AN OUTLINE OF MQL

Victor J. Streeter
School of Management
University of Michigan-Dearborn

1. Introduction

1.1 A Perspective on Data Management

Data management is considered here to involve the collection, maintenance, and manipulation of data for the purposes of current and potential operational and decision support. Typically the data would be organized into one or more files where at least one file would vary in size. Missing data would be recognized in some way. Systems for data management as defined above would include such types of software as editors, compilers, database systems, spreadsheet systems, expert system shells, and statistical packages.

Every program designed to do some kind of data manipulation makes use of one or more language systems through which operations on the data are carried out. These systems can be evaluated through measures that include the following:

1) *Expressivity* describes the types of problems that a language system is intended to address. If a certain type of problem cannot be solved in a language system, the user will have to move to another system, presumably at some inconvenience.
2) *Simplicity* of design involves such objectives as a minimal number of constructs, consistency of definition and use of such constructs, and similarity to language systems with which the user is likely to be familiar. These objectives can, however, conflict with expressivity.
3) *Reliability* of use is a measure of the expected rate of user errors in utilizing the language system for various types of problems. Simplicity should enhance reliability. When a language system is used to manipulate objects not directly describable in the system, simplicity and reliability of applications would be decreased.

In addition, one must consider whether data management is done best through specialized language systems or, at least in theory, is better done through a single language system. The position of this paper is that the different types of software mentioned above are all concerned with the same basic problem, and consequently a single data management system (DMS) to address these areas is not only possible to construct, but also worth constructing and evaluating. This system would make use of an official, textual, command/specification language with which users could specify applications, but one or more related screen-oriented languages utilizing menus and forms would also be made available as front ends to the command language.

1.2 SQL and the DMS Concept

One cannot expect a DMS to be developed quickly, or that if being developed it would attempt a multiple domain capability from the beginning. Thus it is considered here possible that any implementor of SQL, or a standards organization, could describe extensions to SQL to implement an associated programming language, text management, deduction, etc. The actual growth in expressivity of the language since the 1970's, especially in the IBM environment, has been slow.

A circumstance which might tend to impede SQL's adaption to new applications is its use of English keywords for statement identification and modification. One might have more and more difficulty finding an appropriate unused word in an appropriate place to identify a new capability. This same circumstance holds also for COBOL, dBASE, and other languages.

1.3 SQL as a Query Language

SQL was, of course, designed to be an interactive query language and host language interface to the database. Its adoption by virtually every commercial database system of consequence within a relatively short period of time is without precedent. This popularity has also led to a certain amount of critical analysis being directed towards the language, e.g., by Date [Date89, Date90a] and Codd [Codd90]. Date has focused on arbitrary and inconsistent structures in the language, while Codd has criticized SQL's acceptance of duplicate retrieved records, its handling of nested queries, and its inadequate treatment of nulls.

Apparently not included in their concerns is the syntax of the SELECT expression, which because of its recursive definition and the diverse components it may contain must surely rank among the most complex types of expression in any language in use today. This complexity does not support the reliable use of the language interactively where testing of query results is not

common. Alternatives to large SELECT expressions are not encouraged when temporary tables are tedious to define and when certain views cannot be joined together, at least in some implementations.

2. The Current Conceptualization of MQL

2.1 Introduction and Overview

Concern over the pace of development of SQL, its use of complex subqueries, and hence its potential unreliability as an interactive language led to the conceptualization of another system, called MQL for modular query language [Stre86]. The development of this system has up to now focused on its role as an interactive, textual data definition and manipulation language for a conventional database system. The intention, however, is to be able in principle to adapt this system as a command/specification language to the major areas of data management. MQL is being implemented as a single-user, memory-based system using MS-DOS Turbo Pascal.

2.2 Data Structure

An MQL database currently contains two catalog tables describing tables and fields and one or more user data tables. Fields represent either a domain defined on a native data type or a user-defined object. Objects can represent aggregates of data and can specify procedures (methods) associated with the object. Native data types include fixed length types such as character, number, date/time point, and date/time interval. Varying length data essentially would be represented through the type text. Fixed length data which may be missing is defined, where feasible, to have a null-indicating prefix. Any null-permitting field can be redefined as null-prohibiting and thus be readable. In this way many different null values can be recognized for a field and handled through a two-valued logic. The utilization of a null as a default value would also be supported in this way. Fields which are declared primary will be displayed by default when no explicit field list for display is given; fields declared secondary are only displayed explicitly. Secondary fields can overlap other fields.

2.3 A Sample Database

STUDENT(STU#,NAME,SCHOOL,HOURS, GPA,DOB,ADDRESS)
Each student is described through a unique number, his/her name, school, credit hours achieved, grade point average times 100, date of birth, and home address.

STUSECT(STU#,COURSE#,SEC#,HOURS)
Each election of a course by a student is described through the student number, course number, section number, and credit hours.

PROF(PROF#,NAME,TITLE,DEPT,SALARY, OFFICE)
Each instructor is described through a unique number, his/her name, title, department, academic year salary, and office.

2.4 Statement Structure

All statements end with a period. Statements can be seen as implementing a [object] [message] structure. No keywords are utilized, but statements are defined or modified through indicators (messages) beginning with a "@" and other special characters.

2.5 Row-Level Retrieval

The simplest case of retrieval is represented through the table (object) name appearing alone. A null message is assumed in this case that requests the table to display itself. Thus

```
STUDENT.
```

would display all primary fields for all rows of STUDENT, while

```
STUDENT & STUSECT.
```

would display the Cartesian product of STUDENT and STUSECT. The query

```
STUDENT('Student Name'>NAME@2,
    SCHOOL@1,STU#=*)
    & STUSECT(STU#=*, COURSE#)
    @(COURSE# ~ 'IS %' & SCHOOL <>
    'BSAD').
```

shows the name of each student who is taking an IS course but is not in the BSAD school. The IS course being taken is also shown. The STU#=* implements the equijoin, and the "~" represents the SQL "LIKE" operator. The column heading 'Student Name' replaces the default heading "STUDENT", and the resulting records are ordered by school, and within school by name. The condition attached to this join is considered a modifier (or message). Other modifiers include @UNQ to specify no duplicate records, and @MAXREC(*n*) to specify that only the first *n* records meeting the condition are to be displayed. All modifiers of this type are placed in any order at the end of the statement. If the above condition was written as

```
@MAYBE(COURSE# ~ 'IS %' & SCHOOL
    <> 'BSAD')
```

then only those records evaluating to null because of a missing COURSE# or SCHOOL would be displayed.

Deletion of records is carried out, as in dBASE, in two steps. The first step makes use of the @DEL modifier to accomplish a logical deletion, thus

```
STUDENT('Student Name'>NAME@2,
SCHOOL@1,STU# = *) & STUSECT(STU# = *,
    COURSE#) @DEL @(COURSE# ~
      'IS %' & SCHOOL < > 'BSAD').
```

would logically delete and display the student records described above. These records can be restored with the @RECALL modifier, or permanently removed with the @PACK modifier. This two step approach allows the user to consider the effects of the deletion before it is made permanent and can be helpful in simulating various types of deletions without using temporary tables to hold nondeleted records.

Modification of records is implemented using "C" style assignment operators. The indicated modifications will be made for each valid combination of table rows. Care must be taken that a modification takes place the desired number of times. Thus, to increase the salary by 10% of each IS instructor as identified in the table IS_INST and display their names, new salaries, and the new total of their salaries we could type

```
PROF(PROF# = *, NAME,
SALARY* = 1.1; SUM(SALARY)) &
IS_INST(PROF# = *).
```

The user is then shown, with changed fields highlighted, each altered record on the screen. Modification here is not different in essence from the SQL approach, but when done interactively the user is asked if the changes are to be retained with the table. The user may refuse to accept the changes either because they are incorrect or because the user was merely "experimenting" with different modifications.

2.6 Temporary Tables and Views

A temporary or snapshot table is defined by preceeding a retrieval command by the table name followed by a colon. In this way

```
BSAD_STU_BY_GPA: STUDENT(GPA@1,
    NAME, STU#) @(SCHOOL = 'BSAD').
```

will create the table with the three indicated fields from STUDENT with the resulting stored records ordered by GPA. Fields in temporary tables inherit attributes such as name, width, domain, and null status from the defining table expression. Names are checked for uniqueness. Fields for temporary tables can be given new names in the table expression or after the system has discovered duplicate names. A temporary table is dropped automatically when the MQL session is ended unless the user explicitly converts it to a base table.

A view differs from a temporary table in that

1) when created the name is modified by @V
2) references to the view will reflect the most recent changes to the underlying tables (as in SQL)
3) for the sake of simplicity no updates will be permitted through views
4) its definition may include a reference to itself (see Transitive Closure below)

2.7 Groupings and Indexes

A grouping is a partitioning of a base or temporary table through values of one or more fields. Grouping names always begin with an "\" and may not otherwise match any current table names. Once defined, groupings can support queries directed to grouped records. For example,

```
\STUDENT_BY_SCHOOL:
    STUDENT/(SCHOOL).
\STUDENT_BY_SCHOOL(;SCHOOL,
    MAX(GPA), AVG(GPA)).
```

defines a grouping of student records by school and then displays for each school the maximum and average gpa for current students. The query

```
\STUDENT_BY_SCHOOL(;SCHOOL, MAX(GPA),
    AVG(GPA)) @(RATIO(GPA > 300) > .5).
```

will display the preceeding information only for schools where more that half of the students have a gpa exceeding 3.00. The result could also be defined as a temporary table or view.

If we wished to see the top five students in gpa in each of the schools we would enter

```
STUDENTS_ORDERED_BY_GPA:
    STUDENT(NAME, SCHOOL, GPA@1).
\STUDENTS_BY_SCHOOL_ORDERED_BY_GPA:
    STUDENTS_ORDERED_BY_GPA/(SCHOOL).
\STUDENTS_BY_SCHOOL_ORDERED_BY_GPA
    @MAXREC(5).
```

Groupings also play the role of indexes in that they indicate the locations of records for each current value of a grouping (indexed) field. There is thus no separate concept of index in the memory-based implementation of MQL. Groupings, like views, always reflect the most recent changes in their underlying tables.

2.8 Group Joins

The capability to compare groups of records directly is implemented by extending the join operation to produce the Cartesian product of two sets of groups with appropriate conditions to compare the two groups representing each element of the product. If we define

```
\CLASS_BY_STUDENT: STUSECT/(STU#).
```

to group STUSECT rows by student number, and if we then wished to discover if any two students were enrolled in exactly the same sections for the same number of credit hours each, we could enter

```
\CLASS_BY_STUDENT & \CLASS_BY_STUDENT
    @(@EQUAL).
```

The result would be expressed as a two column table of STU#'s (with a name change for the second column) where each row would indicate a pair of students with an identical schedule. Since each student would be paired with himself, it would be desirable to eliminate the identical pairs and express a particular pair just once (rather than repeating B,A after A,B) by

```
\CLASS_BY_STUDENT &
    \CLASS_BY_STUDENT @(@EQUAL
    & @1.STU# < @2.STU#).
```

where @1 and @2 represent the two groupings within the condition. If we wished to identify pairs of students with at least three common classes we would specify

```
\CLASS_BY_STUDENT &
    \CLASS_BY_STUDENT
    @(@MATCHING(>=3) & @1.STU# <
    @2.STU#).
```

2.9 Report Specification

A general report specification capability based on MQL grouping definitions will handle such cases as: for nonadjunct faculty only show: for each title within every department show the faculty names and salaries followed by their count, the average and the total salary; for each department the count of faculty and the sum of their salaries; and for all of the departments combined show the count and the total of the salaries. This report could be specified and produced through

```
NONADJUNCTS: PROF(NAME, SALARY,
    TITLE) @(TITLE !~ 'ADJ%').
NONADJUNCTS(;@COUNT,
    SUM(SALARY)) /(DEPT)(;@COUNT,
    SUM(SALARY)) /(TITLE)(NAME,
    SALARY;@COUNT, @AVG(SALARY),
    @SUM(SALARY)).
```

Additional modifiers could be attached to the definition to provide for a title, special formatting of the summary or detail lines, etc. Report specifications could be saved through a view definition.

2.10 Transitive Closure

Suppose each row of a table FAMILY(PARENT, CHILD) names individuals in a parent-child relationship. The transitive closure of FAMILY could be named FAMILY_TC(ANCESTOR, DESCENDENT) and would display every parent together with each of his/her descendents. This result could be defined through a recursive view definition

```
FAMILY_TC(ANCESTOR,
  DESCENDENT)@V: FAMILY |
    FAMILY_TC(ANCESTOR,
    DESCENDENT=*1) &
    FAMILY(PARENT=*1, CHILD).
```

Another approach would utilize the transitive closure forming function EXTEND_TR (for extend transitive relationship), e.g.

```
FAMILY_TC(ANCESTOR,
    DESCENDENT)@V:
    @EXTEND_TR(FAMILY(PARENT,
    CHILD)).
```

This function would also provide for the specification of columns in the resulting table describing

1) the number of levels (in this example, a level is a generation) separating the objects in each row, and
2) summary measures, e.g. for distance and cost, that describe the separation of the objects in each row

2.11 Outer Join

The conventional approach to defining the outer join, as in [Date89], can be considered as somewhat awkward if in a one-null system all of the participating fields for a table permit null and these fields for some record are already null and cannot be set to null to mark a record in the other table as unmatched.

Aside from MQL's capability to utilize a special null as an unmatched record marker, an alternative approach to this problem in MQL would make use of the one-byte, secondary #TAG field common to all tables. Assuming all #TAG fields for all tables participating in a (inner) join were, for example, blank, the @MARK(!) modifier attached to the join would cause the #TAG fields of all matching records to be set to "!", while the #TAG fields of unmatched records would remain blank. The #TAG field now marks each record's participation in the join.

2.12 Programming Language

It is anticipated that the as yet undesignated MQL programming language (MPL) will be modeled very closely after some existing object-oriented language. The languages MQL and MPL would, of course, work with the identical objects and naming conventions.

2.13 Integrity

At least at the beginning it is expected that integrity specifications will take the form of procedures (triggers) that can be declared to be called after record modification, insertion, and deletion for any table, even for primary key enforcement. The procedures would make use of MQL commands and MPL constructs to provide for maximum expressivity in identifying and responding to integrity problems. This approach also provides uniformity in the declaration of all integrity rules.

2.14 MQL and the Relational Model

MQL's intended domain certainly includes conventional database management and its implementation so far has been centered in that area. Its approach to data management can be compared with certain firmly established tenets of the relational model, as described by Date [Date90a]. The comments below are not intended as a definitive comparison of the MQL and relational data models.

2.14.1 Primary Key Requirement

The requirement that every base table, view, and query result have an associated primary key is apparently a cornerstone of the relational model [Date90a, p. 94]. MQL, like most commercial systems which are commonly termed relational, does not support this requirement. While avoidance of unnecessary sorting to eliminate harmless duplicate records is often cited as sufficient grounds to reject this rule, another factor that bears on this issue is the intended domain of the relational model. Documents intended to be managed only by a screen editor would not require their lines to be numbered in order to be accessed. A statistical database must be able to provide views of its data where individual record keys are not available to the unauthorized user [Adam89]. Since MQL is in principle supportive of text processing and statistical database applications, the universal primary key requirement does not appear appropriate to adopt.

2.14.2 Quota-Directed Queries

The argument for quota-directed queries is that they can be very convenient in testing the correctness of a query, while to implement them no new operational mechanism of any substance is involved. Date [Date90b, p. 359] appears to accept them in principle, but he has apparently not included an implementation among his many suggestions for SQL. In MQL the @MAXREC modifier limits the number of records displayed from a table, or join, or from each group in a grouping, and supports deletions of and modifications to a specified number of records from the beginning of a table or group.

2.14.3 Order-Directed Operations

The relational model, seeing tables as unordered collections of records, seeks to confine information specifications to values in tables, and this goal supports ordering only for query results. While a known sequence of base table records might be exploited by a quota-directed query, views and temporary tables could not be defined to organize their records in any arbitrary sequence. MQL permits ordering for all of its tables, and groupings are ordered automatically by the grouping fields.

2.14.4 Closure

The property of closure for relational operations is properly seen as a strength by the relational model, but closure can encourage undesirable approaches to data management. In one case, closure is used by Warden [Date90a, pp. 462-465] and the designers of SQL to support deeply nested subqueries which might encourage users to express their specifications as a single, large, and complex statement that would be less reliable than several smaller statements that could each be easily checked for accuracy. MQL's design precludes subqueries for the sake of reliable implementation and use.

In another case, closure is seen as preventing reports with different output record structures from being produced from, for example, one SQL SELECT statement. This forces reports to be produced through the assistance of some non-SQL system. MQL, on the other hand, is prepared in principle to generate multiformat results and also to store these results in a temporary table whose records would contain one field of type text.

3. Summary

The intent of the design of MQL is to provide a language system which can serve as a platform for a comprehensive, object-oriented data management system. At the same time, concerns over the reliable use of the system have led, for example, to simple statement designs that prohibit nested structures and force interactive users to see changes in tables before the changes are commited. In all, an attempt is being made to advance expressivity, simplicity, and reliability together.

References

[Adam89] Adam, N. R., and Wortman, J. C., "Security-Control Methods for Statistical Databases: A Comparative Study", *ACM Computing Surveys*, December, 1989.

[Codd90] Codd, E. F., *The Relational Model for Database Management, Version 2*, Addison-Wesley, Reading, MA, 1990.

[Date89] Date, C. J., *A Guide to the SQL Standard*, Addison-Wesley, Reading, MA, 1989.

[Date90a] Date, C. J., *Relational Database Writings 1985-1989*, Addison-Wesley, Reading, MA, 1990.

[Date90b] Date, C. J., *An Introduction to Database Systems*, vol. 1, Addison-Wesley, Reading, MA, 1990.

[Stre86] Streeter, V. J., "Some Relational Query Language Design Issues and the Language MQL," *Proceedings of the ACM Computer Science Conference, 1986.*