AnyDAC macros and scripting

Michaël Van Canneyt

April 9, 2012

Abstract

In a previous article, AnyDAC was presented. One of the topics was the support for macros. In this article, this broad topic is further investigated: it has many applications, in particular, in SQL scripts. AnyDAC offers an SQL script component which can also handle all the macros that the engine supports.

1 Introduction

AnyDAC is a rich set of Delphi components to access a variety of databases. It was introduced in a previous article, where the architecture and basic workings of AnyDAC were explained.

In this article, the focus lies on one of the features of AnyDAC: preprocessing of SQL statements and the use of macros and expressions in the SQL statements that are executed using AnyDAC. In the introduction article, it was shown how macros can be used to change for instance the ordering of a result set by changing the `ORDER BY` clause of a SQL `SELECT` statement. However, much more can be done with the SQL preprocessor of AnyDAC.

AnyDAC focuses on RDBMSes using an SQL interface to the database: all data manipulation is done using SQL statements. Much as SQL is standardized, there are still many differences between SQL engines, some of them quite fundamental. AnyDAC helps removing the differences between the engines by offering a way to preprocess the SQL statements before they are sent to the RDBMS: this allows an AnyDAC application to communicate with any supported database without the need for recompilation.

The applications of the various preprocessing mechanisms in AnyDAC will be demonstrated using the pupil tracker application, presented in the previous article. It will be expanded so it can also work with MySQL. For this, a `TADPhysMySQLDriverLink` component must be dropped on the main form, so the appropriate driver code is linked in. Additionally, a menu item is created that can be used to select the kind of connection before the connection is made.

2 Conditional substitution

One fundamental difference between SQL engines is how they deal with auto-incremental fields: a field whose value is augmented automatically each time a new record is inserted in the table. Both tables in the tracker database contain a primary key which is such an auto-incremental integer field.

Database Engines like MySQL and MS-SQL have a special field-type for this, and a function in the API to retrieve the value of the latest entry. Other engines (Oracle, Firebird, PostGreSQL) do not provide a special field; instead, one uses a normal integer field, and the engine provide sequences (sometimes called identities or generators) to generate new unique values for the field. This value must be inserted in the table using the regular SQL syntax.

The practical upshot of this difference is that for MySQL, an insert in the PUPILS table looks as follows:

```
INSERT INTO PUPILS
  (PU_FIRSTNAME,PU_LASTNAME)
VALUES
 (:PU_FIRSTNAME, :PU_LASTNAME)
```

The value for `PU_ID` will be generated automatically, and can be retrieved with a special function (`mysql_insert_id`).

For Firebird, assuming there is a generator `GEN_PUPILS`, the insert would look as follows:

```
INSERT INTO PUPIL
  (PU_ID, PU_FIRSTNAME,PU_LASTNAME)
VALUES
 (GEN_ID(GEN_PUPILS,1),:PU_FIRSTNAME, :PU_LASTNAME)
```

or, using the newer syntax:

```
INSERT INTO PUPIL
  (PU_ID, PU_FIRSTNAME,PU_LASTNAME)
VALUES
  (NEXT VALUE FOR GEN_PUPILS, :PU_FIRSTNAME, :PU_LASTNAME)
```

Now, if all INSERT statements in an application must be duplicated to be able to work with 2 different database engines, this would be very cumbersome and difficult to maintain. AnyDAC offers a better way to handle this: conditional substitution.

The general syntax is as follows:

```
{if X} Y {fi}
```

Where X can be the name of a macro, or the name of the database engine on which the SQL script is run. The list of possible engine names is presented in the ANYDAC documentation. In this form, `Y` will be included in the SQL statement if the macro X is defined, or if it equals the name of the current SQL engine.

Applied to the above insert statement, this means the insert statements can be written as follows:

```
INSERT INTO PUPIL
  ({if INTRBASE} PU_ID,{fi}
   PU_FIRSTNAME,PU_LASTNAME)
VALUES
  ({if INTRBASE} GEN_ID(GEN_PUPILS,1),{fi}
   :PU_FIRSTNAME, :PU_LASTNAME)
```

`INTRBASE` is the name of the Interbase/Firebird engine in AnyDAC. A second form of conditional subsitution uses `IIF`:

```
{iif X1, Y1, X2, Y2, .. XN,YN [, YE] }
```

If X1 is defined, then Y1 will be substituted. If X1 is not defined, but X2 is, then Y2 will be substituted. If none of the Xn values is defined, then YE will be substituted if it is supplied.

This can for instance be used to mask the difference in syntax for generating a new value between e.g. firebird and PostGreSQL:

```
INSERT INTO PUPILS
  (PU_ID,PU_FIRSTNAME,PU_LASTNAME)
VALUES
  ({iif INTRBASE, NEXT VALUE FOR GEN_PUPILS,
        PG, nextval(GEN_PUPIL) },
  :PU_FIRSTNAME, :PU_LASTNAME)
```

or to mask the difference in some function names. For instance the length of a string. The following statement

```
SELECT * FROM PUPILS
WHERE
  {IIF INTRBASE, OCTET_LENGTH, MYSQL, LENGTH}(PU_FIRSTNAME)<3
```

Will retrieve all pupils in the database whose first name has a byte length of less than 3.

3 Escape sequences

For the last case, where RDBMS functions that perform the same task have different names, it is not necessary to use conditional substitution: AnyDAC offers its own mechanism, escape sequences. Basically, this means that the SQL statement contains the function name in a special form, which AnyDAC will transform to the name that the RDBMS engine expects, before sending the SQL to the RDBMS engine. This is done using the following form

```
{fn FUNCTIONNAME(Args)}
```

The `FUNCTIONNAME` will be transformed to the name that the RDBMS expects. In some cases the `fn` prefix can be left out, and the escape sequence becomes:

```
{FUNCTIONNAME(Args)}
```

For instance, uppercase of a string value is done with UCASE:

```
SELECT {fn UCASE(PU_FIRSTNAME)} FROM PUPIL
```

Which would translate to the following:

```
SELECT UPPER(PU_FIRSTNAME) FROM PUPIL
```

for Firebird.

AnyDAC can also do this for some operations such as string concatenation: different engines use different ways to concatenate strings: MySQL will use a plus sign (+):

```
select PU_FIRSTNAME + ' ' + PU_LASTNAME FROM PUPIL
```

whereas Firebird will use 2 pipe characters (`||`):

```
select PU_FIRSTNAME || ' ' || PU_LASTNAME FROM PUPIL
```

In AnyDAC, this can be expressed as

```
select
  {fn CONCAT(PU_FIRSTNAME, {fn CONCAT(' ',PU_LASTNAME)})}
FROM
  PUPIL
```

The above example shows that it is also possible to nest the functions.

There are a lot of functions which can be used in this way: string manipulations, math functions, type conversion functions. All available functions are listed in the AnyDAC documentation.

To demonstrate the use of escape sequences, a function that searches a pupil in the database is implemented. The search is performed on the pupils first and last name. The function should accept a name to search for, and 2 options:

- to search case sensitively or not
- to search for an exact match or only a part of the name.

To this end, an edit control (`ESearch`) and 2 comboboxes (`CBExact` and `CBCaseSensitive`) are dropped on a form. A button and a DBGrid complete the visual part of form. The query is executed with a `TADQuery` component (`QSearch`), whose `SQL` property is set to the following:

```
SELECT
   PU_ID, PU_FIRSTNAME, PU_LASTNAME
FROM
  PUPIL
WHERE
{if &EXACT}
{if &NOTCASESENSITIVE}
  ({fn UCASE(PU_LASTNAME)} = :LASTNAME) OR
  ({fn UCASE(PU_FIRSTNAME)} = :FIRSTNAME)
{fi}
{if &CASESENSITIVE}
  (PU_LASTNAME = :LASTNAME) OR
  (PU_FIRSTNAME = :FIRSTNAME)
{fi}
{fi}
{if &NOTEXACT}
{if &NOTCASESENSITIVE}
  ({fn UCASE(PU_LASTNAME)} LIKE
    {fn CONCAT('%',{fn CONCAT(:LASTNAME,'%')})}) OR
  ({fn UCASE(PU_FIRSTNAME)} LIKE
    {fn CONCAT('%',{fn CONCAT(:FIRSTNAME,'%')})})
{fi}
{if &CASESENSITIVE}
  (PU_LASTNAME LIKE
    {fn CONCAT('%',{fn CONCAT(:LASTNAME,'%')})}) OR
  (PU_FIRSTNAME LIKE
    {fn CONCAT('%',{fn CONCAT(:FIRSTNAME,'%')})})
{fi}
{fi}
```

AnyDac does not support an else ELSE clause for the escape sequence, so this is emulated using 2 macros, which will have the opposite effect: EXACT and NOTEXACT, and CASESENSITIVE and NOTCASESENSITIVE. The CONCAT function is used to add wildcard characters to the SQL LIKE clause, and the UCASE function is used to search in a case insensitive manner.

Now, by providing a value for the desired macros, the SQL statement can be altered so it performs the desired search. This is done when the SEARCH button is clicked:

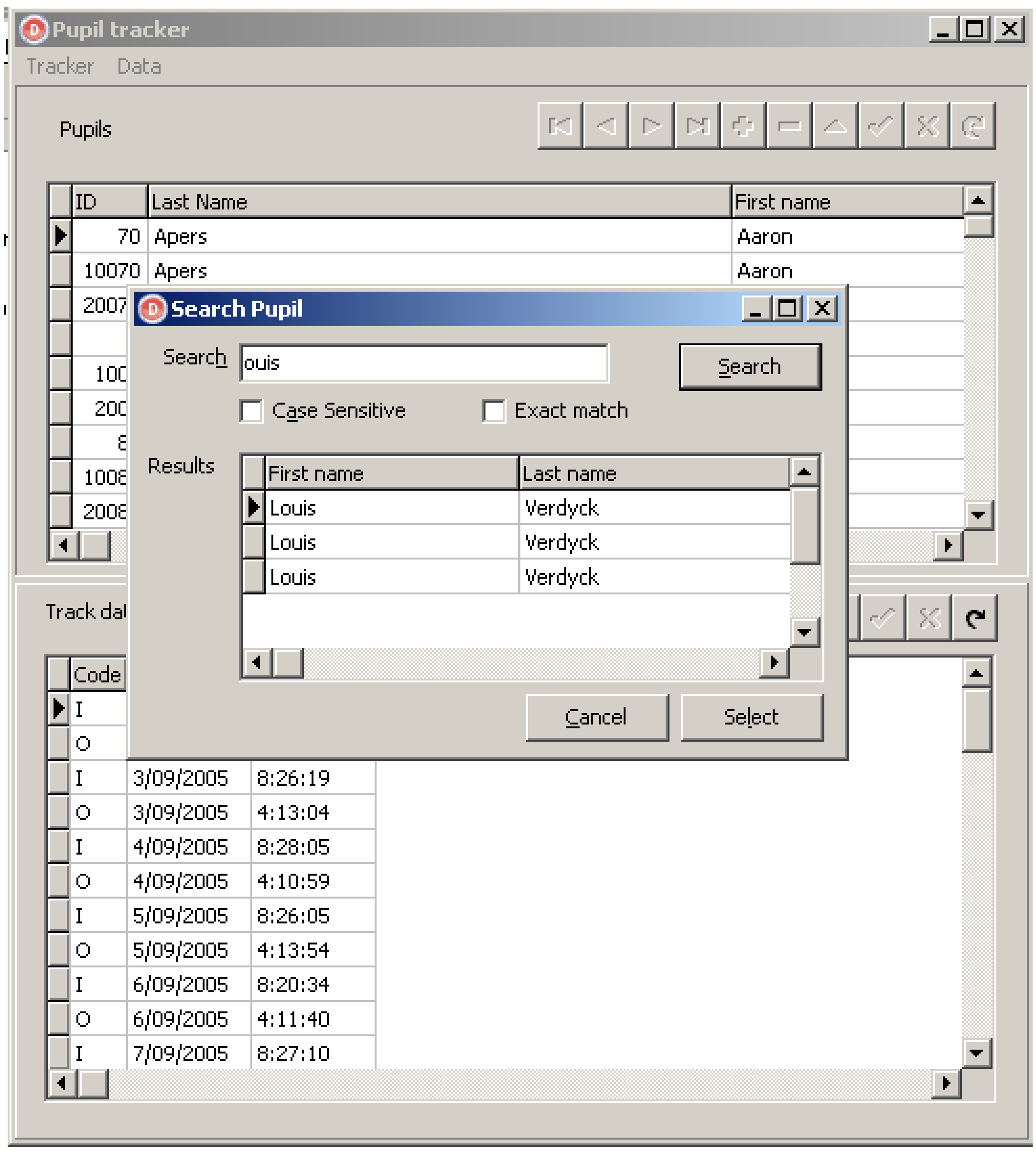
```
procedure TSearchForm.BSearchClick(Sender: TObject);

  Procedure SetM(N : String; B : Boolean);

  begin
    If B then
      QSearch.MacroByName(N).Value:='OK'
    else
      QSearch.MacroByName('NOT'+N).Value:='OK';
  end;

var
  n : String;
  I : integer;
  M : TADMacro;

begin
  With QSearch do
    begin
    For I:=0 to Macros.Count-1 do
      Macros[i].Clear;
    SetM('EXACT',CBExact.Checked);
    SetM('CASESENSITIVE',CBCaseSensitive.Checked);
    Prepare;
    N:=ESearch.Text;
    if Not CBCaseSensitive.Checked then
      N:=UpperCase(N);
    Params.ParamByName('FIRSTNAME').AsString:=N;
    Params.ParamByName('LASTNAME').AsString:=N;
    Open;
    end;
end;
```

The `setM` procedure will set a macro N, or NOTN, based on the boolean `B`. This function is then used to provide a value for the desired macros, based on the values of the checkboxes. After that is done, the parameters are set and the query is opened.

This is the main function in the search form. Some auxiliary functions are needed to pass a connection to the form, and some functionality to be able to select a row and return the ID of the selected pupil. The interested reader can consult the sources of the project.

When all is finished, the result looks as in figure 1 on page 6.

Figure 1: Searching a pupil



Search: ouis

[ ] Case Sensitive [ ] Exact match

ID	Last Name	First name
70	Apers	Aaron
10070	Apers	Aaron
2007		
100		
200		
8		
1008		
2008		

First name	Last name
Louis	Verdyck
Louis	Verdyck
Louis	Verdyck

Code		
I		
O		
I	3/09/2005	8:26:19
O	3/09/2005	4:13:04
I	4/09/2005	8:28:05
O	4/09/2005	4:10:59
I	5/09/2005	8:26:05
O	5/09/2005	4:13:54
I	6/09/2005	8:20:34
O	6/09/2005	4:11:40
I	7/09/2005	8:27:10

4 Constant embedding

As with function names, SQL engines often have different ways of representing constants. Specially date and time values can have different forms. AnyDAC offers special escape sequences to insert constants in an SQL statement. Generally it looks like this:

```
{T value}
```

Here T is the type of the constant. It can be one of

e a float value. The float value must be written as it would be in Delphi, and it will be inserted in the form the RDBMS expects.

d a date value. The value must be written as YYYY-MM-DD, no quotes are needed.

t a time value. The value must be written as hh:mm:ss, where hour is in a 24-hour format.

dt a date-time value, date and time are specified as in the `d` and `t` cases.

l A boolean, which should be 'true' or false, and will be converted to what the RDBMS expects (usually 0 and 1)

s A string. The string will be enclosed with the correct quote char for the RDBMS (usually a single quote).

id An identifier. This is useful for specifying identifier names which are reserved words or which contain spaces. The identifier will be enclosed in the appropriate quoting characters for the RDBMS the form

```
{id ORDER DETAILS}
```

Will be translated to `[Order Details]` for Microsoft SQL server or MS Access, but to `"Order details"` (double quotes) for Firebird, and `` `Order Details` `` (backtick) for MySQL.

5 Scripting and macros

Macros and sequences are also very useful in the script component that AnyDAC supplies.

When working with SQL databases, there comes an inevitable moment when a script must be run on the database. One of these moments is almost certainly the creation of the database tables, other moments can be the update of a database to a new version, or the creation of sample data.

AnyDAC offers a `TADScript` component that allows execution of SQL scripts: a series of SQL commands in a stringlist, or in a file. The `TADScript` component component will parse the SQL script, and will execute the SQL commands one by one - which is the basic functionality one would expect from such a component.

However, it does more than just that. There are some extra functionalities:

1. Macros and escape sequences are handled just as they would be in a `TADQuery` component.

2. There is the possibility to create a log file (called the spool file), with various options. For example, the result of SELECT commands will be output to the log file.

3. Multiple scripts can be specified.
4. Custom commands can be embedded in the script. AnyDAC provides several custom commands out of the box.
5. Validate the script.
6. Step through the script.
7. Display the executed commands on screen.
8. Pass parameters to the script.

There are basically 2 ways to use this component:

1. Call one of the `ExecuteNNN` methods with the name of a script file, or pass it a string list with the commands.
2. Fill the SQLScripts property with a series of scripts and call `ExecuteAll`.

The following execute methods are present:

ExecuteStep Extracts and executes the next command from the `SQLScripts` property.

ExecuteAll Executes all commands in the first script in the `SQLScripts` property. An optional list of arguments can be specified in the `Arguments` property of the SQLScript component.

ExecuteFile This command reads the specified filename and executes it with the optional specified arguments.

ExecuteScript This command can be passed a SQL script as a `TStrings` instance, and executes it with the optional specified arguments.

The last 2 forms will simply

- Clear the SQLScripts property.
- Add the script passed to it (reading it from file in the case of `ExecuteFile`).
- Set the `Arguments` property, if arguments were specified.
- Call `ExecuteAll`.

So it is important to remember that the SQLScripts properties content is destroyed after these calls.

The `SQLScripts` property is a collection of named SQL scripts. Only the first script is executed when `ExecuteAll` is called. To have the statements in the other scripts executed, they must be included using one of the custom AnyDAC commands for the script component. To execute a second script (called 'firebird'), it can be included as follows:

```
@firebird
```

Alternatively, the `INPUT` or `START` commands can be used:

```
INPUT firebird
```

For the tracker application, this can be used to create parts of the database depending on which engine is used. In MySQL, an `AUTO_INCREMENT` field can be used as primary field. For Firebird, an auto-increment field is usually realised using a generator and a trigger, as in the following example:

```
CREATE GENERATOR GEN_PUPIL;

SET TERM ^ ;

CREATE TRIGGER PUPIL_INSERT FOR PUPIL
ACTIVE BEFORE INSERT POSITION 0
AS
begin
IF (NEW.PU_ID IS NULL) then
  NEW.PU_ID = GEN_ID(GEN_PUPIL,1);
end ^
```

For the tracker application, this means that the database create script can be split up in 2 parts: one common part, creating the tables and indexes, and one part, creating the generators and triggers for Firebird.

The following code can be executed when connecting to the database. It checks whether the `PUPIL` table exists. If it does not exit, then the database create script is executed. If the connection is a Firebird database, then a line is added to the main script, which includes the firebird-specific script:

```
procedure TForm1.CheckTables;

begin
  B:=False;
  With TADQuery.Create(Self) do
    try
      Connection:=CTracker;
      try
        Open('select * from PUPIL where (0=1)');
        // if we are here, the table exists, and we can exit.
        exit;
      except
        on E: EADDBEngineException do
          if E.Kind <> ekObjNotExists then
            Raise;
      end;
    finally
      Free;
    end;
  if (CTracker.ConnectionDefName='TrackerFB') then
    ADSCreate.SQLScripts[0].SQL.Add('INput firebird;');
  ADSCreate.ExecuteAll;
end;
```

In case of a database update script, the feature could be used to chain together various updates. The first (main) script can be left empty, and the various scripts needed to update from version to version can be put in scripts named `versionN`. Updating the database is then simple:

```
Procedure TForm1.UpdateDatabase(CurrentVersion : integer);

begin
  With SCUpdate.SQLScripts[0] do
    begin
    if (CurrentVersion<2) then
      Add('input version2');
    if (CurrentVersion<3) then
      Add('input version3');
    end;
  SCUpdate.ExecuteAll;
end;
```

As mentioned earlier, escape sequences and macros are handled in the SQL script engine. This means that the following will work:

```
CREATE TABLE PUPIL (
  PU_ID INTEGER NOT NULL {if mysql}AUTO_INCREMENT{fi},
  PU_FIRSTNAME VARCHAR(50) NOT NULL,
  PU_LASTNAME VARCHAR(50) NOT NULL,
  CONSTRAINT PUPIL_PK PRIMARY KEY (PU_ID)
);
```

The MySQL version of this table definition will include the `AUTO_INCREMENT` option, whereas in firebird it will not be included.

It is important to note that the escaping and macro handling is done only when the script engine passes on the command to the AnyDAC connection for execution: The SQL script itself does not do the escaping. This means that for instance the following

```
{if INTRBASE}
INPUT firebird
{fi}
```

Will not work, because the whole construct will be passed on to the firebird SQL engine, which will not understand the `INPUT` command, is a custom command which must be handled by the `TADSQLScript` component.

The following is the full list of the default commands implemented by AnyDAC:

INPUT Input another script. Alternative names are @ and START

ACCEPT Ask for user input.

CONNECT Connect to another AnyDac connection.

COPY Copy data from or to a file.

DEFINE Define a macro

DELIMITER Set the SQL script command delimiter.

DISCONNECT Disconnect from the connection.

EXECUTE Execute a SQL block.

EXIT Stops script execution and commits changes. The QUIT form does a rollback, the STOP form does nothing.

HELP Lists all commands.

HOST Execute an external program.

PAUSE Pauses script execution and waits for user input.

PROMPT Writes text to the output log.

PRINT Print the names and values of variables.

REMARK include a remark.

SET Set various TADScript component properties from the script.

SPOOL Send output to spoolfile.

UNDEFINE Undefine a macro.

VARIABLE Define a variable.

CREATE DATABASE Create a database. Only for INTRBASE engine.

DROP DATABASE Drop a database. Only for INTRBASE engine.

They are executed by the script engine, and are not sent to the database engine.

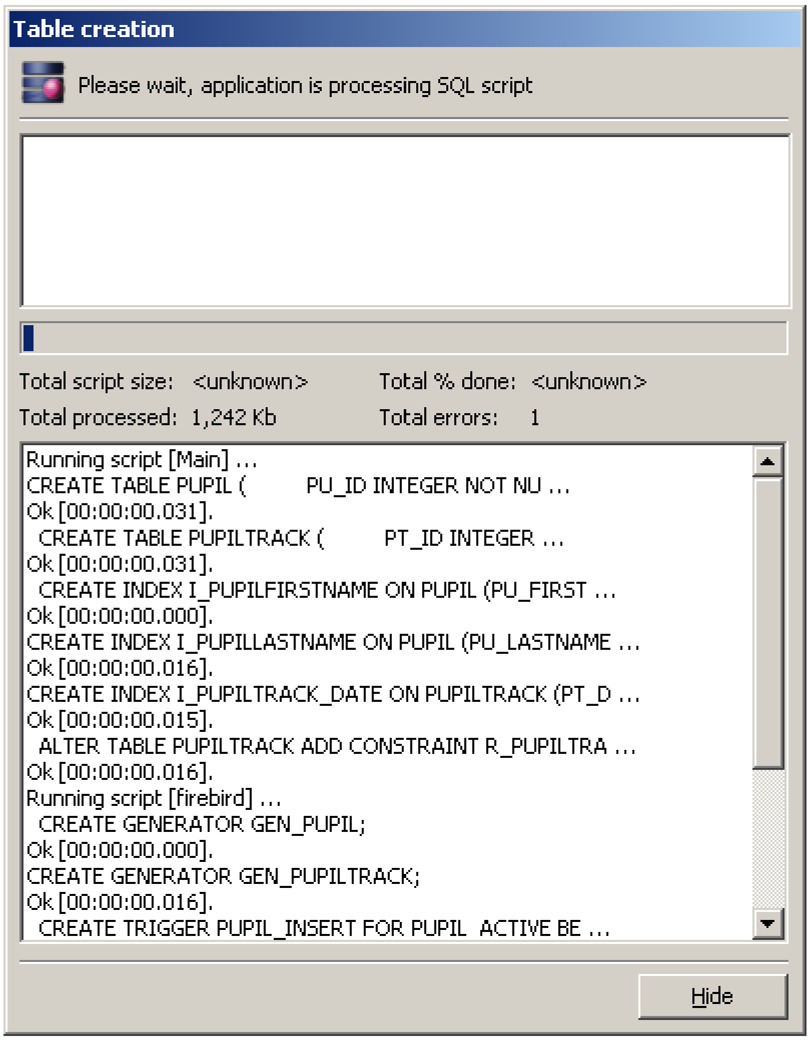
The SQL Script component can log all statements to a dialog (it can also use this dialog to get user input). To do this, a `TADGUIxFormsScriptDialog` component can be dropped on the form, and assigned to the `ScriptDialog` property of the `TADSQLScript` component.

When this is done, and the application connects to a new Firebird database, the result looks as in figure 2 on page 12.

6 Conclusion

In this article, the preprocessor of AnyDAC has been demonstrated. It is implemented on a low level in the AnyDAC structure, so it can be used on several places - not in the least in the SQL script component, which is a versatile and extensible scripting component that should cover most needs when working with SQL databases.

Figure 2: Database creation log



Please wait, application is processing SQL script

Total script size: <unknown> Total % done: <unknown>
Total processed: 1,242 Kb Total errors: 1

```
Running script [Main] ...
CREATE TABLE PUPIL (        PU_ID INTEGER NOT NU ...
Ok [00:00:00.031].
  CREATE TABLE PUPILTRACK (        PT_ID INTEGER ...
Ok [00:00:00.031].
  CREATE INDEX I_PUPILFIRSTNAME ON PUPIL (PU_FIRST ...
Ok [00:00:00.000].
CREATE INDEX I_PUPILLASTNAME ON PUPIL (PU_LASTNAME ...
Ok [00:00:00.016].
CREATE INDEX I_PUPILTRACK_DATE ON PUPILTRACK (PT_D ...
Ok [00:00:00.015].
  ALTER TABLE PUPILTRACK ADD CONSTRAINT R_PUPILTRA ...
Ok [00:00:00.016].
Running script [firebird] ...
  CREATE GENERATOR GEN_PUPIL;
Ok [00:00:00.000].
CREATE GENERATOR GEN_PUPILTRACK;
Ok [00:00:00.016].
  CREATE TRIGGER PUPIL_INSERT FOR PUPIL  ACTIVE BE ...
```