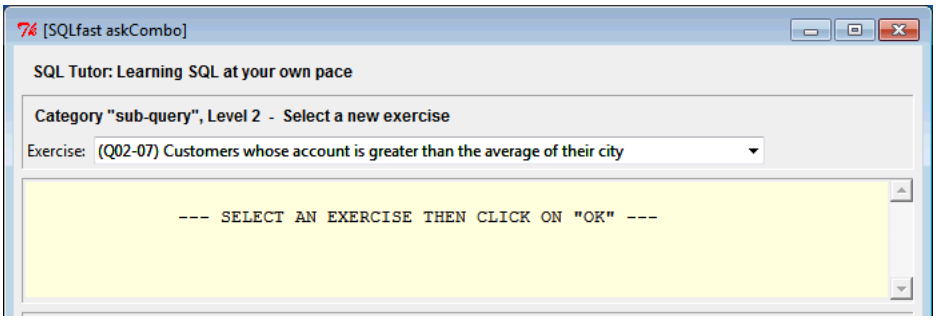

Figure 3.8 - Selecting an exercise to solve

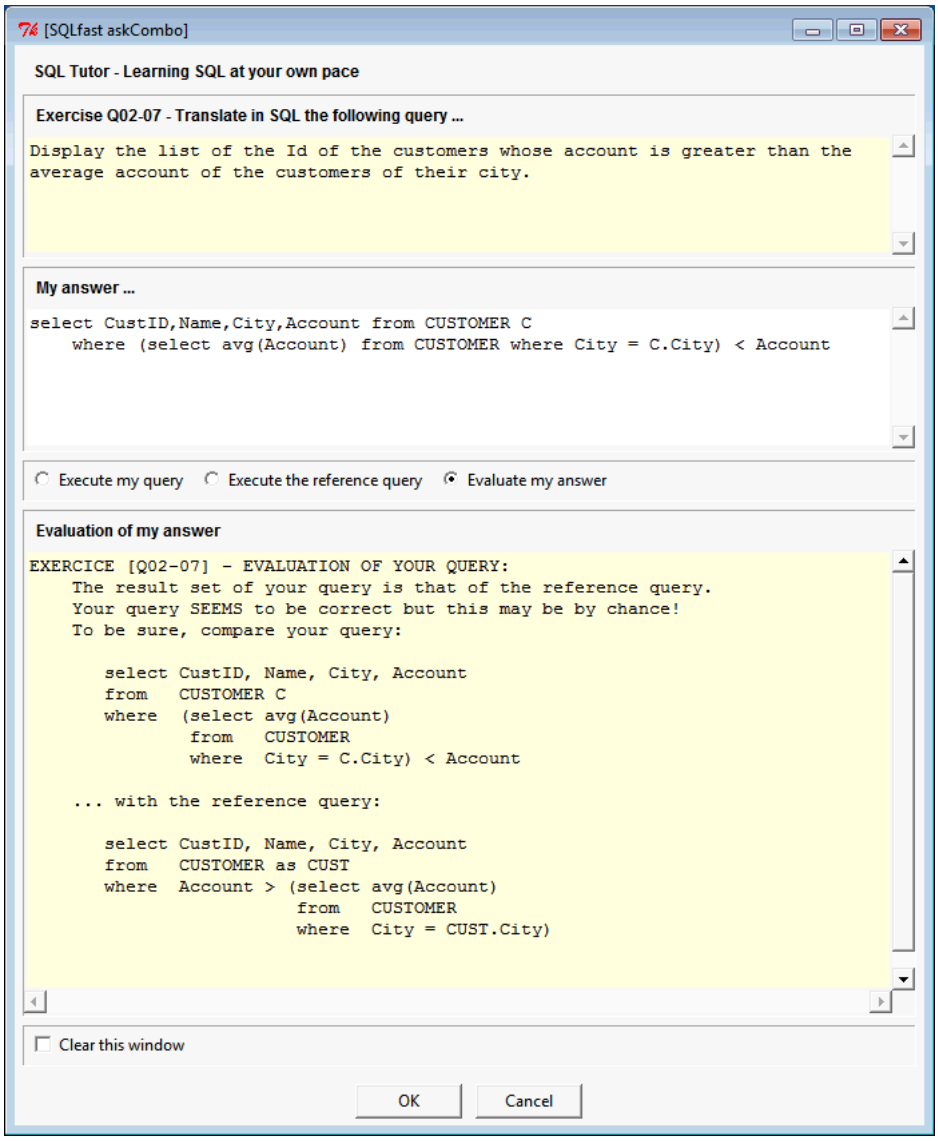

Figure 3.9 - The main dialogue box of the SQL tutor

The window of the SQL interpreter shown in Figure 3.6 is no longer sufficient to let the student practice the exercises of the session. We suggest the new layout of Figure 3.9. The first text area displays the text of the selected exercise. The student enters the SQL query that translates this text in the second area, check button **Evaluate my answer** then clicks on button **OK** to submit it. The tutor evaluates this SQL query and displays its diagnostic in the third text area.

The interface also allows the student to execute the query she just entered (button **Execute my query**) or to execute the reference solution (button **Execute the reference query**).

3.9.3 The tutor engine

Now, we examine the way the tutor will evaluate the SQL query submitted by the student. The analysis developed in Section 3.8 is based on a set of metrics defined as the size of the result sets of combinations of the student's query and the reference query.

Let **rQuery** be the reference query and **sQuery** the query submitted by the student (actually *the names of the variables that contain these queries*). The first one is extracted from the database and the second one from the second text area of the interface. The metrics are computed in Script 3.12:

- **N1**, the size of the result set of **rQuery**
- **N2**, the size of the result set of **sQuery**
- **N12**, the size of the difference of the result sets of **rQuery** and **sQuery**
- **N21**, the size of the difference of the result sets of **sQuery** and **rQuery**

We notice that metrics **N12** and **N21** are not computed on the source queries but on views defined on them. This trick is necessary to avoid set operator confusion when the source queries are themselves based on such operators.

We also note that metrics **N1** and **N2** are defined on multisets (in which duplicates, if any, are preserved) while metrics **N12** and **N21** are defined on pure sets (`union`, `intersect` and `except` eliminate duplicates).

```
extract N1 = select count(1) from ($rQuery$);
extract N2 = select count(1) from ($sQuery$);

create temp view _sview as select * from ($sQuery$);
create temp view _rview as select * from ($rQuery$);

extract N12 = select count(1) from (select * from _rview
                                      except
                                    select * from _sview);

extract N21 = select count(1) from (select * from _sview
                                      except
                                    select * from _rview);
```

Script 3.12 - Computing the raw metrics

The evaluation of **sQuery** derives from the reasoning rules of Script 3.13. They identify seven patterns:

1. *The reference result set is empty*: the diagnostic may be quite imprecise.

2. Considered as pure sets, the result sets are identical: the student's query *may be* considered correct.
3. In addition, considered as multisets, *the result sets are not identical*: the student's query may be considered correct but its result set includes duplicates.
4. The *reference result set contains rows that are not in the student's result set*.
5. The *student's result set contains rows that are not in the reference result set*.
6. The *reference result set is not empty but none of its rows appears in the student's result set*.
7. The r*eference result set includes some of the rows of the student's result set but not all*.

```
if (N1 = 0):
   < reference result set empty; imprecise diagnostic >
else:
   if (N12 + N21 = 0):
      < No row missing; answer probably correct >
      if (N1 = N2):
         < Same size >
      else:
         < Some duplicates >
   else:
      < Row(s) missing in some result set(s). Answer incorrect >
      if (N12 > 0 and N21 = 0):
      < Rows missing in student result set >
      if (N12 = 0 and N21 > 0):
         < Erroneous row(s) in student result set >
      if (N12 > 0 and N21 > 0):
         if (N1 = N12):
            < All the rows of student result set are erroneous >
         if (N1 > N12):
            < Some rows of student result set are erroneous >
```

Script 3.13 - Algorithm of the diagnostic

The state of the interface shown in Figure 3.9 illustrates the case of a student's query that is formally different from the reference query but that produces the same result set without duplicates (pattern n° 2). The diagnostic is appreciative but nevertheless invites the student to compare her query with the reference query. This shows that the tutor is more apt at detecting erroneous answers than identifying correct queries, an approach that is not uncommon among (human) teachers!

3.9.4 Improving the SQL tutor

The SQL tutor we have developed is just a prototype that can be enriched and improved in many ways. Let us mention some of them.

1. **Reducing false positives**

   The main weakness of the SQL tutor is the correct interpretation of patterns n° 1, 2 and 3, when result sets are identical, be they empty of not. The student's query *may* be correct but this can just be a statistical accident, that is, a *false positive*! We can reduce the uncertainty of such diagnostic by executing the queries against **several databases** with the same schema but with different contents. The value of column DB in table EXERCISE will then be a list of database names. The metrics collected for each database are consolidated to build a more reliable diagnostic.

2. **Refining the learning process**

   The training scenario enacted by the SQL tutor is fairly primitive. It would be more flexible, and therefore more efficient, if the tutor could help the student who fails to find the right answer or who explicitly asks for help. Some simple help features can be added: a *short reminder* of the underlying theory or of the SQL syntax, *hints* through which one or two levels of suggestion can be displayed when needed and *comments* associated to the reference solution.

3. **Encrypting reference queries**

   Honest students are those who try to solve the problems by themselves without resorting to dishonest sources. A particular dishonest source would be that provided by this script:

```
openDB SQLtutor.db;
select RefQuery from EXERCISE
where  ExID = 'Q02-07';
```

   The most obvious countermeasure to such work-around is to *encrypt* the values of column RefQuery, as shown below:

```
insert into EXERCISE values(
   'Q02-07',
     ...,
     encrypt('select CustID,Name,City,Account from CUSTOMER
              as CUST where Account > (select avg(Account)
              from CUSTOMER where City = CUST.City',
              'my_password'))
);
```

   The inverse operation will restore the original value:

```
select decrypt(RefQuery,'my_password') from EXERCISE
where  ExID = 'Q02-07';
```

   The database is now more secure, but not by far! A clever student could be tempted to have a look at the *source code* of the scripts, in which she will quickly discover the value of the decrypting key.

   A more secure technique can be developed as a couple of functions stored in Python module UserUDFlib_for_SQL.py or UserUDFlib_for_SQLfast.sql (the source code of which being, of course, unavailable!). The first function appears

to be an innocuous function, for instance that returns the current time (e.g., get_time()). It is called by the main script at start time. In addition to its apparent action, this function stores a kind of *cookie* in the UDF module. The second function (e.g., get_key()) is called to get the decrypting key. It returns the actual key if the cookie exists or a fake key otherwise. Understanding this mechanism and reproducing it to get the reference queries is much more complicated, at least for novice hackers. Considering that this application is not critical, this technique can be quite sufficient to ensure a fairly good level of security.

In the prototype application available in the SQLfast distribution, the reference queries can be, if desired, converted in the database in the *base64* coding scheme:

```
update EXERCISE set RefQuery = b64encode(RefQuery,'std');
```

4. Student management

The SQL tutor focuses on the availability of collections of exercises as well as the evaluation of student's answers. In the scenarios provided by this application, the student is just an external, unidentified user. This extension suggests to integrate students in the tutoring process. This encompasses two aspects:

- first allowing students to suspend, resume and record their sessions
- secondly, allowing teachers to register students, to examine their scores and to evaluate them.

5. Exercise management

A last service application will be welcome: a data entry interface to allow teachers to examine, enter, check, modify and delete exercises.

3.10 The scripts

The algorithms and programs developed in this study are available as SQLfast scripts in directory SQLfast/Scripts/Case-Studies/Case_SQL_Interpreters. Actually, they can be run from main script **SQL-Interpreter-MAIN.sql**, that displays the selection box of Figure 3.10.

The scripts of the SQL tutor have been stored in subdirectory **SQL-Tutor**. The student component (the only one so far) of the SQL tutor is run by executing script **SQL-Tutor-STUDENT.sql**. This script executes an exercise module script that loads in the database the set of exercises the student is interested. in. This script is given a name starting with prefix **'SQL-Exercises-'**. Several exercise scripts, both in English and in French, have been included in the SQLfast distribution.

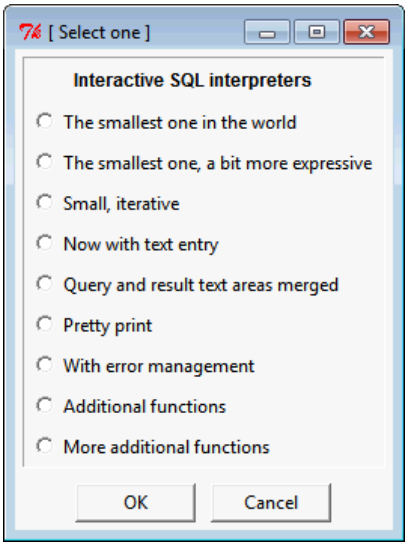

Figure 3.10 - Selecting an SQL interpreter

The scripts of this case study are provided without warranty of any kind. Their sole objectives are to concretely illustrate the concepts of the case study and to help the readers master these concepts, notably in order to develop their own applications.