Writing RDBMS function / procedures for ODBC

It is possible to call functions or stored procedures in your database through the ODBC (Open DataBase Connectivity) layer. All databases support multiple input and output parameters, although the way to write your functions and / or stored procedures may vary to make them successfully callable. The "SQLComponents" library has a few tricks up its sleeve to make the living easier with the interfacing with functions and procedures.

Names

Some RDBMS'es have only functions, some have only stored-procedures and some have both. To ease the reading of this document, we will refer to both stored functions and stored procedures as stored-routines, or simply "routines".

Calling a stored routine

Each database has a different syntax to call a routine. This can be as diverse as a simple variant of the "SELECT" command, or an elaborate syntax that states that we are calling some routine. The following are seen 'in the wild':

- SELECT FROM routine(par1,par2);
- SELECT routine(par1,par2) FROM <table with one row>;
- EXECUTE PROCEDURE routine(par1,par2)
- EXEC routine par1, par2
- CALL routine(par1,par2)

As you can see from the list above: not much standardization is done in this respect. Every RDBMS does as it sees fit. Luckily, here ODBC comes to the rescue. The ODBC API standardizes the calling of a stored routine to the following form of syntax:

- `{ ? = CALL routine (?,?) }`

In this ODBC syntax the question marks stand for the parameter bindings. It is then the intention that the ODBC driver of a RDBMS vendor will translate the part between the '{' and '}' escape markers to the native syntax of that RDBMS. The full Backus-Naur form of this syntax is:

- `{ [?=] CALL <routinename> [( [? [,…]] )] }`

Stated in simple words, this translates to the following rules:

1) If the routine has a return parameter, you can bind that parameter with the "?=" part, before the word "CALL";
2) You must use the command "CALL" explicitly;
3) You then supply the name of the routine (stored function or stored procedure);
4) If no input parameters are involved, the parenthesis can be empty "()" or absent;
5) You then supply a list of parameters binding question marks, separated by commas;

One of the greatest benefits of this syntax is that you can use it for stored functions as well as for stored procedures alike. Even if your RDBMS does not support to call them in the same way!

Well. That is the good news. The bad news is that not all database vendors support this interface in there ODBC drivers. Some support it, some support it half the way and some don't. That makes it not easy to write portable programs that will work on more than just one database.

But luckily we have the SQLComponents library to solve all that!

SQLComponents and SQLQuery::DoSQLCall

To ease the pain of supplying all correct SQL, parameters and doing the parameter binding, the SQL components library offer a handy interface by means of the “DoSQLCall” interface of the SQLQuery class. This takes the burden of binding the parameters and rethinking the needed SQL for calling the database. And it also resolves the problems with those databases that do not support the “call” interface as prescribed by the ODB API.

Here is an example of how to work with the DoSQLCall interface.

```
SQLDatabase dbs;
if(dbs.Open(”mydsn”,”user”,”thepassword”))
{
    SQLQuery query(dbs);

    query.SetParameter(0,bcd(),SQL_PARAM_OUTPUT);
    query.SetParameter(1,bcd(45.667));
    var* result = query.DoSQLCall(”schema”,”calculation”);

    printf(”The result of the calculation is: %s\n”,result.GetAsChar());
}
```

In this simple example we open a database, setup a call with an input and an output parameter and then call the routine with the name “calculation” in some schema. Whether this routine is a stored procedure or a stored function and what the vendor type of the database is, is now irrelevant!

In the paragraphs below, some of the intricacies and details of this interface are described in detail.

Detail 1: The 0th parameter

Normally the ODBC interface is 1-based. Meaning that result columns and parameters all begin there numbering on 1. The zero parameter of the DoSQLCall() is an exception to that rule. It signifies that this special parameter is the return value (left side) of the routine. It also means that if you do not provide the interface with the zero parameter, internally the “[?=]” part of the syntax is not generated, and if the database routine has a return parameter, it will be ignored.

Detail 2: Reserving space for output parameters

Calling a routine has no ‘fetch-the-result’ phase when executing, so it does not resemble the prepare-execute-fetch cycle of a normal query. The downside of this very quick execution is that we cannot request the what the results of the routine will be, like we do in the ‘BindColumn’ phase of a “SELECT” query. We must have the parameters right beforehand. And of the right size!! If the parameter can return e.g. 600 characters, we must allocate a buffer in the parameter of at least 600 characters beforehand. Luckily we have help from the SQLVariant class to do just that, as is shown in the next example:

```
if(dbs.Open(”mydsn”,”user”,”thepassword”))
{
    SQLQuery query(dbs);

    SQLVariant* param(SQL_C_CHAR,600);
    query.SetParameter(0,param,SQL_PARAM_OUTPUT);
    var* result = query.DoSQLCall(”schema”,”item_description”,5629);

    printf(”The description of item [%d] = %s\n”,5629,result->GetAsChar());
}
```

In general the SQL_CHAR, SQL_BINARY and SQL_DECIMAL datatypes are those that need a pre-allocated space before we can go call any stored routines. But the driver will notify you if any parameter does not meet up to the specs needed.

Detail 3: Shorthand forms of DoSQLCall()

There are a few shorthand forms of the "DoSQLCall" interface for much recurring variants of calling this interface. Standard the method has a schema-name and a routine-name parameter that you must fill in. A third boolean parameter exist for suppressing the catching of the return parameter (the 0-th parameter). For the most used forms (currently 1 input parameter in CString or integer form) exist with a return parameter of datatype integer. So if you want to call a routine with 1 character string as input and a integer as output, you do not need to do any plumbing. You can write straight away the following:

```
- SQLVariant* result = query.DoSQLCall("schema","routinename","my input string");
```

Which is quite handy in those cases.

Detail 4: Multiple output parameters

Much harder is the case where we want to return more than just one (1) output parameter. All RDBMS'es support this feature in one form or the other. The ODBC "{ CALL …}" interface was designed to support that feature, but it is not without brambles and tangles. In general the SQLQuery class has been designed to support all database vendors in this feature, but not all database interfaces to stored routines are the same. Most of these interfaces evolved long before the ISO 9075 standard for SQL prescribed anything about it. So there are a lot of differences that are dealt with in the next chapter.

What you must realize in general is that the "SetParameter" interface third parameter guides the way that the ODBC driver – and so the database – will handle our parameter upon completion of the call. These are the options:

3th parameter of SetParameter	Handled by the ODBC driver
SQL_INPUT	Parameter is an input parameter for the routine
SQL_OUTPUT	Parameter is an output parameter for the routine. But some databases will treat it as SQL_INPUT_OUTPUT!
SQL_INPUT_OUTPUT	Parameter is used as an input parameter and will return an answer.
SQL_RESULT	**DO NOT USE**! Currently in ODBC 3.x all drivers use SQL_OUTPUT for the result parameter.

Database dependent details

Calling a stored routine is all very well, but in order to be able to call them, a few quirks and strange implementation dependent details exist. Before we can successfully call the routines through the ODBC driver, we must be aware of the existing details. This chapter give you just those details, ordered per database engine in increasing divergence from the standard.

For each database vendor the “multinout” procedure from the unit test is given.

MS-SQL Server

As you might expect, Microsoft adheres quite well to the ODBC standard, as this standard was originally concocted by this company. Still the MS-SQL Server database, even in its latest incarnation, has few strange things.

You can write stored functions and stored procedures freely. Even return any datatype and as much output parameters as you like, and they all are handled by the “{ CALL … }” interface in the native driver (by the way: the native driver *is* a ODBC driver). But the one restriction is that a stored procedure cannot return anything other than a status integer. No other datatype can be returned by the “RETURN” statement within the stored procedure. Any other value that you try to return will be converted to an integer. This restriction does not hold for stored functions. The functions can return any known datatype.

If you want define an output parameter, you need to put the keyword “OUTPUT” or “OUT” (shorthand) after the definition of the parameter. But be aware that every output parameter is also an input parameter! You can also give it an initial value upon calling the routine.

Lastly, the driver needs a kind of ‘fetch’ for the output parameters. This is done by calling the ::SQLMoreResults() interface of the ODBC driver. This detail is automatically handled by the SQLQuery class for you, but it could change per version of the database. So just keep it in mind.

```
CREATE procedure dbo.multinout(@p text  VARCHAR(200)
                              ,@v_numm  DECIMAL(18,2) OUTPUT
                              ,@p_total VARCHAR(200)  OUTPUT)
AS
  DECLARE @v_plus VARCHAR(200);
  DECLARE @v_dyn  varchar(500);
  SET  @v plus = @p text + @p text;
  SET @p_total = substring(@v_plus,1,100);
  INSERT INTO dbo.testrecord (name) VALUES (@p_total);
  set @v_numm = 77.88;
  return 1;
RETURN
```

Oracle

The handling of stored functions and stored procedures in Oracle is rather standard. The only strange thing is that returned strings in parameters are not delimited. If you define a parameter to have e.g. 500 characters and the routine returns you effectively say 88 characters, you will end up with 88 characters of result with a whopping 412 spaces attached to them.

The SQLQuery class handles the extra spaces by stripping them all away. That would be the normal output mode. But this means that you cannot return a string from a routine with say one or two trailing spaces to them. Not a very severe limitation, but it can become very annoying in production environments.

```
CREATE OR REPLACE FUNCTION multinout(p_text IN VARCHAR2,p_total IN OUT VARCHAR)
RETURN NUMBER IS
  v_plus VARCHAR2(4000);
BEGIN
  v_plus := p_text || p_text;
  INSERT INTO testrecord (name) VALUES (v_plus);
  INSERT INTO testrecord (name) VALUES (p_total);
  COMMIT;
  p_total := SubStr(v_plus,1,100);
  RETURN 88.77;
END multinout;
```

IBM-Informix

The Informix handling of stored routines is rather standard. But be aware that the Informix database also makes it possible to return multiple values with the “RETURN” statement AND multiple parameters within the parameter list. And the multiple return values from the RETURN statement do **NOT** mix with the ODBC “{ CALL … }” interface. If you try to do just that, the result will be that you get an error stating that the routine does not exist. This may

To be able to have multiple output parameters you have to write the parameters on the parameter list pre-pended with the “OUT” keyword. Something like “OUT <parameter> <datatype>”.

```
CREATE FUNCTION informix.multinout(p_string VARCHAR(200),OUT p_text VARCHAR(200))
   RETURNS DECIMAL(18,2);
DEFINE extra VARCHAR(200);
  BEGIN
     LET extra = p_string || p_string;
     INSERT INTO test record(name) VALUES(extra);
  END
  LET p_text = substr(extra,1,100);
  RETURN 77.88;
END FUNCTION;
```

PostgreSQL

The PostgreSQL handling of the “{ CALL … }” interface is limited to input parameters. As long as we supply input parameters, everything works fine. But the handling of output parameters, even of the result parameter of a stored function is non-existent. That’s why the SQLQuery class hands over the handling of the call to the SQLInfoPostgreSQL class. That’s where a “SELECT FROM <routine>” SQL statement is being created and fired against the database. The resulting result-set of that command then contains the returning parameters value.

A second strangeness occurs as the result parameter can only be of type SQL_CHAR. Any results will be converted to that datatype. Stronger: if we try to return two or more return parameters, all results will be pasted together in one long string, as in: “(result1,result2,…)”. That’s why we can return string values, but if we return more than one, no return value can begin with a ‘(‘ character.

The library does the job of separating the various returned results and splitting the answers into the provided output parameters, so that it looks like that the result is exactly the same as for the other database types.

What you must be aware of is, that you must declare a compound return type for all the return values, or declare a variable of type “record” or “table”. You then specify the RETURN command once with the compound variable, effectively returning all output parameters ‘in one go’.

```
create or replace function public.multinout2(x varchar(200))
returns table (one decimal(18,2),two varchar(200)) as $$
declare
   y varchar(200);
begin
   y := x || x;
   insert into public.test_record(name) values (y);
   return query
   select 77.88,y;
end;
$$ language plpgsql;
```

Firebird

The Firebird database diverges the most from the "{ CALL … }" interface, as it is totally absent! Instead you are forced to use one of two interfaces in order to call a stored routine. For a stored function you need to this:

- `SELECT funcname(param1,param2) FROM rdb$database;`

And for a stored procedure you do exactly the same as for PostgreSQL, the non-standard:

- `SELECT FROM procname(param1,param2);`

Where the "rdb$database" stands for the catalog table with one record (being the location of the database). It is the Firebirds implementation of what is the "DUAL" interface in Oracle.

What you must remember is that you write your stored function ending on a RETURN statement, but the stored procedure must end with a "SELECT" statement of the result parameters, followed by the "SUSPEND" statement. This very a-typical way of programming stored procedures makes it possible to write stored procedures that result into a real result-set. Making the stored procedure a kind of virtual table. For the "DoSQLCall" interface to work, you must write a SELECT statement that results into a one (1) row result set, or assign values to all return parameters. Even in the case that you do NOT write a final select, you must still issue the "SUSPEND" statement, to return the values in the parameters.

```
create or alter procedure multinout(x varchar(200))
returns (y decimal(18,2),z varchar(200))
as
  begin
    z = :x || :x;
    insert into test_record (name) values (:z);
    y = 77.88;
    suspend;
  end;
```

A final word on database independence

After we've seen all the database specific diversion of the standard it becomes clear: some planning is needed to write stored functions and procedures that can be truly portable. Some have argued that you should not let constraints from other database vendors lead to limiting what you can / will do in another database, but still… A little sacrifice will lead to truly portable procedures. Seen from a distance we come to the following set of rules:

1) If we do need all datatypes as the return value, we definitely should use stored functions. Reason: In the case of procedures, some engines just return char or integer datatypes;
2) If we do need more than one return parameter, we should plan to do that NOT with the return parameter (the 0-th parameter) but with the output parameters trailing after the input parameters. And use the return parameter as a status parameter like MS-SQL does;
3) We cannot expect string return parameters to return trailing spaces. Reason: Oracle and PostgreSQL do not support that;

But for the rest, we're good to go… ☺