Embedded databases: SQLite

Michaël Van Canneyt

December 5, 2005

Abstract

In this second article about embedded databases, the SQLite database is investigated. This database is one of the easiest to use, and requires very little or no configuration. It's also supported out-of-the box by Lazarus and Delphi.

1 Introduction

SQLite is a C library which implements an in-process SQL engine. It contains most features that one would expect from an SQL database server, with the possible exception of client-server features, but this is not needed for an embedded database. SQLite is public domain code, meaning that anyone is free to use it as he sees fit, no restrictions are placed on usage.

SQLite can be downloaded from

`http://www.sqlite.org/`

There the sources and some command-line utilities can be downloaded. Both the latest version (3.2) and some earlier versions (2.8) can be downloaded.

For distribution, only a library is needed. This library comes pre-installed on many Linux systems (care must be taken that the library version matches the version used in the program, version 2 and 3 are not compatible). For Windows systems, the library must be downloaded, and can be distributed along the application that uses it.

SQLite can be accessed from Lazarus in 3 ways:

1. Using the plain C interface. This has the least overhead. Free Pascal is distributed with the SQLite C header translations for versions 2 and 3 of SQLite.
2. Using the `TSQLiteDataset` component in the sqlitelaz package, which is distributed standard with Lazarus.
3. Using the ZeosLib package. It contains a driver for SQLite, and this can be used.

For Delphi, there are also several ways:

1. Using various `TDataset` descendents. Several can be found on `torry.net`, but most are for the 2.X version of SQLite.
2. Using ZeosLib, the latest development version has support for sqlite.
3. Using the DISQLite package. This is a version of the SQLIte engine which does not need a DLL. There is a personal version as well as a professional version. Only the professional version contains a TDataset descendent.

4. There are several other `TDataset` descendents, which can be found easily through Google.

In this article, the plain C interface will be used, together with the `TSQLIteDataset`.

2 A database model

To test SQLite, and compare it with other database engines in later contributions, a simple database will be created. The same database will be created for all engines, and it will be filled with the exact same data.

The database models a school where pupils check in and out whenever they arrive in school or leave school. For this, 2 tables are created:

PUPIL This table contains the definitions of the pupils.

PUPILTRACK This contains the actual tracking data.

The `PUPIL` table has 3 fields:

PU_ID a unique integer, the primary key of the table.

PU_FIRSTNAME A string field containing the first name of the pupil.

PU_LASTTNAME A string field containing the last name of the pupil.

The `PUPILTRACK` table has 5 fields:

PT_ID a unique integer, the primary key of the table.

PT_PUPIL_FK A reference to the pupil for whom the tracking data is kept.

PU_CODE A 1 character field with 2 possible values, 'I' for entry ('in'), 'O' for exit ('out').

PU_DATE The date of the registration.

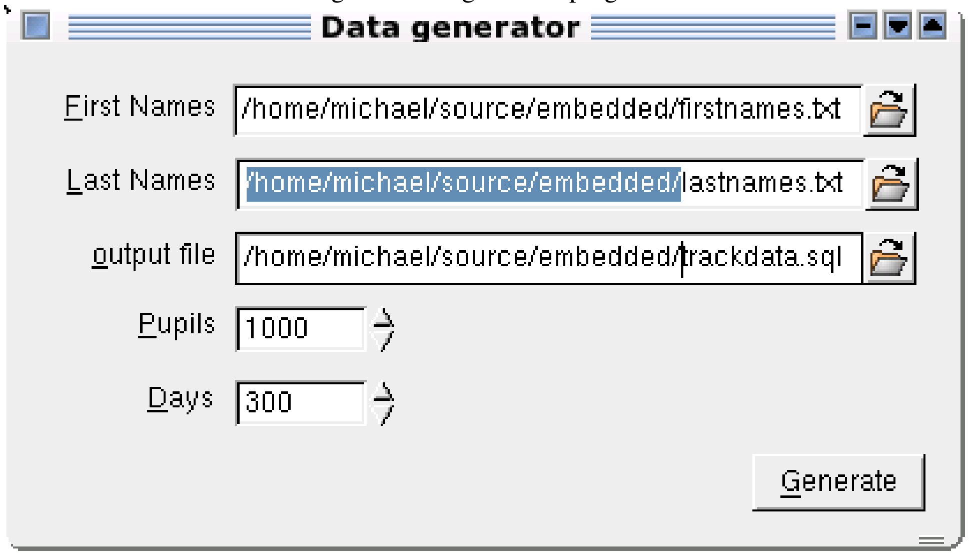
PU_TIME The time of the registration.

The following SQL is used to create the tables:

```
CREATE TABLE PUPIL (
  PU_ID INTEGER NOT NULL,
  PU_FIRSTNAME VARCHAR(50) NOT NULL,
  PU_LASTNAME VARCHAR(50) NOT NULL,
  CONSTRAINT PUPIL_PK PRIMARY KEY (PU_ID)
);

CREATE TABLE PUPILTRACK (
  PT_ID INTEGER NOT NULL,
  PT_PUPIL_FK INTEGER NOT NULL,
  PT_CODE CHAR(1) NOT NULL,
  PT_DATE DATE NOT NULL,
  PT_TIME TIME NOT NULL,
  CONSTRAINT PUPILTRACK_PK PRIMARY KEY (PT_ID),
  CONSTRAINT R_PUPILTRACK_PUPIL
```

Figure 1: The generator program


Data generator

First Names /home/michael/source/embedded/firstnames.txt

Last Names /home/michael/source/embedded/lastnames.txt

output file /home/michael/source/embedded/trackdata.sql

Pupils 1000

Days 300

```
    FOREIGN KEY (PT_PUPIL_FK) REFERENCES PUPIL(PU_ID)
);

CREATE INDEX I_PUPILTRACK_DATE ON PUPILTRACK (PT_DATE);
CREATE INDEX I_PUPILLASTNAME ON PUPIL (PU_LASTNAME);
CREATE INDEX I_PUPILFIRSTNAME ON PUPIL (PU_FIRSTNAME);
```

The statements are in a file db.sql, which is on the CD distributed with this article. The indexes and constraints can also be created after the data has been inserted in the database, to make the inserts faster. Files for this are called db-noindex.sql and db-index.sql, respectively.

A small application ('generator' figure 1 on page 3) is made which generates random data for these tables. The data is written to a file with SQL statements, which can be executed on the database. A medium-to-large school can have about 100 pupils, a year is calculated to be rougly 300 days, so this means that there will be about 600.000 records in the tracking table. This program is used to create a file (trackdata.sql). The produced file will be used to fill all databases, so they all contain the exact same data.

3 An SQL script executor

To demonstrate the use of all the database engines, a small SQL executor program is made. It reads a file with SQL statements, and executes them one by one, keeping track of progress.

A skeleton for the execution program program looks as in figure **??** on page **??**. When the `Execute` button is pushed, the following code is executed:

```
procedure TMainForm.BExecuteClick(Sender: TObject);
begin
  If Not FileExists(FEFile.FileName) then
    Raise Exception.CreateFmt('File "%s" does not exist.',
                              [FEFile.FileName]);
  BTerminate.Visible:=True;
  PBExecute.Visible:=True;
  Try
    ExecuteSQL(FEFile.FileName);
  Finally
    PBExecute.Visible:=False;
    BTerminate.Visible:=False;
  end;
end;
```

After a check whether the provided filename exists, it makes the 'Terminate' button visible as well as the progress bar. It calls then the `ExecuteSQL` function which will do the work:

```
rocedure TMainForm.ExecuteSQl(FileName: String);

Var
  F : TextFile;
  I : Integer;
  Line : String;
  Start,STop : TDateTime;

begin
  FIgnoreErrors:=CBIgnoreErrors.Checked;
  FLogSQL:=CBlogStatements.Checked;
  FTerminate:=False;
  Start:=Now;
  OpenDatabase;
  Try
    AssignFile(f,FileName);
    Reset(F);
    Try
      I:=0;
      While not (EOF(F) or FTerminate) do
        begin
        Inc(I);
        DoProgress;
        Readln(F,Line);
        If FLogSQL then
          LogLine(I,Line);
        ExecuteStatement(Line,FIgnoreErrors);
        end;
    Finally
      CloseFile(F);
    end;
  Finally
    Stop:=Now;
    If FTerminate then
       MLog.Lines.Add(STerminated);
    MLog.Lines.Add(Format(SStatementCount,
                  [I,FormatDateTime('hh:nn:ss.zzz',
                                    Stop-Start)]));
    CloseDatabase;
  end;
end;
```

This procedure is a simple loop. It prepares the database, opens the file, and reads the SQL statements from it: Exactly 1 statement per line, no special parsing is performed. Each line is optionally logged, and then executed. When all is over, the elapsed time is displayed. The `FTerminate` boolean is set when the user presses the 'Terminate' button, and causes the execution to be aborted.

The feedback is moved to the `DoProgress` and `LogLine` procedures. The actual work of setting up the database and executing the statements is done in the `StartDatabase` and `ExecuteStatement` procedures. After the work is done, the connection must be cleaned up in the `StopDatabase` call. These must be implemented for each database separately.

4 The C interface

The executor program will be implemented using the plain C interface of SQLite. Only 3 functions are actually needed to work with a SQLite database:

sqlite3_open Open a database file.

sqlite3_exec Execute an SQL statement.

sqlite3_close Close a previously opened database.

The first call, `sqlite3_open`, takes 2 arguments: a database filename, and an address where to store a database handle (a pointer). It returns an error code which equals `SQLITE_OK` if everything went OK, any other value indicates an error. This call can be used to implement the `OpenDatabase` call:

```
procedure TMainForm.StartDatabase;

Var
  S : String;
  E : Integer;
begin
  S:=FEDB.FileName;
  If (S='') then
    S:='tracker.db';
  E:=sqlite3_open(pchar(S),@FDB);
  if (E<>SQLITE_OK) then
    Raise Exception.CreateFmt(SErrFailedToOpenDatabase,[S,E]);
end;
```

The `FDB` variable is of type `PPSQLite`, an opaque type. It is used in most other SQLite commands. It will be initialized by the `sqlite3_open` call.

The `StopDatabase` call is even more simple, and simply passes the stored `FDB` variable to the `sqlite3_close` call:

```
procedure TMainForm.StopDatabase;

begin
  sqlite3_close(FDB);
end;
```

The execution of the actual statements happens in `ExecuteStatement`:

```
procedure TMainForm.ExecuteStatement(SQL : String;
                                     IgnoreError : Boolean);

Var
  E : Integer;
  P : Pchar;

begin
  Try
    E:=sqlite3_exec(FDB,PChar(SQL),Nil,Nil,@P);
    If (E<>SQLITE_OK) then
      Raise Exception.Create('Code ('+IntToStr(E)+'): '+Strpas(P));
  Except
    On E: Exception do
      begin
      MLog.Lines.add('Error executing statement: '+E.Message);
      If Not IgnoreError then
        Raise;
      end;
  end;
end;
```

The `sqlite3_exec` call is the simplest call to execute SQL statements in SQLite. It takes 5 parameters:

1. A SQLite database handle, as returned by `sqlite3_open`.
2. A null-terminated string containing the SQL statement to be executed.
3. A pointer to a callback routine. This routine will be called for every row in the result set of the SQL statement (if there is a result).
4. A pointer that will be passed as user data to the callback routine.
5. The address of a zero-terminated string that will contain an error message if the function causes an error.

Since the statements to be executed will not return any result, the callback can be `Nil` in the case of the SQL execute program. In general, however, they will return a result. In that case, the callback should look like this:

```
function HandleResult(UserData: Pointer;
                      Columns: Integer;
                      ColumnValues: PPChar;
                      ColumnNames: PPChar): integer; cdecl;
```

The first parameter is the userdata that was passed to `sqlite3_exec`; The meaning of the other three parameters should be obvious from their names: `Columns` is the number of columns in the result set. `ColumnValues` is a pointer to an array of null-terminated strings with the values of the various columns. Similarly, `ColumnNames` is a is a pointer to an array of null-terminated strings with the names of the various columns. Note the mandatory `cdecl` calling convention of this call.

To simply print out the result of a query, one would therefore do:

```
function PrintResult(UserData: Pointer;
                     Columns: Integer;
                     ColumnValues: PPChar;
                     ColumnNames: PPChar): integer; cdecl;

Var
  I : Integer;

begin
  For I:=0 to Columns-1 do
    writeln(ColumnNames[i]:32,' : ',ColumnValues[i]);
  Result:=0;
end;
```

In version 3 of the sqlite engine, support for preparing statements and handling parameters in queries is present, as well as some other auxiliary calls which make programming easier. However, the calls presented here are the main calls needed to program the SQLite engine. The callback system may seem a bit awkward at the beginning, but is otherwise quite easy to work with.

5 A database browser

Lazarus comes with a `TDataset` descendent which is capable of handling a SQLite database: `TSQLiteDataset`, or `TSQLite3Dataset` for version 3 of SQLite. The component can be used as many SQL components, but can also be used to create a table.

The main properties are:

FileName The filename of the SQLite database.

SQLMode A boolean, indicating whether the dataset operates in SQL mode (`True`) or in table mode (`False`).

SQL The SQL to be used in SQL mode.

TableName the name of the table in table mode.

These properties are enough to code many applications. It should be obvious that the `TSQLiteDataset` dataset does not have a centralized connection component: each dataset opens a private connection to the database.

There are some further useful properties:

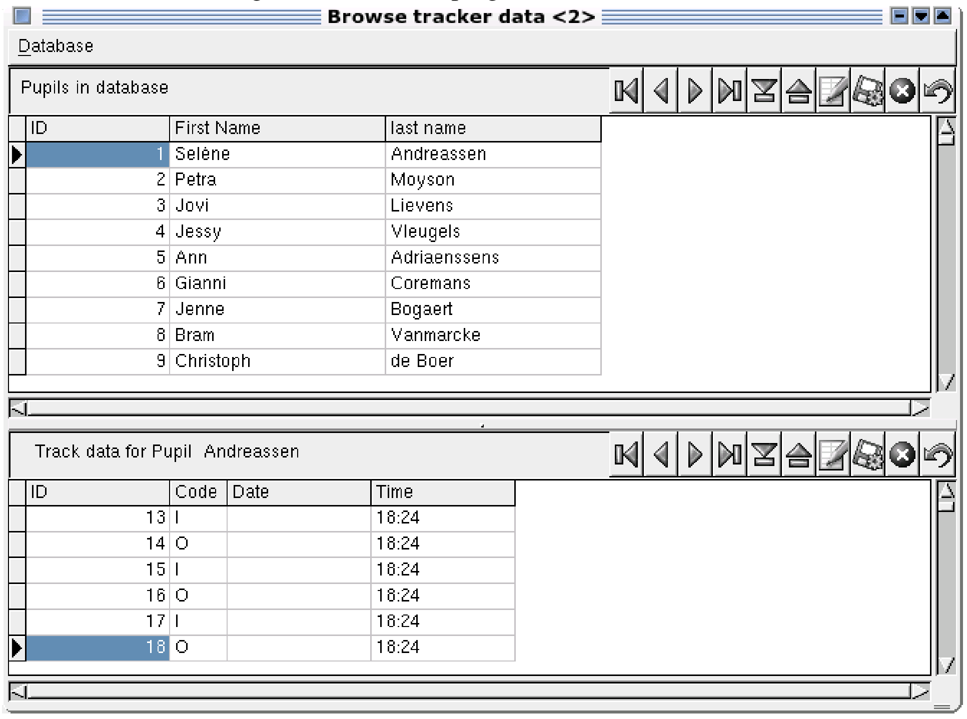
SaveOnClose If `True`, any pending updates to the dataset are posted to the database when the dataset is closed. Pending updates can be posted any time using the `ApplyUpdates` call.

PrimaryKey a list of fields that make up the primary key for the detail table.

MasterSource can be used to create master-detail relationships between various SQLite datasets in table mode.

MasterFields Contains a list of fields in the `MasterSource` table which are used to filter the current table: their values are matched on the values of the index fields of the current table.

Figure 2: A browser program for the track data.



Pupils in database

ID	First Name	last name
1	Selène	Andreassen
2	Petra	Moyson
3	Jovi	Lievens
4	Jessy	Vleugels
5	Ann	Adriaenssens
6	Gianni	Coremans
7	Jenne	Bogaert
8	Bram	Vanmarcke
9	Christoph	de Boer

Track data for Pupil Andreassen

ID	Code	Date	Time
13	I		18:24
14	O		18:24
15	I		18:24
16	O		18:24
17	I		18:24
18	O		18:24

IndexFieldNames a list of fields that make up a key for the detail table. They are matched with the corresponding fields of the `MasterFields` property.

Two of these components are used to compile the application shown in figure 2 on page 8. It contains 2 grids, the top one showing the pupils in the database, the bottom grid showing the tracking data for the currently selected pupil.

The program is quite simple. It uses 2 `TSQLite3Dataset` components, which act as a master-detail relation, although no use is made of the master-detail relation mechanism in `TSQLite3Dataset`. Everything is coded around 2 queries:

```
SELECT * FROM PUPIL
SELECT * FROM PUPILTRACK WHERE (PT_PUPIL_FK=%d)
```

The first query is entered in a first dataset (`SDPupils`). In the `AfterScroll` event of this dataset, the second dataset (`SDPupilTrackData`) is refreshed using the value of the value of the current pupil's ID:

```
procedure TMainForm.SDPupilsAfterScroll(DataSet: TDataSet);

Const
  SSelectTrackData
    = 'SELECT * FROM PUPILTRACK WHERE (PT_PUPIL_FK=%d)';

Var
  ID : Integer;

begin
  SDPupilTrackData.Close;
  if not Dataset.FieldByName('PU_ID').IsNull then
    begin
    ID:=Dataset.FieldByName('PU_ID').AsInteger;
    SDPupilTrackData.SQL:=Format(SSelectTrackData,[ID]);
    SDPupilTrackData.Open;
    end;
end;
```

The other SQLite specific code in the application is opening the database:

```
procedure TMainForm.OpenDatabase;
begin
  With ODFileName Do
    If Execute then
      begin
      FDatabaseName:=FileName;
      InitialDir:=ExtractFilePath(FDatabaseName);
      OpenDatasets;
      end;
end;
```

Where the OpenDatasets method consists of opening the `SDPupils` dataset:

```
procedure TMainForm.OpenDatasets;

begin
  SDPupils.FileName:=FDatabaseName;
  SDPupilTrackData.FileName:=FDatabaseName;
  SDPupils.Open;
end;
```

The second dataset (`SDPupilTrackData`) is opened in the `AfterScroll` event of the pupils dataset.

Remains to close the database, which is simply disconnecting all datasets:

```
procedure TMainForm.CloseDatabase;
begin
  SDPupilTrackData.Close;
  SDPupils.Close;
  FDatabaseName:='';
end;
```

The application contains some other methods, which are simply some actions for the menu; they call the above methods. The interested reader can consult the complete code on the CD-ROM accompagnying this issue.

The careful reader will have noticed that the Date column in the track data grid is empty. This is due to the fact that the SQLite engine does not do any type cheking: it is perfectly possible to store strings in a column declared as an integer, even if the string does not contain a representation of an integer. Likewise, it's possible to store any data in a DATE column. The dates are stored using the `YYYY-MM-DD` format in the database. The `StringToDate` method used to convert the column value of SQL to a TDateTime value fails (it expects a date formatted in the current locale), and therefore the column is shown as empty. It is therefore also possible to store strings of any length in a column declared as a `VARCHAR(10)`

6 Statistics

To be able to compare SQLite with other databases, a number of queries was timed:

1. Inserting the pupil tracking data, about 600.000 records.
2. Retrieving the number of entries per pupil.
3. Retrieving the number of entries before 8:26h on a given date (776 pupils on September 6, 2005) using a left join.
4. Retrieving the number of different pupils that entered school before 8:26h on a given date (776 pupils on September 6, 2005). This is different from the previous query.

The result is given in the following table:

Test	time
1	0:0:42.00
2	0:6:12.75
3	0:0:00.67
4	0:5:59.38

This is by no means a scientific test, but illustrates that SQLite is a very fast engine when it concerns simple queries. By contrast, the second and fourth queries run very slow. The probable cause for this is that joins are not well optimized in SQLit: The same query using a WHERE clause runs equally bad.

7 Conclusion

SQLite is a small, fast engine which can be easily embedded within any application in need of a simple database. Usage is quite simple, both when using the plain C interface as when using `TDataset` components in GUI programs. The performance is quite well to extremely well for simple SQL statements, but SQLite seems to have trouble with some more complex queries. This problem can be overcome by using the appropriate coding style, but may lead to nasty surprises when porting to SQLite from other database engines.

Also the fact that SQLite does not perform any type checks doesn't make it suitable for use with `TDataset` components, which rely on strict adherence to declared types. If the input conforms strictly to the declared types, then this should not be an issue, but for 'unknown' data this may well turn out to be a problem as shown in the example. Even when using strings, one should be careful, since the declared length is not observed: Buffer overflows could very well be the result.

Nevertheless SQLite definitely has it's place, it's speed makes it for instance very usable for large collections of simple data, such as logs or so.