Using the Advantage Database Client Engine

Michaël Van Canneyt

September 7, 2011

Abstract

Advantage Database server has an embedded client engine (Advantage Client Engine) which can be used royalty free for desktop applications. This article shows how it can be used to quickly build an application that needs an SQL database.

1 Introduction

Advantage Database Server is a full-blown SQL database that has been around for a while now. It works on intel Windows 32/64 bit, Linux 32/64 bit, supports stored procedures using SQL programming, external functions in native code, supports reference integrity checking and enforcement of other constraints. Last but not least it offers Full Text Search (FTS) on the database as a native mechanism.

Perhaps less known is that an embedded version of this engine is also available, which allows to create SQL-Enabled applications that are easy-to-deploy: all that is needed is to copy a few DLLs to the application's install directory, and all is ready to go. This embedded engine can be used royalty free, provided it is used in what are essentially single-user applications. As soon as multi-tier or webserver applications are built on top of the engine, a license must be obtained (Details of the licenses used can be found on the website indicated below).

For multi-tier or multi-user applications it may be altogether preferable to use the full Advantage Database Server solution. When starting as a single-user application, if ever the application evolves to a complete client/server application, upgrading is as easy as distributing another connection library and setting a different connection string in the application: Advantage Database is a solution that scales easily.

The client engine (whether embedded or not) can be used from several languages, Object Pascal is just one of them - others include C++, .NET, Python and PHP or Java. Object Pascal support for the Advantage Client Engine is very complete: it comes with it's own `TDataset` descendants, ready for use in Delphi or Lazarus. Object Pascal can also be used to write stored procedures or triggers for an Advantage Database.

The installer is available at

`http://www.advantagedatabase.com/`

(look for Delphi TDataset component). The installer will automatically install the components in the installed Delphi IDE. For Lazarus users, a lazarus package (adsl) is available which must be compiled and installed in the IDE. (Instructions can be found in the installed Help file)

Once the installation was successful, a new tab `Advantage` appears on the component palette. It contains the following components, which will look quite familiar to the experienced database developer:

TAdsConnection This component represents a connection to the database. It is equivalent to the TADOConnection or TSQLConnection components in Delphi or lazarus.

TAdsTable This TDataset descendant represents a table in the ADS database. It is the equivalent of TADOTable or TIBTable in Delphi. In Lazarus, it's roughly equivalent to the TDbf component.

TAdsQuery This TDataset descendant can be used to execute SQL statements. It's equivalent to the TADOQuery, TIBQuery or TSQLQuery components of Delphi and Lazarus.

TAdsStoredProc This component can be used to execute or retrieve data from a stored procedure. Again, it is the equivalent of existing stored procedure components in Delphi, tuned for use with Advantage Database server.

TAdsSettings This component can be used to manipulate the local database engine settings. This is something which can also be done in custom code, but the component makes the process easier.

TAdsDictionary This component encapsulates an Advantage Data Dictionary. More about dictionaries follows below.

There are some other components as well, but the above ones are the most important.

2 Creating a database

Advantage Databases have no single database file as Firebird or MS-SQL Server do. Basically, a database is a directory with a collection of table files. Therefor, it's not really necessary to create a database, other than creating a directory to contain the table files. This kind of database is termed 'Free Tables', and is in fact very similar to a paradox database: a set of flat-file tables that are connected through the program logic only. For simple systems, this may well be enough.

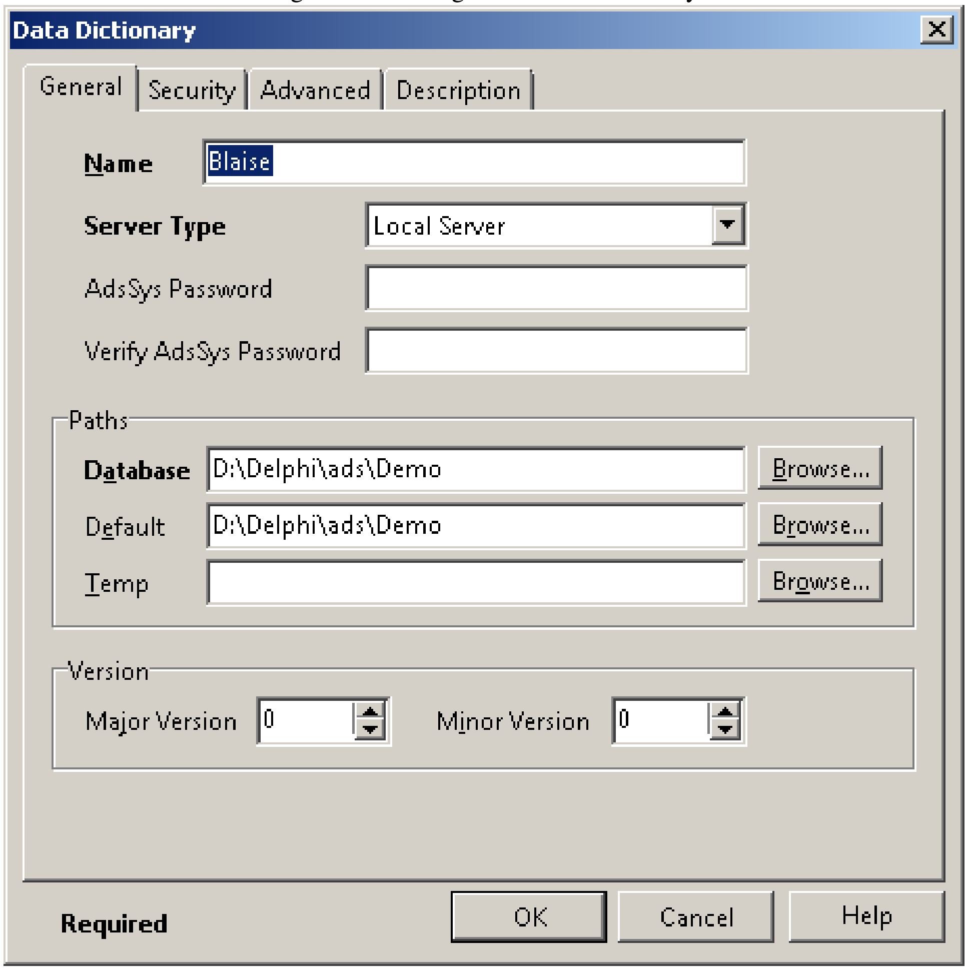
However, a data dictionary can be associated with a database. The data dictionary contains metadata about the tables in the database: it describes constraints on tables, relations between tables, stored procedures, user access rights and many more: all kinds of features one looks for in an RDBMS system.

A data dictionary can be created in one of 3 ways:

1. Through a low-level API call of the Advantage Client Engine library (`AdsDDCreate`).
2. Through a method of the `TAdsDictionary` class. The class is marked deprecated, but it is in fact the easiest way to create a data dictionary in code.
3. Using the Advantage Data Architect. This is a program which must be downloaded separately from the ADS `TDataset` support and installed. It comes with full source, which is an invaluable source of information on using the ADS Delphi components.

Using the Advantage Data Architect is in fact the fastest way to create an Advantage Database. The 'New connection Wizard' exists under the 'File' or 'Connection' menu items. The wizard will start by asking whether a connection must be made to an existing database, or whether a new database must be created. After choosing 'Connection to a new database', the wizard will ask whether a database consisting of free tables must be created, or whether an actual data dictionary must be created. For most purposes, the data dictionary option is the best. The wizard will will then prompt for the necessary parameters such

Figure 1: Creating a new data dictionary


Data Dictionary

General | Security | Advanced | Description

Name Blaise

Server Type Local Server

AdsSys Password

Verify AdsSys Password

Paths

Database D:\Delphi\ads\Demo

Default D:\Delphi\ads\Demo

Temp

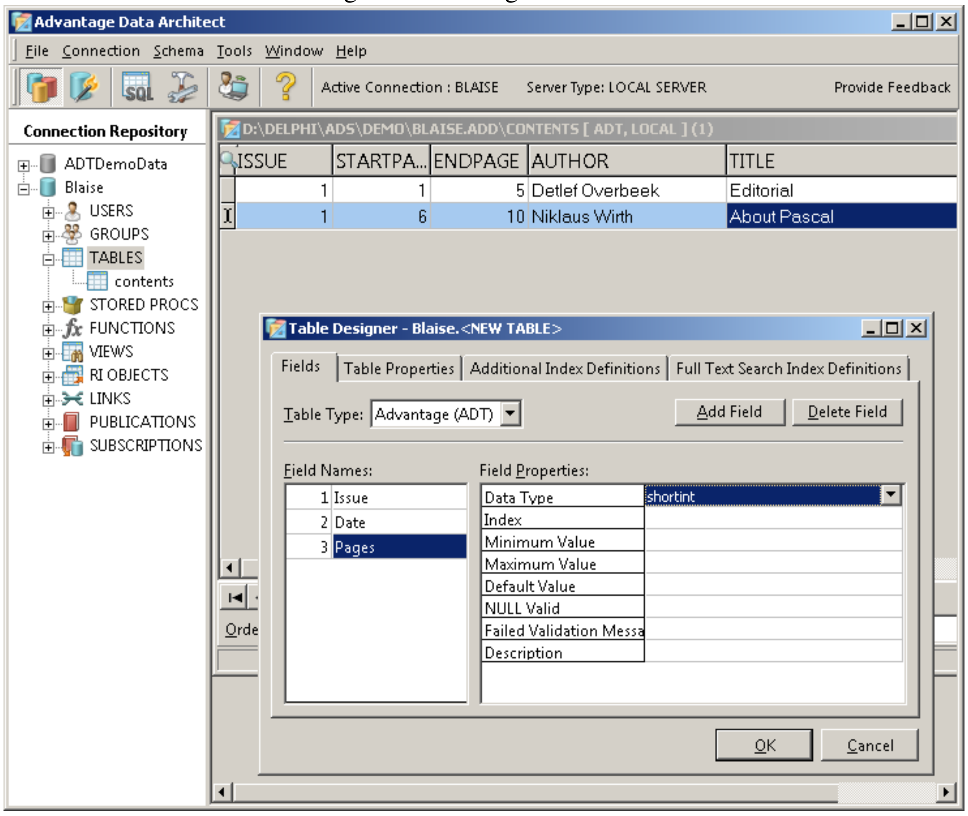
Version

Major Version 0

Minor Version 0

Required

Figure 2: Creating a new table



Active Connection : BLAISE   Server Type: LOCAL SERVER

D:\DELPHI\ADS\DEMO\BLAISE.ADD\CONTENTS [ADT, LOCAL] (1)

ISSUE	STARTPA...	ENDPAGE	AUTHOR	TITLE
1	1	5	Detlef Overbeek	Editorial
1	6	10	Niklaus Wirth	About Pascal

Table Type: Advantage (ADT)

Field Names:

1	Issue
2	Date
3	Pages

Field Properties:

Data Type	shortint
Index	
Minimum Value	
Maximum Value	
Default Value	
NULL Valid	
Failed Validation Messa	
Description	

as the name of the dictionary, and location of the database. The dictionary file will then be saved using the name of the dictionary and an extension .add.

When choosing the 'Free Tables' option, no dictionary file is made. Instead, an alias is made which refers to the database directory.

3 Creating tables

Once the data dictionary is made, tables can be added to it. This can again be done in one of multiple ways:

1. Using the data architect to model the table.
2. Using the data architect to import existing data into a new table.
3. Using code in Delphi: the `TAdsTable` component contains a method to create a table.
4. Using a DDL SQL statement in the Data Architect or from a Delphi program

The first option is doubtless the easiest. A simple set of tables to contain the issues and their contents for the issues of Blaise Pascal magazine can be modeled in less than 5 minutes, and is shown in figure 2 on page 4. A particularly nice thing about the Data Architect is that it can - at will - create code (in several programming languages) to re-create the modeled table in code. It can do the same with SQL: it creates SQL statements which can re-create the whole data dictionary or a simple table: a useful feature also found e.g. in the Lazarus Database Desktop.

Either method can be used to insert code in an application to create an empty Advantage Database when the application is first installed and started on the end-user's system: there is no need to distribute the data dictionary or table files.

The following code, for instance, will recreate the 'contents' table. It uses a `TAdsTable` component, called `TContents`:

```
Procedure TBlaiseModule.CreateContentsTable;

  Function AddDef(AName: String;
                  AType: TFieldType): TFieldDef;

  begin
    Result:=TContents.FieldDefs.AddFieldDef;
    Result.Name:=AName;
    Result.DataType:=AType;
  end;

begin
  With TContents do
    begin
    TableName := 'contents';
    TableType := ttAdsADT;
    AdsTableOptions.AdsCollation := 'ansi';
    end;
  TContents.FieldDeff.Clear;
  AddDef('ISSUE',ftInteger);
  AddDef('STARTPAGE',ftSmallInt);
  AddDef('ENDPAGE',ftSmallInt);
  AddDef('AUTHOR',ftWideString).Size:=50;
  AddDef('TITLE',ftWideString).Size:=100;
  AddDef('ABSTRACT',ftWideMemo).Size:=1;
  AddDef('Keywords',ftWideString).Size:=200;
  AddDef('PageCount',ftSmallInt);
  TContents.CreateTable;
end;
```

The above will look very familiar to programmers that have used the `TTable` or `TDbf` components to create Paradox or DBF tables: first all fields are added to the collection of fielddefs of the `TAdsTable` component. After all needed fields and properties have been set, the `CreateTable` call will create the table in the database and the data dictionary - the `TAdsTable` component should be connected to a `TAdsConnection` instance.

The Advantage Data architect can also create a set of SQL statements to recreate the tables. They can be executed using the `TAdsQuery` component as follows:

```
procedure TBlaiseModule.CreateContentsSQL;

Const
  SCreateSQL = 'CREATE TABLE contents ('+
    '  ISSUE Integer,'+
    '  STARTPAGE Short,'+
    '  ENDPAGE Short,'+
    '  AUTHOR NVarChar(50),'+
    '  TITLE NVarChar(100),'+
    '  ABSTRACT NMemo,'+
    '  Keywords NVarChar(200),'+
    '  PageCount Short) IN DATABASE';

begin
  With QCreate do
    begin
    SQL.Text:=SCreateSQL;
    ExecSQL;
    end;
end;
```

The `QCreate` component should be connected to a `TADSConnection` instance. The `SQL` property (of type `TStrings`) can be filled with a SQL statement that the Advantage Server understands, and `ExecSQL` will execute the query. The supported SQL syntax is documented in the Advantage help file.

The Data Architect tool creates more than 1 SQL statement per table: the table constraints (required fields, indexes and so on) are not included in the `CREATE TABLE` statement, but are created using stored procedures.

For instance, the following procedures will create an index on the field 'AUTHOR' in the contents table and will set the required flag to true:

```
EXECUTE PROCEDURE sp_CreateIndex90(
   'contents','contents.adi','AUTHOR',
   'AUTHOR', '', 2, 512, ':en_US' );

EXECUTE PROCEDURE sp_ModifyFieldProperty ( 'contents',
      'AUTHOR', 'Field_Can_Be_Null',
      'False', 'APPEND_FAIL', 'contentsfail' );
```

The complete list of stored procedures that can be used to modify the data dictionary is included in the ADS help file. Under normal circumstances, it is not necessary to know this list as the Data Architect can be used to create the necessary statements to re-create a table.

Besides tables, it is also possible to create stored procedures. Stored procedures are useful to reduce the execution time of lengthy operations by letting the server execute them. The Advantage Data architect allows to define stored procedures and allows to debug them, which is a very handy feature. The following is an example of a stored procedure that returns all records of the contents table that have a certain word in the keywords field or title field:

```
CREATE PROCEDURE KEYWORDARTICLES(
      akeyword CHAR (50),
      title CHAR (100) OUTPUT,
      author CHAR (50) OUTPUT,
      issue Integer OUTPUT)
BEGIN
DECLARE cursor1 CURSOR AS
  SELECT TITLE,KeyWords,Author,Issue FROM CONTENTS;
DECLARE thelike varchar(55);
thelike=(select rtrim(ltrim(akeyword)) from __input);
thelike='%'+thelike+'%';
```

```
OPEN cursor1;
While FETCH Cursor1 do
  if (Cursor1.TITLE like thelike)
      or (Cursor1.Keywords like thelike) then
    Insert into __Output
    values(Cursor1.title,cursor1.Author,Cursor1.issue);
  end if;
end while;
Close Cursor1;
END;
```

The syntax of this stored procedure is pretty self-explanatory. The use of __input and __output to access the input and output parameters is something to get used to, but other than that, the syntax is straightforward and easy to learn. The stored procedure can then be called like this

```
EXECUTE PROCEDURE KEYWORDARTICLES('Editorial');
```

And it will return a list of all articles which contain the word 'Editorial'. Note that the parameters to this procedure are of type 'CHAR', which necessitates some trimming prior to using the parameter in the condition. The reason for this will be made clear later in this article.

4 Accessing Data

Now that the database has been created, accessing the data can be done using one of 3 components:

TAdsTable This `TDataset` descendant can be used to view all data in a table. It supports filtering and searching using the standard `TDataset` properties and methods (`Filter` and `Locate`). 2 tables can easily be joined together in a master-detail relationship using the `MasterSource` and `MasterFields` properties, a well known mechanism from the Delphi BDE and TDBF components.

TAdsQuery can be used to run a standard SQL SELECT statement if data from multiple tables must be shown in one data set, or if a selection of records of a single table must be shown.

TAdsStoredProc This can be used to read data returned by a stored procedure (or to execute one if it does not return data).

To demonstrate this, a small application can be created. It will connect to the database that was created in the Data Architect. The connection component (of type `TAdsConnection` and named `CBlaise`) and the table components are all placed on a data module (named `BlaiseModule`). This module will contain the methods to create the tables indicated earlier:

Issues is a table that contains information about the various issues of Blaise: Number, date of publication, the theme of the issue and the number of pages.

Contents is a table that holds information about the articles in an issue of Blaise: Title, author, keywords etc. It is linked to the 'Issues' table using a foreign key (a 'RI Object' in the data architect) through the field 'Issue'.

For each of these tables, a `TAdsTable` component is dropped on the module (The components are named `TIssues` and `TContents`), and are linked to a set of `TDatasource` instances (`DSIssues` and `DSContents`). Both are also connected to the `TAdsComponent`

The application is a MDI application, and one of the MDI forms (`TTablesForm`) shows how to browse the issues using a master-detail relationship between the 2 `TAdsTable` components. In order to establish a master-detail relationship, the `TContents` table must have 3 properties set:

IndexFieldNames this must be set to `Issues` : The index is needed to filter the contents table on the `Issue` field.

MasterSource This property must be set to `DSissues` : this tells the `TContents` component that it should filter the shown records depending on what is in the current record in the `TIssues` dataset.

MasterFields this must also be set to `Issues`: This field will be used to filter the records, using the value of the matching field in the `TIssues` dataset. As the user scrolls through the `TIssues` dataset, the `TContents` dataset will automatically refresh itself.

Once this is done, a MDI Chile form can be created. The unit in which the datamodule is defined (dmBlaise) must be added to the uses clause of the form. All that remains to be done is drop a couple of navigators and grid components on the form, connect them to the datasources on the data module and the form is ready to go.

The only thing left to do is make sure the datasets are opened when the form is shown:

```
procedure TTablesForm.FormShow(Sender: TObject);
begin
  BlaiseModule.TIssues.Open;
  BlaiseModule.TContents.Open;
end;
```

Using a menu item in the application's main form, the MDI child can be shown, and should look as in figure 3 on page 9 The application is in fact ready to be used: it's possible to add or edit records in both grids. To make sure that the master-detail relationship is respected when a new records is inserted in the `TContents` dataset, a value is inserted in the `Issues` field:

```
procedure TBlaiseModule.TContentsAfterInsert(DataSet: TDataSet);
begin
  If TIssues.Active then
    TContentsISSUE.AsInteger:=TIssuesIssue.AsInteger;
end;
```

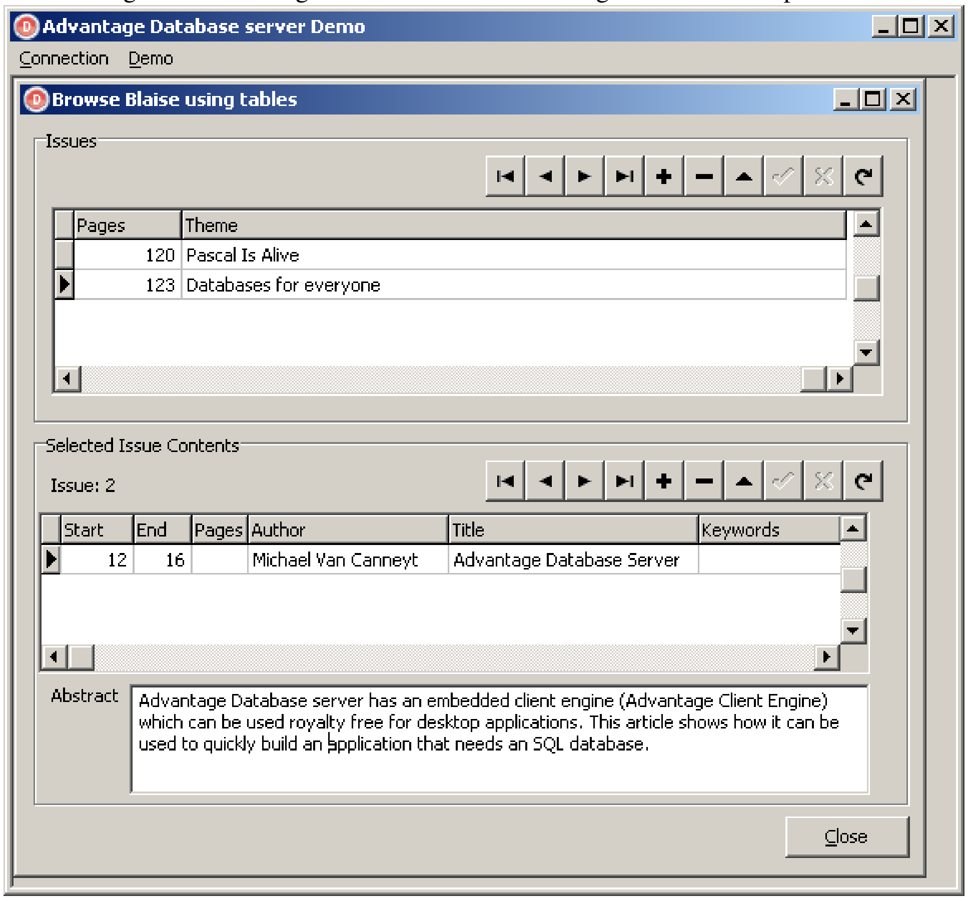
The 'Issue' column in the grid is hidden, so the user cannot override the inserted value.

5 Parametrized queries

The same setup for browsing data can be done using `TAdsQuery` components. In fact, they are even better suited for such situations: the `TAdsQuery` component supports parametrized queries, and this mechanism can be used to create Master-Detail relationships as well.

A Parametrized query is a query which contains a named parameter. For example the following:

Figure 3: Browsing the contents of Blaise using TAdsTable components



Pages	Theme
120	Pascal Is Alive
123	Databases for everyone

Issue: 2

Start	End	Pages	Author	Title	Keywords
12	16		Michael Van Canneyt	Advantage Database Server	

Abstract: Advantage Database server has an embedded client engine (Advantage Client Engine) which can be used royalty free for desktop applications. This article shows how it can be used to quickly build an application that needs an SQL database.

```
SELECT
  Issues.Date, Issues.Issue, Issues.Theme,
  Contents.Title,Contents.Author,Contents.Abstract,Contents.StartPage
FROM
  Contents
  Left Join Issues on (Contents.Issue=issues.Issue)
WHERE
  Keywords like :Keyword
```

The ':keyword' indicates a parameter. It's value is not yet known, but the engine can already prepare the query: parse it, check for syntax errors, allocate resources and calculate the query plan. This needs to be done only once. But the query can be executed multiple times, each time with a different value for the keyword parameter: the engine has already done the preparatory work, and can therefore return a result faster.

In `TAdsQuery`, when one or more parameters are detected in the SQL property, the `Params` collection is filled with an item for each named parameter: this collection can be used to hold values for the parameters. When the query is activated, the value of the parameter is fetched from the item in the `Params` collection and supplied to the database engine.

The above query can be used to implement a window to search for articles in the contents table, based on the keyword. Implementing this window is very simple: one just needs an edit control (`EKeyword`), a button (`BSearch`) and a grid (GSearch). The grid is hooked up to a `TADsQuery` instance (named `QSearch`) which is located on the datamodule of the project. The `OnClick` event handler of the button contains the following code:

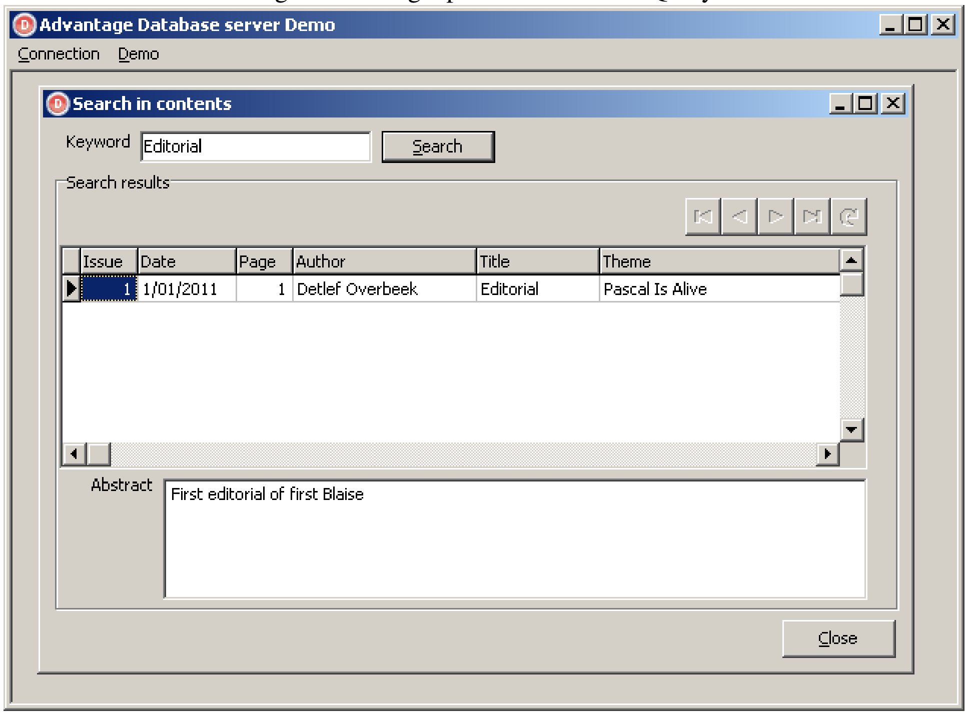
```
procedure TSearchQueryForm.BSearchClick(Sender: TObject);

Var
  S : String;

begin
  S:='%'+EKeyword.Text+'%';
  With BlaiseModule.QSearch do
    begin
    if not Prepared then
      Prepare;
    Close;
    Params.ParamByName('KeyWord').AsString:=S;
    Open;
    end;
end;
```

The value of the search term entered by the user is surrounded by wildcards. After that, the query is prepared, closed (if it was still open from a last search run), the parameter is supplied and then the query is opened again. What happens if the user enters a search term is shown in figure 4 on page 11.

Now, how can this be used to establish master-detail relationships using queries ?

By connecting the Query to another dataset using the `DataSource` property, the `TAdsQuery` will, when opened, fetch the parameter values (for parameters where no value was given explicitly) from the connected dataset. At the same time, when user scrolls through the connected dataset, the query component will react to this and re-open itself with new values for all parameters.

Figure 4: Using a parametrized TAdsQuery



Keyword: Editorial

Issue	Date	Page	Author	Title	Theme
1	1/01/2011	1	Detlef Overbeek	Editorial	Pascal Is Alive

Abstract: First editorial of first Blaise

This can be demonstrated using 2 query components. The first (`QIssues`) has an SQL statement like this:

```
SELECT * FROM Issues
```

and the SQL property of the second query (`QContents`) has a parameter:

```
SELECT * FROM Contents where (Issue=:Issue)
```

The `DataSource` property of the `QContents` query is set to a datasource connected to `QIssues`. After this, a form can be programmed in exactly the same way as it was done with 2 `TAdsTable` components, and the result will look exactly the same. Only this mechanism is vastly more powerful as the mechanism with tables, since the use of parameters allows much more freedom (and is normally also more efficient in terms of speed).

6 Getting data using stored procedures

In the beginning of this article, a stored procedure was created which performs in essence the same operation as the search query presented above. Using the stored procedure is more efficient even than the query, since it is already analyzed and prepared from the moment the database is created (there are even more efficient ways than the stored procedure, using Full Text Search).

It is possible to use a parametrized `TAdsQuery` component to execute the stored procedure and show its results. The following SQL statement would do it:

```
EXECUTE PROCEDURE KEYWORDARTICLES(:KeyWord)
```

The demonstration application shows this.

However, there is a `TDataset` descendant, called `TAdsStoredProc`, which was created specially to deal with stored procedures, whether they return a result or not.

This descendant just needs the name of the stored procedure it should execute or get data from. After that, it behaves like a regular `TDataset`. It also is demonstrated in the sample application. A `TADSStoredProc` component is dropped on the data module and named `SPSearch` after which it is hooked up to the connection component. Finally, the name of the stored procedure is set (`KEYWORDARTICLES`). At this point, the `Params` property of the component will be filled automatically with the names of the parameters.

Note that at the time of writing, `TAdsStoredProc` misses support for Unicode string parameters, which is why the stored procedure was declared using `CHAR` typed parameters instead of `NVARCHAR`.

The form which shows this is identical to the form which searches using a regular query, but is of course hooked to the `TAdsStoredProc` instance, which means there is a slight difference when passing the parameter prior to opening the dataset:

```
procedure TSearchStoredProcForm.BSearchClick(Sender: TObject);
begin
  With BlaiseModule.SPSearch do
    begin
    Close;
    Params.ParamByName('aKeyWord').AsString:=EKeyword.Text;
    Open;
    end;
end;
```

As can be seen, there is no need to enclose it with wildcards. The form will now act and look the same as the first search form.

7 Conclusion

Advantage Database Server is a prime candidate for an embedded SQL database solution: it is easily deployable (2 DLLs suffice for most installations). Added to this the excellent support for stored procedures, embedded functions make it very suitable for deployment with Object Pascal applications. The free licensing scheme, easy upgrade path to a complete client/server solution with remote database, and last but not least its strict adherence to SQL typing make it a better alternative than e.g. sqlite.