Surviving Client/Server: User-Defined SQL Functions

by Steve Troxell

Delphi developers are accustomed to having a lot of programming power at their fingertips: Object Pascal, the VCL, direct access to the Windows API and hundreds of third-party libraries and components that essentially become seamless extensions to the underlying language. In contrast, SQL can be very limiting. Fortunately, many vendors provide a means for us to add some degree of custom functionality to their implementation of the SQL language. With InterBase, we can use user-defined functions (UDFs) to extend the native library of SQL functions, which is fortunate since InterBase provides us with only three built-in SQL functions (`Upper`, `Cast`, and `Gen_ID`).

A UDF is simply a Delphi function written into a DLL and linked into the InterBase server. Therefore we can create a UDF for anything for which we can create a Delphi function. From there, the DLL function can be called from SQL just as though it was a built-in SQL function. InterBase UDFs are not supported on Novell platforms, so we will be concentrating on InterBase for Windows NT and 95, using Delphi 2.

Writing A Simple UDF

Suppose we have a database containing mathematical or statistical data and we wish to know the absolute value of some field:

```
SELECT Abs(Delta) FROM Telemetry
```

InterBase does not natively provide such a function, but we can easily add one through a UDF. Listing 1 shows our implementation of `Abs`.

Our `Abs` function is expecting a single input parameter of the SQL datatype `Double Precision` and returns a single value of the same datatype. In Delphi, this translates to the `Double` type. All input parameters are passed by reference, so we use `var` to denote this in our function declaration. We could just as well have declared `X` to be a pointer to a `Double`, but using `var` is a bit cleaner and safer. InterBase UDFs must be exported using the C calling convention, so we use the `cdecl` directive. Within the function itself we can do whatever Delphi permits us to do with the parameters given to us. In this case, a simple call to Delphi's built-in `Abs` function is all we need.

Deploying The UDF

Once we've compiled the source, we need to place the resulting DLL file such that the InterBase server will find it when our `Abs` function is requested. DLLs must be placed in the \INTRBASE\BIN directory (\IBLOCAL\BIN for Local InterBase) or in a directory specified by the `PATH` environment variable. InterBase loads the DLL on demand the first time any UDF within it is needed by an InterBase user.

Linking The UDF Into SQL

Finally, we need to link the UDF into SQL itself. For this purpose, InterBase has the `DECLARE EXTERNAL FUNCTION` statement. To allow our `Abs` function to be used, we must connect to the database and execute the SQL statement shown in Listing 2. Notice that the function name we use in SQL can differ from the name we gave our function in the DLL. After we identify the SQL name of the function, we list the SQL datatypes of each of the input parameters in order. Here we have only one double precision input parameter.

The `RETURNS` clause tells us the SQL datatype of the value returned by the function. Remember that our Delphi function returns a `Double` value, not a pointer to a `Double`, so the output parameter is passed back by value rather than by reference. To denote this we include the `BY VALUE` keywords. By reference would be assumed if we hadn't said otherwise, which would mean we would have to return a pointer to something.

To make the association between the DLL function and the SQL function, we name the DLL function in the `ENTRY_POINT` clause and the DLL which contains it in the `MODULE_NAME` clause. It is very important to keep in mind that the function name provided for `ENTRY_POINT` is case-sensitive with

➤ *Listing 1*

```
library OurUDFs;
function udf_Abs(var X: Double): Double; cdecl;
begin
  Result := Abs(X);  { Use Delphi's Abs function }
end;
exports
  udf_Abs;
end.
```

➤ *Listing 2*

```
DECLARE EXTERNAL FUNCTION Abs
  DOUBLE PRECISION
  RETURNS DOUBLE PRECISION BY VALUE
  ENTRY_POINT "udf_Abs" MODULE_NAME "ourudfs.dll"
```

how the function was declared in the DLL source code.

At this point we now have a working Abs SQL function for this particular database. UDFs are not inherently available server-wide, so the DECLARE EXTERNAL FUNCTION statement must be executed in every database in which you want to use the given UDF.

As you can see, UDFs are as simple as writing a Delphi function. The only part to be concerned about is making sure you've handled the translation from SQL datatypes to Delphi datatypes correctly. Let's look at a few more examples.

Passing String Parameters

Another valuable SQL function is SubString, which returns a specified portion of a given string. Ironically, Borland thought well enough to provide this function in its local SQL implementation for Paradox and dBase, but not for InterBase. No matter, we'll simply add one via a UDF as shown in Listing 3.

In SQL, string fields are declared as CHAR or VARCHAR and when these fields are passed into a UDF we receive a null-terminated string. Remember that all input parameters are passed by reference, so our parameter declaration is simply PChar. InterBase only allows numeric datatypes to be returned by value from a UDF, so the return value for Substring must be passed by reference and that means returning PChar.

To do the work of the SubString function, we need only use Delphi's built-in StrLCopy function which returns a specified maximum number of characters of a null terminated string. Here we employ a bit of pointer arithmetic to tell StrLCopy to start copying a certain distance into the given string.

The substring is copied into the static global array Buffer. We must have a static region of memory for the function return value to point to. A local variable in the function would not suffice as that memory space would be on the stack. If our function returned a pointer into its own temporary stack storage, that pointer would be invalid as the stack space would have been destroyed as the function terminated. Note also that we needn't be concerned about multiple InterBase users using our UDF at the same time and overwriting each other's data in Buffer. Invocations of our UDF by different users will create separate instances of the DLL and Win32 provides distinct copies of the DLL's variables for each instance.

➤ *Listing 3*

```
Delphi Implementation:
implementation
var
  Buffer: array[0..255] of Char;
function udf_Substring(S: PChar; var Start, Len: SmallInt): PChar; cdecl;
begin
  Buffer[0] := #0;
  if (Start > 0) and (Len > 0) then
    StrLCopy(Buffer, S + Start - 1, Len);
  Result := Buffer;
end;

SQL Data Definition:
DECLARE EXTERNAL FUNCTION Substring
  VARCHAR(255), SMALLINT, SMALLINT
  RETURNS VARCHAR(255)
  ENTRY_POINT "udf_Substring" MODULE_NAME "ourudfs.dll"
```

➤ *Listing 4*

```
Delphi Implementation:
function udf_DateDiff(var Date1, Date2: TIBDateTime): SmallInt; cdecl;
begin
  Result := Abs(Date2.NumDays - Date1.NumDays);
end;

SQL Data Definition:
DECLARE EXTERNAL FUNCTION DateDiff
  DATE, DATE
  RETURNS SMALLINT BY VALUE
  ENTRY_POINT "udf_DateDiff" MODULE_NAME "ourudfs.dll"
```

Reading And Returning DATE Fields

The InterBase DATE fields don't map directly to a Delphi type, so how do we deal with them? Most references on UDFs tell you to access the InterBase API directly and call the routines that translate date/time values into C date/time structures. However, there is a much easier way. The InterBase DATE datatype contains date and time information in 8 bytes. The first four bytes contain the number of days since 17 Nov 1858. The next four bytes contain the number of 0.0001 second intervals since midnight (with there being 864,000,000 of them to a 24 hour day). Date values passed into a UDF can be represented by the following type:

```
TIBDateTime =
  record
    NumDays: LongInt;
    DayFrac: LongInt;
  end;
```

Let's make a simple UDF which returns the number of days between two dates. Listing 4 shows how we might accomplish this. Notice that by using Delphi's Abs function, we don't care which of the two dates is the larger date.

Writing a UDF that returns a date/time is just as easy. We simply compute a TIBDateTime value and return a pointer to it. Remember, only simple numeric datatypes can be returned by value, all other datatypes must be returned by reference. Listing 5 shows the CalcDate function which returns a date that is a given number of days from a given date. Once again, we must use a static global variable to contain the returned date so we can pass back a pointer to it.

Other Datatypes

Table 1 shows the InterBase datatypes and the equivalent

Delphi types for purposes of UDF development. Note that for the exact precision datatypes `NUMERIC` and `DECIMAL`, there is no direct Delphi equivalent. So we must be satisfied with an approximation using the `Double` type.

BLOBs

When BLOB fields are manipulated by a UDF, the process is somewhat different. If a BLOB is to be passed through a parameter of the UDF, you do not receive a pointer to the BLOB data itself. Rather, you get a pointer to a *BLOB Control Structure* as shown in Listing 6.

The BLOB Control Structure contains data about the BLOB as well as function pointers, or callbacks, which allow your UDF to access the BLOB. UDFs cannot access BLOB data directly but must do so through the BLOB callback functions.

The data fields in the BLOB structure should be fairly obvious: you can get the total size of the BLOB in bytes, the number of segments and the size in bytes of the largest BLOB segment. `BlobHandle` is a reference to the BLOB itself, but you cannot access the BLOB directly via this pointer. `BlobHandle` is used with the callback functions to identify the BLOB. If the BLOB is null in the database, then we will get a nil pointer for `BlobHandle`. It's important that we check for this because passing a nil handle into the callback functions will result in an access violation in the server.

The `BlobGetSegment` function is used to import the actual BLOB data into our UDF. We pass in the BLOB handle, a pointer to our internal data buffer, its length, and a variable to hold the number of bytes actually copied into the buffer. `BlobGetSegment` gives us back one BLOB segment at a time and returns `True` as long as there are more BLOB segments to read. So, to get the entire BLOB, we simply keep calling `BlobGetSegment` in a loop until we have no more segments to read.

Listing 7 shows a BLOB UDF which returns a count of the number of occurrences of a particular string within the BLOB. Note that we're assuming we've been given a text BLOB, as there is nothing we can do to check the subtype of the BLOB passed in.

Maintaining UDFs

InterBase does not load the DLL and look for the given function until the function is first used in an SQL statement, so you will not get any error messages from `DECLARE EXTERNAL FUNCTION` if you've given the wrong function or DLL name. If you've made a mistake in binding a UDF to the database, you must first drop the existing declaration using, for example:

```
DROP EXTERNAL FUNCTION Abs
```

before attempting a new `DECLARE EXTERNAL FUNCTION` statement.

If you wish to change the implementation of one or more UDFs within a given DLL, then you must shut down the InterBase server to

➤ *Table 1: SQL datatypes and Delphi equivalents*

InterBase Type	Delphi Equivalent
Char	PChar
Date	(see *Reading And Returning Date Fields* section)
Decimal	Double*
Double Precision	Double
Float	Single
Integer	LongInt
Numeric	Double*
SmallInt	SmallInt
VarChar	PChar
**With possible loss of precision*	

➤ *Listing 5*

```
Delphi Implementation:
implementation
var
  ReturnDateTime: TIBDateTime;
function udf_CalcDate(var BaseDate: TIBDateTime; var Days: SmallInt):
PIBDateTime; cdecl;
begin
  ReturnDateTime := BaseDate;
  Inc(ReturnDateTime.NumDays, Days);
  Result := @ReturnDateTime;
end;

SQL Data Definition:
DECLARE EXTERNAL FUNCTION CalcDate
  DATE, SMALLINT
  RETURNS DATE
  ENTRY_POINT "udf_CalcDate" MODULE_NAME "ourudfs.dll"
```

➤ *Listing 6: The BLOB Control Structure*

```
TBLOBStruct =
  record
    BlobGetSegment: TGetSegmentProc;
    BlobHandle: Pointer;
    NumberOfSegments: LongInt;
    MaxSegmentLength: LongInt;
    TotalSize: LongInt;
    BlobPutSegment: TPutSegmentProc;
  end;
TGetSegmentProc = function(BlobHandle: Pointer; Buffer: Pointer;
  BufLen: SmallInt; var BytesWritten: SmallInt): WordBool; cdecl;
TPutSegmentProc = procedure(BlobHandle: Pointer; Buffer: Pointer;
  BufLen: SmallInt);
```

get it to unload the DLL (if it has been loaded because one or more of the UDFs have been used). Then copy the new DLL into place and re-start the server. If there have been no changes to the UDF interfaces, then everything will work normally. If there have been changes to the DLL function names or parameters, then you'll have to drop the affected UDF definition (`DROP EXTERNAL FUNCTION`) and redefine it (`DECLARE EXTERNAL FUNCTION`) in each database where it will be used.

➤ *Listing 7*

```
function udf_BlobStrCount(SubStr: PChar; var Blob: TBLOBControlStruct):
  SmallInt; cdecl;
var
  Buffer: PChar;
  BufLen: SmallInt;
  S: PChar;
  BytesBack: SmallInt;
  L: Integer;
begin
  Result := 0;
  L := StrLen(Substr);
  with Blob do begin
    if BlobHandle = nil then Exit;     { Check for a null BLOB }
    BufLen := MaxSegmentLength + 1;
    GetMem(Buffer, BufLen);
    try
      { Get each BLOB segment }
      while BlobGetSegment(BlobHandle, Buffer, BufLen, BytesBack) do begin
        { Count the occurrences of the substring }
        S := StrPos(Buffer, Substr);
        while S <> nil do begin
          Inc(Result);
          S := StrPos(S + L, Substr);
        end;
      end;
    finally
      FreeMem(Buffer, BufLen);
    end;
  end;
end;
```

Conclusion

User-Defined Functions allow you to create very powerful extensions to InterBase's SQL language. Just remember to be careful about when parameters are passed by reference and when they are passed by value and you shouldn't have too much trouble creating your own huge library of SQL functions.

Steve Troxell is a Senior Software Engineer with TurboPower Software. He can be reached by email at stevet@turbopower.com or on CompuServe at 74071,2207