A CRITIQUE OF THE SQL DATABASE LANGUAGE

C.J.Date

PO Box 2647, Saratoga
California 95070, USA

December 1983

Abstract

The ANS Database Committee (X3H2) is currently at work on a proposed standard relational database language (RDL), and has adopted as a basis for that activity a definition of the "structured query language" SQL from IBM [10]. Moreover, numerous hardware and software vendors (in addition to IBM) have already released or at least announced products that are based to a greater or lesser extent on the SQL language as defined by IBM. There can thus be little doubt that the importance of that language will increase significantly over the next few years. Yet the SQL language is very far from perfect. The purpose of this paper is to present a critical analysis of the language's major shortcomings, in the hope that it may be possible to remedy some of the deficiencies before their influence becomes too all-pervasive. The paper's standpoint is primarily that of formal computer languages in general, rather than that of database languages specifically.

1. INTRODUCTION

The relational language SQL (the acronym is usually pronounced "sequel"), pioneered in the IBM prototype System R [1] and subsequently adopted by IBM and others as the basis for numerous commercial implementations, represents a major advance over older database languages such as the DL/I language of IMS and the DML and DDL of the Data Base Task Group (DBTG) of CODASYL. Specifically, SQL is far easier to use than those older languages; as a result, users in a SQL system (both end-users and application programmers) can be far more productive than they used to be in those older systems (improvements of up to 20 times have been reported). Among the strongpoints of SQL that lead to such improvements we may cite the following:

* simple data structure

* powerful operators

* short initial learning period

* improved data independence

* integrated data definition and data manipulation

* double mode of use

* integrated catalog

* compilation and optimization

These advantages are elaborated in the appendix to this paper.

The language does have its weak points too, however. In fact, it cannot be denied that SQL in its present form leaves rather a lot to be desired -- even that, in some important respects, it fails to realize the full potential of the relational model. The purpose of this paper is to describe and examine some of those weak points, in the hope that such aspects of the language may be improved before their influence becomes too all-pervasive.

Before getting into details, I should like to make one point absolutely clear: **The criticisms that follow should not be construed as criticisms of the original designers and implementers of the SQL language.** The paper is intended solely as a critique of the SQL language as such, and nothing more. Note also that the paper applies specifically to the dialect of SQL implemented by IBM in its products SQL/DS, DB2, and QMF. It is entirely possible that some specific point does not apply to some other implemented dialect. However, most points of the paper do apply to most of the dialects currently implemented, so far as I am aware.

The remainder of the paper is divided into the following sections:

* lack of orthogonality: expressions
* lack of orthogonality: builtin functions
* lack of orthogonality: miscellaneous items
* formal definition
* mismatch with host languages
* missing function
* mistakes
* aspects of the relational model not supported
* summary and conclusions

Reference [3] gives some background material -- specifically, a set of principles that apply to the design of programming languages in general and database languages in particular. Many of the criticisms that follow are expressed in terms of those principles. **Note:** Some of the points apply to interactive SQL only and some to embedded SQL only, but most apply to both. I have not bothered to spell out the distinctions; the context makes it clear in every case. Also, the structure of the paper is a little arbitrary, in the sense that it is not really always clear which heading a particular point belongs under. There is also some repetition (I hope not too much), for essentially the same reason.

2. LACK OF ORTHOGONALITY: EXPRESSIONS

It is convenient to begin by introducing some nonSQL terms.

* A _table-expression_ is a SQL expression that yields a table -- for example, the expression

```
SELECT *
FROM   EMP
WHERE  DEPT# = 'D3'
```

* A _column-expression_ is a SQL expression that yields a single column -- for example, the expression

```
SELECT EMP#
FROM   EMP
WHERE  DEPT# = 'D3'
```

A column-expression is a special case of a table-expression.

* A _row-expression_ is a SQL expression that yields a single row -- for example, the expression

```
SELECT *
FROM   EMP
WHERE  EMP# = 'E2'
```

A row-expression is a special case of a table-expression.

* A scalar-expression is a SQL expression that yields a single scalar value -- for example, the expression

```
SELECT AVG (SALARY)
FROM   EMP
```

or the expression

```
SELECT SALARY
FROM   EMP
WHERE  EMP# = 'E2'
```

A scalar-expression is a special case of a row-expression and a special case of a column-expression.

Note that these four kinds of expression correspond to the four classes of data object (table, column, row, scalar) supported by SQL -- though incidentally SQL is inconsistent as to whether its expressions yield values or references, in general. Note too that (as pointed out in [3]) the four classes of object can be partially ordered as follows:

$$\begin{tikzcd} & \text{table (highest)} \arrow[dl] \arrow[dr] & \\ \text{column} \arrow[dr] & & \text{row} \arrow[dl] \\ & \text{scalar (lowest)} & \end{tikzcd}$$

(columns are neither higher nor lower than rows with respect to this ordering).

As explained in [3] (again), a language should provide, for each class of object it supports, at least all of the following:

* a constructor function, i.e., a means for constructing an object of the class from literal (constant) values and/or variables of lower classes;

* a means for comparing two objects of the class;

* a means for assigning the value of one object in the class to another;

* a selector function, i.e., a means for extracting component objects of lower classes from an object of the given class;

* a general, recursively defined syntax for expressions that exploits to the full any closure properties the object class may possess.

The table below shows that SQL does not really measure up to these requirements.

\ opn obj\	constructor	compare	assign	selector	gen expr
table	no	no	only via INSERT - SELECT	yes	no (see below)
column	only as arg to IN (host vbles & consts only)	no	no	yes	no
row	only in INSERT & UPDATE (host vbles & consts only)	no	only to/ from set of host scalars	(yes)	no
scalar	N/A	yes	only to/ from host scalar	(yes)	no

Let us consider table-expressions in more detail. The SELECT statement, which, since it yields a table, may be regarded as a table-expression (possibly of a degenerate form, e.g., as a column-expression), currently has the following structure:

```
SELECT scalar-expression-commalist
FROM   table-name-commalist
WHERE  predicate
```

(ignoring numerous irrelevant details). Notice that it is just _table-names_ that appear in the FROM clause. Completeness suggests that it should be _table-expressions_ (as Gray puts it [8], "anything in computer science that is not recursive is no good"). This is not just an academic consideration, by the way; on the contrary, there are several practical reasons as to why such recursiveness is desirable.

* First, consider the relational algebra. Relational algebra possesses the important property of closure -- that is, relations form a closed system under the operations of the algebra, in the sense that the result of applying any of those operations to any relation(s) is itself another relation. As a consequence, the operands of any given operation are not constrained to be real ("base") relations only, but rather can be any algebraic expression. Thus, the relational algebra allows the user to write _nested relational expressions_ -- and this feature is useful for precisely the same reasons that nested expressions are useful in ordinary arithmetic.

* Now consider SQL. SQL is a language that supports, directly or indirectly, all the operations of the relational algebra

(i.e., SQL is relationally complete). However, the table-expressions of SQL (which are the SQL equivalent of the expressions of the relational algebra) cannot be arbitrarily nested. Let us consider the question of exactly which cases SQL does support. Simplifying matters slightly, the expression SELECT - FROM - WHERE is the SQL version of the nested algebraic expression

```
projection ( restriction ( product ( table1, table2, ... ) ) )
```

(the product corresponds to the FROM clause, the restriction to the WHERE clause, and the projection to the SELECT clause; table1, table2, ... are the tables identified in the FROM clause -- and note that, as remarked earlier, these are simple table-names, not more complex expressions). Likewise, the expression

```
    SELECT ... FROM ... WHERE ...
    UNION
    SELECT ... FROM ... WHERE ...
      .....
```

is the SQL version of the nested algebraic expression

```
    union ( tabexp1, tabexp2, ... )
```

where tabexp1, tabexp2, ... are in turn table-expressions of the form shown earlier (i.e., projections of restrictions of products of named tables). But it is not possible to formulate direct equivalents of any other nested algebraic expressions. Thus, for example, it is not possible to write a direct equivalent in SQL of the nested expression

```
    restriction ( projection ( table ) )
```

Instead, the user has to recast the expression into a semantically equivalent (but syntactically different) form in which the restriction is applied before the projection. What this means in practical terms is that the user may have to expend time and effort transforming the "natural" formulation of a given query into some different, and arguably less "natural", representation (see Example below). What is more, the user is therefore also required to understand exactly when such transformations are valid. This may not always be intuitively obvious. For example, is a projection of a union always equivalent to the union of two projections?

Example: Given the two tables

```
    NYC ( EMP#, DEPT#, SALARY )
    SFO ( EMP#, DEPT#, SALARY )
```

(representing New York and San Francisco employees, respectively), list EMP# for all employees.

"Natural" formulation (projection of a union):

```
    SELECT EMP# FROM ( NYC UNION SFO )
```

SQL formulation (union of two projections):

```
    SELECT EMP# FROM NYC
    UNION
    SELECT EMP# FROM SFO
```

We remark in passing that allowing both formulations of the query would enable different users to perceive and express the same problem in different ways (ideally, of course, both formulations would translate to the same internal representation, for otherwise the choice between the two would no longer be arbitrary).

* The foregoing example tacitly makes use of the fact that a simple table-reference (i.e., a table-name) _ought_ to be just a special case of a general table-expression. Thus we wrote

```
    NYC UNION SFO
```

instead of

```
    SELECT * FROM NYC UNION SELECT * FROM SFO    ,
```

which current SQL would require. It would be highly desirable for SQL to allow the expression "SELECT * FROM T" to be replaced by simply "T" wherever it appears, in the style of more conventional languages. In other words, SELECT should be regarded as a _statement_ whose function is to retrieve a table (represented by a table-expression). Table-expressions per se -- in particular, _nested_ table-expressions -- should not require the "SELECT * FROM". Among other things this change would improve the usability of the EXISTS builtin function (see later). It would also be clear that INTO and ORDER BY are clauses of the SELECT _statement_ and not part of a table- (or column-) expression; the question of whether they can appear in a nested expression would then simply not arise, thus avoiding the need for a rule that looks arbitrary but is in fact not.

* A nested table-expression is permitted -- in fact required -- in current SQL as the argument to EXISTS (but strangely enough not as the argument to the other builtin functions; this point is discussed in the next section). Nested _column-expressions_ ("subqueries") are (a) _required_ with the "ANY" and "ALL" operators (includes the IN operator, which is just a different spelling for =ANY); and (b) _permitted_ with scalar comparison operators (<, >, =, etc.), if and only if the column-expression yields a column having at most one row. Moreover, the nested expression is allowed to include GROUP BY and HAVING in case (a) but not in case (b). More arbitrariness.

* Elsewhere I have proposed some extensions to SQL to support the outer join operation [4]. The details of that proposal do not concern us here; what does concern us is the following. If the user needs to compute an outer join of three or more relations, then (a) that outer join is constructed by performing a sequence of <u>binary</u> outer joins (e.g., join relations A and B, then join the result and relation C); and (b) it is essential that the user indicate the sequence in which those binary joins are performed, because different sequences will produce different results, in general. Indicating the required sequence is done, precisely, by writing a suitable nested expression. Thus, nested expressions are <u>essential</u> if SQL is to provide direct (i.e., single-statement) support for general outer joins of more than two relations.

* Another example (involving outer join again): Part of the proposal for supporting outer join [4] involves the use of a new clause, the PRESERVE clause, whose function is to preserve rows from the indicated table that would not otherwise participate in the result of the SELECT. Consider the tables

```
COURSE   ( COURSE#, SUBJECT )
OFFERING ( COURSE#, OFF#, LOCATION )
```

and consider the query "List all algebra courses, with their offerings if any". The two SELECT statements following (neither of which is valid in current SQL, of course) represent two attempts to formulate this query:

```
SELECT   ALGEBRA.COURSE#, OFF#, LOCATION
FROM   ( SELECT COURSE#
         FROM   COURSE
         WHERE  SUBJECT = 'Algebra' ) ALGEBRA, OFFERING
WHERE    ALGEBRA.COURSE# = OFFERING.COURSE#
PRESERVE ALGEBRA

SELECT   COURSE.COURSE#, OFF#, LOCATION
FROM     COURSE, OFFERING
WHERE    COURSE.COURSE# = OFFERING.COURSE#
AND      SUBJECT = 'Algebra'
PRESERVE COURSE
```

Each of these statements does list all algebra courses, together with their offerings, for all such courses that do have any offerings. The first also lists algebra courses that do not have any offerings, concatenated with null values in the OFFERING positions; i.e., it preserves information for those courses (note the introduced name ALGEBRA, which is used to refer to the result of evaluating the inner expression). The second, by contrast, preserves information not only for <u>algebra</u> courses with no offerings, <u>but also for all courses for which the subject is not algebra</u> (regardless of whether those courses have any offerings or not). In other words, the first preserves information for algebra courses only (as required), the second produces a lot of unnecessary output. And note that the first cannot even be formulated (as a single statement) if nested expressions are not supported.

* In fact, SQL does already support nested expressions in a kind of "under the covers" sense. Consider the following example:

Base table:

```
S ( S#, SNAME, STATUS, CITY )
```

View definition:

```
CREATE VIEW LONDON_SUPPLIERS
    AS SELECT S#, SNAME, STATUS
       FROM   S
       WHERE  CITY = 'London'
```

Query (Q):

```
SELECT *
FROM   LONDON_SUPPLIERS
WHERE  STATUS > 50
```

Resulting SELECT statement (Q'):

```
SELECT S#, SNAME, STATUS
FROM   S
WHERE  STATUS > 50
AND    CITY = 'London'
```

The SELECT statement Q' is obtained from the original query Q by a process usually described as "merging" -- statement Q is "merged" with the SELECT in the view definition to produce statement Q'. To the naive user this looks a little bit like magic. But in fact what is going on is simply that the reference to LONDON_SUPPLIERS in the FROM clause in Q is being replaced by the expression that _defines_ LONDON_SUPPLIERS, as follows:

```
SELECT *
FROM ( SELECT S#, SNAME, STATUS
       FROM   S
       WHERE  CITY = 'London' )
WHERE  STATUS > 50
```

This explanation, though both accurate and easy to understand, cannot conveniently be used in describing or teaching SQL, precisely because SQL does not support nesting at the external or user's level.

* UNION is not permitted in a subquery, and hence (among other things) cannot be used in the definition of a view (although

strangely enough it can be used to define the scope for a cursor in embedded SQL). So a view cannot be "any derivable relation", and the relational closure property breaks down. Likewise, INSERT ... SELECT cannot be used to assign the union of two relations to another relation. Yet another consequence of the special treatment given to UNION is that it is not possible to apply a builtin function such as AVG to a union. See the following section.

We conclude this discussion of SQL expressions by noting a few additional (and apparently arbitrary) restrictions.

* The predicate C BETWEEN A AND B is equivalent to the predicate A <= C AND C <= B -- except that B (but not A or C!) can be a column-expression (subquery) in the second formulation but not in the first.

* The predicate "field comparison (subquery)" must be written in the order shown and not the other way around; i.e., the expression "(subquery) comparison field" is illegal.

* If we regard SELECT, UPDATE, and INSERT all as special kinds of assignment statement -- in each case, the value of some expression is being assigned to some variable (a newly created variable, in the case of INSERT) -- then source values for those assignments can be specified as scalar-expressions (involving database fields, host variables, constants, and scalar operators) for SELECT and UPDATE, but must be specified as simple host variables or constants for INSERT. Thus, for example, the following is valid:

```
SELECT :X + 1
FROM   T
   ...
```

and so is:

```
UPDATE T
SET    F = :X + 1
   ...
```

but the following is not:

```
INSERT INTO T ( F )
       VALUES ( :X + 1 )
```

* Given the tables:

```
S ( S#, SNAME, STATUS, CITY )
P ( P#, PNAME, COLOR, WEIGHT, CITY )
```

the SELECT statement

```
    SELECT COLOR
    FROM   P
    WHERE  CITY =
         ( SELECT CITY
           FROM   P
           WHERE  P# = 'P1' )
```

is legal, but the UPDATE statement

```
    UPDATE P
    SET    COLOR = 'Blue'
    WHERE  CITY =
         ( SELECT CITY
           FROM   P
           WHERE  P# = 'P1' )
```

is not. Worse, neither is the UPDATE statement

```
    UPDATE P
    SET    CITY =
         ( SELECT CITY
           FROM   S
           WHERE  S# = 'S1' )
    WHERE  ...
```

Even worse, given:

```
    EMP     ( EMP#, SALARY )
    BONUSES ( EMP#, BONUS )
```

the following (potentially very useful) UPDATE is also illegal:

```
    UPDATE EMP
    SET    SALARY = SALARY + ( SELECT BONUS
                               FROM   BONUS
                               WHERE  EMP# =  EMP.EMP# )
```

(Actually there is a slight problem in this last example. Suppose a given employee number, say e, appears in the EMP table but not in the BONUSES table. Then the parenthesized expression will evaluate to null for employee e, and the UPDATE will therefore set e's salary to null as well -- whereas what is wanted is clearly for e's salary to remain unchanged. To fix this problem, we need to replace the parenthesized expression by (say)

```
    ROW_MAX ( ( SELECT BONUS ... EMP.EMP# ) , 0 )
```

where ROW_MAX is a function that operates by (a) ignoring any of its arguments that evaluate to null and then (b) returning the maximum of those that are left, if any, or null otherwise. Note that ROW_MAX is different in kind from the builtin functions currently provided in SQL -- it is in fact a scalar-valued function, whose arguments are scalar-expressions.)

3. LACK OF ORTHOGONALITY: BUILTIN FUNCTIONS

Frankly, there is so much confusion in this area that it is difficult to criticize it coherently. The basic point, however, is that the argument to a function such as SUM is a column of scalar values and the result is a single scalar value; hence, orthogonality dictates that (a) any column-expression should be permitted as the argument, and (b) the function-reference should be permitted in any context in which a scalar can appear. However, (a) the argument is in fact specified in a most unorthodox manner, which means in turn that (b) function references can actually appear only in a very small set of special-case situations. In particular, function-references cannot appear nested inside other function-references. In addition to this fact, functions are subject to a large number of peculiar and apparently arbitrary restrictions.

Before getting into details, we should point out that SQL in fact supports two distinct categories of function, not however in any uniform syntactic style. We refer to the two categories informally as _column_ and _table_ functions, respectively. We discuss each in turn.

Column functions

Column functions are the ones that one usually thinks of whenever functions are mentioned in connexion with SQL. A column function is a function that reduces an entire column of scalar values to a single value. The functions in this category are COUNT (excluding COUNT(*)), SUM, AVG, MAX, and MIN. A functional notation is used to represent these functions; however, as suggested above, the scoping rules for representing the argument are somewhat unconventional. Consider the following database (suppliers and parts):

```
S  ( S#, SNAME, STATUS, CITY )
P  ( P#, PNAME, COLOR, WEIGHT, CITY )
SP ( S#, P#, QTY )
```

and consider also the following query:

```
SELECT SUM (QTY)
FROM   SP
```

The argument to SUM here is in fact the entire column of QTY values in table SP, and a more conventional representation would accordingly be:

```
SUM ( SELECT QTY
      FROM   SP )
```

(though once again the keyword SELECT seems rather obtrusive; QTY FROM SP, or -- even better -- simply SP.QTY, would be more orthodox). As another example, the query:

```
   SELECT SUM (QTY)
   FROM   SP
   WHERE  P# = 'P2'
```

would more conventionally be represented as

```
   SUM ( SELECT QTY
         FROM   SP
         WHERE  P# = 'P2' )
```

or (better) as:

```
   SUM ( SP.QTY WHERE SP.P# = 'P2' )
```

As it is, the argument has to be determined by reference to the context. An immediate consequence of this fact is that a query such as "Find parts supplied in a total quantity of more than 1000" cannot be expressed in a natural style. First, the syntax:

```
   SELECT P#
   FROM   SP
   WHERE  SUM (QTY) > 1000
```

clearly does not work, either with SQL's rules for argument scope or with any other rules. The most logical formulation (but retaining a SQL-like style) is:

```
   SELECT DISTINCT SPX.P#
   FROM   SP SPX
   WHERE  SUM ( SELECT QTY
                FROM   SP SPY
                WHERE  SPY.P# = SPX.P# )
          > 1000
```

(The DISTINCT is required because of SQL's rules concerning duplicate elimination.) However, the normal SQL formulation would be:

```
   SELECT P#
   FROM   SP
   GROUP  BY P#
   HAVING SUM (QTY) > 1000
```

Note that the user is not really interested in grouping per se in this query; by writing GROUP BY, he or she is in effect telling the system how to execute the query, which is counter to the general philosophy of the relational model. To put this another way, the statement begins to look more like a prescription for solving the problem, rather than a simple description of what the problem is.

More important, it is necessary to introduce the HAVING clause, the justification for which is not immediately apparent to the user ("Why can't I use a WHERE clause?"). The HAVING clause --

and the GROUP BY clause also, come to that (see later) -- are needed in SQL only as a consequence of the column-function argument scoping rules. As a matter of fact, it is possible to produce a SQL formulation of this example that does not use GROUP BY or HAVING at all, and is fairly close to "the most logical formulation" suggested earlier:

```
   SELECT DISTINCT P#
   FROM   SP SPX
   WHERE  1000 <
        ( SELECT SUM (QTY)
          FROM   SP SPY
          WHERE  SPY.P# = SPX.P# )
```

As mentioned earlier, current SQL requires the predicate in the outer WHERE clause to be written as shown (i.e., in the order "constant - comparison - (subquery)", instead of the other way around).

An important consequence of all of the foregoing is that SQL cannot support arbitrary retrievals on arbitrary views. Consider the following example.

View definition:

```
   CREATE VIEW PQ ( P#, TOTQTY )
       AS SELECT P#, SUM (QTY)
          FROM   SP
          GROUP  BY P#
```

Attempted query:

```
   SELECT *
   FROM   PQ
   WHERE  TOTQTY > 1000
```

This query fails (it is syntactically invalid), because the "merging" process described earlier leads to something like the following:

```
   SELECT P#, SUM (QTY)
   FROM   SP
   WHERE  SUM (QTY) > 1000
   GROUP  BY P#
```

and this is not a legal SELECT statement. Likewise, the attempted query:

```
   SELECT AVG (TOTQTY)
   FROM   PQ
```

also does not work, for similar reasons.

The following is another striking example of the unobviousness of the scoping rules. Consider the following two queries:

```
SELECT SUM (QTY)                   SELECT SUM (QTY)
FROM   SP                          FROM   SP
                                   GROUP  BY P#
```

In the first case, the query returns a single value; the argument to the SUM invocation is the entire QTY column. In the second case, the query returns multiple values; the SUM function is invoked multiple times, once for each of the groups created by the GROUP BY clause. Notice how the meaning of the syntactic construct "SUM(QTY)" is dependent on context. In fact, SQL is moving out of the strict tabular framework of the relational model in this second example and introducing a new kind of data object, viz. _a set of tables_ (which is of course not the same thing as a table at all). GROUP BY converts a table into a set of tables. In the example, SUM is then applied to (a column within) each member of that set. A more logical syntax might look something like the following:

```
APPLY ( SUM, SELECT QTY
             FROM ( GROUP SP BY P# ) )
```

where "GROUP SP BY P#" produces the set of tables, "SELECT QTY FROM (...)" extracts a corresponding set of columns, and APPLY applies the function specified as its first argument to each column in the set of columns specified as its second argument, producing a set of scalars -- i.e., another column. (I am not suggesting a concrete syntax here, only indicating a possible direction for a systematic development of such a syntax.)

As a matter of fact, GROUP BY would be logically unnecessary in the foregoing example anyway if column function invocations were more systematic:

```
SELECT DISTINCT SPX.P#, SUM ( SELECT QTY
                              FROM   SP SPY
                              WHERE  SPY.P# = SPX.P# )
FROM   SP SPX
```

This formulation also shows, incidentally, that it might be preferable to declare aliases (range variables) such as SPX and SPY by means of separate statements before they are used. As it is, the use of such variables may often precede their definition, possibly by a considerable amount. Although there is nothing logically wrong with this, it does make the statements difficult to read (and write).

Yet another consequence of the scoping rules (already touched on a couple of times) is that it is not possible to nest column function references. Extending the earlier example of generating the total quantity per part (i.e., a column of values, each of which is a total quantity), suppose we now wanted to find the _average_ total quantity per part -- i.e., the average of that column of values. The logical formulation is something like:

```
   AVG ( APPLY ( SUM, SELECT QTY
                      FROM ( GROUP SP BY P# ) ) )
```

But (as already stated) existing SQL cannot handle this problem at all in a single expression.

Let us now leave the scoping rules and consider some additional points. Each of SUM, AVG, MAX, and MIN can optionally have its argument qualified by the operator DISTINCT. (COUNT _must_ have its argument so qualified, though it would seem that there is no intrinsic justification for this requirement. For MAX and MIN such qualification is legal but has no semantic effect.) If (and only if) DISTINCT is _not_ specified, then the column argument can be a "computed" column, i.e., the result of an arithmetic expression -- for example:

```
   SELECT AVG ( X + Y )
   FROM   T
      ...
```

And (again) if and only if DISTINCT is _not_ specified, the function reference can itself be an operand in an arithmetic expression -- for example:

```
   SELECT AVG ( X ) * 3
   FROM   T
      ...
```

In current SQL, null values are always eliminated from the argument to a column function, regardless of whether DISTINCT is specified. However, this should be regarded as a property of the existing functions specifically, rather than as a necessary property of all column functions. In fact, it would be better to _not_ to ignore nulls but to introduce a new function whose effect is to reduce a given column to another in which nulls have been eliminated (and, of course, to allow this new function to be used completely orthogonally).

Table functions

Table functions are functions that operate on an entire table (not necessarily just on a single column). There are four functions in this category, two that return a scalar value and two that return another table. The two that return a single value are COUNT(*) and EXISTS.

* COUNT(*) is basically very similar to the column functions discussed above. Thus, most of the comments made above apply here also. For example, the query:

```
   SELECT COUNT(*)
   FROM   SP
```

would more logically be expressed as

```
   COUNT ( SELECT *
           FROM SP )
```

or (better) as:

```
   COUNT ( SP )
```

COUNT(*) does *not* ignore nulls (i.e., all-null rows) in its argument.

* EXISTS, interestingly enough, does use a more logical syntax. For example:

```
   SELECT *
   FROM   S
   WHERE  EXISTS
        ( SELECT *
          FROM   SP
          WHERE  SP.S# = S.S# )
```

-- though the EXISTS argument would look better if the "SELECT * FROM" could be elided:

```
   SELECT *
   FROM   S
   WHERE  EXISTS ( SP WHERE SP.S# = S.S# )
```

or (better still):

```
   S WHERE EXISTS ( SP WHERE SP.S# = S.S# )    .
```

EXISTS takes a table as its argument (though that table *must* be expressed as a SELECT-expression, not just as a table-name) and returns the value *true* if that table is nonempty, *false* otherwise. Because there is currently no BOOLEAN or BIT data type in SQL, EXISTS can be used only in a WHERE clause, not (e.g.) in a SELECT clause (lack of orthogonality once again).

Now we turn to the functions that return another table, viz. DISTINCT and UNION.

* DISTINCT takes a table and returns another which is a copy of that first table except that redundant duplicate rows have been removed (rows that are entirely null are considered as duplicates of each other in this process -- that is, the result will contain at most one all-null row). Once again the syntax is unconventional. For instance:

```
   SELECT DISTINCT S#
   FROM   SP
```

instead of:

```
   DISTINCT ( SELECT S#
              FROM   SP )
```

or (better):

```
   DISTINCT ( SP.S# )
```

There is an apparently arbitrary restriction that DISTINCT may appear at most once in any given SELECT statement.

* UNION takes two tables (each of which must be represented by means of a SELECT-expression, not just as a simple table-name) and produces another table that is their union. It is written as an infix operator. Because of the unorthodox syntax, it is not possible (as mentioned before) to apply a column function such as AVG to a union of two columns.

Note: We consider UNION, alone of the operators of the relational algebra, as a function in SQL merely because of the special syntactic treatment it is given. SQL is really a hybrid of the relational algebra and the relational calculus; it is not precisely the same as either, though it leans somewhat toward the calculus -- a dialect of the calculus that does not lend itself very neatly to support of UNION, however, which is precisely why the special treatment is necessary.

4. LACK OF ORTHOGONALITY: MISCELLANEOUS ITEMS

Indicator variables

Let F be a database field that can accept null values, and let HF be a corresponding host variable, with associated indicator variable HN. Then:

```
   SELECT F
   INTO   :HF:HN
     ...
```

is legal, and so are

```
   INSERT ...
   VALUES ( :HF:HN ... )
```

and

```
   UPDATE ...
   SET    F = :HF:HN
    ...
```

But the following is not:

```
   SELECT ... ( or UPDATE or DELETE )
     ...
   WHERE  F = :HF:HN
```

References to current data

Let C be a cursor that currently identifies a record of table T. Then it is possible to designate the "CURRENT OF C" -- i.e., the record currently identified by C -- as the target of an UPDATE or DELETE statement, e.g., as follows:

```
   UPDATE T
   SET    ...
   WHERE  CURRENT OF C
```

Incidentally, a more logical formulation would be

```
   UPDATE CURRENT OF C
   SET    ...
```

Specifying the table-name T is redundant (this point is recognized in the syntax of FETCH, see later), and in any case "CURRENT OF C" is not the same kind of construct as the more usual WHERE-predicate (e.g., "SALARY > 20000"). Nor is it permitted to combine "CURRENT OF C" with other predicates and write (e.g.) "WHERE CURRENT OF C AND SALARY > 20000". But to return to the main argument: Although the (first) UPDATE statement above is legal, the analogous SELECT statement

```
SELECT ...
FROM   T
WHERE  CURRENT OF C
```

is not. Nor can fields within the "CURRENT OF C" be directly referenced -- e.g., the following is also illegal:

```
SELECT *
FROM   EMP
WHERE  DEPT# =
     ( SELECT DEPT#
       FROM   DEPT
       WHERE  CURRENT OF D )
```

Turning now to the FETCH statement, we have here an example of bundling. "FETCH C INTO ..." is effectively a shorthand for a sequence of two distinct operations --

```
STEP C TO NEXT
SELECT * INTO ... WHERE CURRENT OF C
```

-- the first of which (STEP) advances C to the next record in T in accordance with the ordering associated with C, and the second of which (SELECT) then retrieves that record. As noted above, that SELECT does not logically require any FROM clause. Replacing the FETCH statement by two more primitive statements in this way would have the following advantages:

(a) it is clearer;

(b) it is a more logical structure (incidentally, "FETCH C" does not really make intuitive sense -- it is not the _cursor_ that is being fetched);

(c) it would allow SELECTs of individual fields of the current record (i.e., "SELECT field-name" as well as "SELECT *");

(d) it would allow selective (and repeated) access to that current record (e.g., "SELECT F" followed by "SELECT G", both selecting fields of the same record);

(e) it would be extendable to other kinds of STEP operation -- e.g., STEP C TO PREVIOUS (say).

In fact I would go further. First, note that "CURRENT OF C" is an example of a row-expression. Let us therefore introduce a (new) FETCH statement, whose argument is a row-expression (as opposed to SELECT, whose argument is a table-expression), and whose function is to retrieve the row represented by that expression. Next, outlaw SELECT where FETCH is really intended. Next, introduce "(row-expression).field-name" -- e.g., (CURRENT OF C).F -- as a new form of scalar-expression. Finally, support all of these constructs orthogonally. Thus, for example, all of the following would be legal:

```
   FETCH CURRENT OF C INTO ...

   FETCH (CURRENT OF C).F INTO ...

   SELECT *
   FROM   EMP
   WHERE  DEPT# = (CURRENT OF C).DEPT#

   UPDATE CURRENT OF C
   SET    ...

   DELETE CURRENT OF C
```

The examples illustrate the point that "CURRENT OF C" is really a very clumsy notation, incidentally, but an improved syntax is beyond the scope of this paper. See [5] for a preferable alternative.

ORDER BY in cursor declaration

Specifying ORDER BY in the declaration of cursor C means that the statements UPDATE/DELETE ... CURRENT OF C are illegal (in fact, the declaration of C cannot include a FOR UPDATE clause if ORDER BY is specified). The rationale for this restriction is that ORDER BY may cause the program to operate on a copy instead of on the actual data, and hence that updates and deletes would be meaningless; but the restriction is unfortunate, to say the least. Consider a program that needs to process employees in department number order and needs to update some of them as it goes. The user is forced to code along the following lines:

```
   EXEC SQL DECLARE C CURSOR FOR
                      SELECT EMP#, DEPT#, ...
                      FROM   EMP
                      ORDER BY DEPT# ;

   EXEC SQL OPEN C ;
   DO WHILE more-to-come ;
      EXEC SQL FETCH C INTO :EMP#, :DEPT#, ... ;
      if this record needs updating, then
      EXEC SQL UPDATE EMP
           SET    ...
           WHERE  EMP# = :EMP# /* instead of CURRENT OF C */ ;
   END ;
   EXEC SQL CLOSE C ;
```

The UPDATE statement here is an "out-of-the-blue" UPDATE, not the CURRENT form. Problems:

(a) The update will be visible through cursor C if and only if C is running through the real data, not a copy.

(b) If cursor C is running through the real data, and if the UPDATE changes the value of DEPT#, the effect on the position of cursor C within the table is apparently undefined.

We remark also that the FOR UPDATE clause is a little mysterious (its real significance is not immediately apparent); it is also logically unnecessary. The whole of this area smacks of a most unfortunate loss of physical data independence.

The NULL constant

The keyword NULL may be regarded as a "builtin constant", representing the null value. However, it cannot appear in all positions in which a scalar constant can appear. For example, the statement

```
SELECT F, NULL
FROM   T
```

is illegal. This is unfortunate, since the ability to select NULL is precisely what is required in order to construct an outer join (in the absence of direct support for such an operation). See [4].

Empty sets

Let T be a table-expression. If T happens to evaluate to an empty set, then what happens depends on the context in which T appears. For example, consider the expressions

```
SELECT SALARY            and          SELECT AVG (SALARY)
FROM   EMP                            FROM   EMP
WHERE  DEPT# = 'D3'                   WHERE  DEPT# = 'D3'
```

and suppose that department D3 currently has no employees. Note that the second of these expressions represents the application of the AVG function to the result of the first; as pointed out earlier, it would more logically be written as

```
                                      AVG (SELECT SALARY
                                           FROM   EMP
                                           WHERE  DEPT# = 'D3')
```

* The statement

```
EXEC SQL SELECT SALARY
         INTO   :S:SN
         FROM   EMP
         WHERE  DEPT# = 'D3' ;
```

gives "not found" (SQLCODE = +100, host variables S and SN unchanged).

* The statement

```
    EXEC SQL SELECT AVG (SALARY)
             INTO   :S:SN
             FROM   EMP
             WHERE  DEPT# = 'D3' ;
```

sets host variable SN to an unspecified negative value to indicate that the value of the expression is null. The effect on host variable S is unspecified.

* The statement

```
    EXEC SQL SELECT ...
             INTO   :S:SN
             FROM   ...
             WHERE  field IN
                  ( SELECT SALARY
                    FROM   EMP
                    WHERE  DEPT# = 'D3' ) ;
```

gives "not found" (at the _outer_ level).

* The statement

```
    EXEC SQL SELECT ...
             INTO   :S:SN
             FROM   ...
             WHERE  field =
                  ( SELECT SALARY
                    FROM   EMP
                    WHERE  DEPT# = 'D3' ) ;
```

also gives "not found" (at the outer level), though there is a good argument for treating this case as an error, as follows: The parenthesized expression "(SELECT SALARY ...)" should really be regarded as a shorthand for "UNIQUE (SELECT SALARY ...)", where UNIQUE is a quantifier (analogous to EXISTS) meaning "there exists _exactly one_" -- or, in other words, a function whose effect is to return the single element from a singleton set and to raise an error if that set does not in fact contain exactly one member. Note that an error _would_ be raised in the example if the parenthesized expression yielded a set having more than one member (which in general, of course, it would).

* The statement

```
    EXEC SQL SELECT ...
             INTO   :S:SN
             FROM   ...
             WHERE  field =
                  ( SELECT AVG (SALARY)
                    FROM   EMP
                    WHERE  DEPT# = 'D3' ) ;
```

also gives "not found" at the outer level.

<u>Inconsistent syntax</u>

Compare the following:

```
   SELECT * FROM T ...

   UPDATE        T ...

   DELETE   FROM T ...

   INSERT   INTO T ...

 ( FETCH   C   ... )
```

A more consistent approach would be to define "table-expressions" (as suggested earlier), and then to recognize that SELECT, UPDATE, etc., are each operators, one of whose arguments is such a table-expression. (A problem that immediately arises is that a simple table-name is currently <u>not</u> a valid table-expression! -- i.e., instead of being able to write simply T, the user has to write SELECT * FROM T. This point has been mentioned before, and is of course easily remedied.)

Note too that the syntax UPDATE T SET F = ... does not extend very nicely to a form of UPDATE in which an entire record is replaced en bloc (SET * = ... ?). And this touches on yet another point, viz: SQL currently provides whole-record SELECT (and FETCH) and INSERT operators, but no whole-record UPDATE operator. (DELETE of course <u>must</u> be "whole-record".)

<u>Long fields (LONG VARCHAR, or VARCHAR(n) with n ≥ 254)</u>

Long fields are subject to numerous restrictions. Here are some of them (this may or may not be an exhaustive list). A long field:

- cannot be referenced in a predicate

- cannot be indexed

- cannot be referenced in SELECT DISTINCT

- cannot be referenced in GROUP BY

- cannot be referenced in ORDER BY

- cannot be referenced in COUNT, MAX, MIN (note: SUM and AVG would make no sense)

- cannot be involved in a UNION

- cannot be involved in a "subquery" (column-expression)

- cannot be INSERTed from a constant or SELECT-expression

- cannot be UPDATEd from a constant (UPDATE from NULL _is_ legal, however)

UNION restrictions

UNION is not permitted on long fields or in a subquery (in particular, in a view definition). Also, the data types of corresponding items in a UNION must be _exactly_ the same:

- if the data type is DECIMAL(p,q), then p must be the same for both items and q must be the same for both items

- if the data type is CHAR(n), then n must be the same for both items

- if the data type is VARCHAR(n), then must be the same for both items

- if NOT NULL applies to either item, then it must apply to both

Given these restrictions, it is particularly unfortunate that a character string constant such as 'ABC' is treated as a _varying_ length string -- a varying string, moreover, for which _nulls are allowed_.

Note also that UNION always eliminates duplicates. There is no "DISTINCT/ALL" option as there is with a simple SELECT; and if there were, the default would have to be DISTINCT (for compatibility reasons), whereas the default for a simple SELECT is ALL.

GROUP BY restrictions

GROUP BY:

- only works to one level (it can construct a "set of tables" but not a "set of sets of tables", etc.)

- can only have simple fields as arguments (unlike ORDER BY)

The fact is, as indicated in the discussion of functions earlier, an orthogonal treatment of GROUP BY would require a thorough treatment of an entirely new kind of data object, namely the "set of tables" -- presumably a major undertaking.

NULL anomalies

* Null values are implemented by hidden fields in the database. However, it is necessary to expose those fields in the interface to a host language such as PL/I, because PL/I has no notion of null. As an example, if F and G are two fields in table T, the

UPDATE statement to set F equal to G is:

```
   EXEC SQL UPDATE T
            SET    F = G ...
```

but the UPDATE statement to set F equal to a host variable H is (for instance):

```
   EXEC SQL UPDATE T
            SET    F = :H:HN ...
```

(assuming in both cases that the source of the assignment might be null).

* Indicator variables are not permitted in all contexts where host variables can appear (as already discussed).

* To test (in a WHERE clause) whether a field is null, SQL provides the special comparison "field IS NULL". It is not intuitively obvious why the user has to write "field IS NULL" and not "field = NULL" -- especially as the format "field = NULL" _is_ used in the SET clause of the UPDATE statement to update a field to the null value. (In fact, the WHERE clause "WHERE field = NULL" is illegal syntax.)

* Null values are considered as duplicates of each other for the purposes of UNIQUE and DISTINCT and ORDER BY but not for the purposes of WHERE and GROUP BY. Null values are also considered as greater than all nonnull values for the purposes of ORDER BY but not for the purposes of WHERE.

* Null values are always eliminated from the argument to a builtin function such as SUM or AVG, regardless of whether DISTINCT is specified in the function reference -- except for the case of COUNT(*), which counts all rows, including duplicates and including all-null rows. Thus, for example, given:

```
   SELECT AVG (STATUS) FROM S   --  Result: x

   SELECT SUM (STATUS) FROM S   --  Result: y

   SELECT COUNT(*)     FROM S   --  Result: z
```

there is no guarantee that x = y/z.

* As a consequence of the foregoing, the function reference SUM(F) (for example) is _not_ semantically equivalent to the expression

```
   f1 + f2 + .. + fn
```

where f1, f2, ..., fn are the values appearing in field F at the time of evaluation. Perhaps even more counterintuitively, the expression

```
SUM (F1 + F2)
```

is not equivalent to the expression

```
SUM (F1) + SUM (F2)    .
```

Host variables

Host variables are permitted in the INTO clause (of SELECT and FETCH), the SET clause (of UPDATE), and the WHERE clause (of SELECT, UPDATE, and DELETE), but nowhere else. In particular, table-names and field-names cannot be represented by host variables.

Introduced names

The user can introduce names (aliases) for tables (e.g., FROM T TX) but not for scalars (e.g., SELECT F FX). This latter facility would be particularly useful when the scalar is in fact represented as an operational expression -- e.g., SELECT A+B C. The name C could be used in ORDER BY or in GROUP BY or as an inherited name in CREATE VIEW (etc., etc.).

Legal INSERTs/UPDATEs/DELETEs

Certain INSERT, UPDATE, and DELETE statements are not allowed. For example, consider the requirement "Delete all suppliers with a status less than the average". The statement:

```
DELETE
FROM   S
WHERE  STATUS <
     ( SELECT AVG (STATUS)
       FROM   S )
```

is illegal, because the FROM clause in the subquery refers to the table against which the deletion is to be done. Likewise, the UPDATE statement

```
UPDATE S
SET    STATUS = 0
WHERE  STATUS <
     ( SELECT AVG (STATUS)
       FROM   S )
```

is also illegal, for analogous reasons. Third, the statement

```
INSERT INTO T
       SELECT * FROM T
```

which might be regarded as a perfectly natural way to "double up" on the contents of a table T, is also illegal, again for analogous reasons.

5. FORMAL DEFINITION

As indicated earlier in this paper, it would be misleading to suggest that SQL does not possess a detailed definition. However, as was also indicated earlier, that definition [10] was produced "after the fact". In some respects, therefore, it represents a definition of the way implementations actually work rather than the way a "pure" language ought to be (although it must be said that many of the criticisms of the present paper have indeed been addressed in [10]). At the same time it provides definitive answers to some questions that are not in agreement with the way IBM SQL actually works! Furthermore, there still appear to be some areas where the definition is not yet precise enough. We give examples of all of these aspects below.

* Cursor positioning

Let C be a cursor that is currently associated with a set of records of type R. Suppose moreover that the ordering associated with C is defined by values of field R.F. If C is positioned on a record r and r is deleted, C goes into the "before" state -- i.e., it is now positioned "before" record r1, where r1 is the immediate successor of r with respect to the ordering associated with C -- or, if there is no such successor record, then it goes into the "after" state -- i.e., it is "after" the last record in the set (note: the "after" state is possible even if the set is empty).

Questions:

(a) If C is "before r1" and a new record r is inserted with a value of R.F such that r logically belongs between r1 and r1's predecessor (if any), what happens to C? [Answer: Implementation-defined.]

(b) Does it make a difference if the new record r logically precedes or follows the old record r that C was positioned on before that record was deleted? [Answer: Implementation-defined.]

(c) Does it make a difference if C was actually running through a copy of the real set of records? [Answer: Implementation-defined.]

Note for cases (a)-(c) that it _is_ guaranteed that the next "FETCH C" will retrieve record r1 (provided no other DELETEs etc. occur in the interim).

(d) What if the new r is not an INSERTed record but an UPDATEd record? [Answer: Not defined.]

(e) If C is positioned on a record r and the value of field F in that record is updated (not via cursor C, of course), what happens to C? [Answer: Not defined.]

* <u>LOCK statement</u>

Does LOCK SHARED acquire an S lock or an SIX lock [9]? If the answer is S, are updates permitted? When are locks acquired via LOCK TABLE released?

* <u>Name resolution</u>

First, consider the two statements:

```
    SELECT S#
    FROM   S
    WHERE  CITY = 'London'

    SELECT P#
    FROM   P
    WHERE  CITY = 'London'
```

The meaning of the unqualified name CITY depends on the context -- it is taken as S.CITY in the first of these examples and as P.CITY in the second. But now suppose the columns are renamed SCITY and PCITY respectively, so that now the names are globally unique, and consider the query "Find suppliers located in cities in which no parts are stored". The obvious formulation of this query is:

```
    SELECT S#
    FROM   S
    WHERE  NOT EXISTS
         ( SELECT *
           FROM   P
           WHERE  PCITY = SCITY )
```

However, this statement is invalid. SQL assumes that "SCITY" is shorthand for "P.SCITY", and then complains that no such field exists. The following statement, by contrast, is perfectly valid:

```
    SELECT S#
    FROM   S
    WHERE  NOT EXISTS
         ( SELECT *
           FROM   P
           WHERE  PCITY = S.SCITY )
```

So also is:

```
    SELECT S#
    FROM   S SX
    WHERE  NOT EXISTS
         ( SELECT *
           FROM   P
           WHERE  PCITY = SX.SCITY )
```

Is the following legal?

```
SELECT *
FROM   S
WHERE  EXISTS ( SELECT *
                FROM   SP SPX
                WHERE  SPX.S# = S.S#
                AND    SPX.P# = 'P1'
                AND    EXISTS ( SELECT *
                                FROM   SP SPX
                                WHERE  SPX.S# = S.S#
                                AND    SPX.P# = 'P2' ) )
```

What if "FROM SP SPX" is replaced by "FROM SP" (twice) and all other occurrences of "SPX" are replaced by "SP"? And is the following legal?

```
SELECT *
FROM   S
WHERE  EXISTS ( SELECT *
                FROM   SP SPX
                WHERE  SPX.S# = S.S#
                AND    SPX.P# = 'P1' )
AND    EXISTS ( SELECT *
                FROM   SP SPX
                WHERE  SPX.S# = S.S#
                AND    SPX.P# = 'P2' )
```

(etc., etc.). In other words: What _are_ the name scoping rules for "aliases" (range variables)?

There is another point to be made while on the subject of name resolution, incidentally. Consider the statement:

```
SELECT S.S#, P.P#
FROM   S, P
WHERE  S.CITY = P.CITY
```

(we now go back to the unqualified name CITY in each of the two tables). This statement is (conceptually) evaluated as follows:

- form the product of S and P; call the result TEMP1

- restrict TEMP1 according to the predicate S.CITY = P.CITY; call the result TEMP2

- project TEMP2 over the columns S.S# and P.P#

But how can this be done? The predicate "S.CITY = P.CITY" does not refer to any columns of TEMP1 (it refers to columns of S and P, obviously). Similarly, S.S# and P.P# are not columns of TEMP2. In order for these references to be interpreted appropriately, it is necessary to introduce certain _name inheritance rules_, indicating how result tables inherit column-names from their source tables (which may of course may themselves also be [intermediate] result tables, with inherited column-names of their own). Such rules are currently defined only very informally, if at all. Such rules become even more important if SQL is to provide support for nested expressions.

* _Base vs. copy data_

When exactly does a cursor iterate over the real "base data" and when over a copy?

* _Binding of "SELECT *"_

When exactly does "*" become bound to a specific set of field-names? [Answer: Implementation-defined -- but this seems an unfortunate aspect to leave to the implementer, especially as the binding is likely to be different for different _uses_ of the feature (e.g., it may depend on whether the "*" appears in a program or in a view definition).]

6. MISMATCH WITH HOST LANGUAGE

The general point here is that there are far too many frivolous distinctions between SQL and the host language in which it happens to be embedded; also that in some cases SQL has failed to benefit from lessons learned in the design of those host languages. Generally, orthogonality suggests that what is useful on one side of the interface (in the way of data structuring and access for "permanent" [i.e., database] data) is likely to be useful on the other side also (for "temporary" [i.e., local] data); thus, a distinct sublanguage is the wrong approach, and a two-level store is wrong too (fundamentally so!). Some specific points:

* SQL does not exploit the exception-handling capabilities of the host (e.g., PL/I ON-conditions). This point and (even more so) the following one mean that SQL does not exactly encourage the production of well-structured, quality programs, and that in some respects SQL programming is at a lower level than that of the host.

* SQL does not exploit the control structures of the host (loop constructs in particular). See the previous point.

* SQL objects (tables, cursors, etc.) are not known and cannot be referenced in the host environment.

* Host objects can be referenced in the SQL environment only if:

  - they are specially declared (may not apply to all hosts)

  - they are scalars or certain limited structures (in particular, they are not arrays)

  - the references are marked with a colon prefix (admittedly only in some contexts -- but in my opinion "some" is worse than "all")

  - the references are constrained to certain limited contexts (e.g., they can appear in a SELECT clause but not a FROM clause)

  - the references are constrained to certain limited formats (e.g., no subscripting, only limited dot qualification, etc.)

* SQL object names and host object names are independent and may clash. SQL names do not follow the scoping rules of the host.

* SQL keywords and host keywords are independent and may clash (e.g., PL/I SELECT vs. SQL SELECT).

* SQL and host may have different name qualification rules (e.g., T.F in SQL vs. F OF T in COBOL; and note that the SQL form must be used even for host object references in the SQL environment).

* SQL and host may have different data type conversion rules.

* SQL and host may have different expression evaluation rules (e.g., SQL division and varying string comparison differ from their PL/I analogs [at least in SQL/DS]).

* SQL and host may have different Boolean operators (AND, OR, and NOT in SQL vs. &, |, and ~ in PL/I).

* SQL and host may have different comparison operators (e.g., COBOL has IS NUMERIC, SQL has BETWEEN [and many others]).

* SQL imposes statement ordering restrictions that are alien to the host.

* SQL DECLARE cannot be abbreviated to DCL, unlike PL/I DECLARE.

* Null is handled differently on the two sides of the interface.

* Function references have different formats on the two sides of the interface.

* SQL name resolution rules are different from those of the host.

* Cursors are a clumsy way of bridging the gap between the database and the program. A much better method would be to associate a query with a conventional _sequential_ _file_ in the host program, and then let the program use conventional READ, REWRITE, and DELETE statements to access that file (maybe INSERT statements too).

* The "structure declarations" in CREATE TABLE should use the standard COBOL or PL/I (etc.) syntax. As it is, it is doubtful whether they can be elegantly extended to deal with minor structures (composite fields) or arrays, should such extensions ever prove desirable (they will).

* The SQL parameter mechanism is regressive, clumsy, ad hoc, restrictive, and different from that of the host.

7. MISSING FUNCTION

(Note: It is obviously possible to extend the existing language to incorporate most if not all of the following features. We mention them for completeness.)

* Ability to override WHENEVER NOT FOUND at the level of an individual statement.

* "Whole-record" UPDATE.

* Procedure call instead of GO TO on WHENEVER.

* Cursor stepping other than "next".

* Cursor comparison.

* Cursor assignment.

* Cursor constants.

* Cursor arrays.

* Dynamically created cursors and/or cursor stacks.

* Reusable cursors.

* Ability to access a unique record and keep a cursor on it without having to go through separate DECLARE, OPEN, and FETCH: e.g., "FETCH UNIQUE (EMP WHERE EMP# = 'E2') SET (C) ;".

* Fine control over locking.

8. MISTAKES

* Null values

I have argued against null values at length elsewhere [6], and I will not repeat those arguments here. In my opinion the null value concept is far more trouble than it is worth. Certainly it has never been properly thought through in the existing SQL implementations (see the discussion under "Lack of Orthogonality: Miscellaneous Items", earlier). For example, the fact that functions such as AVG simply ignore null values in their argument violates what should surely be a fundamental principle, viz: The system should never produce a (spuriously) precise answer to a query when the data involved in that query is itself imprecise. At least the system should offer the user the explicit option either to ignore nulls or to treat their presence as an exception.

* Unique indexes

Field uniqueness is a logical property of the data, not a physical property of an access path. It should be specified on CREATE TABLE, not on CREATE INDEX. Specifying it on CREATE INDEX is an unfortunate bundling, and may lead to a loss of data independence (dropping the index puts the integrity of the database at risk).

* FROM clause

The only function of the FROM clause that is not actually redundant is to allow the introduction of range variables, and that function would be better provided in some more elegant manner. (The normal use, as exemplified by the expression SELECT F FROM T, could better be handled by the expression SELECT T.F, especially since this latter expression -- with an accompanying but redundant FROM clause -- is already legal SQL.)

* Punning

SQL does not make a clear distinction between tables, record types, and range variables. Instead, it allows a single symbol to stand for any one of those objects, and leaves the interpretation to depend on context. Conceptual clarity would dictate that it at least be possible always to distinguish among these different constructs (i.e., syntactically), even if there are rules that allow such punning games to be played when intuitively convenient. Otherwise it is possible that -- for example -- extendability may suffer, though I have to admit that I cannot at the time of writing point to any concrete problems. (But it shouldn't be necessary to have to defend the principle of a one-to-one correspondence between names and objects!)

While on the subject of punning, I might also mention the point that SQL is ambivalent as to the meaning of the term "table".

Sometimes "table" means, specifically, a base table (as in CREATE TABLE); at other times it means "base table or view" (as in COMMENT ON TABLE). Since the critical point about a view is that it is a table (just as the critical point about a subset is that it is a set), I would vote for the following changes:

(a) Replace the terms "base table" and "view" by "real table" and "virtual table", respectively;

(b) Use the term "table" generically to mean "real table or virtual table";

(c) In concrete syntax, use the expressions [REAL] TABLE and VIRTUAL TABLE (where it is necessary to distinguish them), with REAL as the default.

* "SELECT *"

This is a good example of a situation in which the needs of the end-user and those of the application programmer are at odds. "SELECT *" is fine for the interactive user (it saves keystrokes). I believe it is rather dangerous for the programmer (because the meaning of "*" may change at any time in the life of the program). The use of "ORDER BY n" (where n is an integer instead of a field-name) in conjunction with "SELECT *" could be particularly unfortunate. Similar remarks apply to the use of INSERT without a list of field-names.

Incidentally, I believe that the foregoing are the only situations in the entire SQL language in which the user is dependent on the left-to-right ordering of columns within a table. It would be nice to eliminate that dependence entirely (except possibly for "SELECT *", for interactive queries only).

* =ANY (etc.)

The comparison operators =ANY, >ALL, etc., are totally redundant and in many cases actively misleading. The following example is taken from "IBM Database 2 SQL Usage Guide" (IBM Form No. GG24-1583): "Select employees who are younger than any member of department E21" (irrelevant details omitted).

```
SELECT EMPNO, LASTNAME, WORKDEPT
FROM   TEMPL
WHERE  BRTHDATE >ANY ( SELECT BRTHDATE
                       FROM   TEMPL
                       WHERE  WORKDEPT = 'E21' )
```

This SELECT does not find employees who are younger than any employee in E21 (at least in the sense that this requirement would normally be understood in colloquial English) -- it finds employees who are younger than some employee in E21.

To illustrate the redundancy, consider the query: "Find supplier names for suppliers who supply part P2". This is a very simple problem, yet it is not difficult to find no less than seven at least superficially distinct formulations for it (see below). Of course, the differences would not be important if all formulations worked equally well, but that is unlikely.

```
    1. SELECT SNAME
       FROM   S
       WHERE  S# IN
            ( SELECT S#
              FROM   SP
              WHERE  P# = 'P2')

    2. SELECT SNAME
       FROM   S
       WHERE  S# =ANY
            ( SELECT S#
              FROM   SP
              WHERE  P# = 'P2')

    3. SELECT SNAME
       FROM   S
       WHERE  EXISTS
            ( SELECT *
              FROM   SP
              WHERE  S# = S.S# AND P# = 'P2')

    4. SELECT DISTINCT SNAME
       FROM   S, SP
       WHERE  S.S# = SP.S# AND P# = 'P2')

    5. SELECT SNAME
       FROM   S
       WHERE  0 <
            ( SELECT COUNT(*)
              FROM   SP
              WHERE  S# = S.S# AND P# = 'P2')

    6. SELECT SNAME
       FROM   S
       WHERE  'P2' IN
            ( SELECT P#
              FROM   SP
              WHERE  S# = S.S# )

    7. SELECT SNAME
       FROM   S
       WHERE  'P2' =ANY
            ( SELECT P#
              FROM   SP
              WHERE  S# = S.S# )
```

In general, the WHERE clause

```
    WHERE x $ANY ( SELECT y FROM T WHERE p )
```

(where $ is any one of =, >, etc.) is equivalent to the WHERE clause

```
   WHERE EXISTS ( SELECT * FROM T WHERE (p) AND x $ T.y )
```

Likewise, the WHERE clause

```
   WHERE x $ALL ( SELECT y FROM T WHERE p )
```

is equivalent to the WHERE clause

```
   WHERE NOT EXISTS ( SELECT * FROM T WHERE (p)
                                 AND NOT ( x $ T.y ) )
```

As a matter of fact, it is not just the comparison operators =ANY (etc.) that are redundant; the entire subquery construct could be removed from SQL with effectively no loss of function. (Nested table- and column-expressions etc. would of course still be required, as argued earlier.) This is ironic, since it was the subquery notion that was the justification for the "Structured" in "Structured Query Language" in the first place.

9. ASPECTS OF THE RELATIONAL MODEL NOT SUPPORTED

There are several aspects of the full relational model (as defined in, e.g. [2]) that SQL does not currently support. We list them here in approximate order of importance. Again, of course, most of these features can be added to SQL at some later point -- the sooner the better, in most cases. However, their omission now leads to a number of situations in current SQL that are extremely ad hoc and may be difficult to remedy later on, for compatibility reasons.

* Primary keys

Primary keys provide the sole record-level addressing mechanism within the relational model. That is, the only system-guaranteed method of identifying an individual record is via the combination (R,k), where R is the name of the containing relation and k is the primary key value for the record concerned. Every relation (to be a relation) is required to have a primary key. Primary keys are (of course) required to be unique; in the case of real (base) relations, they are also required to be (wholly) nonnull.

SQL currently provides mechanisms that allow users to apply the primary key discipline for themselves (if they choose), but does not itself understand the semantics associated with that discipline. As a result, SQL support for certain other functions is either deficient or lacking entirely, as we now explain.

1. Consider the query

```
SELECT P.P#, P.WEIGHT, AVG (SP.QTY)
FROM   P, SP
WHERE  P.P# = SP.P#
GROUP  BY P.P#, P.WEIGHT
```

The "P.WEIGHT" in the GROUP BY clause is logically redundant, but must be included because SQL does not understand that P.WEIGHT is single-valued per part number (perhaps only a minor annoyance, but it could be puzzling to the user).

2. Primary key support is prerequisite to foreign key support (see the following subsection).

3. An understanding of primary keys is required in order to support the updating of views correctly. SQL's rules for the updating of views are in fact disgracefully ad hoc. We consider projection, restriction, and join views in turn below. Further discussion of this topic can be found in [7].

3(a). A projection is logically updatable if and only if it preserves the primary key of the underlying relation. However, SQL supports updates, not on projections per se, but on what might be called column subsets -- where a "column subset" is any subset of the columns of the underlying table for which duplicate elimination is not requested (via DISTINCT) -- with a "user beware" if that subset does not in fact include the underlying primary key. (Actually the situation is even worse than this. Even a column subset is not updatable if the FROM clause in the definition of that subset lists multiple tables. Moreover, updates are prohibited if duplicate elimination is requested, even if that request can have no effect because the column subset does include the underlying primary key.)

3(b). Any restriction is logically updatable. SQL however does not permit such updates if duplicate elimination is requested (even though such a request can have no effect if the underlying table does have a primary key), nor if the FROM clause lists multiple tables. What is more, even when it does allow updates, SQL does not always check that updated records satisfy the restriction predicate; hence, an updated (or inserted) record may instantaneously vanish from the view, and moreover there are concomitant security exposures (e.g., a user who is restricted to accessing employees with salary less than $40K may nevertheless _create_ a salary greater than that value via INSERT or UPDATE). [Note: The CHECK option, which is intended to prevent such abuses, cannot always be specified.] Also, the fact that SQL automatically supplies null values for missing fields in inserted records means that it is _impossible_ for such records to satisfy the restriction predicate in some cases (consider, for example, the view "employees in department D3", if the view does not include the DEPT# field). However, these latter deficiencies are nothing to do with SQL's lack of knowledge of primary keys per se.

3(c). A join of two tables on their primary keys is logically updatable. So also is a join of one table on its primary key to another on a matching foreign key (though the details are not totally straightforward). However, SQL does not allow _any_ join to be updated.

* _Foreign keys_

Foreign keys provide the principal referencing mechanism within the relational model. Loosely speaking, a foreign key is a field in one table whose values are required to match values of the primary key in another table. For example, field DEPT# of the EMP table is a foreign key matching the primary key (DEPT#) of the DEPT table.

SQL does not currently provide any kind of support for the foreign key concept at all. I regard lack of such support as the major deficiency in relational systems today (SQL is certainly not alone in this regard). Proposals for such support are documented in some detail in [7].

* Domains

SQL currently provides no support for domains at all, except inasmuch as the fundamental data types (INTEGER, FLOAT, etc.) can be regarded as a very primitive kind of domain.

* Relation assignment

A limited form of relation assignment is supported via INSERT ... SELECT, but that operation does not overwrite the previous content of the target table, and the source of the assignment cannot be an arbitrary algebraic expression (or SQL equivalent).

* Explicit JOIN

We mentioned earlier that explicit support for the (natural) join operation was desirable. At that point we were tacitly discussing the *inner* or regular natural join. The observation is still more applicable to *outer* join. Reference [4] shows how awkward it is to extend the circumlocutory SELECT-style join to handle outer joins. Thus, support for an explicit JOIN operator is likely to become even more desirable in the future than it is already.

* Explicit INTERSECT and DIFFERENCE

These omissions are not particularly important (equivalent SELECT-expressions exist in each case); however, symmetry would suggest that, since UNION *is* explicitly supported, INTERSECT and DIFFERENCE ought to be explicitly supported too. Some problems are most "naturally" formulated in terms of explicit intersections and differences. On the other hand, as indicated earlier, it is usually not a good idea to provide a multiplicity of equivalent ways of formulating the same problem, unless it can be guaranteed that the implementation will recognize the equivalences and will treat all formulations equally, which is probably unlikely.

10. SUMMARY AND CONCLUSIONS

This paper has discussed a large number of deficiencies in the SQL language as currently defined, in the hope that such a discussion can serve as a step toward remedying those deficiencies. In fact (as remarked earlier), the ANS Database Committee (X3H2) has already remedied some of them in its "RDL" proposal; a secondary objective for the present paper is thus to serve as a document of justification for the changes X3H2 has already made.

Of course, I realize that many of the shortcomings identified in this paper will very likely be dismissed as academic, trivial, or unimportant by many people, especially as SQL is so clearly superior to older languages such as the DML of DBTG. However, experience shows that "academic" considerations have a nasty habit of becoming horribly practical a few years further down the road. The mistakes we make now will come back to haunt us in the future. Indeed, the language in its present form is already proving difficult to extend in some (desirable) ways because of limitations in its current structure. A very trivial example is provided by the problems of adding support for composite fields (i.e., minor structures).

In conclusion, let me repeat the point that many other database languages suffer from similar shortcomings; SQL is (as stated before) certainly not the sole offender. But the fact remains that, if SQL is adopted on a wide scale in its present form, then we will to some degree have missed the relational boat, or at least failed to capitalize to the fullest possible extent on the potential of the relational model. That would be a pity, because we had an opportunity to do it right, and with a little effort we could have done so. The question is whether it is now too late. I sincerely hope not.

ACKNOWLEDGMENTS

I am grateful to my friends and colleagues Ted Codd, Phil Shaw, and Sharon Weinberg for their helpful comments and criticism.

REFERENCES

1. M.M.Astrahan et al. "System R: Relational Approach to Database Management." ACM TODS 1, No. 2 (June 1976).

2. E.F.Codd. "Extending the Database Relational Model to Capture More Meaning." ACM TODS 4, No. 4 (December 1979).

3. C.J.Date. "Some Principles of Good Language Design." Submitted to ACM SIGMOD Record.

4. C.J.Date. "The Outer Join." Proc. 2nd International Conference on Databases (ICOD-2), Cambridge, England (August-September 1983).

5. C.J.Date. "An Introduction to the Unified Database Language (UDL)." Proc. 6th International Conference on Very Large Data Bases, Montreal, Canada (October 1980).

6. C.J.Date. "Null Values in Database Management" (invited paper). Proc. 2nd British National Conference on Databases (BNCOD-2), Bristol, England (July 1982).

7. C.J.Date. *A Guide to DB2.* Addison-Wesley (to appear 1984).

8. J.N.Gray. Private communication.

9. J.N.Gray et al. "Granularity of Locks in a Large Shared Data Base." Proc. 1st International Conference on Very Large Data Bases, Framingham, Mass. (September 1975).

10. X3H2 (American National Standards Database Committee). Draft Proposed Relational Database Language. Document X3H2-83-152 (August 1983).

APPENDIX: SQL STRONGPOINTS

Simple data structure

SQL is based on the relational model, and as such supports the simple tabular data structure of that model. It does **not** support any user-visible links between tables.

Powerful operators

SQL also supports (indirectly) all the operators of the relational algebra, including in particular the operators SELECT (i.e., RESTRICT), PROJECT, and (natural) JOIN (these are the ones required most often in practice). Each of these operators is **very high-level**, in the sense that it treats entire sets of records as single operands.

Short initial learning period

It is very easy to learn enough of the SQL language to "get on the air" and start doing real, useful work; thus, the initial learning period is typically very short indeed -- certainly hours rather than days or weeks.

Improved data independence

Users are insulated, to a greater degree than with earlier languages, from the physical structure of the database (physical data independence). This fact means that: (a) Users can concentrate on the logic of their application without having to concern themselves with irrelevant physical details; (b) the physical structure of the database can be changed without necessitating any corresponding reprogramming. Users are also insulated to some extent from the **logical** structure of the database (logical data independence); this means that users can concentrate on just that portion of the data that is of interest to them (they may not even be aware of other portions), and it also means that some limited changes can be made to the logical structure of the database without very much reprogramming (probably not without any, however).

Integrated data definition and data manipulation

SQL imposes comparatively few artificial boundaries between definition functions and manipulation functions. For example, the creation of a view (a definition function) involves essentially the same SELECT operation as does the formulation of a query (a manipulation function). This uniformity, again, makes the language easier to learn and use.

Double mode of use

SQL can be used both interactively (i.e., as a query language) and embedded in a program (i.e., as a database programming language). This property is desirable for several reasons. First, it improves communication: End-users and application programmers are "speaking the same language". Second, it makes programmers, as well as end-users, more productive -- the benefits sketched above (e.g., the provision of high-level operators) apply to programmers too. And third, the interactive interface provides a very convenient programmer debugging facility; that is, application programmers can take the SQL portions of their program and debug them interactively at the terminal.

Integrated catalog

Since the database catalog is represented just like any other data in the system (i.e., as a collection of tables), it can be interrogated by means of SQL SELECT statements, just like any other data in the system. Users do not have to learn two languages, one for querying the dictionary (for the catalog is in effect exactly that, a rudimentary, online, active dictionary), and one for querying the database.

Compilation and optimization

SQL is capable of efficient implementation, via the by now well-known compilation/optimization techniques pioneered in the IBM prototype System R. Moreover, the fact that SQL is compiled, and hence that systems such as System R are "early binding" systems, does not compromise the flexibility of those systems. If a change is made to the database (such as the dropping of an index) that invalidates an existing compiled program, then that program -- or, more accurately, the SQL statements within that program -- will automatically be recompiled and rebound on the next invocation. Thus the system can provide the flexibility of late binding without incurring the interpretation overheads normally associated with such systems.