Embedded Databases: NexusDB

Michaël Van Canneyt

July 28, 2006

Abstract

NexusDB is a database engine from NexusDB, specifically designed for use in Delphi. It runs as an embedded database server, but can also be used as a client/server setup; It supports a variety of client/server protocols, and can be used with or without SQL engine. In this last article about Embedded Databases we compare it with other engines.

1 Introduction

The NexusDB engine has it's roots in the - discontinued - FlashFiler product from TurboPower. It was designed specifically as an embedded engine, but has many powerful options for running it as a stand-alone engine as well. It runs solely on Windows platforms, and with any version of Delphi. It comes with it's own memory manager, which is one of the reasons why porting it to Free Pascal/Lazarus is not an option at this time. It would take a huge effort to do this.

Despite the fact that it is designed specifically for Delphi, the NexusDB database engine can also be accessed through ODBC, PHP, ADO.NET: drivers for each of these exist (although it's not quite clear whether they will work with the embedded version)

NexusDB has a very rich feature set: Transactional support, foreign key handling, SQL 2003 compliance, stored procedures, extensive monitoring and logging mechanisms, live backups. The engine can be extended with custom plugins, and so is easily extendable.

NexusDB is a commercial product, but can be downloaded for free - without sourcecode - from their website:

`http://www.nexusdb.com/`

The free version is usable only for embedded use: no client/server capabilities are provided. There are 2 versions of the NexusDB engine: an older version (V1) and a newer version (V2).

Installation is very simple, the product installs itself and provides a large number of help files, which provided detailed information in how the product works.

In the next section, the architecture of the Nexus DB components is outlined. After that, the SQL executor application which was coded for the other database engines will be coded for the Nexus DB engine as well, after which performance will be tested on the Pupil Tracker database.

2 Architecture

NexusDB is built very modular: the different parts of the system are quite independent of each other. This means that it is possible to tune the application to a large degree: only the functionality that is actually needed can be included in the final binary. No external libraries or drivers are needed. The engine is directly included in the application executable.

Since everything is built with Delphi components, this modularity reflects itself in the components that are installed in the component palette of Delphi by the installer program.

The main components are the following:

TnxServerEngine This is the Nexus Server engine. It is the core of the Nexus DB architecture, and contains the actual file and data management routines. It is only needed if the application will function as a server application - which is of course the case for any embedded application. Pure clients would not need this part.

TnxDatabase This represents a Nexus Database. It's mainly the definition of the database, as the actual data is handled by the server engine. It's always needed, also on a pure client.

TnxTransContext This is a transaction context. If this component is present, the use of transactions is enabled, and the component offers the methods to start, commit or rollback transactions. Note that DDL statements (such as `CREATE TABLE`) can not be executed in the context of a transaction.

TnxSession This represents a single connection to a database. It is always needed in a client application. If a single application has multiple threads, each thread should use it's own session.

TnxSqlEngine This component should be added if one wishes to execute SQL statements. It is possible to avoid the use of SQL altogether and access the tables in a database much as one would access DBase files. If a database should be SQL enabled, it should point to an SQL engine component by means of it's `SQLEngine` property.

TnxseAllEngines (or `Tnx1xAllEngines`) are 2 components that are needed to link the various needed server engines in the executable.

This seems like a lot of components - compared to other database engines where only 1 component is needed to establish a connection, but this is a consequence of the modular design of Nexus DB. This could be alleviated by creating some descendents which group some of these components into a single component.

The above components are all needed to be able to connect to a Nexus DB database. The following 2 components are the `TDataset` descendents which actually allow access to the data:

TnxTable This component allows direct access to the tables of the Nexus Database. It does not need any SQL capacities, but does allow complex filters to be built and applied. Likewise, master-detail relationships can be built. It is the Nexus equivalent of the BDE

TnxMemTable This component allows to create and manipulate in-memory tables. These tables are kept in-memory, are accessible throughout simultaneous sessions: they are handled at the server, and disappear when the server is stopped.

TnxQuery This component allows to specify an SQL statement and either directly execute that statement, or open a dataset with the results of the statement.

TnxStoredProc This component allows to execute stored procedures on the server, and possibly retrieve results generated by this procedure.

There are many other components: monitoring and logging components, server plugin components. The help files and examples give more information about most of them.

Additionally, there are various transport mechanisms to create true client/server applications: data transport between client and server can be achieved through COM, Shared memory, TCP/IP, named pipes and so on. All that is needed is to drop the desired transport component on the form, set some properties, and that is it.

3 The SQL Executor

As for the previous applications, a SQL file executor program is constructed. It will read SQL statements one by one from a file (one statement per line) and execute them on the database.

Building the SQL executor application is straightforward, but needs slightly more components as for the other engines. The following components were dropped on the main form of the SQL executor:

1. A `TnxServerEngine` named `SEExecutor`. Instances of `TnxseAllEngines` and `Tnx1xAllEngines` are dropped on the form as well.
2. A `TnxSqlEngine`, needed to enable support for SQL. It's called `SQLEngine`, and it's `Enabled` properyt should be set to `True`.
3. A `TnxSession` component named `SExecutor`.
4. A `TnxTransContext` component named `TCExecutor`. It's session property is set to `SExecutor`
5. A `TnxDatabase` named `DBExecutor`. It's `AliasPath` property should be set to a directory where the database should be created. This property is mutually exclusive with the `AliasName` property, which can be used to enter an alias which is known to the server engine. The `SQLengine` property is set to the `SQLEngine` component. The `Session` property is set to `SExecutor`, and the `TransContext` set to the `TCExecutor` component.
6. Finally, a `TnxQuery` component called `QExecutor`. This component will actually execute the statements. It's `Database` property should be set to `DBExecutor`

Remains to code the 3 routines in the SQL executor that are specific to the database engine. The `StartDatabase` method is first:

```
procedure TMainForm.StartDatabase;

begin
  SQLEngine.Active:=True;
  DBExecutor.AliasPath:=ExtractFilePath(ParamStr(0))+'Data';
  DBExecutor.Active:=True;
  TCExecutor.Active:=True;
  TCExecutor.StartTransaction();
end;
```

As can be seen in the code above, all needed components must be activated before they can be used. The database path is set to the Data subdirectory of the application directory.

The `StopDatabase` performs the inverse operation, in the reverse order:

```
procedure TMainForm.StopDatabase;

begin
  TCExecutor.Commit;
  TCExecutor.Active:=False;
  DBExecutor.Active:=False;
  SQLEngine.Active:=False;
end;
```

Note that to execute DDL statements, the transaction context should not be activated, because then the statements will fail.

Executing a statement is just as simple:

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

begin
  Try
    If (Trim(SQL)<>'') then
      begin
      QExecutor.SQL.Text:=SQL;
      QExecutor.ExecSQL;
      end;
  Except
    On E: Exception do
      begin
      MLog.Lines.add('Error executing SQL statement: '+SQL);
      MLog.Lines.Add('Error message : '+E.Message);
      If Not IgnoreError then
        Raise;
      end;
  end;
end;
```

And with that, the SQL executor is ready. It can be used to create the database, and populate them.

There are 2 caveats in this:

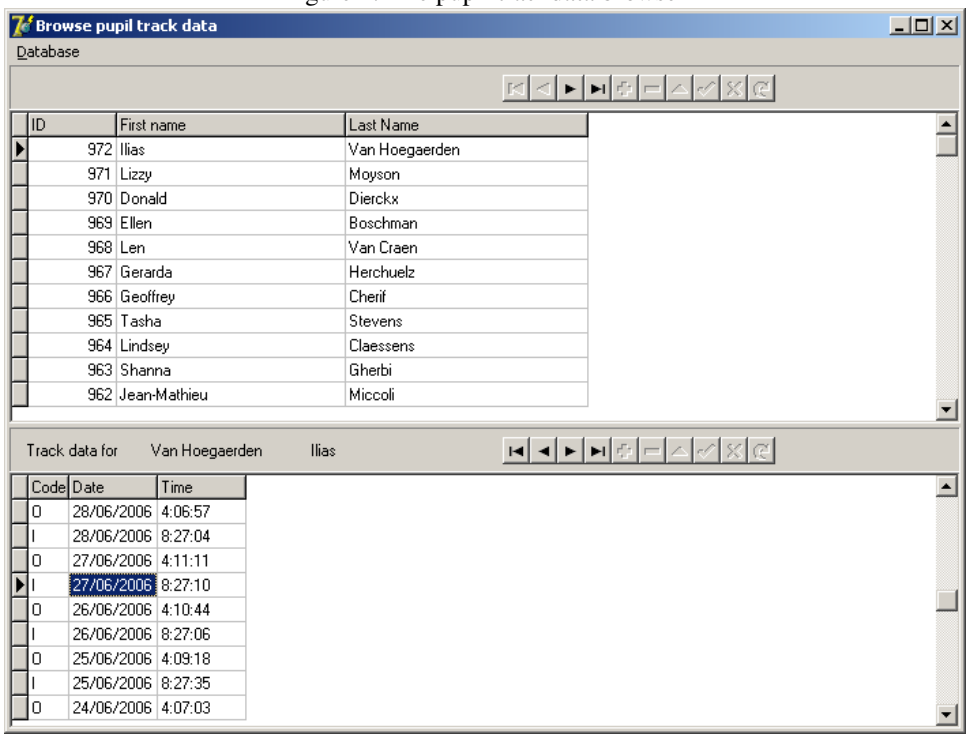
1. The transaction must be disabled when executing the database creation scripts; DDL statements cannot be executed in the context of a transaction.
2. The SQL syntax for DATE and Time fields is not compatible with the standards as followed by the other database engines. It is necessary to change the scripts so a date field is specified as follows:

   ```
   DATE '2006-01-01'
   ```

   Note the DATE identifier in front of the constant. Likewise a time value must be explicitly indicated:

   ```
   TIME '08:27:00'
   ```

Figure 1: The pupil trackdata browser



Other than that, all SQL statements execute without change.

The pupil trackdata browser program is made like all the other programs. The same amount of components is needed as for the SQL Executor program, and the queries are no different than in the previous versions: the NexusDB query component supports master-detail relations based on parameter names, just as the regular `TQuery`, which means no additional coding is needed to display the tracker data of the current pupil. Opening and closing the database is done in much the same way as for the SQL executor: The open and close methods for the database are similar to the ones in the SQL executor:

```
procedure TMainForm.AOpenDBExecute(Sender: TObject);
begin
  SETracker.Active:=True;
  DBTracker.Active:=True;
  QPupils.Open;
  QPupilTrack.Open;
end;

procedure TMainForm.ACloseDBExecute(Sender: TObject);
begin
  QPupilTrack.Close;
  QPupils.Close;
  DBTracker.Active:=False;
  SETracker.Active:=False;
end;
```

The running application can be seen in figure **??** on page **??**.

4 Performance

The performance of the Nexus DB engine is compared with the performance of the other databases. As usual, 4 tests were run. Since the queries needed for the tests need some date/time constants, the tests are repeated here, with corrected SQL statements.

1. Inserting all data in the table (601.000 insert queries).
2. Retrieve the number of tracking items per pupil:

```
SELECT
  PU_ID,COUNT(PT_ID)
from
  pupil
  left join pupiltrack on (PU_ID=PT_PUPIL_FK);
```

3. Number of tracking entries on 6 september 2005, before 8:28 AM.

```
SELECT
  COUNT(PT_ID)
FROM
  PUPILTRACK
WHERE
  (PT_DATE= DATE '2005-09-06')
  AND (PT_CODE='I')
  AND (PT_TIME<= TIME '08:28:00');
```

4. Number of different pupils entering school on 6 september 2005, before 8:28 AM.

```
SELECT
  COUNT(PU_ID)
FROM
  PUPIL LEFT JOIN PUPILTRACK ON (PT_PUPIL_FK=PU_ID)
WHERE
  (PT_DATE= DATE '2005-09-06')
  AND (PT_CODE='I')
  AND (PT_TIME<= TIME '08:28:00');
```

The results can be compared in the following table:

Test	SQLite	Firebird	MySQL	ADS	Nexus
1	0:0:42.00	0:1:29.47	0:0:40.48	0:4:27.00	00:22:39.42
2	0:6:12.75	0:0:02.58	0:0:02.40	0:0:6.54	0:0:37.906
3	0:0:00.67	0:0:00.44	0:0:02.43	0:0:0.63	0:0:0.28
4	0:5:59.38	0:0:00.48	0:0:02.44	0:0:7.18	0:0:0.328

As can be seen, the read results are very, even extremely, good. The insert results, on the other hand, are rather disappointing. Here it was found that the speed of inserts depends largely on whether the table was already used once or not: at the start of the inserts, the table is cleared. If the table was filled from a previous run, the insert time can be halved - which is the opposite effect of the Advantage Database server, where the second run was much more slow when compared to the first run.

5 Nexus DB: Conclusion

Nexus DB is a very fast engine, with excellent modularity in it's design. This can be intimidating at first due to the number of components that are needed to get started, but can

be remedied with some simple coding. The upside is that Nexus DB can be finely tuned to one's needs. It can be upscaled to full client/server, making it a safe bet for embedded work when the application can be upscaled in the near future. The fact that no additional files need to be deployed make it very easy to distribute an application that uses it.

The test results show engine performs very well, compliance to SQL standards is reasonable, but has some quircks which the developer will need to get used to.

The only drawback is that it's very Windows bound; A linux version or Lazarus/Free Pascal would be a definite plus: Linux is, after all, a popular server platform.

All in all it would be fair to say that Nexus DB is a good choice for Embedded work, with definite possibilities for upscaling, unless one has cross-platform development in mind.

6 Embedded database engines: Round up

At the end of these series of embedded databases, the various engines can be compared; Each has their drawbacks and advantages:

SQLite is very easy in deployment (a single DLL) , is very fast for simple queries and inserts. On the downside one should mention that it has issues with difficult queries, and does no typechecking whatsoever on the data entered. In a Pascal environment, this is not recommendable. It cannot be upscaled, so should only be used when this is not an issue. Support for Delphi and Lazarus is good.

Firebird is easy to deploy, works fast, is feature rich. It is cross-platform, and there is no problem upscaling to full client-server. Support for Delphi and Lazarus is very good to excellent.

MySQL is hard to deploy as an embedded engine, is not available for all platforms in the latest version. It's however very fast, and is definitely upscalable. Support for Delphi and Lazarus is medium to good: the issues with various versions are a problem; not all components handle all versions of MySQL.

ADS - Advantage Database Server is very easy to deploy. Works fast - some quircks aside -, is feature rich and is easily upscalable. As a commercial product, support is readily available. Support for Delphi and Lazarus is very good to excellent, and it works both on Windows and Linux. It's available for embedded use for free.

Nexus is probably the easiest to deploy. Works very fast, again, some quircks aside, is also feature rich and is easily upscalable. It is a commercial product, so support is readily available. Support for Delphi is excellent, but support for Linux or Lazarus is non-existent. It's available for embedded use for free.

Which of these engines is to be recommended is hard to say, they are all closely tied. Each has some disadvantages and advantages. It's easier to warn that SQLite and MySQL have some issues that make it less suitable for use with Delphi and/or Lazarus.