USER HELP

Using the ORACLE Call Interface Effectively

David McGoveran is co-founder of DataBase Associates, a strategic consulting services firm in Morgan Hill, California. He is also President of Alternative Technologies, a development services firm based in Santa Cruz, California.

With any 3rd-generation language (3GL) software project that involves relational database management, developers must not only design a supporting database schema and load it with data, but also implement a number of access routines in order to enable application programs to manipulate the data in the database.

These routines are usually implementations of a specific application function, rather than reusable tools. Common practice among developers is to implement these routines by writing them in a 3GL and embedding the SQL statements in the source file. To better handle these foreign statements, the source files are "pre-processed" prior to compilation. This is the "embedded SQL" approach of Oracle's Pro*C.

An alternative approach is programming with a function call interface, such as the ORACLE Call Interface (OCI). While many developers consider this approach heresy, I personally have found it to be the most effective. Vendor-supplied function libraries are familiar to most programmers, whereas the "embedded SQL" approach with precompilation takes some getting used to. The OCI offers a means to alleviate a number of other problems found with the embedded approach.

Although I will use 3GL projects as the prototypes in this article, many of the same comments are relevant to 4th-generation language (4GL) implementations as well. Most 4GLs (including Oracle's SQL*Forms) require some kind of "compilation" step. However, access to SQL via a 4GL usually looks like 3GL-written code (albeit at a higher level of utility) using dynamic embedded SQL. This approach does not alleviate certain problems, which I discuss below.

Throughout this article, I will use the term 'command' or 'SQL command' to stand for a sequence of SQL statements. For example, PL/SQL blocks certainly qualify as SQL commands. Although this article discusses SQL, the discussion applies equally to any DML (data manipulation language) or DDL (data definition language) for relational DBMSs.

Nature of the Problem

One of the objectives of embedded SQL is to provide for portability. This is a fine objective, especially if you are using more than one DBMS product throughout a company, or if you are a software vendor whose product must work with several relational DBMSs. Unfortunately, portability is difficult to attain. Even if the DBMSs use SQL statements that are compatible *syntactically*, the chances are good that *semantics* (results and transaction behavior) will differ.

In my experience, the ability to reuse and maintain code effectively is more important than portability to end users. However, embedded SQL makes reusable

and maintainable code difficult to attain, especially on large applications.

Even with dynamic embedded SQL using the PREPARE construct or its equivalent, there are certain limitations that restrict the flexibility of the resulting modules. In most implementations, PREPARE takes no more than one SQL statement, so 3rd-generation language code is written to support each occurrence.

There are a number of problems with the embedded SQL approach:

- *Since each of the supporting 3GL routines has a specific rigid function, they tend to proliferate.*

If the SQL pre-processor generates "compiled" SQL, it may use the program name to catalog and invoke the compiled SQL.

This defeats the modularity of the 3GL (or 4GL) code, making it very difficult—or even impossible—to create shared libraries of database access routines—with profound effects on the structure of applications, the time and resources required to design and develop them, and their uniformity and efficiency. In fact, I have found that it promotes bottom-up replacement of record-at-a-time file I/O with relational data access routines, rather than top-down design of relational applications. The use of positioned cursors especially promotes such anti-relational design.

- *The pre-compilation phase is cumbersome, adding a development phase that is not always compatible with software management tools.*

For example, symbolic debuggers do not show the original embedded SQL source code, but rather its processed form, making it difficult to track down compilation and runtime errors.

- *The embedded relational database language is "mixed" with the 3GL so that source code control is difficult. The difference between the procedural and non-procedural components of the code creates what is sometimes referred to as an "impedance mismatch."*

Coordination of function between the 3GL and SQL is difficult to manage when either language is changed. Even if programs are treated as database-dependent objects, this information is not available to 3GL source code management tools.

Of course, various CASE products alleviate this problem, but there is no general solution. In a UNIX environment, there is no reasonable way to manage this mixture short of maintaining all DDL and DML under RCS or SCCS, and similarly, using the make compilation utility to keep track of database-dependent objects, as well as 3GL code.

Another aspect of this problem is that the need to load/unload data structures such as trees and linked-lists (which are common in C programs) causes the module to be specialized. It also is relatively inefficient if host variables cannot be referenced as pointers in a SELECT...INTO.

- *A programmer must know not only the 3GL, but also the relational database language and the characteristics of the pre-processor.*

If the programmer does not understand how to write both languages, the interaction between the two is likely to reflect the weaker understanding. This is especially true in cases in which the 3GL programmer must explicitly assert transaction or lock control. If the programmer's knowledge of the 3GL is weak, the code interspersed between SQL statements will be inefficient, and locks will be held for unnecessary periods. If the programmer's SQL skills are weak, unnecessary calls will be made to the database, resulting in lower system throughput. Furthermore, finding and retaining programmers with both SQL and 3GL skills can be difficult and costly.

- *The programmer will probably have to obtain help in optimizing the SQL statements and then successfully translating the statements into appropriate embedded statements in the context of the 3GL code.*

This process requires a unique skill, since the syntax of the SQL dialect when embedded in a 3GL is quite different from the syntax when SQL is used interactively with, for example, SQL*Plus. With SQL, the use

of CURSORs, FETCH, etc., may lead to translation problems. It is not sufficient to have database personnel optimize the individual SQL statements, since it is the *work unit* as implemented in a sequence of SQL statements that must be efficient.

A proficient SQL coder, knowledgeable about the schema and the product being used, may achieve the desired function more efficiently using a different sequence of SQL statements than an application programmer would use. Since application programmers and database personnel are often in separate workgroups with different skill sets, the coupling between the two kinds of code makes task division quite difficult when managing development, deployment, and maintenance.

- *Source code must be recompiled and the entire system relinked if any changes are made to the embedded SQL.*

Even with dynamic embedded SQL, there are limitations to how much a SQL statement can be changed before the supporting 3GL must be changed. Where the rigidity sets in depends upon the vendor implementation. While it is possible with dynamic embedded SQL to write reasonably general-purpose modules, the result is not as clean as purely 3GL modules would be. Explanations include the availability of a cursor context across modules, the subtle differences in transaction management between applications, and problems with error handling (e.g., can a general purpose error handling routine be used as the GOTO action in a WHENEVER clause?).

- *It is costly to move embedded SQL code from one relational DBMS product to another or between versions of the same product.*

Many programmers develop efficient code at the expense of portability. The features that differentiate products also tend to make applications non-portable. Developing an application around the least common denominator—the common subset—severely restricts both the creativity of the application designer and the benefits of product evaluation and selection. The product then cannot perform to the full potential intended by its designers.

- *The source code is "mixed" with the database schema.*

This last item is by far the most costly. Large applications will consist of many "database access routines." When the database administrator decides to modify the relational database schema, each of these routines will have to be examined to see if they now access some modified data element in an inappropriate manner. If a table is normalized into two tables, consider how this affects each SELECT...FOR UPDATE and each UPDATE...WHERE CURRENT OF.

Short of a full data dictionary—such as that envisioned with IRDS and partially supplied with SQL*Dictionary—this impossibly complex task leads to redundancy between the development and database management environments. Even with such a repository, making the necessary changes to the source code can consume many man-hours.

If the cost of this maintenance is high enough, many organizations will forbid changes to the schema in order to avoid the cost in time, expertise, or potential disruption of the business. This coupling between application code and database schema effectively removes one of the primary benefits of a relational database—its flexibility.

Generalizing Database Access

The perspective promoted here is that the application has responsibility for:

- Determining what data is sent to the database
- Determining what to do with data returned from the database
- Specifying, in a functional sense only, what is to be done by the database
- *Nothing else* pertaining to the database

The application code should not be coupled to SQL specifics, nor to the database design. At the same time, it is understood that existing applications must be migrated to the database environment; hence, the ability to use SQL-specific code need not be absolutely precluded.

Coding unique routines for each application SQL block is superfluous. Indeed, failing to isolate code from data often leads to maintenance inefficiencies. Vendors that follow the ANSI committee approach support various techniques using embedded SQL. SQL is embedded explicitly in the code, and a preprocessor (such as the Oracle precompiler) is used to convert the lines (sometimes preceded by a special symbol) into function calls to the database.

Oracle's precompiler performs this process, substituting the appropriate function calls and data declarations. These function calls are built on top of the OCI. Some vendors, including Oracle, allow the embedding of certain statements by reference so that they can be altered during the run of the application. This is called dynamic SQL.

Fortunately, Oracle Corporation and a few other relational DBMS vendors allow the programmer to code the function calls directly to the database. The OCI is one example of a documented runtime function call interface. Documented in the Pro*C manuals, it is a powerful set of functions providing for logon/logoff, cursor open and close, the parsing and execution of arbitrary SQL statements, and the dynamic description and fetching of SELECT results. More recently, powerful "array fetch" and "transaction execute" functions have been added.

Direct manipulation of results data, update parameters, and 3GL data structures can be very powerful and efficient with OCI. 3GL programmers can use structures such as trees and linked-lists in a manner familiar to them. Functionality is not lost and may even be improved over embedded SQL. This method requires no additional coding—a library of high-level functions using OCI can actually eliminate tedious and proliferating EXEC SQL statements, reducing the amount of detail to which a programmer must attend.

It is a common error for the programmer to hardcode the SQL statement as an argument to the database vendor supplied function. These errors can be eliminated by the development of a flexible development library using OCI. There is no need to recode the vendor-supplied function calls for each application if developers pay proper attention to the usual rules of cohesion within modules and loose coupling between modules. Data, including SQL statements, should never be hardcoded.

Data coupling of the application code to the relational DBMS is an error that can occur whether a developer uses a function call interface or embedded SQL. Data coupling (or binding) can occur at preprocessor time, compile time, link time, or runtime, with the latter being the only truly flexible method.

Even if the data is relatively isolated by creating a macro-defined symbol that the 3GL pre-processor (e.g.,

Pro*C) will expand at compile time, the code becomes strongly coupled to the eccentricities of the DML (including bugs) and to the database design.

This latter error is severe; changes to the database design invariably lead to modifications of the application code. If the cost of the application modifications required to implement a change in the database design is great enough, the database design becomes fixed, eliminating one of the key benefits of a relational database: its mutability.

If the database vendor supports stored blocks, procedures or scripts that can be stored (perhaps in the database) and invoked by name, SQL can be removed from the code altogether. For example, PL/SQL can be stored in a client file or database and automatically loaded for execution at program initialization, or as each procedure is referenced.

However, two problems remain. First, the code supporting the execution of the SQL command is sensitive to the particular SQL statements within the SQL command. Second, the linkage between code data structures and the data structures SQL requires to interface to the vendor database is defined within the code and remains to couple the database schema to the application, and vice-versa. Both problems can be addressed using dynamic allocation of variables, access to the SQLCA structure, and the OCI functions ODEFIN, ODSC, and OBNDRN.

Even with such requirements for careful design, the use of external SQL procedures is justified by the resulting efficiency and database schema independence. When routines for processing SQL are written once, they can be highly optimized for performance. The application also obtains database schema independence, since all SQL code is localized and may be managed within the database.

Database schema independence helps IS managers assign SQL experts to SQL and 3GL experts to the use of 3GL functions. When a SQL command or procedure needs to be changed to reflect a change in the database schema, no changes to 3GL code need be made as long as the inputs and outputs remain unaltered. External SQL procedures can also provide a measure of relational database support for object-oriented programming techniques, since they can be treated as methods for objects.

Conclusions

The most important tool for developing a relational application is a sharable library for processing SQL requests transparently. The tool should be accessible from a number of 3GL languages and should not be bound to any particular database or sequence of SQL statements. The OCI is ideally suited to achieving such objectives. In fact, using OCI has been my preferred method of developing ORACLE applications. ■