Surviving Client/Server: Stored Procedures Part 2

by Steve Troxell

Last month we took a look at some of the benefits of using stored procedures for the SQL queries in your system. This month, we're going to take a closer look at how to construct a few typical Interbase stored procedures and how to access them from Delphi. The techniques used here are specific for Interbase and will undoubtedly be very different for other back-end servers. Consult your server's manuals for specific information.

All of the Interbase examples use the sample EMPLOYEE.GDB database that ships with Delphi. This file is found in the C:\IBLOCAL\EXAMPLES directory. For the purposes of this article, we'll assume you are working with a copy of this database called MYEMPL.GDB and that there is a corresponding BDE alias called MYEMPL. The examples in this article can be found on the free disk with this issue.

Using Interbase Script Files

The best way to manage stored procedures in an Interbase database is to write a script file containing the SQL statements to create the stored procedures. We can create the script file using any ASCII text editor and then run it through ISQL to perform the SQL statements we have stored in our script. In this way we have a "source file" from which we can make changes to the stored procedures and recreate them in the database. Without a script file, it would be very difficult to modify an existing stored procedure.

For this example, we'll write a stored procedure that, given an employee number, returns a few columns from the `Employee` table pertaining to that employee. Listing 1 shows the script file we'll use (LISTING1.SQL on the disk). The first statement that must appear in the script file is a `CONNECT` statement. This is an Interbase SQL statement to connect to the database in which we will run the script. After the `CONNECT` verb, supply the full path and name for the database file. Then we must supply the username and password with which we will connect to database. Be sure to end this entire statement with a semi-colon. Unlike single SQL statements executed interactively with ISQL, SQL statements within script files need to be properly terminated.

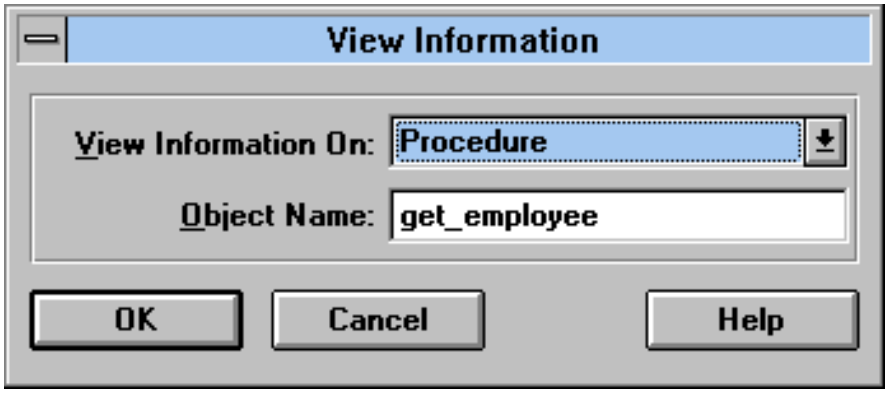
Before we can use the `CREATE PROCEDURE` statement to actually create a stored procedure, we must redefine the SQL terminator character using the `SET TERM` statement. We have to do this because the `CREATE PROCEDURE` statement will have other SQL statements embedded within it (the body of the stored procedure). The script file parser is not very intelligent and it will consider the end of the first stored procedure statement to be the end of the entire `CREATE PROCEDURE` statement. To get around this, we redefine the SQL terminator character for our script file. We continue to use the semi-colon terminator for SQL statements that make up the body of our stored procedure, but "outer" SQL commands, like our `CREATE PROCEDURE` statement, will use the redefined terminator character to signal their end. In this example, we are defining the `^` character to be our new terminator character (notice how the `SET TERM` statement itself is terminated with a semi-colon, the "old" terminator).

Now we are finally ready to make our stored procedure. The `CREATE PROCEDURE` statement is followed by the name of the stored procedure and a comma-separated list of input parameters enclosed within parentheses. Each parameter is given a name and followed by the SQL datatype for that parameter. The `RETURNS` clause is used to identify any output parameters returned by the procedure (again, a comma separated list of parameters enclosed within parentheses).

The statements within the `AS BEGIN...END` block define the statements that make up the stored procedure. In this case, the procedure consists of a single `SELECT` statement. In the `WHERE` clause, we refer to the input parameter `EmpNo` by preceding it with a colon (in much the same way as Delphi identifies parameters within `TQueries`). Also, we use the special `INTO` clause to copy the selected columns into the output parameters. Again, each parameter name is preceded with a colon. The order of the columns listed in the `SELECT` must match the

➤ *Listing 1*

```
CONNECT "c:\iblocal\examples\myempl.gdb"
  USER "SYSDBA" PASSWORD "masterkey";
SET TERM ^;
CREATE PROCEDURE Get_Employee(EmpNo integer)
RETURNS(First_Name varchar(15), Last_Name varchar(20),
        Phone_Ext varchar(4), Dept_No char(3))
AS
BEGIN
  SELECT First_Name, Last_Name, Phone_Ext, Dept_No
    FROM Employee
    WHERE Emp_No = :EmpNo
    INTO :First_Name, :Last_Name, :Phone_Ext, :Dept_No;
END^
```



➤ *Figure 1*

```
EXECUTE PROCEDURE Get_Employee(15)

FIRST_NAME      LAST_NAME            PHONE_EXT DEPT_NO
=============== ==================== ========= =======
Katherine       Young                231       623
```

➤ *Figure 2*

➤ *Figure 3*

order of the parameters listed in the `INTO`.

Note that we terminate the `SELECT` statement with a semi-colon; all SQL statements comprising the body of the stored procedure are terminated with a semi-colon. Note further that the `AS BEGIN...END` block is terminated with a `^`. This uses our redefined terminator character to terminate the overall `CREATE PROCEDURE` statement.

To run the completed script file through ISQL, go to the `File` menu and select `Run an ISQL Script`. Enter the path and name of your script file. It is not necessary to be connected to the database since we are required to use an explicit `CONNECT` statement within the script file. However, the connection made by the `CONNECT` statement only lasts for the duration of the script file. Once the script completes, ISQL will be left in an unconnected state. You will generally want to connect through ISQL anyway, since you can't do anything with the database after you've run your script file unless you do.

Once you've run the script, you can see the stored procedure in the database by selecting `View | Metadata Information` from the menu. Select `Procedure` from the combo box and enter `Get_Employee` for the object name (see Figure 1). Once you click `Ok`, the stored procedure definition appears in the ISQL `Output` window.

To test our stored procedure, go back to the `SQL Statement` window and run the SQL query shown in Figure 2. This is the Interbase SQL statement to run our stored procedure. We are passing an arbitrary value of 15 for the input parameter (the employee number). Note that we do not include anything for the output parameters. In this case, the output of the stored procedure is viewed as a single row result set, where the output parameter names serve as the column headers.

To use this stored procedure from a Delphi application, use a `TStoredProc` component setup as shown in Figure 3 (or see the EXAMPLE1.DPR project on the disk). Set the `DatabaseName` property to the `MYEMPL` alias to connect the component to the database. In the `StoredProcName` property, select the `GET_EMPLOYEE` procedure from the drop down list.

With Interbase, once you select the stored procedure, the `Params` property is automatically populated. The `Params` property editor is shown in Figure 4. The parameters are listed alphabetically and the parameter type and data type for each is already filled out. Delphi is not able to obtain complete parameter information for all back-end servers, so you may have to fill out some of this information by hand if you're not using Interbase.

Ironically, although this stored procedure technically appears to produce a single row result set in ISQL, you do not use `Open` to run it. Since the data for the result set is passed back through parameters, there is no result set as such obtainable by Delphi. So, we execute this procedure by using `ExecProc`. Confused? Suffice it to say that Interbase stored procedures do not produce result sets that are directly accessible by `TStoredProc`. As we'll see soon, other back-end servers do, and this explains why `TStoredProc` has all those data set capabilities built into it.

To run this stored procedure, we simply call the `ExecProc` method and examine the output parameters through the `ParamByName` method. The code shown in Listing 2 executes this stored procedure, examines the first name field to see if a matching record was found (assuming no record can have a null name field), and copies the values into edit controls. Since there is no data set associated with this stored procedure, we cannot use a `TDataSource` component to link data-aware controls to its output.

Multiple Row Result Sets

What about returning multiple row result sets? If there is no data set associated with Interbase stored procedures, how can we handle this? As it turns out, you cannot

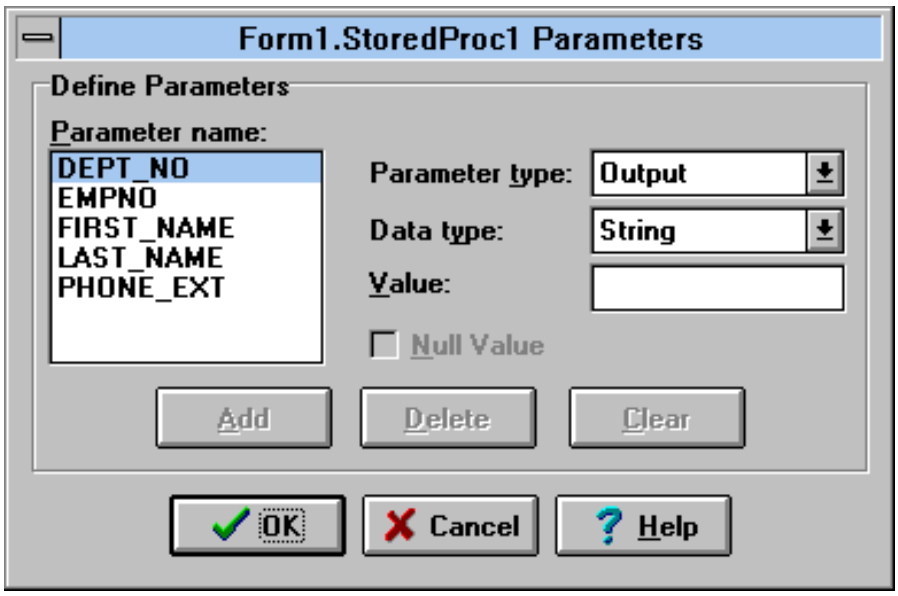
use TStoredProc at all with Interbase stored procedures that return multiple row result sets. If you try, you'll only see the first row if you use ExecProc, or you'll get an *"Error creating cursor handle"* exception if you use Open. Fortunately, we can use a TQuery to execute the stored procedure with a special form of the SELECT statement to obtain a result set.

First we'll need to modify the stored procedure itself to enable it to return more than one row. Let's write a new stored procedure called Get_Employees_ByDept that returns all the employees for a given department. Take a look at Listing 3.

This is very similar to our first stored procedure, except we have added a looping construct (the FOR-DO SUSPEND) around the SELECT statement. Because the column values are passed back through output parameters, only one set of values can be returned at one time. The FOR-DO SUSPEND logic causes control to return to the calling program between each row returned by the query. So, if this query produces a result set of say 10 rows, the first row is passed back through the output parameters and the SUSPEND statement returns control to the calling program (the query remains open and active). The program is then expected to issue an SQL FETCH command to get each successive row. This repeats one row at a time until all the rows are retrieved.

Although this sounds like a lot of work, we can encapsulate it all within a single SELECT statement where we substitute the stored procedure name for the table name (see Figure 5).

This SELECT statement performs all the overhead of FETCHing the subsequent rows from the stored procedure and returns a complete result set for all rows found by the query. Note that this is a special form of the SELECT statement for Interbase, it may or may not be supported (or needed) on other SQL servers. Now all we have to do to access this stored procedure from Delphi is use this SELECT statement within a TQuery. Since we will



➤ *Figure 4*

```
procedure TForm1.btnFindClick(Sender: TObject);
begin
  with StoredProc1 do begin
    ParamByName('EmpNo').AsInteger := StrToInt(edtEmpNo.Text);
    ExecProc;
    if ParamByName('First_Name').IsNull then
      raise Exception.Create('Employee not found');
    edtName.Text := ParamByName('First_Name').AsString + ' ' +
                    ParamByName('Last_Name').AsString;
    edtDept.Text := ParamByName('Dept_No').AsString;
    edtExtension.Text := ParamByName('Phone_Ext').AsString;
  end;
end;
```

➤ *Listing 2*

```
CONNECT "c:\iblocal\examples\myempl.gdb"
  USER "SYSDBA" PASSWORD "masterkey";
SET TERM ^;
CREATE PROCEDURE Get_Employees_ByDept(Dept char(3))
RETURNS(Emp_No smallint, First_Name varchar(15),
        Last_Name varchar(20), Phone_Ext varchar(4))
AS
BEGIN
  FOR
    SELECT Emp_No, First_Name, Last_Name, Phone_Ext
      FROM Employee
      WHERE Dept_No = :Dept
      INTO :Emp_No, :First_Name, :Last_Name, :Phone_Ext
  DO SUSPEND;
END^
```

➤ *Listing 3*

```
SELECT * FROM Get_Employees_ByDept('623')

EMP_NO FIRST_NAME      LAST_NAME           PHONE_EXT
====== =============== =================== =========
    15 Katherine       Young               231
    29 Roger           De Souza            288
    44 Leslie          Phong               216
   114 Bill            Parker              247
   136 Scott           Johnson             265
```

➤ *Figure 5*

be getting a true result set, we Open the query and manipulate it with all the normal data set navigation methods (see Listing 4).

So why does TStoredProc have the same result set handling methods as TQuery and TTable if it apparently can't be used with result set producing stored procedures? In actuality, this limitation is a function of the back-end server. Stored procedures in Sybase and Microsoft SQL Server, for example, can produce result sets that TStoredProc can manipulate.

Listing 5 shows a Microsoft SQL Server version of the Interbase stored procedure shown in Listing 3 (note the subtle differences in SQL syntax). In this case, an "implied result set" is created and output parameters are not necessary. You can access the columns listed in the SELECT statement through TStoredProc's Fields or FieldByName methods, and you can navigate through the result set with First, Next etc (just as you would had you executed the same SELECT statement through a TQuery).

The two stored procedures shown in Listings 3 and 5 illustrate the point I made at the start of this article: stored procedure implementations can and do vary widely between vendors. You have to consult the SQL manuals for your server to know exactly how to write stored procedures.

```
with Query1 do begin
  { Normally, the SQL code would be set at design time
    through the property editor }
  SQL.SetText('SELECT * FROM GetEmployee(:Dept)');
  ParamByName('Dept').AsString := '623';
  Open;
  while not Eof do begin
    { Do something with...} FieldByName('Last_Name').AsString;
    { Do something with...} FieldByName('First_Name').AsString;
    { etc }
    Next;
  end;
  Close;   { Close Query1 }
end;
```

➤ *Listing 4*

SQL Code:

```
CREATE PROCEDURE Get_Employees_ByDept(@Dept char(3))
AS
BEGIN
  SELECT Emp_No, First_Name, Last_Name, Phone_Ext
    FROM Employee
    WHERE Dept_No = @Dept
END
```

Delphi Code:

```
with StoredProc1 do begin
  ParamByName('@Dept').AsString := '623';
  Open;
  while not Eof do begin
    { Do something with...} FieldByName('Last_Name').AsString;
    { Do something with...} FieldByName('First_Name').AsString;
    { etc }
    Next;
  end;
  Close;  { Close StoredProc1 }
end;
```

➤ *Listing 5*

Multiple Queries In One Procedure

You shouldn't look at a stored procedure as simply a wrapper around a single SQL query. As we saw last month, it is possible and often desirable to perform many queries, possibly containing flow control statements, within a single stored procedure.

Listing 6 shows a stored procedure taken from the Interbase sample database EMPLOYEE.GDB. It performs the task of deleting an employee from the system. As you can see, this procedure does more than merely delete a row from one table.

First, it makes sure that the employee is not associated with any sales orders (they must be re-assigned to another salesman, something that should be done by a human being). Then, if the employee was a department manager, they are removed from that table (note that the department is not deleted, it just no longer has a manager). The same process occurs for any projects for which the employee might have been a team leader (again, the projects remain, they are simply leaderless). The employee is removed from any projects they may have been assigned to and all previous salary history is removed. Finally, the actual employee record is removed.

This is known as a "cascading delete", where the act of deleting a master record from a table "cascades" into several deletes across the whole system. It is a way of enforcing referential integrity so that no orphaned records are produced and no invalid links across foreign keys remain in the system. This procedure takes advantage of one of the principal benefits of stored procedures to encapsulate all of the logic involved with deleting an employee. It is also a much cleaner solution than submitting six separate queries from the application, even if those queries could be encapsulated at a single point in the application.

Several more interesting stored procedure examples can be found in your Delphi installation in the directory:

\IBLOCAL\EXAMPLES\TUTORIAL\PROCS.SQL

Summary

Stored procedures provide a whole new layer of functionality to your system. If you're careful about how you organize and name your stored procedures, you can even hide changes to the database schema.

```
CREATE PROCEDURE Delete_Employee (Emp_Num integer)
AS
  DECLARE VARIABLE Any_Sales integer;
BEGIN
  Any_Sales = 0;
  SELECT COUNT(PO_Number)
    FROM Sales
    WHERE Sales_Rep = :Emp_Num
    INTO :Any_Sales;
  IF (Any_Sales > 0) THEN
    EXCEPTION Reassign_Sales;
  UPDATE Department
    SET Mngr_No = NULL
    WHERE Mngr_No = :Emp_Num;
/* If the employee is a project leader, update project. */
  UPDATE Project
    SET Team_Leader = NULL
    WHERE Team_Leader = :Emp_Num;
/* Delete the employee from any projects. */
  DELETE FROM Employee_Project
    WHERE Emp_No = :Emp_Num;
/* Delete old salary records. */
  DELETE FROM Salary_History
    WHERE Emp_No = :Emp_Num;
 /* Delete the employee. */
  DELETE FROM Employee
    WHERE Emp_No = :Emp_Num;
END
```

➤ *Listing 6*

Reorganization of the underlying tables could be transparent to the application.

Hopefully these articles have shown you that stored procedures are valuable tools for the client/server developer. The cost of these benefits is an added dimension to the development process, but this can be minimized by high-quality SQL development tools. There are as yet no standards for stored procedures and some vendor's implementations are better than others. As always, check everything against you're particular server's manuals.

In next month's issue we'll look at aliases and the `TDatabase` component, including a brief look at using the BDE API to set up an alias entirely from code (to give your installation programs that extra pizazz).

Steve Troxell is a Software Engineer with TurboPower Software where he is developing Delphi Client/Server applications using InterBase and Microsoft SQL Server for parent company Casino Data Systems. Steve can be reached on the internet at stevet@tpower.com and also on CompuServe at 74071,2207