Craig S. Mullins & Associates, Inc.
Database Performance Management

Fall 1994

The open enterprise survival guide

Pursuing the Procedural SQL

The Good, the Bad, and the Inevitable

By Craig S. Mullins

One of the greatest benefits derived from moving to a Relational Database Management System (RDBMS) is the ability to operate on sets of data with a single line of code. Consider, for example, Structured Query Language (SQL) — the predominant RDBMS query language. Multiple rows can be retrieved, modified, or removed in one fell swoop using a single SQL statement! However, this very capability also limits SQL's functionality.

Thankfully, this limitation is quickly becoming a thing of the past. Several major RDBMS vendors support a procedural dialect of SQL that adds looping, branching, and flow of control statements. For example, Sybase SQL Server provides this support in Transact-SQL. Procedural SQL has major implications for database design.

Sample procedural SQL

Procedural SQL will look familiar to anyone who has ever written any type of SQL or coded using any type of programming language. Typically, procedural SQL dialects contain constructs to support looping (while), exiting (return), branching (goto), conditional processing (if...then...else), blocking (begin...end), and variable definition and usage.

Consider the sample Transact-SQL code shown in Figure 1. This piece of logic will retrieve rows from the titles table, format the output, and write the output. It utilizes looping, flow control, exiting, blocking, and cursors.

Figure 1: Procedural Transact-SQL

```
create proc book_list
as
declare title_cursor cursor for
        select title_id, title, pub_id
        from titles
declare      @tid char(6),@title varchar(80),@pubid char(4), @outline varchar(92)

open title_cursor
fetch title_cursor into @tid, @title, @pubid

while @@sqlstatus !=2
begin
        if @@sqlstatus = 1
        begin
           print "Select failed"
           return
        end
        else
        begin
           select @outline = @tid + " " + @title + " " + @pubid
           print@outline
           fetch title_cursor into @tid, @title, @pubid
        end
end

close title_cursor
deallocate cursor title_cursor
return
```

The benefits of procedural SQL

The most useful procedural extension to SQL is the addition of procedural flow control statements. Flow control within a procedural SQL is handled by typical programming constructs that can be mixed with standard SQL statements. These typical constructs enable programmers to:

- embed SQL statements within a loop
- group SQL statements together into executable blocks
- test for specific conditions and execute one set of SQL statements when the condition is true, another set when the condition is false (if...else)
- suspend execution until a predefined condition occurs or a preset amount of time expires
- perform unconditional branches to other areas of the procedural code

The addition of procedural commands to SQL provides a more flexible environment for application developers. Major components of an application can frequently be delivered using only SQL. Stored procedures and complex triggers can be coded using procedural SQL, thereby reducing the lines of host language code (COBOL, C, PowerBuilder, etc.) required.

Additionally, when triggers and stored procedures can be written employing only SQL, more developers will be inclined to use these features. Some RDBMS products require triggers and/or stored procedures to be written in a traditional third generation programming language like COBOL or C. This scares off many potential developers. Most DBAs I know avoid programming like the plague. But since writing SQL isn’t really programming, they might be willing.

In addition to SQL-only triggers and stored procedures, procedural SQL extensions also permit more complicated business requirements to be coded using only SQL. For example, ANSI SQL provides no mechanism to examine each row of a result set during processing. Using cursors and looping, a procedural SQL can accomplish this quite handily.

Yes, you read that correctly! It is possible to implement cursors in procedural SQL. Sybase SQL Server 10 does this quite well. Cursor support enables developers to code robust applications, using nothing but the procedural SQL.

The drawbacks of procedural SQL

The greatest drawback of procedural SQL is that it is not currently part of the ANSI SQL standard, although it is being developed for SQL3. The result is that each DBMS vendor supports a different flavor of procedural SQL. If your shop has standardized on one particular DBMS or does not need to scale applications across multiple platforms, then this may not be a problem. But how many shops does this actually describe? Not many, I'd guess!

The bottom line is that portability will suffer when applications are coded using non-standard extensions — like procedural SQL. It is a non-trivial task to re-code applications that were designed to use stored procedures and triggers written using procedural SQL constructs. This is exactly what must be done, if an application needs to be ported to a platform that utilizes a DBMS that does not support procedural SQL. Consider, for example, the steps necessary to move an SQL Server application, written in Transact-SQL using triggers and stored procedures, to a DB2 platform. I get a headache just thinking about it!

Another potential drawback is the risk of performance degradation. The first time an SQL Server stored procedure is executed, for example, it is optimized. The optimized form of the procedure is stored in the procedure cache and will be re-executed for subsequent users. But the procedure was optimized for the particular data request that was issued for the first execution, and it is very likely that future executions of the procedure are for different amounts and types of data. If the logic were instead embedded into an application program and compiled statically, the performance would be optimized for all ranges of local variables. Of course, SQL Server provides an option to optimize (the **with recompile** option), but then dynamic SQL is always used — and this can cause different types of performance problems.

One solution is to provide a form of static SQL for stored procedures that is not optimized for a particular type of request.

Other performance drawbacks can occur when using procedural SQL if the developer is not careful. For example, careless cursor coding can cause severe performance problems. The classic example is the program that codes multiple cursors to perform a programmatic join instead of simply coding a single multi-table join. But this can happen just as easily when cursors are used inside a host language. The problem is more inherent to application design that it is to procedural SQL.

Finally, even procedural SQL dialects are not computationally complete. Most dialects of procedural SQL lack programming constructs to control users' screens and mechanisms for data input/output (other than to relational tables).

Synopsis
Whether you favor procedural SQL extensions is really a moot point, because procedural SQL is here to stay. It simply offers too many benefits to be ignored. And, if you seriously doubt the veracity of that

assessment, re-read this article in a couple of years — by then the ANSI standard will include procedural SQL statements