Surviving Client/Server: Stored Procedures Part 1

by Steve Troxell

Stored procedures are a way of extracting SQL statements and logic from an application and embedding them within the database. Any application written in any language which has the facilities to execute SQL code can run these stored procedures and manipulate their results.

This month, we're going to explore the pros and cons of stored procedures in general and how to handle them in Delphi in particular. Next month, we'll go into greater detail by exploring the various types of stored procedures, how to create them with Interbase and how to manipulate them from Delphi.

While the concepts of stored procedures are common across many back-end servers, the syntax and functionality varies widely from vendor to vendor. Stored procedures have not been a formal part of the SQL standard, so vendors have been free to implement them however they see fit. You should view the material presented here as an overview to stored procedures and consult your SQL server's reference manuals for specific information.

What Is A Stored Procedure?

Whereas a table is a database object containing data, a stored procedure is a database object containing SQL code which usually manipulates the data in one or more tables. It is really a server-based SQL query, but may also include flow control logic or data computations. In this way, some portions of the system logic are separated from the Delphi application and placed in the back-end database. It's not out of the question that all SQL logic in a system could be located in stored procedures and the only SQL statements remaining in the application would simply be calls to invoke those stored procedures.

You generally pass information between the application and the stored procedure through parameters in the procedure (not unlike a parameterized `TQuery`). Some back-end servers, like Sybase, can produce "implied results sets", meaning their stored procedures can produce multi-row result sets indistinguishable from a result set produced by a regular SQL query. Delphi provides the `TStoredProc` component to help you manage your stored procedures. This component has most of the functionality you'll find in `TQuery`, which is not surprising since stored procedures are just server-based SQL queries. In addition to `TStoredProc`, you can also call stored procedures through a `TQuery` with regular SQL commands.

To illustrate how the use of stored procedures affects the Delphi code you'll write, we'll take a simple database task and solve it with SQL embedded within the application and with a stored procedure.

The task is this: You have a table called `Customers` from which you can look up a particular customer by the `CustomerNo` field. Also in this table is a `CreditLimit` field showing the maximum amount of credit which has been authorized for the customer. In addition, you have an `Extensions` table with a one-to-many relationship to `Customers` (linked through `CustomerNo`). This table contains one row for every credit extension (loan) given to the customer. The `BalanceDue` column shows how much of the loan remains to be paid (completely paid loans have a `BalanceDue` of zero). If we take the credit limit found in the `Customers` table and subtract out all the outstanding balances found in the `Extensions` table, we get the amount of available credit for the customer.

Listing 1 shows one way to approach this problem using SQL code embedded in the Delphi application. `Query1` and `Query2` are `TQuery` components on the form. The SQL statements we're executing each return a result set containing one value, so we use `Open` to execute them. After that, it's a simple calculation using the values returned to determine the available credit. All currency amounts in this example are assumed to be whole dollar amounts, so we use `integer` types.

Listing 2 shows the definition of an Interbase stored procedure to do this same task. The stored procedure accepts an input parameter, `CustNo`, defining the customer we are interested in. It also returns two output parameters, `CreditLimit` and `CreditAvail`, returning the information we need. We use a local variable, `CreditUsed`, to temporarily hold the amount of credit used.

Finally, Listing 3 shows the Delphi code that we use to call this stored procedure (this would replace the code shown in Listing 1). `StoredProc1` is a `TStoredProc` component on the form.

OK, so the Delphi code is more concise, but all we've done is shift the load from one area to another. Also, you could argue that it takes more effort to develop the stored procedure version. So why go to the trouble of splitting the SQL code out into a stored procedure?

Encapsulated Logic

Since a calculation like this could easily be needed at many points in an application, or across multiple applications in an integrated system, we have encapsulated its logic at a single point in the database itself. In so doing, we have removed a *business rule* from the software

```
function TForm1.GetCreditAvail(CustomerNum: LongInt): LongInt;
begin
  Result := 0;
  with Query1 do begin { Typically, the SQL code would be set through the property editor }
    SQL.Clear;
    SQL.Add('SELECT CreditLimit FROM Customers');
    SQL.Add('  WHERE CustomerNo = :CustNo');
    ParamByName('CustNo').AsInteger := CustomerNum;
    Open;
    try
      with Query2 do begin
        SQL.Clear;
        SQL.Add('SELECT SUM(BalanceDue) FROM Extensions');
        SQL.Add('  WHERE CustomerNo = :CustNo');
        ParamByName('CustNo').AsInteger := CustomerNum;
        Open;
        try
          { This check is needed in case there are no credit extensions on file for the customer }
          if Query2.FieldByName('Sum').IsNull then
            Result := Query1.FieldByName('CreditLimit').AsInteger
          else
            Result := Query1.FieldByName('CreditLimit').AsInteger -
                      Query2.FieldByName('Sum').AsInteger;
        finally
          Close;  { Close Query2 }
        end;
      end;
    finally
      Close;   { Close Query1 }
    end;
  end;
end;
```

➤ *Listing 1*

```
CREATE PROCEDURE GetCreditInfo(CustNo integer)
RETURNS(CreditLimit integer, CreditAvail integer)
AS
  DECLARE VARIABLE CreditUsed integer;
BEGIN
  SELECT CreditLimit FROM Customers
    WHERE CustomerNo = :CustNo
    INTO :CreditLimit;
  SELECT SUM(BalanceDue) FROM Extensions
    WHERE CustomerNo = :CustNo
    INTO :CreditUsed;
  /* This check is needed in case there are no credit extensions on file for the customer */
  IF (CreditUsed IS NULL) THEN CreditUsed = 0;
  CreditAvail = CreditLimit - CreditUsed;
END
```

➤ *Listing 2*

```
function TForm1.GetCreditAvail(CustomerNum: LongInt): LongInt;
begin
  { Typically, this would be set through the property editor }
  with StoredProc1 do begin
    StoredProcName := 'GetCreditInfo';
    ParamByName('CustNo').AsInteger := CustomerNum;
    ExecProc;
    Result := ParamByName('CreditAvail').AsInteger;
  end;
end;
```

➤ *Listing 3*

and centralized it in the database, where any application can use it. By extracting business rules like this from the software, all applications can instantly respond to logic changes without the need to re-code, re-compile, and redistribute EXEs or DLLs. We can simply change the logic in the stored procedure and all affected applications will automatically respond to the new logic.

By the same token, encapsulated business rules simplify the reusability of the software when installed at more than one customer site. Let's say you developed a system containing the GetCreditInfo stored procedure and deployed it for Customer A and Customer B, but Customer B wants to include more factors in determining available credit than Customer A does. All you have to do is modify Customer B's stored procedure, none of the application software needs to be modified.

It is this concept of encapsulated business rules in the database that is commonly referred to as 'two-tier architecture'. The first tier is the client application which interacts with the user. The second tier is the business rule logic at the server which interacts with the client application.

Improved Performance

A second advantage of this stored procedure is that it executes more quickly than if the individual SELECT statements were sent as queries from the client. To understand why this is so, we must look at how an SQL query is processed.

When an SQL statement is received from the client, the server parses the statement and validates it, checking its syntax and database object references. It then formulates an execution plan for the query by deciding which, if any, indexes it will use to process the request. Some servers may attempt to reorganize the statement, compare the execution costs of various strategies and select the optimal approach. Finally the query is compiled and executed.

However, with a stored procedure, all the steps of parsing, validating, optimizing, and compiling are performed when the procedure is created. When a client application executes a stored procedure, the server skips those steps since they have already been done and simply executes the pre-compiled SQL statements. This is the same reason why using Prepare and Unprepare with a TQuery improves performance for repetitive parameterized queries (see last month): Prepare takes care of all the steps up through compilation. But, those steps must still be performed at least once at runtime.

With a stored procedure, those steps are all handled in advance and are not needed at runtime.

Another way in which stored procedures can improve performance is by reducing network traffic. For example, if your application needed to manipulate several rows of data to arrive at a decision and that manipulation could be encapsulated within a stored procedure, then only the decision arrived at needs to be sent back to the client. The rows never need to leave the server in order to arrive at the decision. Thus, network congestion for this activity is greatly reduced.

For example, suppose a stored procedure runs several `SELECT` statements to gather values from several different tables into local variables. It then does some calculations and comparisons of these values to arrive at a simple yes/no decision. In this case, all the operations are performed at the server and only the yes/no answer is sent back to the client. If this task were done using SQL embedded in the application, the results of all the `SELECT` statements would be sent back to the client, which would examine the values and arrive at the yes/no decision.

Enhanced Data Security

Client/server databases enforce system security by granting or revoking data access privileges (select, insert, update, delete) for each table to specific users or groups of users. Normally, if the application you are developing allows the user to modify a particular table, that user must be granted the appropriate permissions. However, since client/server systems are inherently 'open' to any ODBC application, this also allows a particularly devious user to log into the database using an external application to modify the table directly, outside the control of *your* application. So, a knowledgeable, unscrupulous user could generate transactions to gain some benefit for themselves or someone they know and promptly remove all trace of accountability for those transactions, through some external program.

However, a stored procedure can read or modify a table independent of any permissions the user might have. While at first glance this may seem to be a design flaw in the security of the database, it's actually an intentional feature to permit even tighter control over data access. A user might only be granted read access to a particular table, but be able to execute a stored procedure to update the same table. The user can modify the table only by running the stored procedure. This allows users to modify database tables under the control of an application written around the stored procedures, but prevents them from altering the data with independent third-party software.

In addition, stored procedures, like all database objects, generally can only be altered by the user who owns them, or the system administrator. So if you include stored procedures in your database, you aren't exposing critical parts of your application to unauthorized modification.

Drawbacks

The principal drawback of stored procedures is reduced flexibility. Since the SQL code in the procedure is fixed, you lose the power of dynamically constructing SQL statements at runtime in response to user actions (see *Dynamic SQL* last month). Also, there is slightly more development overhead since the stored procedures are inherently separated from the application and a certain amount of interface scaffolding is necessary.

On the other hand, it wouldn't be unusual for a large project to have a dedicated SQL developer responsible for all the stored procedures in the system. The Delphi developers can then concentrate on the user interface and not concern themselves with database access other than the task of interfacing to the stored procedures.

TStoredProc

Stored procedures are most commonly accessed in a Delphi application through the `TStoredProc` component. Once the `TStoredProc` component is bound to a database through the `DatabaseName` property, you simply select the stored procedure from the drop-down list in the `StoredProcName` property editor.

With Interbase, Delphi automatically populates the stored procedure's parameter names, types (whether they are input or output parameters) and data types in the `Params` property. However, with some other SQL back-ends, like Sybase and Microsoft, the Borland Database Engine is unable to determine the stored procedure's parameter types (input or output), so you have to assign them manually through the `Params` property editor. If you fail to do this, Delphi raises the exception *'No parameter type for parameter xxx'* when you try to execute the procedure.

Once these connections are defined, you use the `TStoredProc` component very much the same way you use the `TQuery` component. You assign input values to or read output values from the procedure's parameters through the `Params` or `ParamByName` methods. To execute the procedure, you call the `Open` method if the procedure returns a result set, or the `ExecProc` method if it does not. A common error in working with stored procedures (or `TQueries` for that matter) is to confuse these two methods of execution. If you get the exception *'Error creating cursor handle'* when executing a stored procedure, the procedure most likely does not return a result set and it should be executed with `ExecProc`.

TQuery

`TStoredProc` isn't the only mechanism you have to run stored procedures from Delphi. All client/server databases which support stored procedures also provide SQL commands to execute them. Using SQL syntax, you could always execute any stored procedure from a `TQuery`. For example, to run an Interbase stored procedure which deletes a record (does not return a result set), you would use the following SQL statement:

```
EXECUTE PROCEDURE
  DeleteCustomer(:CustomerNo)
```

If we have the TStoredProc component why would we need to use a TQuery to run a stored procedure? First, as we'll see next month, there are certain kinds of stored procedures that simply cannot be executed with TStoredProc. Second, TStoredProc is very closely bound to the Borland Database Engine. As new releases of back-end servers appear, the BDE may not be up-to-date with any changes made in the stored procedure functionality. Since TQuery provides direct access to the stored procedures through the vendor's own SQL command set, you can bypass any shortcomings in TStoredProc's connectivity. Third, in the early releases of Delphi, the BDE and SQL Links drivers had several bugs in communicating with some back-end servers that could be avoided by calling the stored procedure directly through TQuery.

Triggers

Triggers are a special form of stored procedure. Whereas stored procedures are executed on demand by an explicit call from the application, triggers are bound to specific data events on a table and are executed automatically by the server when those events occur. Generally, a trigger can be assigned to execute whenever a record is added, changed, or deleted in a given table. An application cannot explicitly execute a trigger and cannot prevent a trigger from being executed if the proper data event has occurred. Triggers act somewhat like event-handlers in Delphi. Just as you can assign a Delphi procedure to execute when certain application events (like OnClick) occur, you can assign database triggers to execute when certain database events occur. Unlike Delphi event handlers, triggers have no input or output parameters. As they are controlled purely through database events, there is no direct communication between an application and a trigger.

Triggers can contain essentially the same SQL statements and logic which can be used within a stored procedure. Some example uses of triggers would be: posting an audit trail record so that a history is maintained of all changes to a table, auto-generating a unique primary key value for new records, posting a time stamp of record modification, or cascading changes across related tables.

```
CREATE TRIGGER Save_Salary_Change FOR Employee
AFTER UPDATE
AS
BEGIN
  IF (Old.Salary <> New.Salary) THEN
    INSERT INTO Salary_History
      (Emp_No, Change_Date, Updater_ID, Old_Salary, Percent_Change)
      VALUES (
        Old.Emp_No,
        'Now',
        User,
        Old.Salary,
        (New.Salary - Old.Salary) * 100 / Old.Salary);
END
```

➤ *Listing 4*

Listing 4 shows a typical trigger for an Interbase table. This example comes directly from Delphi's sample employee database, file EMPLOYEE.GDB. Whenever the Salary field in the Employee table is changed, this trigger adds a record to the Salary_History table.

This code creates a trigger called Save_Salary_Change which is executed whenever a record in the Employee table is updated. Like a stored procedure, the actual trigger code is defined within the AS BEGIN END block. Interbase supplies two 'virtual' tables, Old and New, to hold the record's contents before and after the update respectively. These virtual tables are available only within triggers. We use them here to determine if the Salary field has changed and, if so, to insert a record into the Salary_History table. The INSERT statement uses the Interbase literal Now to provide the current date and time and the literal User to capture the username of the user making the change.

A common technique in the client/server world is to use a trigger tied to the record insert event to assign an auto-incrementing numeric value to the primary key of the record. In Delphi, at least with data-aware controls and TTable, this poses a bit of a problem. Delphi uses the primary key to keep track of the records. Since the primary key for the inserted record is assigned outside of Delphi's purview by the trigger, Delphi loses track of the record and raises a *'Record/Key Deleted'* exception. To avoid this, rather than use a trigger to assign the key value, you may want to use a regular stored procedure to return the auto-increment value, have Delphi use this to obtain the key value, and assign it to the record's primary key field in TTable's OnNewRecord event handler.

Summary

A client/server system can benefit from stored procedures by encapsulating business rules, improving performance and enhancing data security. However, they reduce your ability to dynamically create queries at runtime and may complicate the development process. Since there are many variations among SQL server vendors, do make sure you check your own vendor's reference manuals.

Next month we'll continue our look at stored procedures by rolling up our sleeves and walking through several Interbase examples, digging into both the SQL and Delphi ends of the business. We'll also take a look at some Microsoft SQL Server examples to illustrate just how different stored procedures can be from server to server.

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