M Database Access Methods: Which One Should You Use?

by Bruce B. Evans

M has come a long way since the days it was an operating system, a language, and a database. Today, under increasing pressure to offer open systems, vendors have responded with different access methods for use by M and non-M developers alike. This article will discuss three external access methods and highlight why you might consider using one over another. Finally, this article will discuss the choice and the architecture that was made for CorVision, an application generator, and the implications of the decision.

Direct Access to M

The first broad category is direct-access APIs (application programming interface). With this method, a non-M application written in, say C or COBOL, directly calls M-vendor-written service subroutines that perform the essential functions of the M language. For example, there would be a callable subroutine to perform $DATA, another to do $ORDER, another for SET, and so on. This method has only one advantage: It allows non-M programs to access an M database. It does have several disadvantages, however. To use M functionality, the developer really does have to understand the M database and the language since direct access closely mimics M language functionality. Since direct access is not a standard, not all M systems have it, and it is different for those that do. This means that code written to access one M system is not portable to the others or any other database for that matter. Finally, depending on how direct access is implemented by the vendor and used by the developer, it can be slow. Every time a direct-access function is called, the boundary between the current environment and M must be crossed. It is this "boundary crossing" or call interface where variables must be bound and unbound that can make the difference between a practical and an impractical interface, especially when performance is critical.

SQL Access

The second method, SQL access, has attracted the most attention recently. Many, if not all, M-system vendors already have released, or are about to release, an SQL interface. This usually takes one of two forms, embedded SQL in M code or dynamic SQL access from another language such as C or COBOL. The clear advantage to an SQL interface is that it allows non-M developers to gain access to M data without requiring knowledge of M itself or its global structures. In essence by using SQL, M can be viewed as a relational database. If that can be said, M can take its place alongside other mainstream technologies fully opening it to enterprise-wide applications in a standard fashion. Turning M into an open relational database does not come without a price, however, as will be shown.

A relational database's strength is M's weakness and vice versa. Within the bounds of physical limitations, any proper SQL statement will be processed by a relational database and return predictable, correct results. To do this, it must first examine its internal information (metadata) about tables, indices, and cardinality to determine how best to satisfy the request. It must then construct the request internally and execute it. All this requires resources and thus decreases efficiency. Reduced efficiency is the price of flexible data access at run time. The price of using M is that data-access methods are determined by database structure and therefore must be known at development time. This decreased flexibility often translates to faster and more efficient access. In general, relational databases must be highly tuned and somewhat "denormalized" to obtain the performance of a reasonably designed M database. So, it is important to realize that layering SQL on top of M generally will require more storage, memory, and speed to compare with the previous performance of straight M access.

Thus when evaluating SQL interfaces to M, efficiency of the SQL layer is very important. The portion of the database engine that decides how to satisfy an SQL request is called the optimizer. Optimizing an optimizer is by no means a trivial problem. Relational-database vendors have spent literally years perfecting it such that response time for most requests is reasonable. Therefore, efficiency will vary greatly from vendor to vendor depending on the experience and level of investment.

Any external SQL access to M requires an API to transport SQL to M and return results. Examples of this are Oracle's

OCI (Oracle Call Interface) or Microsoft's ODBC (Open DataBase Connectivity). When evaluating SQL for M, it is important to select an API that is sufficiently general to allow non-M applications to access multiple databases, or even switch databases without modifying source code. Note also that not all SQLs are standard, so selecting an interface that uses an SQL that complies with the ANSI SQL standard, or comes close to it, is essential. If this can be accomplished, applications using this API will be more homogeneous, more maintainable, and will not be tied to a specific database. In other words, they will be truly open.

All SQL interfaces require that any current M data structures be mapped into a relational model. There are several factors that may make this difficult. For instance, M does not impose fixed fields, records, data types, or any degree of normalization on the developer. M data structures also may change at run time, which is in stark contrast to relational databases that are designed for third normal form data, fixed data structures, and defined data types. Therefore, it is entirely possible that portions of particular M data structures cannot be used in conjunction with an SQL interface.

Not all SQL interfaces are appropriate if writing to M is a requirement. Two elements in particular must be considered for writes to be effective: locking and transaction management. Locks issued by an SQL-initiated transaction must be of the same type as other applications accessing the same database or conflicts will exist. Transaction management exists in many forms with varying features such as rollback, journalling, and two-phase commit. While it would be outside the scope of this article to describe them all, this too must be given careful consideration as it will have a major effect on efficiency and functionality.

An External Call to an M Routine

The third method is an external call to an M routine. An external application calls an M routine that accesses M data and returns a value. This value may then be processed by the calling routine. This method also has its advantages and disadvantages.

Among the advantages are that, unlike direct access, there are only two boundary crossings: one for the call and the other for the return, and not two for every M function. This increases efficiency. The M code is portable from vendor to vendor, and since the M database and language were designed to be used together, it is efficient. Among its disadvantages is, like the External Call API, that only M databases may be accessed in this manner. Thus, applications that call M routines will never be really open.

Even if the M routines are written to be portable, changes still may have to be made in three areas when accessing different M databases. Maximum string length varies from vendor to vendor, so care must be taken to ensure that the return value will never exceed this length. Further, since the method of returning values is not standard, code in this area may have to be changed. Finally, the method by which M routines are linked to the external application will vary greatly between vendors. Currently, the MUMPS Development Committee is considering popular remote-procedure call mechanisms for inclusion in the next M standard. If implemented, this would make calling M routines standard across all vendors and thus open up external applications.

Again, an efficiency versus openness judgment must be made. Is it better to have flexible data access at run time via SQL or can data-access methods be fixed at development time? If data-access methods are known at development time, is there sufficient expertise to write and maintain the M code that accesses the database? If the answer to the last question is yes, then calling an M routine generally will be more efficient than SQL access.

Virtual Relational Structure without Sacrificing Performance

When it came time to interface CorVision to M, one of the above choices had to be made. CorVision prefers relational-data structures to be effective, so an SQL interface seemed to be the clear choice. Indeed, there was already an SQL interface for most relational databases. For reasons of portability and efficiency, however, it was decided to generate and call M routines. These M routines would emulate relational access without all the overhead of an SQL.

The primary function of these routines is the two-way mapping between M hierarchical structures and virtual normalized relational structures. These routines also are charged with calling any other required preexisting M application routines to obtain computed column results when required. CorVision M access routines are created by first storing dictionary, mapping, structure, and access methods in an external repository. This repository is then used as input by a generator to produce callable M routines. The routines are then installed on the M system to be used.

Since the M code CorVision uses is all ANSI standard, the same generated routines can be used to access M databases from multiple vendors with little effort. Further, since the data-access method is usually known by most production applications at development time, the overhead of SQL can be avoided. Therefore, relational access was achieved with greater efficiency in one access method with only a minor sacrifice in flexibility.

Having decided on the method, the next decision was the implementation architecture. See figure 1. A prime consideration was that M data be treated the same as other data whether they be from sequential files or from relational databases. This would make the high-level application code for screens, reports, and batch procedures completely database-independent, as well as allow for merging, joining, and transaction management of data from disparate sources. To do all this required a multilevel-access architecture.

At the highest level, the application architecture contains data-source-independent code for screens, reports, batch procedures, menus, and so forth. Any time data access is required, a standard set of generic input/output routines is called. These routines are the same for any database including M.

Each generic I/O routine is generated to know what data source to access. There are generic I/O routines for most database functions including reading, writing, selecting, deleting, locking, and transaction and cursor management. These generic I/O routines then call database-specific routines at a lower data-manager level.

The code at the data-manager level makes calls to M routines in a vendor-approved manner. Data sent to M for writing are in the form of strings. Data returned from M are also a series of strings that are converted by the data manager into the appropriate data types and then placed into record buffers for processing by the application level.

At the lowest level, generated M routines themselves handle all database access including reads, writes, deletes, and locking. For `Reads`, the appropriate M routine assembles data from globals, nodes, pieces of nodes, routines, and so forth, to create a relational row that is then passed back to the caller. `Writes` do just the opposite: an M routine takes a relational row and disassembles it into its component nodes, or pieces of nodes. `Locks` lock developer-specified objects that are native to the M environment. Thus CorVision applications easily coexist with other native M applications.

This multitiered approach has several other advantages. Since the M code is generated from an external repository, every time a structure changes, the M code as well as generated documentation follows along. Further, since the application-level code is data-source independent it can be made to point at different data sources without changing code.

What's Best?

This article has described three external-access methods to M and some reasons for selecting one over another. But there is no single method that is clearly superior to another in every situation. Thus, it is critical to define your goals and objectives prior to making any choice. M

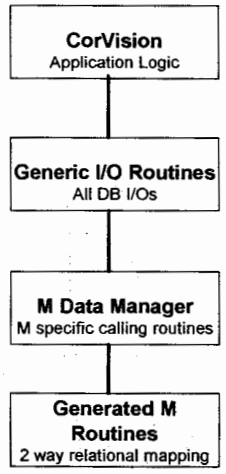

Figure 1. CorVision Multitiered Access Structure

What Is CorVision?

CorVision is a tool that generates applications across multiple databases, such as M, Rdb, ORACLE, Ingres, RMS, and C/ISAM. Additionally, it generates applications for Open VMS and UNIX. Users can generate applications either as client/server or character cell, depending on their requirements.

Developers describe their applications by describing data sources, screens, windows, and menus, for example. These descriptions, called metadata, are automatically stored in a repository.

The CorVision generator then takes the metadata from the repository to generate applications that include source code (Microsoft Windows Resource files, C application code, SQL, M data-access code, and so on), developers and user documentation, as well as online help.

This generated code is then compiled and tested. Based on test feedback, developers can rapidly change repository descriptions and regenerate those portions of the changed application. Generated code is rarely, if ever, modified by hand. Instead, CorVision allows the developer to specify event points where code, if inserted, is necessary. With each generation, these event points are included automatically in the application.

Bruce Evans is an independent consultant with a master's degree in mathematics/computer science. He designed the CorVision/M interface, a product of International Software Group, Inc., Waltham, Massachusetts. He lectures and consults on relational databases and on M. He also designs and implements with M, CorVision, and PowerBuilder. He can be reached at Sherborn Consulting Associates; the phone number is 508-655-3633.