Flexible Relational Database Applications

In any software project involving relational database management and third-generation language (3GL) programs, access routines must be developed to allow application programs to manipulate database data. It is common practice to build these routines by embedding SQL statements in 3GL application code. This *embedded SQL* approach has limitations which restrict the flexibility of these access routines. This article discusses these limitations and proposes a development approach that supports the creation of shareable and flexible database access routines.

by David McGoveran

The Problem

There are several problems associated with implementing database access routines using embedded SQL in application 3GL source code:

1. *Each 3GL routine performs a specific rigid application function, and routines, therefore, tend to proliferate.*

3GL source modules containing embedded SQL are usually passed through an SQL preprocessor prior to compilation. Some SQL preprocessors generate *compiled* SQL which is stored in the DBMS under the name of the 3GL source program. This defeats the modularity of the 3GL code, making it difficult or impossible to create shared libraries of database access routines. (Note: although this article discusses the use of embedded SQL in 3GL applications, many of the same comments apply to 4GL applications.) Other preprocessors create compiled SQL, but do not use the source module name for controlling the SQL module names. DB2, for example, uses this approach. In DB2, a compiled SQL module is related to the name of the final linked application, not the original source program name.

The inability to create shareable database access routines affects the structure of applications, the time and resources required to design and develop them, and their uniformity and efficiency. It promotes bottom-up replacement of record-at-a-time file I/O operations with relational data access routines, rather than top-down design of relational applications.

2. *The preprocessor phase is cumbersome, adding a development step which is not always compatible with software management tools.*

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

3. *The embedded relational database language is "mixed" with the third-generation language so that source code control is difficult — this is sometimes referred to as an "impedance mismatch."*

It is difficult to modify the 3GL source code when the SQL is changed, and vice versa. Even if programs are treated as database dependent objects, this information is not available to 3GL source code management tools. For example, in a Unix environment, there is no reasonable way to manage this mixture, short of maintaining all data definition language (DDL) and data manipulation language (DML) statements under RCS or SCCS, and using the *make* compilation utility to keep track of dependent database objects as well as 3GL code.

4. *A programmer must know not only the third-generation language, but also the relational database language and the characteristics of the preprocessor.*

If the programmer does not understand how to write both languages, this will be reflected in the interaction between them. This is particularly true where the 3GL programmer must explicitly assert transaction control. If the 3GL programming is weak, the code between SQL statements will be less than efficient, and could allow locks to be held during transaction processing for unnecessary periods of time. If the SQL programming is weak, unnecessary calls could be made to the database, resulting in reduced system throughput. Furthermore, finding and retaining individuals with both sets of skills can be difficult and costly.

5. *The programmer will have to obtain help in optimizing the SQL, and will then have to translate the optimized SQL into the appropriate syntax for embedding in third-generation language code.*

This requires a unique skill, since the SQL syntax when embedded in a third generation language may be quite different from the syntax when the SQL is used interactively. With embedded SQL, the use of CURSORs, FETCH statements, etc. may lead to translation problems from the interactive version. It is not sufficient to have database personnel optimize the individual SQL statements since it is the transaction implemented as a sequence of SQL statements which must be efficient.

A proficient SQL coder, knowledgeable about the database schema and the product being used, may be able to achieve the desired function more efficiently with a different sequence of SQL statements from that which the applications programmer would use. Since the applications programmer and database personnel are often in separate work groups with different skill sets, the coupling between the two kinds of code makes task division more difficult when managing application development, deployment, and maintenance.

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

Even with dynamic embedded SQL, there are limitations to how much an SQL statement can be changed before the underlying 3GL code must be changed.

7. *Source code is costly to move from one relational database management system product to another.*

While the ANSI/ISO SQL standardization efforts certainly help with portability, efficient code is always developed at the expense of portability. Those features which differentiate products also make applications non-portable. Developing an application around the least common denominator — the common subset — severely restricts the creativity of the applications designer, and often removes the benefits identified during DBMS product evaluation and selection. The product will not, therefore, perform to its full potential.

8. *The source code is dependent on 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. Short of a full data dictionary such as that envisioned with IRDS, this is an impossibly complex job, and leads to redundancy between the development and database management environments. Even with such a repository, making the necessary changes to the source code can be very time consuming.

If the cost of this maintenance is high, changes to the schema will be forbidden in order to avoid that cost, whether it be 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 — flexibility.

Generalizing Database Access

Data, including SQL statements, should never be hardcoded in an application program. The application should have responsibility for

- determining what data is sent to the database,
- deciding what to do with data returned from the database,
- specifying in a functional sense only what is to be done by the database, and
- *nothing else* to do with the database.

The application code should not be coupled to SQL specifics or to the database design. The coding of unique routines for each application SQL command is superfluous. Indeed, failing to isolate code from data leads to maintenance inefficiencies.

DBMS products conforming to the SQL ANSI/ISO standard support the use of embedded SQL. Here, SQL is embedded explicitly in the code, and a preprocessor (sometimes called a precompiler) is used to convert the SQL (sometimes preceded by a special symbol) into *function calls* to the database. Several vendors 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.

Some vendors allow the programmer to code the function calls to the database directly — this is called a *runtime function call interface*. It is a common error for the programmer to hardcode the SQL statement as an argument to the database vendor supplied function call. These errors can be eliminated by the creation of a flexible development library — there is no need to re-code the vendor supplied function calls for each application.

Strong data coupling of the application code to the relational DBMS is an error that can occur whether a function call interface or embedded SQL is used, but is hard to avoid with embedded SQL. The process by which data coupling (or binding) is accomplished can occur at preprocessor time, compile time, link time, or execution time — the latter being the only truly flexible method. Even if the data is relatively isolated by creating a macro-defined symbol which the 3GL preprocessor will expand at compile time, the code becomes strongly coupled to both the eccentricities of the DML (including bugs) and the database design.

If the DBMS vendor supports SQL stored procedures (or commands or scripts) that can be stored in the database and invoked by name (*call-by-name* syntax), the SQL might be removed from the code altogether. Two problems remain to be resolved, however. First, the code supporting the execution of the SQL stored procedure is sensitive to the particular SQL statements within the procedure. Second, the linkage between the application code data structures and the data structures used by the stored procedure SQL to interface to the vendor DBMS, is defined within the application code itself, i.e., these structures couple the database schema to the application and vice versa. Nonetheless, the use of SQL stored procedures can be justified from the standpoint of efficiency and improved database schema independence. They also provide a measure of relational database support for object oriented programming techniques.

The most important tool that can be developed for a relational application is a shareable library (or server) of general purpose database access routines which eliminate the problems outlined above, including the two that remain when using stored procedures. The tool should be accessible from a number of 3GL languages. Such a tool is referred to as a *database access manager.*

Database Access Manager Characteristics

To those unfamiliar with the benefits of relational databases (and non-procedural programming languages in general), it might appear that a set of callable general purpose database access routines would be too low-level for direct use by applications. However, the fact is that the specific DML commands issued (or requested) by the application can serve to differentiate one function from another. The idea is to treat a group of SQL statements (i.e., an SQL command) as a "database function."

A database access manager enables applications to be written which are as independent of the database schema as possible. Applications developed this way retain the flexibility of the relational DBMS.

The database access manager should provide a small number of function, subroutine, or procedure calls that an applications programmer needs to learn. These functions should isolate the third-generation language code from the relational database language code. They should provide a high-level standard interface for the programmer to use in accessing the relational database. They should also isolate vendor specific relational features from the application so that a relational database management system vendor can be changed without modifying third-generation language code.

The major emphasis of the design of a database access manager is to satisfy the requirement that application programs be able to handle many different types of data structures and multiple SQL statements as a unit. In order to achieve this aim, the concepts of object oriented programming, in particular "data abstraction," are used extensively.

A database access manager should be written so as to maximize the efficiency and simplicity of relational database access and modification (set-at-a-time and non-procedural) as requested by applications (single record-at-a-time and procedural). It should make extensive use of the data dictionary, so that changes to the database do not affect its functional integrity.

The use of SQL commands, stored commands, stored programs, and stored procedures supported by a specific DBMS vendor, can all provide the *call-by-name* syntax required by the database access manager concept — see Figure 1. These features isolate the implementation details of the SQL *function* to be executed from that of the 3GL calling function which manages the execution of the SQL. The required SQL is defined and possibly compiled independently of the application, so that it can be changed without altering the program. Data structures which hold either input or output data are defined externally to the program in a loadable format and given a name. The name of the module of SQL statements, the name of the block of parameters that must be passed to it, and the name of input and output data structures become the arguments of a database access routine. As a result, the application

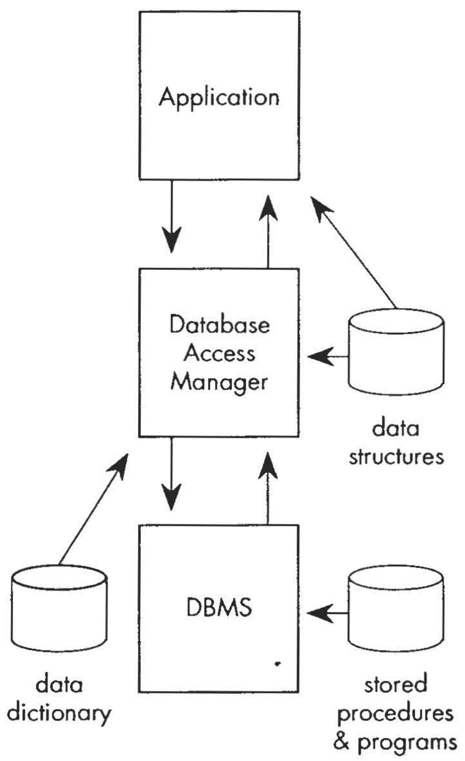

Figure 1. Database access manager architecture

becomes relatively database schema independent. Independent optimization of SQL statements and commands, and the promotion of robust database access (standard error and recovery handling, transaction management, for example) becomes possible.

A database access manager should improve the ability to meet the rules for flexible applications listed in Figure 2. These rules describe the nature of procedures which can be invoked from a 3GL or 4GL relational DBMS application, and which execute a group of DDL or DML statements. A collection of such procedures constitutes a proposed database access manager.

All of this can be done without sacrificing performance. The size of applications is minimized by reducing redundant database access code. The code can be written so that it is portable across environments. In effect, the intended flexibility of relational databases can be not only preserved, but extended to the application code. The cost of maintenance is decreased, debugging time is reduced, and neither the application programmer nor the database administrator need be concerned about unnecessary coupling between the application and the relational database. They can each do that portion of the work which they know best.

In a development environment which uses a database access manager, clearly defined roles for information systems personnel are created. These roles are separated by function. Managers can employ and train individuals to meet the specific needs of these roles. As a result, resource and budget management becomes easier than when individuals must acquire multiple skills. Highly trained relational database professionals are hard to find and demand higher than average salaries. The skills they possess should not be used for tasks which a proficient programmer can accomplish. Indeed, it is extremely difficult to train an individual in the intricacies of database design, SQL coding, SQL optimization, 3GL coding, and the application-specific functionality, let alone recruit the larger numbers of such personnel that are needed on medium- to large-scale relational DBMS projects. In particular, three significantly different roles for programming professionals are created: the applications programmer, the SQL programmer, and the database administrator.

The applications programmer writes code only in a non-database language, such as COBOL. When database access is required by the design, the applications programmer

- specifies the functional SQL requirements (though not the SQL) for the SQL programmer,
- defines the input and output data structures,
- writes the database access manager function calls and code skeleton, and
- specifies and codes data structure allocation or processing functions.

The SQL programmer takes over where the applications programmer leaves off. This individual is the interface between the applications programmer and the database administrator, and must be familiar with the current database schema as defined by the database administrator. The SQL programmer

- converts the functional SQL requirements into schema-specific, optimized SQL commands,
- ensures that the applications programmer's input and output data structures are properly interpreted by the SQL function,
- maintains the SQL commands as the database schema is altered,
- determines the performance load on the database with the database administrator, and
- implements the appropriate transaction management.

The database administrator (DBA) has a much more traditional role. The DBA

- designs and modifies the database schema to meet the needs of all applications,
- monitors and optimizes the load on database resources,
- manages database security, and
- manages database recovery and availability.

Database Access Manager Functions

A database access manager should provide an interface between the application program and the database. The purpose is to increase coding productivity by minimizing the need to know details of the access methods or of the schema of the database being accessed. The major functions of the tool should include opening and closing the database, binding of variables and data structures, the execution of data manipulation language statements, and the retrieval of pending data. The tool should not consume unnecessary memory or storage space, and be implemented either as a shared library or as a re-entrant server.

The detailed elements of database access manager functionality are determined in part by the method of implementation. If it is implemented via a shared library the database access manager should include

- multi-tasking initialization and termination,
- input and output program variable binding external to the application program,
- support for arrays of records and records of arrays,
- support for any "flat" data structure including linked lists, tree structures, and so forth,
- multi-record reads and writes,
- standard error processing and user-defined exception processing,
- 3GL procedural call interface.

A server-based tool can also enhance fault tolerance, availability, recoverability, and system administration through

- automatic deadlock recovery,
- asynchronous time-out and recovery,
- automatic retry after errors,
- forced table and database locking, and virtual record "locking,"
- virtual stored commands (preparsed queries not stored in the database),

1. *Schema Transparency:* Changes to the DBMS schema have no effect on procedure invocation and execution when such changes preserve information and theoretically permit unimpairment.
2. *DML and DDL Transparency:* The DML and DDL used to define the procedure have no effect on the invocation or execution of the procedure, even if the language syntax is changed, for example, if SQL is used in place of QUEL.
3. *DBMS Location Transparency:* The location of tables referenced by a procedure does not affect the invocation or execution of the procedure.
4. *Procedure Transparency:* The procedure is maintained in the database system catalog like any other database object, can be shared by all users, and can be executed in a manner consistent with the syntax of the DML.
5. *Domain Transparency:* Column domain changes do not affect the invocation or execution of a procedure, or the parameter definitions of a procedure.
6. *Syntax Transparency:* Procedure definition syntax changes do not affect the method of invoking or executing a procedure where changes theoretically permit unimpairment.
7. *Complexity Independence:* Procedure invocation and execution is independent of the complexity of the procedure definition.
8. *Detailed Diagnostics:* Detailed error information about procedure invocation and execution is provided.
9. *Full DDL and DML Support:* All DML and DDL statements can be executed via a procedure.
10. *No DBMS Imposed Restrictions:* There are no practical limits on the size of a procedure, or on the number of parameters or statements it can contain.
11. *Complete Security Support:* A means of controlling the permission to execute a procedure is provided that is consistent with the syntax of the DML and DDL.
12. *Transaction Scope Independence:* Transactions can span and be embedded in procedures.
13. *Database Code Isolation:* Database code is isolated from non-database code.
14. *Error Processing Transparency:* The error processing required when accessing the relational DBMS does not effect the invocation or execution of a procedure.
15. *Application Data Structure Transparency:* Changes to the application data structures that are used to access or modify the database have no effect on the invocation or execution of the procedure when such changes preserve information and theoretically permit unimpairment, nor do they require recompilation or relinking of the application.
16. *Performance Transparency:* Procedure optimization has no effect on the invocation or execution of the procedure.
17. *Application Transaction Management Support:* A facility to manage the priority, recovery, and relative scheduling of applications is provided that does not have to be hardcoded.

(Note: A procedure is a facility for executing a group of DDL and DML statements from a 3GL or 4GL relational DBMS application.)

Figure 2. Rules for supporting flexible relational DBMS applications

- soft failover to a *hot standby* database instance,
- an application level transaction definition language,
- general application transaction management.

What a Database Access Manager Should Not Do

While a database access manager will free the user from excessive concern with the intricacies of database software, it can only *encourage* good program structure and use of the relational database. It cannot *force* the user to write optimal code; it does not generate code, nor can it ensure that the database schema is properly designed. Security issues are considered to be in the domain of the database management system and the operating system. However, properly used, the tool will provide many benefits.

While the principal focus is, in fact, the execution of SQL (or other DMLs and DDLs), these routines should contain no intelligence whatsoever regarding the proper design and use of SQL commands. A production runtime environment is not the place for design and debugging. However, external utilities can be provided for interactive development of the SQL. Since most relational DBMSs provide such utilities, interactive design, development and testing of SQL should be encouraged. It is even possible to provide outboard translation of the SQL from one vendor dialect to another, but the overhead cost of doing this at runtime is undesirable.

It should not be the purpose of the database access manager to provide error checking which is application-specific. For example, the following should not be internal functions:

- defining procedural qualifications of data prior to writing to the database, e.g., edit checks
- defining procedural qualifications of data prior to acceptance of data retrieved from the database
- executing non-server related processing such as application specific exception processing or mirroring to host application files

A database access manager accepts SQL commands which are made

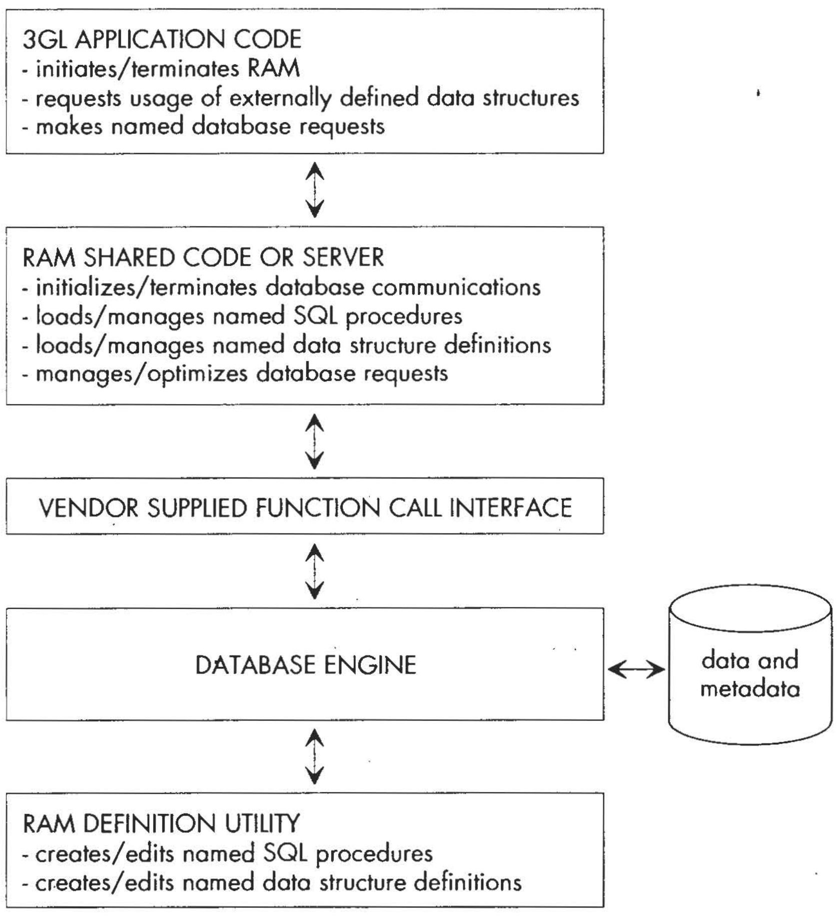
3GL APPLICATION CODE
- initiates/terminates RAM
- requests usage of externally defined data structures
- makes named database requests
RAM SHARED CODE OR SERVER
- initializes/terminates database communications
- loads/manages named SQL procedures
- loads/manages named data structure definitions
- manages/optimizes database requests
VENDOR SUPPLIED FUNCTION CALL INTERFACE
DATABASE ENGINE
data and metadata
RAM DEFINITION UTILITY
- creates/edits named SQL procedures
- creates/edits named data structure definitions

Figure 3. RAM architecture

specific by a named access routine argument list (the message) and *not* by the name of the function call. If access for a specific application purpose to the vendor database is accomplished through named function calls, co-mingling of data structures and control structures occurs, with the degradation of the software architecture being the final result.

An Existing Solution

One implementation of a database access manager is known as the Relational Access Manager (RAM) — see Figures 3 and 4. The RAM was designed and developed by Alternative Technologies over a period of eight years. It not only meets all the guidelines for a database access manager discussed above, but also has served as a repository for much of the expertise we have acquired in developing mission critical and complex relational database applications.

At the present time RAM libraries are available for ORACLE, ShareBase (formerly Britton Lee) and the Sybase SQL Server, each of which supports the currently *necessary* function call interface to the database. Forthcoming improvements to the embedded capabilities of RTI's INGRES DBMS may allow support for this product. By removing the need for a function call interface, products like DB2 could also be supported. Under VAX/VMS, the RAM supports all languages which meet the VAX calling standards — a uniform means of invoking subroutines and passing arguments. The C language is supported in Unix environments, with other languages such as FORTRAN, COBOL, Pascal, and Ada where possible.

There are two versions of the RAM:

- the *standard product* which consists of a library of (in most environments) shareable functions, and

```
* RAM-INIT needs be executed only once and initializes the interfaces.
* Each set of input and output data structures used by a RAM application may
* be defined either at runtime using calls to RAM-BIND or externally using
* a utility designed for the purpose.
* RAM-LOADDEFS is used to load and identify all named data structures and
* SQL procedures from the database.
* A particular data structure is made active with a call to RAM-SETDEF.
* RAM-QUERY and the "PERFORM...UNTIL" loop execute the required SQL queries.
* RAM-NEXTBUF handles the return of multiple rows of data from the database,
* converting the data into the proper output data structure automatically.
*
IDENTIFICATION DIVISION.
PROGRAM-ID.    CALLING-PROG.
ENVIRONMENT DIVISION.
DATA DIVISION.
* ram-tokens.h contains WORKING-STORAGE SECTION declarations and values
INCLUDE ram-tokens.h
PROCEDURE DIVISION.
INITIALIZATION.
  CALL "RAM-INIT" USING...GIVING RETURN-CODE
    IF LOAD-FROM-DATABASE = TRUE
* load buffer definitions
      CALL "RAM-LOADDEFS" USING... GIVING RETURN-CODE
* load command definitions
      CALL "RAM-LOADDEFS" USING... GIVING RETURN-CODE
* set input buffer definition
      CALL "RAM-SETDEF" USING... GIVING RETURN-CODE
* set output buffer definition */
      CALL "RAM-SETDEF" USING... GIVING RETURN-CODE.
* alternatively, create them in-line at run-time.
    ELSE
* bind input variables
      CALL "RAM-BIND" USING... GIVING RETURN-CODE
* bind output variables
      CALL "RAM-BIND" USING... GIVING RETURN-CODE.
*
BEGIN-MAIN.
* perform query processing while RAM-MORESTMTS
  PERFORM DATABASE-PROCEDURES
    UNTIL RETURN-CODE NOT = RAM-MORESTMTS.
END-MAIN.
*
TERMINATION.
* terminate the connection to RAM, close files, etc.
  CALL "RAM-CLOSE" USING... GIVING RETURN-CODE.
EXIT PROGRAM.
*
DATABASE-PROCEDURES.
* execute SQL queries regardless of kind
  CALL "RAM-QUERY" USING... GIVING RETURN-CODE
* if data is pending on return from RAM-QUERY get data
* using RAM-NEXTBUF while RAM-MOREDATA is returned
  IF RETURN-CODE = RAM-MOREDATA
    PERFORM DB-NEXTBUF USING...
      UNTIL RETURN-CODE NOT = RAM-MOREDATA.
* set up the call to RAM-NEXTBUF as a procedure.
DB-NEXTBUF.
  CALL "RAM-NEXTBUF" USING... GIVING RETURN-CODE.
```

Figure 4. RAM COBOL code skeleton

- the *extended product* which provides access to the library via a server process with various extensions such as global transaction management, and significantly more robust error recovery.

For some database products, the tool can eliminate constraints on what kind of database language statement can be processed from within a so-called stored command or procedure. For example, while Sharebase does not support the creation of tables within stored commands, RAM provides a means by which this may be accomplished. For products which do not support stored procedures, scripts become *virtual* stored procedures which can be created and maintained independently of either the application or the database code, and reside either on the host file system or in the database.

RAM also provides a scheme for implementing object-oriented interfaces in various relational database environments. While this scheme only works given adequate database design, control over data access, control over DML command creation and maintenance, and relational access manager routines that are not particularly sensitive to the number of parameters in the message, it is extremely powerful.

We have found that RAM leads to rapid prototyping and development as well as lowering the costs of maintenance. Very often, a simple and fairly standard 3GL code skeleton suffices to implement the relational application prototype. The details are then completed with extensive SQL, as the database schema is further defined and modified. Application functional prototyping and database design become parallel tasks. Since RAM provides a means for defining the data structures externally to the application code, recompilation and relinking are rarely needed.

Features such as automatic translation of SQL dialects, distributed application transaction management, and CL/1 support are planned.

David McGoveran is President of Alternative Technologies, a company that provides consulting services to users and vendors of relational DBMS software. Relational Access Manager and RAM are trademarks of Alternative Technologies.