SQL access in Borland C++Builder

Written by Mike Destein. Revised by Charles Gallant.

Outline

1. Introduction

The database components of Borland C++Builder were created with SQL and client/server development in mind. Because of this, you will find extensive support to let you leverage the features of a remote server. C++Builder has implemented this support through two methods. First, the native commands of Borland C++Builder allow the developer to navigate tables, set ranges, delete and insert records, and modify existing records. Second, via pass-through SQL, a string can be passed to a server to be parsed, optimized, and executed, with results returned as appropriate.

This paper focuses on the second method of database access, pass-through SQL. While, it is not intended to be a course on SQL syntax or usage, this paper provides many examples of how to use the TQuery and the TStoredProc components. In doing so, many SQL concepts are presented such as selects, inserts, updates, views, joins, and stored procedures. The paper also introduces transaction control and connecting to a database. To begin, let's create a simple SELECT query and display the results.

2. TQuery Component

The TQuery and TTable components are both descended from TDataset which provides much of their database access functionality. TDatasource is used to prepare the data for the visual components. Because both the TTable and TQuery components rely on TDatasource, these components have many common traits. For instance, both provide a DatabaseName property through which an alias can be used to specify what server and database a query will access.

SQL Property

The unique functionality of TQuery includes a property titled ‘SQL.’ The SQL property is used to store a SQL statement. The steps below describe how to use a SQL statement to query for all employees who have a salary greater than a specified amount.

1. Drag and drop a TQuery object onto a form.

2. Set the DatabaseName property to an alias. (This example uses IBLOCAL which points to the sample database EMPLOYEE.GDB.

3. Select the SQL property and click the ‘detail’ button labeled with an ellipsis (“…”). The String List Editor dialog will appear.

4. Enter `Select * from EMPLOYEE where SALARY>50000`. Click OK.

5. Select the Active property and set it to “TRUE.”

6. Drag and drop a TDatasource object onto the form.

7. Set the Dataset property of the TDatasource to “Query1.”

8. Drag and drop a TDBGrid onto the form.

9. Set the Datasource property of the TDBGrid to “Datasource1”.

The SQL property is of type TStrings. A TStrings object is a list of strings, similar to an array. The TStrings data type offers commands to add lines, load from a text file, and exchange data with another TStrings object. Another component that uses TStrings is a TMemo. In the following listing, the user would enter the SQL Statement in the ‘Memo1’ TMemo control and click the Do It button. The results of the query can be displayed in a grid.

Listing 1 displays the code in the Do It button.

Listing 1

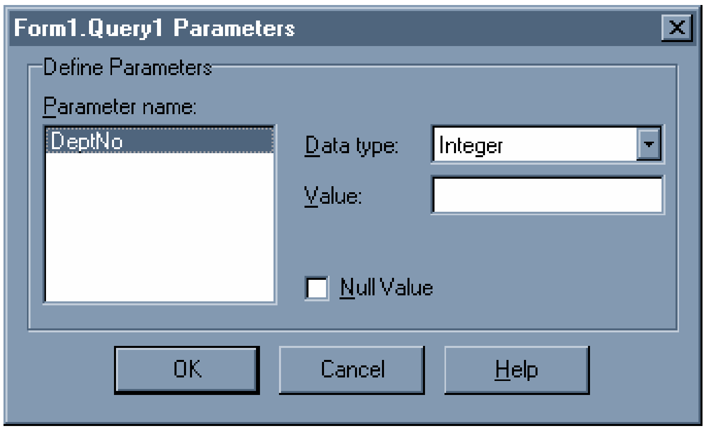
```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   // Check for a blank entry.
   if (strcmp(Memo1->Lines->Strings[0].c_str(), "") == 0)
   {
      MessageBox(0, "No SQL Statement Entered", "Error", MB_OK);
      return;
   }
   else
   {
      // Deactivate the query as a precaution.
      Query1->Close();

      // Empty any previous query
      Query1->SQL->Clear();

      // Assign the SQL property the memo's text.
      Query1->SQL->Add(Memo1->Lines->Strings[0].c_str());
   }

   try
   {
      // Execute the statement and open the dataset
      Query1->Open();
   }

   catch(EDBEngineError* dbError)
   {
      for (int i = 0; i < dbError->ErrorCount; i++)
      {
         MessageBox(0, dbError[i].Message.c_str(), "SQL Error", MB_OK);
      }
   }
}
```

Params Property

This would suffice for a user who knows SQL. Most users, however, do not. So your job as the developer is to provide an interface and construct the SQL statement. In Borland C++Builder, you can use a dynamic query to construct the SQL statement on the fly. Dynamic queries allow the usage of parameters. To specify a parameter in a query, use a colon (“:”) preceding a parameter name. Below is an example SQL statement using a dynamic parameter:

```
Select * from EMPLOYEE
where DEPT_NO = :Dept_no
```

When you want to test or set a default value for the parameter, select the Params property of Query1. Click the '...' button. This presents the parameters dialog. Select the parameter DeptNo. Then select Integer from the data type drop-down list. To set a default, make an entry in the Value edit box.



Bind parameters provide run-time access to modify a SQL statement. The parameters can be changed and the query re-executed to update the data. To directly modify the parameter value, use either the Params property or ParamByName method. The Params property is a pointer to a TParams object. So to access a parameter, you need to access the Item property of the TParams object specifying the index of the Items property. For example,

```
Query1->Params->Items[0]->AsInteger = 900;
```
or,
```
Query1->Params->Items[0]->AsInteger = atoi(Edit1->Text.c_str());
```

The AsInteger property reads the data as an Integer (nicely self documenting). This does not necessarily mean that the field type is an integer. If the field type is ANSIString, C++Builder will do the data conversion. So the above example could have been written as:

```
Query1->Params->Items[0]->AsString = "900";
```
or,
```
Query1->Params->Items[0]->AsString = Edit1->Text;
```

When you would rather use the parameter name instead of the index number, use the ParamByName method. This method will return the TParam object with the specified name. For example,

```
Query1->ParamByName("DEPT_NO")->asInteger = 900;
```

Listing 2 shows a complete example.

Listing 2

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   // Deactivate the query.
   Query1->Close();

   if (! Query1->Prepared)
   {
      // Make sure the query is prepared;
      Query1->Prepare();

      // Take the value entered by the user and replace the parameter
      // with  the value.
      if (strcmp(Edit1->Text.c_str(),"") == 0)
      {
         Query1->ParamByName("DEPT_NO")->AsInteger = 0;
         Edit1->Text = 0;
      }
      else
      {
         Query1->ParamByName("DEPT_NO")->AsString = Edit1->Text.c_str();
      }

      // Trap for errors.
      try
      {
         // Execute the statement, and open the dataset.
         Query1->Open();
      }
      catch(EDBEngineError* dbError)
      {
         for (int i = 0; i < dbError->ErrorCount; i++)
         {
            MessageBox(0, dbError[i].Message.c_str(), "SQL Error", MB_OK);
         }
      }
   }
}
```

Notice the procedure first determines if the query is prepared. When the prepare method is called, Borland C++Builder sends the SQL statement to the remote server. The server will then parse and optimize the query. The benefit of preparing a query is to allow the server to parse and optimize it once. The alternative would be to have the server prepare it each time the query is executed. Once the query is prepared, all that is necessary is to supply new parameter values and execute the query.

Data Source

In the previous example, the user could enter a department number and when the query is executed, a list of employees in that department is displayed. What about using the DEPARTMENT table to help the user scroll through the employees and departments.

> ***Note:*** *The next example has a TTable called Table1. Table1's Databasename is IBLOCAL and the Tablename is DEPARTMENT. DataSource2 is the TDatasource bound to Table1. The table is also active and displaying records in a TDBGrid.*

The way to connect the TQuery to the TTable is through the TDatasource. There are two main techniques to do this. First, place code in the TDatasource's OnDataChange event. For example, Listing 3 demonstrates this technique.

Listing 3 - Using the OnDataChange event to view child records

```
   void __fastcall TForm1::DataSource2DataChange(TObject *Sender,
         TField *Field)
   {
      Query1->Close();

      if (!Query1->Prepared)
      {
         Query1->Prepare();
         Query1->ParamByName("Dept_no")->AsInteger = Table1->Fields[0]->AsInteger;

         try
         {
            Query1->Open();
         }
         catch(EDBEngineError* dbError)
         {
            for (int i = 0; i < dbError->ErrorCount; i++)
            {
               MessageBox(0, dbError[i].Message.c_str(), "SQL Error", MB_OK);
            }
         }
      }
   }
```

While the technique of using OnDataChange is very flexible, there is an easier way to connect a query to a table. The TQuery component has a Datasource property. By specifying a TDatasource for the Datasource property, the

TQuery object will compare the parameter names in the SQL statement to the field names in the TDatasource. Where there are common names, those parameters will be filled in automatically. This would release the developer from having to perform the code in Listing 3 above.

In fact, the technique of using the Datasource requires no additional code at all. Perform the steps in listing 4 to connect the query to the table by DEPT_NO.

Listing 4 - Binding a TQuery to a TTable via the Datasource Property With the Query1, select the SQL property and enter:

```
select * from EMPLOYEE
where DEPT_NO = :DEPT_NO
```

Select the Datasource property and choose the datasource bound to Table1 (Datasource2 in the sample). Select the Active property and choose TRUE. That's all you need to do for this type of relation. There are some limitations on parameterized queries, however. Parameters are limited to values. You cannot use a parameter for a Column name or Table name for example. To create a query that dynamically modifies the table name, one technique could be to use string concatenation. Another technique is the use the Format command.

sprintf Function (from the C++ RTL)
The sprintf function will replace the format parameter (%s, %d, %n, etc.) with a value passed. For example,

```
sprintf(sqlStr, "select * from %s", "EMPLOYEE");
```

The result of the above command would be "Select * from EMPLOYEE". The function will do a literal replacement of the format parameter with the arguments . When using multiple format parameters,
replacement is done from left to right. For example,

```
tblName = "EMPLOYEE";
fldName = "EMP_ID";
fldValue = 3;

sprintf("select * from %s where %s = %d", tblName, fldName, fldValue)
```

The result of this sprintf function is "Select * from EMPLOYEE where EMP_ID = 3". This functionality provides an extremely flexible approach to dynamic query execution. The example in Listing 5 below lets the user to show the salary field in the result. The user also can enter a criteria for the salary field.

Listing 5 - Using sprintf to create a SQL statement procedure

```
void __fastcall TForm1::Button1Click(TObject *Sender)
{
   // this will be used to hold the SQL Statement.
   //
   char* sqlStr = new char[250];

   // These will be used to pass values to sprintf.
   //
   char* fmtStr1 = new char[50];
   char* fmtStr2 = new char[50];

   // If the Salary check box has a check.
   //
   if (showSalaryChkBox->Checked)
   {
      strcpy(fmtStr1, ", SALARY");
   }
   else
   {
      strcpy(fmtStr1, "");
   }

   // If the Salary Edit box is not empty
   //
```

```
      if (!(strcmp(salaryEdit->Text.c_str(),"") == 0))
      {
         strcpy(fmtStr2, salaryEdit->Text.c_str());
      }
      else
      {
         strcpy(fmtStr2, "> 0");
      }

      // Deactivate the query as a precaution.
      //
      Query1->Close();

      // Erase the previous query.
      //
      Query1->SQL->Clear();

      // Build the SQL statement using the sprintf() function.
      //
      sprintf(sqlStr, "Select EMP_NO %s from EMPLOYEE where SALARY %s", fmtStr1,
             fmtStr2);

      Query1->SQL->Add(sqlStr);

      try
      {
         Query1->Open();
      }
      catch(EDBEngineError* dbError)
      {
         for (int i = 0; i < dbError->ErrorCount; i++)
         {
            MessageBox(0, dbError[i].Message.c_str(), "SQL Error", MB_OK);
         }
      }
   }
```

In this example, we are using the Clear and Add methods of the SQL property. Because a prepared query uses resources on the server and there is no guarantee that the new query will use the same tables and columns, Borland C++Builder will unprepare the query any time the SQL property is changed. When a TQuery has not been prepared (i.e. the Prepared property is False), C++Builder will automatically prepare it each time it is executed

Open vs. ExecSQL

In the previous examples, the TQueries performed a Select statement. Borland C++Builder treats the result of the Select query as a Dataset, like a table would. This is just one class of SQL statements that are permissible. For instance, the Update command updates the contents of a record, but does not return records or even a value. When you want to use a query that does not return a dataset, use ExecSQL instead of Open. ExecSQL will pass the statement to the server to be executed. In general, if you expect to get data back from a query, then use Open. Otherwise, it is always permissible to use ExecSQL, although using it with a Select would not be constructive. Listing 6 provides an excerpt from an example.

Listing 6

```
   void __fastcall TForm1::Button1Click(TObject *Sender)
   {
      // deactivate the query, and clear the current SQL statement.
      //
      Query1->Close();
      Query1->SQL->Clear();

      // change the SQL statement to perform an update (which returns
      // no dataset).
      //
      Query1->SQL->Add("update EMPLOYEE set SALARY = (SALARY * (1 + :raise)) where (SALARY <
:salary)");
      Query1->ParamByName("salary")->AsString = Edit1->Text.c_str();
      Query1->ParamByName("raise")->AsString = Edit2->Text.c_str();

      // execute the new SQL statement.
      //
      try
      {
         Query1->ExecSQL();
```

```
        }
        catch(EDBEngineError* dbError)
        {
           for (int i = 0; i < dbError->ErrorCount; i++)
           {
              MessageBox(0, dbError[i].Message.c_str(), "SQL Error", MB_OK);
           }
        }

        // deactivate the query, and set the SQL statement back to the
        // original SQL statement.
        //
        Query1->Close();
        Query1->SQL->Clear();

        Query1->SQL->Add("select * from EMPLOYEE");

        // open the Query.
        //
        try
        {
           Query1->Open();
        }
        catch(EDBEngineError* dbError)
        {
           for (int i = 0; i < dbError->ErrorCount; i++)
           {
              MessageBox(0, dbError[i].Message.c_str(), "SQL Error", MB_OK);
           }
        }
     }
```

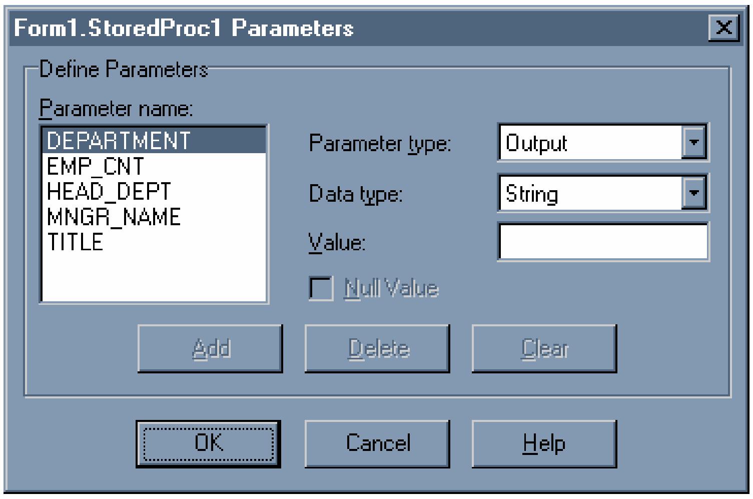
The examples provided here only introduce the subject of using queries in your application. They should give you a good foundation to begin using TQueries in your applications. SQL servers offer additional features including stored procedures and transactions. The next two sections briefly introduce some of these features.

3. TStoredProc Component

A stored procedure is a listing of SQL, or server-specific, commands stored and executed on the server. Stored procedures are not much different in concept than other kinds of procedures. The TStoredProc is descended from TDataset so it will share many traits with the TTable and TQuery. The similarity to a TQuery is especially noticeable. Since stored procedures aren't required to return a value, the same rules apply for the ExecProc and Open methods. Each server will implement stored procedure usage a little differently. For example, if you are using Interbase as the server, stored procedures are executed via Select statements. To view the result of the stored procedure, ORG_CHART, in the example EMPLOYEE database use the following SQL statement:

```
Select * from
ORG_CHART
```

With other servers such as Sybase, you can use the TStoredProc component. This component has properties for Databasename and the name of the stored procedure. If the procedure takes any parameters, use the Params property to input values.


Form1.StoredProc1 Parameters

Define Parameters

Parameter name:
DEPARTMENT
EMP_CNT
HEAD_DEPT
MNGR_NAME
TITLE

Parameter type: Output

Data type: String

Value:

Null Value

4. TDatabase

The TDatabase component provides functionality in addition to that of the TQuery and TStoredProc component. Specifically, a TDatabase allows the application to create a BDE alias local to the application thus not requiring an alias to be present in the BDE Configuration file. This local alias can be used by all TTables, TQueries, and TStoredProcs in the application. The TDatabase also allows the developer to customize the login process by suppressing the login prompt or filling in parameters as necessary. Lastly and most importantly, a TDatabase can keep a single connection to a database funneling all database operations through one component. This allows the database controls to support transactions.

A transaction can be thought of as a unit of work. The classic example of a transaction is a bank account transfer. The transaction would consist of adding the transfer amount to the new account and deleting that amount from the original account. If either one of those steps fails, the entire transfer is incomplete. SQL servers allow you to rollback commands if an error occurs, never making the changes to the database. Transaction control is a function of the TDatabase component. Since a transaction usually consists of more that one statement, you have to mark the beginning of a transaction and the end. To mark the beginning of a transaction, use TDatabase->StartTransaction() . Once a transaction has begun, all commands executed are in a temporary state until either TDatabase->Commit() or TDatabase->Rollback() is called. If Commit is called, any changes made to the data are posted. If Rollback is called, any changes are discarded. The example in Listing 7 below uses the PHONE_LIST table. The function shown will change the location of the office that was entered in the EditOldLocation TEdit control to the location entered in the EditNewLocation TEdit control.

Listing 7

```
   void __fastcall TForm1::Button1Click(TObject *Sender)
   {
      try
      {
         // this will be used to hold the SQL Statement.
         //
         char sqlStr[250];

         Database1->StartTransaction();
         Query1->SQL->Clear();

         // change Location of the office that was entered
         // into EditOldLocation to the value entered into
         // EditNewLocation.
         //
         sprintf(sqlStr, "update PHONE_LIST set LOCATION = \"%s\" where (LOCATION = \"%s\")",
EditNewLocation->Text.c_str(), EditOldLocation->Text.c_str());
         Query1->SQL->Add(sqlStr);
         Query1->ExecSQL();
         Query1->SQL->Clear();

         // change Location of the office that was entered
         // into EditNewLocation to the value entered into
         // EditOldLocation.
         //
         sprintf(sqlStr, "update PHONE_LIST set LOCATION = \"%s\" where (LOCATION = \"%s\")",
EditOldLocation->Text.c_str(), EditNewLocation->Text.c_str());
         Query1->ExecSQL();

         // commit all changes made to this point.
         //
         DataBase1->Commit();
         Table1->Refresh();
         Table2->Refresh();
      }
      catch(EDBEngineError* dbError)
      {
         for (int i = 0; i < dbError->ErrorCount; i++)
         {
            MessageBox(0, dbError[i].Message.c_str(), "SQL Error", MB_OK);
         }
         Database1->Rollback();
         return;
      }
      catch (Exception* exception)
      {
         MessageBox(0, exception->Message.c_str(), "Error", MB_OK);
         Database1->Rollback();
         return;
      }
   }
```

The last step necessary is connecting to the database. In the example above, a TDatabase was used to provide the single conduit to the database, thus allowing a single transaction. To accomplish this, an Aliasname was specified. The alias holds the connection information such as Driver Type, Server Name, and User Name. This information is used to create a connect string. To create an alias, you can use the BDE Config utility or, as the next example shows, you can fill in parameters at runtime.

The TDatabase component provides a Params property that stores connection information. Each row in the Params property is a separate parameter. In the example below, the user puts their User Name in Edit1 and the Password in Edit2. To connect to the database, the code in Listing 8 is executed:

Listing 8

```
   void __fastcall TForm1::Button1Click(TObject *Sender)
   {
      try
      {
         // create two buffers, one for user name,
         // and the other for password entries.
         //
         char* nameStr = new char[20];
         char* passStr = new char[20];

         //Close the Database, and set the params.
         //
         Database1->Close();
         Database1->DriverName = "INTRBASE";
         Database1->KeepConnection = true;
         Database1->LoginPrompt = false;
         Database1->Params->Add("SERVER NAME=d:\\ebony\\IntrBase\\EXAMPLES\\EMPLOYEE.GDB");
         Database1->Params->Add("SCHEMA CACHE=8");
         Database1->Params->Add("OPEN MODE=READ/WRITE");
         Database1->Params->Add("SQLPASSTHRU MODE=SHARED NOAUTOCOMMIT");
        sprintf(nameStr, "USER NAME=%s", Edit1->Text.c_str());
         Database1->Params->Add(nameStr);
         sprintf(passStr, "PASSWORD=%s", Edit2->Text.c_str());
         Database1->Params->Add(passStr);

         // Re-Open the Database, and re-open the Table
         //
         Database1->Open();
         Table1->Open();
         // Fill a ComboBox with the names of the tables in the
         // Database.
         //
         Database1->Session->GetTableNames(Database1->DatabaseName, "*",
                                           true,
                                           true,
                                           ComboBox1->Items);

      }
      catch(EDBEngineError* dbError)
      {
         for (int i = 0; i < dbError->ErrorCount; i++)
         {
            MessageBox(0, dbError[i].Message.c_str(), "Error", MB_OK);
         }
      }
   }
```

This example shows how to connect to a server without creating an alias. The key points are to specify a DriverName and fill out the Params with the necessary information to connect. You don't need to specify all the parameters, you just need to specify those that are not set with the driver in the BDE Configuration. Making an entry in the Params property will override any settings in BDE Configuration. By omitting a parameter, C++Builder will fill in the rest of the parameters with the settings in the BDE configuration. The example above also introduces the new concepts of sessions and the GetTableNames method. The session variable is a handle to the database engine and is discussed further in your Borland C++Builder documentation.

One other point to be made is the use of SQLPASSTHRU MODE. This parameter of a database determines how native database commands such as TTable.Append and TTable.Insert will interact with TQueries connected to the same database. There are three possible values: NOT SHARED, SHARED NOAUTOCOMMIT, and SHARED AUTOCOMMIT. NOT SHARED means that native commands are using one connection to the server and the queries are using another connection. The server sees this as two different users effectively. Any time that a transaction is active, native commands are not committed until that transaction is posted. If a TQuery is executed, any changes it might make are posted to the database, separate from the transaction.

The other two modes SHARED NOAUTOCOMMIT and SHARED AUTOCOMMIT. AUTOCOMMIT will share the same connection to the server with the native commands and queries. The difference between the two is whether native commands are automatically committed upon execution. If SHARED AUTOCOMMIT is the selected mode, then it would be pointless to begin a transaction that uses native commands to delete a record and then try to issue Rollback. The record would be deleted and the change would be committed prior to the Rollback command. If you need to issue native commands within a transaction and have those commands included with the transaction, make sure the SQLPASSTHRU MODE is set to SHARED NOAUTOCOMMIT or NOT SHARED.

4. Conclusion

C++Builder provides many services to use the SQL language with your database servers. At this point, you should have a good framework to begin using SQL in your C++Builder applications.