dbExpress Tips And Tricks

by Guy Smith-Ferrier

Following on from last month's introduction to dbExpress by Bob Swart, here is a selection of more advanced tips and tricks to help you get the most from dbExpress.

Working Without dbxdrivers And dbxconnections

In last month's article Bob explained how dbExpress driver information (for configuring the InterBase, MySQL, Oracle and DB2 drivers) is contained in the dbxdrivers configuration file. He also mentioned that connection information (analogous to a BDE alias) is contained in the dbxconnections file. In Linux these files are located in the .borland directory beneath the home directory, so if you have installed Kylix as the root user then they will be in root/.borland. In Windows, both files are suffixed with .INI and are located in C:\Program Files\Common Files\Borland Shared\DBExpress. You can change this location by setting `DLLPath` in

```
HKEY_CURRENT_USER\SOFTWARE\
  BORLAND\DBEXPRESS
```

in the registry.

Kylix and Delphi developers have expressed some concern that these configuration files are open to the same kinds of problems that the BDE's IDAPI.CFG file is and that the solution for dbExpress is the same as for the BDE; that is, don't use the configuration files. This section attempts to explain how these files are used and how this solution is not a wise approach.

As Bob mentioned, to use a `TSQLConnection` just drop one on a form and set its `ConnectionName` property. At this point the dbxconnections and dbxdrivers files are read. Even if you set just the `Drivers` property there is no avoiding reading the dbxdrivers file. Drop a `TSQLClientDataSet`, `TDataSource` and `TDBGrid` onto the form and connect them up so that you can see a table in the grid. Now rename the two configuration files and run the program. The program runs just fine, despite the fact that the configuration files have been renamed (you could have deleted the files to be absolutely sure but that is a little bit destructive). The point is that, at runtime, the application *does not* read either of the configuration files.

By default, these configuration files are only read at design-time. In the Form Designer, right-click and select `View as Text` and you will see that all of the configuration information has been read from the configuration files and saved as properties in the form. The result is that your deployed application does not need to have the dbxdrivers and dbxconnections files installed onto your users' machines.

Now we know that these files aren't needed at runtime it begs the opposite question: what if I want to be able to reconfigure my application's database connection after it has been deployed ? If the application doesn't read the configuration files how can I change the location of the database, for example? You have three choices: first, you can solve the problem yourself by reading the appropriate information in the `TSQLConnection.BeforeConnect` event; or, secondly, you can set `TSQLConnection.LoadParamsOnConnect` to `True` to force the configuration files to be read; or thirdly, you can use `TSQLConnection.LoadParamsFromINIFile` to load the parameters from a configuration file which either has a different name or a different location. This last choice has the benefit that you can have a 'private' configuration file which is only used by your application and which does not interfere with other applications and is not interfered with by other applications.

Using dbExpress MetaData

BDE developers will have noted that dbExpress has no equivalent to the BDE's `TSession` component. One of the benefits of this component is that it allows you to retrieve metadata (schema information) about the database. In dbExpress this same information is provided by `TSQLConnection`. The information can be accessed by five methods (`GetTableNames`, `GetFieldNames`, `GetIndexNames`, `GetProcedureNames`, `GetProcedureParams`) and one property (`MetaData`). At first sight, the methods appear easy to use. Add a `TSQLConnection` and three `TListBox` controls to a form, configure the `TSQLConnection` and open it. Add an `OnCreate` event to the form:

```
SQLConnection1.GetTableNames(
  ListBox1.Items);
```

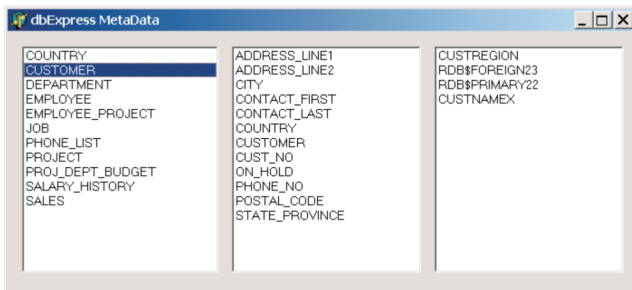
Add an `OnClick` event to `ListBox1`

```
SQLConnection1.GetFieldNames(ListBox1.Items[ListBox1.ItemIndex], ListBox2.Items);
SQLConnection1.GetIndexNames(ListBox1.Items[ListBox1.ItemIndex], ListBox3.Items);
```

➤ *Above: Listing 1*

➤ *Below: Listing 2*

```
var
  List: TList;
  Params: TParams;
  intParam: integer;
begin
  ListBox5.Clear;
  List:=TList.Create;
  SQLConnection1.GetProcedureParams(ListBox4.Items[ListBox4.ItemIndex], List);
  Params:=TParams.Create;
  LoadParamListItems(Params, List);
  for intParam:=0 to Params.Count - 1 do
    ListBox5.Items.Add(Params.Items[intParam].Name);
  Params.Free;
  FreeProcParams(List);
end;
```



➤ *Figure 1*

(see Listing 1). Run the program and click on a table in `ListBox1` (see Figure 1).

`GetProcedureNames` works the same way, but this is where the simplicity stops. The code in Listing 2 dumps the parameters of the selected stored procedure (`ListBox4`) into a new listbox, `ListBox5`.

This routine dumps the stored procedure parameters into a `TList`. `LoadParamListItems` then copies the `TList` items into a `TParams` and finally the list of parameters is added to the `TListBox` (not forgetting to call `FreeProcParams` to free the original `TList`).

TSQLDataSet.SetSchemaInfo

The eager amongst us will, at this point, be wondering how to get at all of the other schema information we want, apart from the tables, indexes and fields. What about views and primary keys? This information is available, but you need to employ a little cunning.

When we used `GetFieldNames` we simply retrieved a list of the field names. We all know that there is more information about fields than just their names. To return this information dbExpress takes the same attitude that ADO does and returns rectangular data as a dataset. Add a `TSQLDataSet` to the last application and set `SQLConnection` to `SQLConnection1`. Add a `TDataSetProvider` and set `DataSet` to `SQLDataSet1`. Add a `TClientDataSet` and set `ProviderName` to `DataSetProvider1`. Add a `TDataSource` and a `TDBGrid` to show the contents of `ClientDataSet1`. Add a button with the following code:

```
SQLDataSet1.SetSchemaInfo(
  stColumns, 'CUSTOMER', '');
ClientDataSet1.Open;
```

`SetSchemaInfo` instructs the dataset to retrieve schema information instead of data. In this example (Listing 3) it is the columns of the `CUSTOMER` table. The result is that `ClientDataSet1` contains all of the information about the columns of the `Customer` table.

We can use Listing 3, for example, to get a list of views, or all of the index information for the `Customers` table sorted in index name order (Listing 4).

In InterBase the `CUSTREGION` index has two fields and so there are two rows for the `CUSTREGION` index. To get a list of primary keys for the `Customers` table you have to be especially cunning. The code in Listing 5 retrieves a list of indexes for the `Customers` table and filters the list so that only those indexes which exist to support a primary key are included.

Executing SQL Directly

As you will be aware, dbExpress datasets are read-only. Any attempt to edit or insert a record generates a *'Cannot modify a read-only dataset'* exception (delete, however, does nothing and does not raise an exception). So, to update data you must either use a `TClientDataSet` or a `TSQLClientDataSet`, or execute your own SQL directly. For this latter purpose we have `ExecuteDirect` and `Execute`. `ExecuteDirect` allows us to execute a statement immediately without supplying any parameters (Listing 6).

The return result indicates the success of the execution. A positive value is the number of records and a negative value is the dbExpress error code. To supply parameters to a statement we use `Execute` (Listing 7).

dbExpress Error Codes

dbExpress error codes are positive numbers. Each error code has a corresponding error string in the `SQLConst` unit. Also in this unit is an array called `DBXError` which is a list of all of the error strings. The array is in the same numeric order as the error numbers, so a dbExpress error 3 is the fourth element in the array (element 0 is not an error and is therefore just an empty string). A consequence of this is that it is easy to write a simple dbExpress error code translator (Listing 8).

Connection Cloning

`TSQLConnection` has a property called `ActiveStatements` which reveals the number of statements which are using the connection at

```
SQLConnection1.TableScope:=[tsView];
SQLDataSet1.SetSchemaInfo(stTables, '', '');
ClientDataSet1.Open;
```

➤ *Above: Listing 3*

➤ *Below: Listing 4*

```
SQLDataSet1.SetSchemaInfo(stIndexes, 'CUSTOMER', '');
ClientDataSet1.IndexFieldNames:='INDEX_NAME';
ClientDataSet1.Open;
```

```
SQLDataSet1.SetSchemaInfo(stIndexes, 'CUSTOMER', '');
ClientDataSet1.IndexFieldNames:='INDEX_NAME';
ClientDataSet1.Filter:='PKEY_NAME IS NOT NULL';
ClientDataSet1.Filtered:=True;
ClientDataSet1.Open;
```

➤ *Above: Listing 5*

➤ *Below: Listing 6*

```
var
  intResult: integer;
begin
  intResult:=SQLConnection1.ExecuteDirect(
  'UPDATE CUSTOMER SET CONTACT_FIRST = "Marge" '+
  'WHERE CUSTOMER = "Mrs. Beauvais"');
  if intResult >= 0 then
    ShowMessage('Success ('+IntToStr(intResult)+')')
  else
    ShowMessage('Failed  ('+IntToStr(intResult)+')');
end;
```

any one time. For example, if you open a TSQLDataSet then ActiveStatements will be increased by one until the dataset is closed. When you execute a statement using Execute or ExecuteDirect then the number of active statements is increased by one for the duration of the execution. For databases such as InterBase this might be mildly interesting, but it is of little consequence. Instead, it has a significant effect on databases such as MySQL. TSQLConnection has another property called MaxStmtsPerConn, which returns the maximum number of statements which the database server will allow to use the connection. For InterBase, Oracle and DB2, this value is 0, indicating that the server does not impose a limit, but for MySQL this value is 1, indicating that a single statement can use this connection at any one time. In theory, if you were to attempt to have two TSQLDataSets using the same connection then the second dataset to open would generate an exception. If this were true then it would radically alter the way developers would use dbExpress by forcing developers to have a TSQLConnection for every TSQLDataSet. Fortunately, TSQLConnection has a built-in solution to this problem in the form of AutoClone. When AutoClone is True, which is the default, TSQLConnection will create subsequent clones of itself as necessary to cope with the additional need for connections. Consequently, you can continue to use TSQLConnection in the same way for MySQL as for InterBase without yielding to the limits of the lowest common denominator (ie MySQL).

Alternatively, you can set AutoClone to False and take control of the cloning process yourself. You could, for example, create a BeforeConnect event for your TSQLConnection and check to see if ActiveStatements is equal to MaxStmtsPerConn and, if so, call CloneConnection to create a new clone. The benefit of taking control of this process is that it would allow you to impose an upper limit on the maximum number of connections for a given user.

With all this said, it should also be noted that the very design of dbExpress means that the number of active statements is typically lower than that of the BDE. If, for example, you intend to allow your users to scroll back and forth through the result set, then in dbExpress you will be using either a TClientDataSet directly or a TSQLClientDataSet. Either way, the data will be read into the ClientDataSet and the necessary SELECT statement closed immediately.

```
var
  Prms: TParams;
  intResult: integer;
begin
  Prms:=TParams.Create;
  try
    Prms.CreateParam(ftString, 'CONTACT_FIRST', ptInput);
    Prms.ParamByName('CONTACT_FIRST').AsString:='Marge';
    Prms.CreateParam(ftString, 'CUSTOMER', ptInput);
    Prms.ParamByName('CUSTOMER').AsString:= 'Mrs. Beauvais';
    intResult:=SQLConnection1.Execute('UPDATE '+
      'CUSTOMER SET CONTACT_FIRST = :CONTACT_FIRST '+
      'WHERE CUSTOMER = :CUSTOMER', Prms);
    if intResult >= 0 then
      ShowMessage('Success ('+IntToStr(intResult)+')')
    else
      ShowMessage('Failed  ('+IntToStr(intResult)+')');
  finally
    Prms.Free;
  end;
end;
```

➤ *Above: Listing 7*

➤ *Below: Listing 8*

```
function dbxErrorCodeToStr(intdbxError: integer): string;
begin
  if (intdbxError > 0) and (intdbxError <=  DBX_MAXSTATICERRORS) then
    Result:=DBXError[intdbxError]
  else
    Result:='Unknown error ('+IntToStr(intdbxError)+')';
end;
```

➤ *Listing 9*

```
INTERBASE - isc_attach_database
INTERBASE - isc_dsql_allocate_statement
INTERBASE - isc_start_transaction
select * from CUSTOMER
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_allocate_statement
SELECT 0, '', '', A.RDB$RELATION_NAME, A.RDB$INDEX_NAME, B.RDB$FIELD_NAME,
  B.RDB$FIELD_POSITION, '', 0, A.RDB$INDEX_TYPE, '', A.RDB$UNIQUE_FLAG,
  C.RDB$CONSTRAINT_NAME, C.RDB$CONSTRAINT_TYPE FROM RDB$INDICES A,
  RDB$INDEX_SEGMENTS B FULL OUTER JOIN RDB$RELATION_CONSTRAINTS C ON
  A.RDB$RELATION_NAME = C.RDB$RELATION_NAME AND C.RDB$CONSTRAINT_TYPE =
  'PRIMARY KEY' WHERE (A.RDB$SYSTEM_FLAG <> 1 OR A.RDB$SYSTEM_FLAG IS NULL) AND
  (A.RDB$INDEX_NAME = B.RDB$INDEX_NAME) AND (A.RDB$RELATION_NAME =
  UPPER('CUSTOMER')) ORDER BY RDB$INDEX_NAME
INTERBASE - isc_dsql_prepare
INTERBASE - isc_dsql_describe_bind
INTERBASE - isc_dsql_execute
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_dsql_fetch
INTERBASE - isc_commit_retaining
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_free_statement
INTERBASE - isc_dsql_fetch
```

TSQLMonitor

One of the truly useful utilities which comes with the Enterprise version of Delphi is SQL Monitor. This tool allows you to view the interaction between the BDE and a SQL Links driver. It is excellent for dispelling myths about how TTable and TQuery really work. dbExpress has a TSQLMonitor component which has the same purpose. Drop a TSQLMonitor on a form and set SQLConnection to SQLConnection1 and Active to True. Add a TMemo and a button with the following code:

```
Memo1.Lines:=
  SQLMonitor1.TraceList;
```

Run the program and click the button and, assuming you have a

TSQLTable which gets opened, you will see Listing 9 in the memo.

By default, TSQLMonitor adds all activity to its TraceList TStrings but you can have greater control over the process than this if you wish: add the Listing 10 OnTrace event to the TSQLMonitor.

```
procedure TForm1.SQLMonitor1Trace(
  Sender: TObject;
  CBInfo: pSQLTRACEDesc;
  var LogTrace: Boolean);
begin
  Memo1.Lines.Add(CBInfo.pszTrace);
  LogTrace:=False;
end;
```

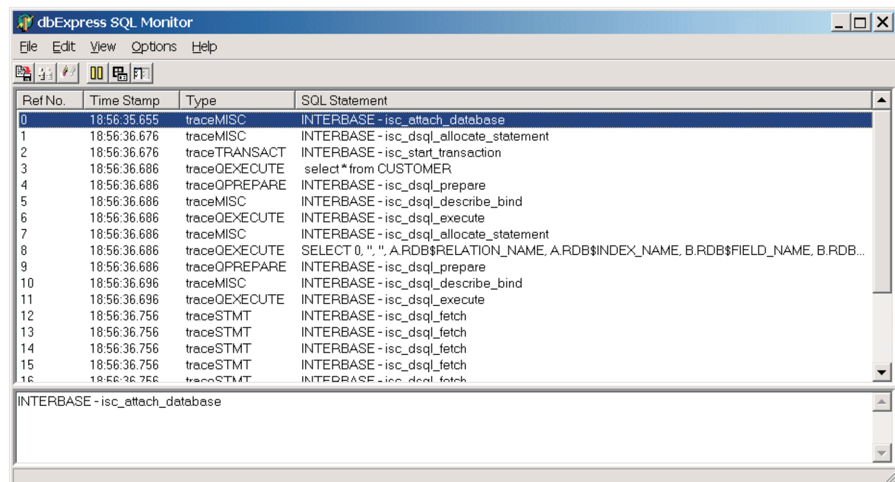
➤ *Listing 10*

This event adds the text from the CBInfo parameter to the memo. The result is the same as for the previous example, except that the memo is updated continuously. The OnTrace event is called before the string is added to TraceList and gives you the opportunity of preventing the string from being added to TraceList by setting LogTrace to False. This is advisable if you have no intention of using the TraceList property. The CBInfo parameter is a SQLTraceDesc record which has the event description (pszTrace), the length of the description (uTotalMsgLen), some internal information (ClientData), and the event category (eTraceCat). Unfortunately, eTraceCat is not set in the current version of dbExpress so you cannot use this for filtering events. Another limitation of the current version is that you can't use it to view parameter data being passed back and forth. This is especially useful for spying on the data used in UPDATE, INSERT and DELETE statements.

TSQLMonitorDialog

TSQLMonitor has no preconceptions about how the data it receives will be used by the developer. Consequently, as you have seen from the previous example, it has no built-in user interface, so you must provide some means by which the log can be viewed. It occurred to me that many developers will simply want to view this log in the same way that the BDE's SQL Monitor allows the log to be viewed, so I wrote a simple component, TSQLMonitorDialog, which behaves like the BDE's SQL Monitor: it is included on this month's disk. Figure 2 is a screenshot of the same sequence shown for the TSQLMonitor example. The Type column is a guess made by interpreting the string. As such it may not yield the same results as when this feature is properly hooked up in dbExpress.

➤ *Figure 2*



TSQLMonitor Analysis

If you look back at the list of instructions displayed by TSQLMonitor in the memo you can see what dbExpress has to do to open a table. The first SQL statement which is executed is select * from CUSTOMER. The next statement executed is rather curious. Just as for SQL Monitor in the BDE, you can see what dbExpress is doing but it doesn't necessarily follow that you can understand it. This second SQL statement retrieves a list of the fields which make up the indexes of the CUSTOMER table. The relevant columns of the result set are shown in Table 1.

What is curious about this second SQL statement is the columns which are used for padding. For example, the first two columns always return 0 and a comma. We can only guess as to why the SQL statement is formulated in this way but one possible explanation is that the code which uses the result set retrieves columns by their ordinal position and not by their field name. The SELECT statement might have been changed since it was originally written and in order to maintain the ordinal position of the columns padding columns have had to be used. But this is just supposition.

From the trace we can see that this SELECT statement is followed by six fetches. This is because there are five rows in the result set and the sixth fetch is needed to hit the end of file. After this is a single fetch which is the retrieval of the first record in select * from CUSTOMER statement.

What is interesting about this trace is that no additional metadata is retrieved for the fields in the result set. This is one of the goals of dbExpress, to avoid the problems of metadata caching which the BDE endured. It is also interesting to note that if you close the dataset and reopen it (without shutting down the application) then the trace you see is exactly the same the second time around. This illustrates that dbExpress does not cache metadata.

If you exchange the TSQLTable for a TSQLClientDataSet then you will see that the SELECT statement used to retrieve the data is opened, all of the data is retrieved and then closed. When the client dataset's ApplyUpdates is executed you can monitor the UPDATE, INSERT and DELETE statements which result.

Performance

One of the main goals of dbExpress is to provide high performance. One of the strategies for achieving this lies in its design. The only way to read result sets in dbExpress is through a read-only, forwards only

RDB$RELATION_NAME	RDB$INDEX_NAME	RDB$FIELD_NAME	RDB$FIELD_POSITION
CUSTOMER	CUSTREGION	COUNTRY	0
CUSTOMER	RDB$FOREIGN23	COUNTRY	0
CUSTOMER	RDB$PRIMARY22	CUST_NO	0
CUSTOMER	CUSTNAMEX	CUSTOMER	0
CUSTOMER	CUSTREGION	CITY	1

➤ *Table 1*

cursor. For most database engines this cursor is the fastest available and thus dbExpress is making the best use of its DBMSs. In addition, dbExpress avoids reading extra schema information.

A direct performance comparison between dbExpress and the BDE or ADO is problematic at best, as it is difficult to find examples which truly compare like with like. Even in such examples it can be argued that the test favours one database middleware over another since it ignores all of the features which are not getting used in the other. In short, the subject of performance is a minefield and this subject alone could take a whole article or more. However, in an attempt to provide some minimal answer to this question I will say that dbExpress's design certainly lends itself to better performance in theory and that in some tests which I have performed, comparing dbExpress connection, opening and traversing InterBase result sets with similar operations for the BDE, dbExpress has equal or better performance.

dbExpress Deployment

One of the fundamental goals of dbExpress is ease of installation and uninstall. dbExpress achieves this goal in spades. To deploy your dbExpress application you copy the relevant dbExpress library (for InterBase, DBEXPINT.DLL on Windows, libsqllib.so.1 on Linux) to the end-user's machine and place it where it can be found. There is no messing around with the registry or INI files. If you intend to use client datasets (`TClientDataSet`, `TSQLClientDataSet`) then you must also copy MIDAS.DLL or midas.so to the end-user's machine. Lastly you must install your DBMS on the user's machine. In the case of InterBase this means copying GDS32.DLL (on Windows) or libgds.so.0 (on Linux) to the user's machine. To uninstall this example on Windows you need simply to delete DBEXPINT.DLL, MIDAS.DLL and GDS32.DLL. Once again, there is no messing around with the registry or INI files.

Under normal circumstances, you do not need to install dbxdrivers or dbxconnections on the user's machine. If, however, you have set `LoadParamsOnConnect` to `True` or have used `LoadParamsFromINIFile`, then you will need to deploy these files as well.

As for the dbExpress footprint, it is tiny. The sum of DBEXPINT.DLL (116Kb), MIDAS.DLL (273Kb) and GDS32.DLL (353Kb) is 742Kb which is barely a fraction of the BDE size.

Statically Linking dbExpress

An alternative to distributing the EXE with a set of DLLs is to statically link the DLLs into the EXE. For an InterBase application, simply include `dbexpint` in the `uses` clause and the dbExpress InterBase functions will be included in the EXE. You can see this by checking the size of the EXE before and after. In addition, rename dbexpint.dll and you'll see the statically linked EXE will run without error (whereas a dynamically linked EXE would fail because the DLL is missing). There are similar units for each of the dbExpress drivers. In addition to this, you can statically link MIDAS.DLL into your program by including `midaslib` and `crtl` in the uses clause. The result is a single EXE which requires no configuration and no installation (assuming the DBMS is already installed). This is a significant step forward in application deployment.

dbExpressPlus

Before I wrap up this article I want to mention dbExpressPlus. This is a freeware set of components developed by Thomas Miller which enhance dbExpress. It is expected to be on the Delphi 6 Companion CD and will also be available from www.bss-software.com after the release of Delphi 6. Its components include `TSQLMetaData`, `TSQLScript`, `TSQLDataPump` and `TSQLAsciiPump` and it is well worth a look.

Conclusion

dbExpress's goals were to have a small footprint, be easy to install and uninstall, be cross-platform and to have exceptional performance. It is fair to say that dbExpress has certainly achieved the first three goals and appears to have achieved the last.

At the time of writing, dbExpress is showing its youth, as various features have not yet been implemented (for example, trace category in `TSQLMonitor`, tracing of parameters in `TSQLMonitor`, support for other transaction isolation levels via xilCUSTOM, drivers for SQL Server, Sybase, Informix, support for the most recent MySQL), but you can rest assured that this is simply a matter of time.

Guy Smith-Ferrier is a Senior Delphi Consultant for Borland's Professional Services Organisation in the UK. Contact Guy at gsmithferrier@capellasoft.com

© 2001 Capella Software Ltd

The opinions of the author are not necessarily the opinions of Borland