Serving SQL

by John Clemens

Toward Simpler Interoperability

Just as the minicomputer revolution of the 1970s reinvented the wheels of the prior mainframe technology, improving them along the way, so did the microcomputer revolution of the 1980s reinvent and improve the wheels of the minicomputer technology. The wheel that was often most improved was the steering wheel, that is, the user interface. There is now great demand for data residing on mainframes and minicomputers as well as microcomputers to be made available to user-friendly software products running on commodity-priced PCs sitting on millions of desktops. Microcomputer technology has progressed from file servers to database servers. A leading contender for a language or protocol to talk to database servers is some form of Structured Query Language (SQL).[1]

To meet this demand as well as to provide simpler interoperability with software products in the more immediate environment, Digital's Digital Standard Mumps (DSM) Product Group decided to implement an SQL access method for DSM databases. Following is a description of this implementation and some of the necessary considerations.

Terminology

In speaking about databases, theorists often make a distinction between *data* and *metadata*, where for a particular data model, the metadata describe the organization of and the relationships between the data elements. Of course the metadata are data and often the metadata descriptions describe the metadata as well as the data (a formal requirement for relational databases). SQL assumes a relational data model and the metadata entities —relations, domains, and attributes— refer to tuples (the data). We may also speak of tables instead of relations, rows instead of tuples, and columns for attributes, or for the values of attributes in a row.

Database languages are classified as data definition languages (DDL) if they describe and manipulate metadata; they are called data manipulation languages (DML) if they retrieve and modify data. The data repository for metadata is usually called a data dictionary. To refer to data models, this article will use the terms hierarchical, network, and flat (rather than relational), since it is between you and your conscience whether the data you present comply with all of Codd's rules.[2] SQL will not generally notice, but necessitates first normal form (fields must be atomic, not repeating).

SQL is an ANSI standard language that serves as both a DDL and a DML for relational databases. M is a general purpose computer language with a built-in hierarchical storage facility for permanent data. As such, M is an extremely facile DML, but there is no intrinsic DDL or built-in data dictionary. With DSM, Digital distributes DSM Application Software Library (DASL), a 4GL query and reporting facility based on a data dictionary, which maps globals as flat records. We decided that we should support DASL as a source of metadata, but since M makes it easy to roll one's own data dictionary and many users have done so, we decided to support other data dictionary sources as well.

Implementation

A newly released Digital product, RdbAccess for Custom Drivers, provided a means of implementing SQL access. This product consists of a "non-SQL data server" (NSDS) engine, originally developed to support the product RdbAccess for RMS (Digital's Record Management Services). The NSDS engine allows the development of a metadata driver (to tell NSDS about metadata when requested), and a data driver (to fetch data when requested), via a relatively simple call interface. NSDS talks in turn to client software using Digital Standard Relational Interface (DSRI), in short pretending to be something like Rdb, Digital's relational database product.

The advantages of being "under the Rdb hood" are twofold: (1) many DSRI-layered software products such as SQL, DECQUERY, and RALLY can access DSM data; and (2) accessibility to cross-platform database-access protocols supported by Digital. This should allow an implementor to easily integrate SQL applications that reside on any of the eight platforms supported by Digital's Network Application Support (NAS) product family. These include DOS, Microsoft Windows, Apple Macintosh, SUN SPARCstation, OS/2, ULTRIX, OpenVMS VAX, and OpenVMS Alpha AXP systems. This cross-platform support should continue into the future regardless of which protocol wins the standards "war": Microsoft's ODBC, Borland's IDAPI, the SAG standard [1], or any other database access protocol.

Consequently, the product group wrote NSDS metadata and data drivers in C (because they had to be linked, shareable images), but used the DSM Call Interface to call in to DSM to do all the "dirty work." A typical NSDS driver routine receives requests from NSDS, sets some DSM local variables, executes a DSM routine entry point, reads the value of some DSM local variables, and finally returns data to NSDS. It is left entirely up to the DSM routines to specify the metadata and the data. The interface is simpler than an SQL interface—parsing SQL, doing joins, and optimizing queries all happen at higher levels. A read-only DASL version of the DSM routines was the first implementation, and an equivalent package for VA FileMan is under development. Publishing the call interface should allow users to develop data and metadata routines for their own databases.

The DSM entry points are listed in the following table:

Entry	To/From	Description
		Metadata Routines
INIT	From	Database attach
FF	To	Fetch global fields (domains)
FDR	To	Fetch dictionary relations
FRF	To	Fetch relation fields (attributes)
		Data Access Routines
INIT	From	Database attach
QUERYR	To	Relation and primary key info
DBKLEN	To	DBKEY length
AUXKEY	To	Auxiliary key info
OPEN	From	Open access context
CLOSE	From	Close access context
SAP	From	Set access path
DBKGET	To	Get tuple by DBKEY
GETNXT	To	Get next tuple

Table 1. DSM routine entry points.

The "To/From" refers to whether the information flow is to or from NSDS. These are the calls necessary for a read-only version. NSDS supports modification of data (but *not* metadata). So additional routines will be supported for a read/write version now under development, as shown in Table 2.

Entry	To/From	Description
TRSTART	From	Start transaction
COMMIT	From	Commit transaction
ABORT	From	Abort transaction
INSERT	From	Insert tuple
DELETE	From	Delete tuple (by DBKEY)
UPDATE	From	Update tuple (by DBKEY)

Table 2. Modify data routines.

In the read/write version, there is a need for transaction rollback whether DSM transactioning is used or not. A problem for both DASL and FileMan is the semantics of INSERTing or UPDATEing a tuple that contains pointer fields; DELETE is less of a problem. All data modification calls might require the specification of an M action routine, written by someone familiar with the database, to preserve referential integrity.

This cross-platform support should continue into the future regardless of which protocol wins the standards "war" . . .

The read-only routines thus far implemented can be relatively simple, but there are some special considerations.

- Datatypes—M does not have any, but DASL and FileMan do, based on the use of the datum. The metadata driver tells NSDS that all fields are one of three data types—STRING, NUMERIC, and DATE—and performs the necessary conversions.

- DBKEYS—When fetching tuples for SQL, a dbkey (a unique identifier for the tuple) must be returned along with the tuple. A later request in the SQL session might request the tuple by dbkey. For DASL, the dbkey is the delimited concatenation of the "primary keys" (subscripts); for FileMan the dbkey will be the unique internal entry number. Depending on external circumstances, dbkeys have to be valid for the life of a transaction or of a session. This interface then assumes dbkey validity for the life of the session.

Continued on page 54

- Access paths—NSDS permits the specification of one primary key and multiple auxiliary keys to take advantage of existing indices or cross-references. For DASL, the primary key is the first "primary key" (highest-level subscript); for FileMan it is the cross-reference for the NAME (.01) field. The call interface is at a lower level than SQL—parsing has all been figured out for you, but the "WHERE" clause peeks through in the case of access paths. For example, if the data driver tells NSDS that for relation WINES there is an auxiliary key for attribute VINTAGE, the query "SELECT * FROM WINES WHERE VINTAGE BETWEEN 1970 AND 1975" will cause a set access path call conveying the upper- and lower-bound information for that key, and these bounds must be honored in the "get next tuple" call returns.
- Collating Sequence—The M collating sequence for subscripts posed some problems, at least in the DASL implementation. Even though the tuples in a relational database are not ordered, in the case where the primary key is logically a string but has some canonic numbers, an ASCII order function had to be written in M to satisfy access path requests for that key. Of course, in DSM, a global may be string-collated but many are not so collated, and it may not be an option to convert existing globals.

Figure 1 diagrams a sample architecture for this implementation (using a field test version of an ODBC driver for Rdb).

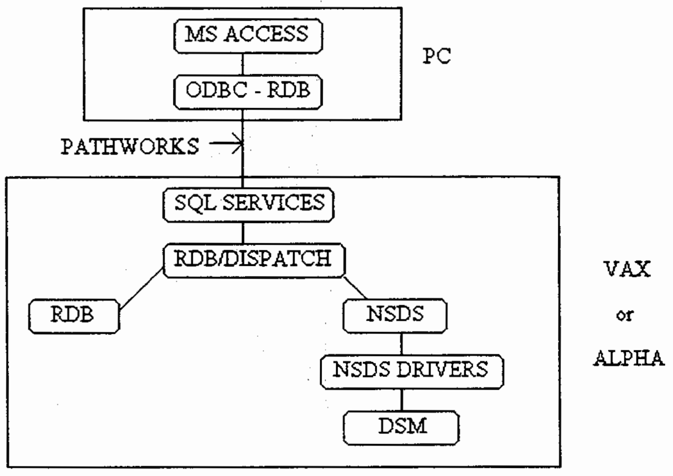

Figure 1. Architectural overview.

DASL

The DASL version was relatively free of problems, since the DASL data dictionary had been designed to support SQL-like queries. Tables are defined with a set of fixed-length fields based on data dictionary fields containing global references. DASL, like other M tools, tends to be user friendly and flexible. This flexibility may cause conflicts when interfacing with more rigid standards. The DASL implementation presents two examples.

Since field definitions in DASL tables reference data dictionary fields, it seemed natural for the metadata driver to tell NSDS about all the data dictionary fields as global fields (domains), and define the table fields referencing the domains. But since DASL allows overriding the datatype and length of the data dictionary field, the table definition of the field could be in conflict with the global field definition as far as NSDS was concerned. This resulted in having no global fields. They aren't required; NSDS will invent domains.

DASL supports several types of cross-references and some of them are case-insensitive; this is ideal for looking up names, for example. But since such cross-references are not ASCII-sorted with respect to the original data, they cannot be easily used for auxiliary key access paths defined by NSDS.

FileMan

FileMan provides an interesting contrast with DASL. In DASL, data storage is assumed to be hierarchical, but the data model is flat. In FileMan the data storage for a file is flat, but the data model (because of subfiles) is hierarchical. There are four particular problems facing the FileMan implementation—subfiles, pointer fields, field lengths, and fields that are arbitrarily long.

DASL, like other M tools, tends to be user friendly and flexible.

Multivalued fields, implemented as subfiles, violate first normal form; one either omits them or finds some way to present flat tables. Two ways come to mind: one is "flattening" and the other is "normalization." Flattening means that for each value of a subfile, the M data driver would present a separate tuple of the parent file entry along with that value. For example, flattening a file entry with two subfiles, each of one hundred entries, would result in ten thousand tuples. Normalization consists of defining internal entry numbers as external table attributes and presenting subfiles as separate tables with the parent's entry number as one of its attributes; the parent table could then be joined with the table representing its subfile. For example, if there were a patient file with a nickname subfile, these would be represented as a patient table (with the internal entry number as one of the attributes) and a separate nickname table (with the patient file internal entry number as one of the attributes and the table name invented by the M driver). Subfiles can have subfiles recursively: the New Person file defined in Kernel V. 7.0 has dozens of subfiles at as many as three levels below the top, summing to hundreds of fields. This could present a problem if totally flattened, because although Rdb doesn't have any particular column limit, SQL SERVICES does and so does Microsoft Access. Normalization of this same file could lead to many passes over the same actual file to provide joins with multiple subfiles represented as separate tables.

In short, total flattening could exceed certain limits, and total normalization could cause abysmal performance in some cases.

Pointer fields are not a problem if the default representation is to supply the NAME field value of the file entry pointed to, but if you want "extended pointers" as described in Davis's work, the information you need is not in the data dictionary (although it might be in a print template).[3]

Field lengths are another problem because maximum lengths are not stored in the data dictionary, although input edits for text fields often contain "`$L(X)>30`" from which a maximum can be inferred. Numeric fields often have print justification, e.g., "J10," from which a length can extracted. But field lengths are hard to determine or indeterminate in the general case.

Word-processing fields are arbitrarily long. Rdb actually supports such fields (called in SQL "LIST OF BYTE VARYING," which is a list of BLOBS each over 65,000 octets long). Sadly, NSDS does not, so we cannot use them. NSDS does support text fields of up to 32,765 characters, but maybe for the first version we will limit fields to 512 characters (guess why).

The solution to these problems could be defining a default behavior in each case, which could then be overridden by an entry in an SQL presentation file (hierarchical, of course) with entries for each file, and subfiles for the fields. This file would be used by the M drivers to control how they respond to SQL queries. The default presentation for subfiles might be normalization, but in this file you could specify flattening. Likewise the default behavior for pointer fields could be overridden. The default for field length could be a maximum, say 512, but the SQL presentation file could specify an actual maximum. This file would also be useful for defining files and fields that SQL need not see.

Another way to resolve the flattening versus normalization issue would be for the M drivers to present a table for the totally flattened file and (inventing table names as necessary) tables for the completely normalized file. **M**

Endnotes

1. SAG is the SQL Access Group, a nonprofit standards organization that works in conjunction with X/Open. Its members include Digital and Microsoft. Standards being developed include an API (application programming interface) for embedded SQL, a CLI (call level interface) dynamic SQL interface, and FAP (Formats and Protocols)—a client/server communication protocol for SQL remote database access. Microsoft's ODBC (Open Database Connectivity) is based on SAG's CLI. Borland's IDAPI (Integrated Database Application Programming Interface) is also based on SAG standards with extensions for nonrelational databases. A number of companies, including IBM, have jumped on this bandwagon. Interface programmers love standards—as many as possible.

2. E. F. Codd, "A Relational Model of Data for Large Shared Databanks," *CACM*, 13:6 (1970), 377-387, and subsequent papers, raising the hurdle a relational database must clear.

3. R. G. Davis, *FileMan User Manual, Volume II*, 1990.

John Clemens is an independent consultant currently associated with the DSM Product Group. He has been writing computer programs for over thirty years. Many of these programs were written in M (MUMPS). He can be reached at 508-256-8044 or clemens @dsm.enet.dec.com.