NISTIR 4494

MARCH, 199

SQL3 Support for CALS Applications

Leonard Gallagher

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Computer Systems Laboratory
Database and Graphics Group
Gaithersburg, MD 20899

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

NIST

NISTIR 4494

SQL3 Support for CALS Applications

Leonard Gallagher

U.S. DEPARTMENT OF COMMERCE
National Institute of Standards and Technology
Computer Systems Laboratory
Database and Graphics Group
Gaithersburg, MD 20899

December 1990

Issued February 1991

U.S. DEPARTMENT OF COMMERCE
Robert A. Mosbacher, Secretary
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
John W. Lyons, Director

SQL3 Support for CALS Applications

by

Leonard Gallagher

Information Systems Engineering Division
National Computer Systems Laboratory
National Institute of Standards and Technology

December 21, 1990

FY90 CALS Task 5.2.3 "Work with SQL committees to inject CALS requirements into SQL3 and RDA"

Deliverable: Report on SQL3 requirements

- Abstract -

Previous reports to CALS have identified the importance of Database Language SQL in CALS Phase II requirements. In particular, a July 1989 point paper on SQL and RDA identified features in the existing SQL standard and its near-term SQL2 replacement that are most appropriate to CALS data management concerns. This report focuses on SQL3, a follow-on standardization project for major new SQL enhancements that is expected to be adopted by ANSI, ISO, and as a FIPS in the mid 1990's. Many of the proposed SQL3 features are of particular importance to the Standard for the Exchange of Product model data (STEP) because of that standard's unique data modeling and data access requirements. Existing and planned features in SQL3 may not satisfy all STEP requirements, but they should provide an appropriate base from which many requirements can be suitably addressed. Since features in SQL3 are just now being specified, they are open to modification and improvement to best suit CALS needs. This report identifies the major enhancements under consideration by the ANSI and ISO SQL standardization committees and relates them to known manufacturing and product management requirements. It also discusses the status of these features in the SQL3 specification and indicates opportunities available to CALS to influence further development.

- Table of Contents -

1. Introduction 1

2. Encapsulation 2

    2.1 Assertions 2
    2.2 Triggers 3
    2.3 External Procedures 4
    2.4 Temporary Tables and Views 6

3. Data Abstraction 7

    3.1 String Operations 7
    3.2 Datetime Data Types 10
    3.3 Cast Function 11
    3.4 Simple Domains 11
    3.5 User-Defined Data Types 12

4. Inheritance 13

    4.1 Domain Hierarchies 13
    4.2 Table Hierarchies 14

5. Recursive Expressions 16

6. Existential and Universal Quantifiers 19

7. Roles and Data Security 20

8. Savepoints and Subtransactions 23

9. Distributed Database Management 24

    9.1 Remote Database Access 25
    9.2 Distributed Database Requirements 26

10. Database Export and Import 27

11. Other Proposed Features 28

Bibliography 33

1. Introduction

Previous reports to CALS [7, 14] have identified the importance of Database Language SQL in CALS Phase II requirements. In particular, a July 1989 point paper on SQL and RDA [14] identified features in the existing SQL standard and its near-term SQL2 replacement that are most appropriate to CALS data management concerns. Each previous report recognized that CALS requires a logically integrated database of diverse data (e.g. documents, graphics, alphanumeric records, images, voice, video) stored in geographically separated data banks under the management and control of heterogeneous data management systems. Yet, they also conclude that Database Language SQL is appropriate for the definition and management of a large class of this data, especially data that is structured into repeated occurrences having common data structure definitions.

A major conclusion in previous reports is that future database management systems need to support broader application areas such as CAD/CAM and graphics, which are critically important to STEP. One way to accomplish this is to extend database management technology to support object-oriented concepts currently receiving a high degree of acceptability in the programming language arena. The object-oriented concepts of encapsulation, data abstraction, and inheritance are emphasized in this report as desirable features in the emerging SQL3 enhancement of Database Language SQL.

Existing SQL and SQL2 Specifications

Database Language SQL was first adopted in 1986 by the American National Standards Institute (ANSI). Later, in 1987, an identical specification was adopted by the International Organization for Standardization (ISO) and by NIST as a Federal Information Processing Standard (FIPS). A revised standard [1], with optional integrity enhancements, was adopted by ANSI in October 1989. An Embedded SQL specification [2], for embedding SQL statements into six programming languages, was adopted by ANSI in April 1989. These two ANSI standards were adopted by the NIST as a revised FIPS [9] in February, 1990. These standards are mutually compatible and are referenced subsequently in this report as SQL-1989.

A substantial upward-compatible enhancement [11], often called SQL2, has already been specified by the ANSI and ISO SQL development committees. It will standardize a number of SQL features not included in the original specification because they were not commonly available in SQL products. The technical specification for SQL2 is quite stable; only a very few features are controversial and subject to modification. SQL2 is intended to be a superset of SQL-1989 that replaces the existing SQL standard. SQL2 was registered as an ISO/IEC Committee Draft (CD) in early 1989 and is currently undergoing an ISO/IEC national body ballot to raise its status to Draft International Standard (DIS). Formal adoption of SQL2 as an ISO/IEC standard is expected in 1992. The features of SQL-1989 and SQL2 are discussed in [14].

Emerging SQL3 Specification

A second substantial SQL enhancement [12], often called SQL3, is under development by ANSI and ISO SQL specification committees, with publication expected in the 1995 time frame. SQL3 is a forward looking SQL enhancement that intends to provide additional facilities for managing object-oriented data and for forming the basis of "intelligent" database management systems, a CALS Phase III requirement. It includes generalization and specialization hierarchies, multiple inheritance, user defined data types, triggers and assertions, support for knowledge based systems, recursive query expressions, additional data administration tools, standardized database export/import facilities, and progress toward distributed database management. These features are preliminary and subject to significant modification and improvement before final adoption.

This report focuses on SQL3 features through examples taken from a manufacturing environment. Each section addresses a category of functionality of importance to CALS applications.

2. Encapsulation

In object-oriented programming languages, encapsulation is the packaging of data, constraints, and procedures into an application object so that the object can be manipulated by high level operations without concern for lower level access and integrity control or object self-management. If application objects, or portions of them, are represented in an SQL database, then the SQL language requires additional tools to support encapsulation of these representations. This is particularly true in STEP applications where production data and the run-time environment for factory objects will often be represented in SQL tables. New features proposed for SQL3 that support encapsulation include Assertions, Triggers, and External Procedure Calls. The temporary table facility in SQL2 also supports encapsulation.

2.1 Assertions

An Assertion is an integrity constraint that is triggered by a specific database action. Assertions in SQL3 are more powerful than CHECK constraints in SQL2 because they introduce the notion of before and after images of the database and because they can be activated at specific times. Assertions may be used for sophisticated access and integrity control over application objects.

For example, an assertion could be used to check that the new value of a machine parameter is greater than the old value after an update. The following assertion definition in the database schema would enforce this requirement separately from the application that updates the database:

```
Create Assertion
    During Update of Pitch on Lathe_Variables
    Referencing New Lathe_Variables as X
    Check (X.Pitch > Pitch)
```

An assertion could also be used to enforce the requirement that the Requisitions table only be modified during regular business hours:

```
Create Assertion
    Before Update, Insert, Delete on Requisitions
    Check (Current Time Between 8:30 and 17:00)
```

or that a deletion never delete the last item of a given type from an inventory:

```
Create Assertion
    During Delete from Inventory
    Referencing Old Inventory as OldInv
    Check (Exists (Select * From Inventory Where Inventory.Part_Type = OldInv.Part_Type))
```

or that the addition of a new part to the parts tray only be done by the machine that makes that part:

```
Create Assertion
    During Insert into Parts_Tray
    Referencing New Parts_Tray as NewPart
    Check (User = (Select Part_Maker From Part Where Part_Id = NewPart.Part_Id))
```

or that each machine have less than three of its parts on the parts tray at any one time:

```
Create Assertion
        After Insert into Parts_Tray
        Check ( Count( Select * From Parts_Tray Where User = Part_Maker) < 3 )
```

Each above assertion is "dynamic" in that it depends upon temporary data values that exist only at the time the invoking statement is executed. It may be possible, but likely quite clumsy, to achieve the same effect with just the existing "static" constraints available in SQL2. Using only SQL2 facilities, a database designer would have to find a way to incorporate the dynamic information as persistent values in the database.

The current SQL3 specification has not yet fully resolved how the Before, During, and After options on assertion invocation relate to the New and Old options on the Referencing clause. One requirement might be that a Referencing clause may not be specified unless the During option has been specified and that New means the assertion is checked before each individual row modification and Old means the assertion is checked after each individual row modification. Other restrictions might be that New not be specified with Delete and that Old not be specified with Insert.

In each of the above cases, if the assertion fails to be true when it is invoked, then the database is rolled back to its state immediately prior to execution of the invoking statement and an exception condition is returned to the application program. The program could then execute a Get Diagnostics statement to retrieve information about the constraint that caused the failure.

2.2 Triggers

A Trigger is a sequence of database actions that is initiated by a specific database action. In the current SQL3 specification, a trigger can be activated by an Insert or Delete statement on an underlying table, or by an Update on specified columns. When a trigger is activated, a When clause determines if the list of actions should be executed. The list may include any Insert, Update, or Delete statements on any updateable tables or views in the database. Using triggers, an inventory database could manage its own back-orders or maintain supplies at levels appropriate for a given season. Machines on the factory floor could then order parts from the inventory without concern for inventory management.

For example, the following two trigger definitions in the database schema might serve as a starting point for an inventory database's self-management. The first trigger ensures that inventory supplies are properly updated before a new requisition is allowed to be inserted into the Requisition table. The second trigger ensures that new parts are ordered when the supply is reduced to five or fewer items and that all activity is properly documented in the Activity_Log table.

```
Create Trigger
        During Insert into Requisition
        Referencing New Requisition as NewReq
        Update Inventory Set Supply = Supply - NewReq.Quantity
                Where Inventory.Part_Id = NewReq.Part_Id

Create Trigger
        During Update of Supply in Inventory
        Referencing New Inventory as NewInv
        When ( Inventory.Supply > 5 And NewInv.Supply <= 5 )
        Insert into Backorder
                Select * From Parts
                Where Parts.Part_Id = NewInv.Part_Id;
```

```
Update Activity_Log
        Set Time = Current, Part = NewInv.Part_Id
        Where ID = User
```

In the above trigger example, the second trigger definition inserts a new item into the Backorder table, but it doesn't specify the quantity to be ordered. The following additional trigger definition might address that situation by setting a *null* Quantity value to be some default quantity previously specified for that item.

```
Create Trigger
    After Insert into Backorder
    Update Backorder
        Set Quantity = (Select Quantity From Defaults Where Part_Id = Backorder.Part_Id)
        Where Quantity is Null
```

In this trigger definition, the Update on Backorder is triggered only after the Insert into Backorder is otherwise completed.

Any one of the statements in a trigger definition could violate an integrity constraint on the rest of the database, thus forcing a rollback to the point prior to the statement that initiated the triggered actions. For example, suppose Supply in the Inventory table was always required to be non-negative; then if an application attempted to place a requisition for 23 rolls of toilet paper when the current supply was only 15 rolls, the update statement in the first trigger definition above would fail, thereby generating an exception condition for the Insert statement to the Requisition table that attempted to place the order. The ensuing statement-level rollback would ensure that the inserts and updates of the second trigger definition and the updates of the third trigger definition above are never executed. The user could query Diagnostics to obtain information about the action and the constraint that caused the failure.

Triggers can play a very important roll in modeling application objects in a relational database system. Previously, if an object was represented by several tables, any operation on that object would have to consider the underlying actions on each representing table. Now it is possible for object self-management to be fully specifed in the database schema, with all of the accompanying abstraction, performance and optimization benefits available to the underlying database management system.

2.3 External Procedures

An external procedure is a procedure written in a programming language outside of SQL that can be called from within SQL. The current SQL3 specification accepts SQL data items as input values, passes those values to the external procedure, and accepts a single SQL data item as its result. The result can then be used anyplace that an elementary data type value, or Boolean truth value, can be used in SQL. The specification allows external procedure calls to be defined in the SQL schema, allows appropriate privilege declarations for access by other users, defines how exception conditions are returned to the application program, and includes special rules for passing *true*, *false*, and *null* values.

No database management system can supply all of the specialized functions and operations needed by a specific application area. For example, text processing systems or geographic information system (GIS) databases may need specialized facilities not available in every data manager. An external procedure allows an application environment to define exactly those procedures that it needs to operate efficiently.

A simple example of the usefulness of external procedures might be an application environment that requires access to the Fortran Scientific Subroutine Package (Fortran SSP) from within SQL. With external procedures, the database administrator could define a special schema, e.g. SQLSSP, with procedure calls for each Fortran trigonometric and exponential function, each appropriate function in the Fortran SSP, and each application specific function defined by the database designer. For example:

```
Create Schema SQLSSP

    Declare External SINE
            Real
            Returns Real
            Language Fortran

    Declare External EXPONENTIATE
            Real, Integer
            Returns Real
            Language Fortran

    Declare External FITS
            Real, Real
            Returns Character (1)
            Language Fortran
```

The SQLSSP functions SINE, EXPONENTIATE, and FITS would be linked to Fortran subroutines that calculate the desired values. For example, FITS might be defined as:

```
Subroutine  FITS(TWIST, WEDGE, RESULT)
      Real  TWIST, WEDGE
      Char  RESULT
If (SIN(TWIST) .GT. SIN(WEDGE)**2) Then
      RESULT = '1'
      Else RESULT = '0'
End If
```

A FITS subroutine that returns a '1' is interpreted by SQL as a *true* truth value, whereas a '0' is interpreted as *false*. A user may then specify either of the following to return parts from the database that satisfy the desired expression:

```
Select * From Parts
      Where SQLSSP.SINE(TWISTANGLE) >
SQLSSP.EXPONENTIATE(SQLSSP.SINE(WEDGEANGLE),2)
```

or

```
Select * From Parts
      Where SQLSSP.FITS(TWISTANGLE, WEDGEANGLE)
```

The choice of whether to define one comprehensive procedure that tests a specific condition, versus separate calls to more elementary trigonometric and exponent functions is up to the database designer; in current implementations this decision is based on performance. Many commercial database systems are unwilling to give complete control to the user for user-defined external procedures unless that procedure executes in a workspace separated from the database workspace by software protection schemes. This "firewall" is necessary to protect the integrity of the database kernel but adds expensive overhead to each external procedure call. This problem can be avoided if, at database procurement time, the vendor is given a collection of generic external procedure calls, defined in a special schema, that can be tested for safety and integrated into the database kernel without the need for "firewall" protections. Special external procedures to satisfy text processing or GIS applications could even be proprosed as standards by application groups that have special requirements.

2.4 Temporary Tables and Views

In the existing SQL-1989 standard, if users want to have the effect of temporary tables, they can either define a permanent table in the schema and then empty it and fill it with temporary values during an SQL user session, or they can create a new table in the SQL user session, use it, and then destroy it at the end of the session. Either of these approaches is sufficient in many situations for handling temporary data, but both have drawbacks in that they do not fully satisfy the requirements of true temporary tables.

In one case, it may be desirable to define a global temporary table in the schema and have it simultaneously accessible by hundreds of different SQL user sessions. If the table were defined as a base table instead of a temporary table, then there would be a concurrency logjam if each user session first empties the table and then populates it with data specific to that user session, or users may get spurious data if they define multiple transactions over the table. If instead, users create a new base table in each user session, there might be name conflicts and performance implications.

The SQL2 specification provides a temporary table facility that recognizes three different types of temporary tables: global, created local, and declared local. Global and created local temporary tables are both named tables defined by a

```
CREATE [ { GLOBAL | LOCAL } TEMPORARY ] TABLE
```

definition either in the database schema or in an application program. Each of them is effectively materialized only when first referenced in an SQL user session. Their contents cannot be shared among different simultaneous SQL user sessions. The difference between them is that the contents of a global temporary table are shared among all modules in a single SQL user session, whereas a created local temporary table results in a distinct instance of the temporary table for each module in that session.

A declared local temporary table is defined by the following special temporary table definition:

```
DECLARE LOCAL TEMPORARY TABLE <table name> <table elements>
  [ ON COMMIT { PRESERVE | DELETE } ROWS ]
```

A declared local temporary table may not be defined in a database schema; instead, it must always be declared in a module or in an embedded SQL application program. A declared local temporary table is never associated with a schema name; instead, it is always tied directly to its implicit SQL module. A declared local temporary table is accessible only from procedures in the same module that contains its definition. These restrictions allow system performance optimizations because table definition data need not be stored in the system catalog. The ON COMMIT clause permits the contents of the temporary table to be emptied at the end of each successful transaction, or if desired, retained into the next transaction.

Temporary tables are effectively empty when they are first materialized in an SQL user session. Table instances are destroyed at the end of each SQL user session, so no temporary table instance data may persist beyond the end of an SQL user session.

The SQL3 specification also allows definition of temporary views. A temporary view may be defined in terms of base tables or in terms of other temporary tables in the same SQL user session. A temporary view may not be defined in a database schema; instead, it must always be declared in a module or in an embedded SQL application program. A temporary view is never associated with a schema name; instead, it is always tied directly to its implicit SQL module. The prime advantage of a temporary view is that it allows construction of "what if" situations involving some persistent data from the database and some temporary data existing only in that user session. A temporary view is updatable if and only if it would be updatable as a non-temporary view, and updates that propagate to base tables underlying the view may be committed permanently to the database.

3. Data Abstraction

Data abstraction is a programming technique that hides the details of a data object behind a set of high-level operations on that object. A programmer defines an abstract data type consisting of a data structure and a set of procedures used to access and manipulate the data structure. Abstract data modeling suppresses the details of the internal representation, thereby providing a clean, well-defined interface to the user of the data object. SQL is itself a data abstraction in that the data objects are tables and the procedures are operations to Create or Drop tables and to Select, Update, Insert or Delete table data.

Data abstraction is supported in existing SQL-1989 by a rather elaborate view mechanism that allows pre-specification of database operations to produce a simple table at execution time. It is possible to build hierarchies of view definitions, including summary data, so that each application has an external view of exactly the appropriate data object, ignoring the complex operations that may be necessary to retrieve or update the desired data.

Data abstraction is improved in SQL2 with the ability to define simple Domains in the schema, with the inclusion of new data types Date, Time, Timestamp, and Bit String, and with additional semantics for handling variable length strings. Data abstraction could be further enhanced in SQL3 with the acceptance of new proposals for user-defined data types, application specific *null* values, an Enumeration data type, and optional "strong type checking" in Domains. Some of these proposals are included in the SQL3 specification in preliminary form while others are just working papers or ideas for further discussion. Several of these topics are discussed further in this section; others are described in Section 11.

3.1 String Operations

The existing SQL-1989 standard provides limited support for manipulation of character string data. It includes equality comparisons, inequality comparisons with ordering based on an implementor-defined collating sequence, and a LIKE predicate that allows limited pattern matching capabilities. Pattern matching includes support for special "wildcard" characters that match either a single arbitrary character or a string of arbitrary characters. The LIKE predicate allows an application to search data values for occurrences of key words or phrases, or existence of a special pattern, but it does not easily allow construction of sophisticated search patterns.

The SQL2 specification adds variable length character strings and bit strings as new data types and includes support for international character sets. SQL2 also specifies concatenation and substring operations,

folding between upper and lower case, and allows the invocation of externally defined character sets and collation sequences. The following example demonstrates how these new SQL2 functions might be used to construct mailing labels from a table containing structured mailing information.

String Manipulation Example

Suppose Experts is a table that represents name and address information for people who are regarded as experts in different aspects of manufacturing technology. Experts is represented as follows:

```
Experts  ( Last_Name, First_Name, Middle_Names, Pre_Title, Post_Title,
           Company_Name, Line1_Address, Line2_Address, Mail_Stop, PO_Box,
           City, State, Country, Zip_Base, Zip_Extension, Description)
```

The last column of the Experts table, the Description column, is a variable length character string that describes in text the area of expertise of the person represented by that row. Any non-trivial query or manipulation of the Description column requires text manipulation capabilities beyond those specified in SQL-1989. The facilities provided in SQL2 may not satisfy all string processing requirements, especially those involving "chunk" expressions addressing words, lines, and paragraphs, and distances between such objects; however, they do make complex string expressions possible. Enhanced facilities for sophisticated text processing involving "chunk" expressions may be considered for SQL3, but they are not contained in the current draft.

The following query would return a list of mailing labels for all experts in the Boston area with expertise related to "friction", "corrosion", or "bearings":

```
Select   Pre_Title || '. ' || First_Name || ' ' ||
             Case
                  When Middle_Names is Null Then Null
                  Else Substring(Middle_Names From 1 For 1) || '. '
             End
             || ' ' || Last_Name ||
             Case
                  When Post_Title is Null Then 'CR/LF'
                  Else ', ' || Post_Title || 'CR/LF'
             End
             || Company_Name || 'CR/LF' ||
             Case
                  When Line2_Address is Null Then Line1_Address || ' ' || Mail_Stop || 'CR/LF'
                  Else Line1_Address || 'CR/LF' || Line2_Address || ' ' || Mail_Stop || 'CR/LF'
             End
             || Case When PO_Box is Null Then Null Else PO_Box || 'CR/LF' End
             || City || ', ' || State || '  ' || Zip_Base ||
             Case
                  When Zip_Extension is Null Then 'CR/LF'
                  Else '-' || Zip_Extension || 'CR/LF'
From Experts
Where
      (Lower(Translate(Description Using Universal_To_ASCII)) Like '%friction%'
           Or Lower(Translate(Description Using Universal_To_ASCII)) Like '%corrosion%'
           Or Lower(Translate(Description Using Universal_To_ASCII)) Like '%bearing%')
```

```
And (City = 'Boston' Or Zip_Base Like '02___')
```

The above query expression uses the SQL2 concatenation operator (||) to construct a single derived column of output. In addition, it uses the Substring function on Middle_Names to extract just the first character as a middle initial, the Lower function to translate ASCII, mixed-case values to lowercase, and the Translate function to invoke an externally defined translation from the implementor-defined Universal character set to the ASCII character set.

The implementor-defined Universal character set might include the union of all possible national and international character sets known to the system so that the description of a person's expertise can be consistently and correctly stored in the database, even if it contains special fonts and character sets with accents, inflection marks, or other special symbols. The translation definition then maps these characters to ASCII characters so that the string can be tested without diacritical marks. The translation from non-latin character sets to ASCII is meaningful only if there exists some locally accepted interpretation of non-latin characters as a latin string.

Case Expression

The above example also uses the case expression in SQL2. The case expression has one of the following two general forms:

```
Form 1:     Case
                When <predicate 1> Then <result>
                ...
                When <predicate n> Then <result>
                Else <result>
            End

Form 2:     Case <value expression 1>
                When <value expression 2> Then <result>
                ...
                When <value expression n> Then <result>
                Else <result>
            End
```

In Form 1 different predicates are tested and the <result> corresponding to the first predicate that evaluates to *true* is the result of the case expression; otherwise, the Else clause determines the result. In Form 2 <value expression 1> is evaluated initially and its value is tested against the other value expressions in the list. The <result> corresponding to the first expression that matches the initial expression is the result of the case expression; otherwise, the Else clause determines the result.

A case expression always returns a single result, an SQL value expression that reduces to a single SQL value. Therefore, the case expression can appear in the SQL language at any place a single value can appear.

Using only the SQL-1989 specification, without string manipulation features and without the case expression, the above mailing list example would require development of a rather sophisticated application program. Each row of the table would have to be read into the application program and tested to decide if that row satisfies the desired search criteria. Also, the mailing label would have to be constructed within the application program. These SQL2 string manipulation capabilities, together with the case expression, provide desirable abstraction tools for CALS applications.

Similar Predicate

The SQL3 specification provides additional capabilities for pattern matching in bit strings and character strings with the Similar predicate. The Similar predicate is analogous to the Like predicate in SQL2, but is much more powerful. It takes the general form:

<match value> [NOT] SIMILAR TO <pattern>
[ESCAPE <escape character>]

where <match value>, <pattern>, and <escape character> are all comparable character strings, i.e. character strings from character sets with compatible collation definitions. The <pattern> allows construction of regular expressions equivalent to regular expressions in the emerging Posix standard under development in ISO/IEC JTC1/SC22.

Using the Similar predicate, the first part of the Where clause in the experts mailing example above could be restated as follows:

Lower(Translate(Description Using Universal_To_ASCII))
Similar To (%friction%|%corrosion%|%bearing%)

The Similar predicate is specified separately from the Like predicate in SQL2, because it is not possible to extend the definition of Like in an upward compatible way to be consistent with the Posix definition of regular expressions.

3.2 Datetime Data Types

The new Date, Time, Timestamp, and Interval data types in SQL2 follow the common rules associated with dates and times and yield valid datetime or interval results based on the Gregorian calendar. The Date data type consists of fields: Year, Month, and Day; the Time data type consists of fields: Hour, Minute, Second, Timezone_Hour, and Timezone_Minute; and the Timestamp data type consists of all fields contained in both Date and Time. The Time and Timezone data types allow an arbitrary specification of Seconds precision so that fractions of a second can be maintained as needed.

The Interval data type may be a year-month interval or a day-time interval. The need for two types of intervals is to accommodate the variable number of days in a month. A year-month interval specifies a duration measured in years, months, or years and months. A day-time interval specifies a duration measured in a combination of days, hours, minutes, and/or seconds. An interval occurs naturally as the subtraction of two date, time, or timestamp data types.

Time and Timestamp data types are maintained in Universal Coordinated Time (UTC), commonly called Greenwich Mean Time, with an explicit or implied timezone part. The timezone part specifies the difference between UTC and the actual date and time in the timezone represented by the data item. Time and Timestamp data types may be specified with or without timezone information, but the absence of timezone information implies an implicit timezone equal to the local timezone of the active SQL-user session.

Intervals may be added or subtracted from datetime data types to produce a new datetime. Intervals may be added or subtracted from each other or multiplied or divided by a numeric value to produce a new interval. Operations involving a datetime and an interval preserve the timezone of the datetime operand.

3.3 Cast Function

In the SQL2 specification, a cast function may be used to convert a value from one data type to another. All data type values may be cast to a character string value and all fixed point numeric values may be cast to approximate numeric values or to various field values in datetime data types. The SQL2 proposal contains a table of all valid conversions from one data type to another. The following are examples of valid conversions:

```
CAST DATE '1990-10-21' AS CHARACTER = '1990-10-21'
CAST B'011011101' AS CHARACTER = '011011101'
CAST 29.36781 AS NUMERIC (5,2) = 29.37
      Note: Rounding or truncating is implementor defined.
CAST NULL AS INTEGER = NULL
      Note: SQL does not support declaration of non-typed null values.
CAST 'John' AS CHARACTER (7) = 'John   '
CAST 'John   ' AS CHARACTER VARYING = 'John'
CAST INTERVAL '206' YEARS AS CHARACTER = '206 YEARS'
CAST INTERVAL '-45:12:15:34.6' DAYS TO SECONDS AS CHARACTER = '-45:12:15:34.6'
```

The cast function provides an additional abstraction tool for CALS applications and makes it possible to pass any SQL data value to any programming language, either as a character string value or as a value in a data type native to the programming language. This conversion capability supports application interoperability.

3.4 Simple Domains

Domains in SQL2 are somewhat restricted in that they cannot be nested; instead, each domain must be defined directly over a base SQL data type. In addition, there is no concept of the union of domains, which would allow values from two or more base SQL data types in a single column. Even with the above restrictions, domains provide a valuable enhancement for data abstraction in CALS applications.

For example, domain definitions can be used to hide the underlying character string representation of a part identifier or to enforce format constraints such as hyphens or other symbols in specific positions of the string, such as:

```
Create Domain Part_Id
      As Character (20) Not Null
      Check (Substring(Value From 1 for 4) = 'NIST'
            And Substring(Value From 17 for 1) = '-')
```

Similarly, domains could be used to define and enforce particular bit string representations of raster graphics, diagrams, voice prints, or video images.

In addition, domains provide a feasible alternative to the Boolean or Enumeration data types found in many programming languages. For example, the domain definition

```
Create Domain Month
      As Character (3)
      Check (Value In ('Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun',
                  'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec'))
```

represents an enumeration data type for months of the year; similarly, the domain definition

```
Create Domain Boolean
      As Character (1)
      Check ( Value In ('T', 'F'))
```

plays the role of a Boolean data type. Even the Boolean operations And, Or, and Not are simulated by the normal SQL predicates, since Boolean expressions like "Not(A And B) Or C" can be represented by the SQL predicate:

```
Not (A = 'T' And B = 'T') Or (C = 'T')
```

Even *unknown* values are supported by the above domain definitions because the Check constraint does not prohibit *null* values; that is, the Check must evaluate to *false* before the constraint is violated. An evaluation to *unknown* is does not violate the constraint. An additional Not Null constraint in the domain definition would restrict the domain to non-*null* values, if desired.

The above domain definition capabilities are specified in the emerging SQL2 specification [11]. A more sophisticated domain facility, with domain hierarchies, operational inheritance, and optional "weak" or "strong" type checking, is proposed for SQL3 (see Section 4.1).

3.5 User-Defined Data Types

User-defined data types in SQL3 allow composite data types, such as complex numbers and matrices, to be constructed from the existing base types or new abstract types such as Full Text, Spatial Data, Video, or Voice to be constructed from scratch. With the appropriate use of external procedure calls mentioned in the previous section, user-defined data types might include sophisticated integrity constraints and high-level operations that could be used for retrieval and other database manipulations. Operations may range from something as straight forward as trigonometric expressions on complex numbers or matrix arithmetic to sophisticated splicing of Video and Voice images.

A proposal currently under consideration for SQL3 would allow user-defined types to be constructed from existing data types or from previously defined user types. The single construction allowed is a sequence of data types, but it can be nested to any level so that highly complex structures are possible. Capabilities for additional "constructor" data types, such as Set, Variable List, Choice, Union, etc., as specified in [13], may be considered later as SQL3 develops.

The user-defined data type definition can include sizing qualifiers, such as the dimensions of an array, so that various sizes can be specified when the data type is used. An instance of any such data type would always be considered as a single value, storable as a column value in a table. Operations on the newly defined types are defined by passing parameters to external procedures and are invoked by use of a functional style. Operations can be defined to handle *null* values and to return error codes compatible with the SQL exception handling mechanism. User defined types can be combined with the Domain feature to provide encapsulation of the data structure, constraints, and operations as a single package.

The following is a simple example of how two user defined types, Vector and Frame, might be defined from more elementary data types and domains. The Vector data type has two operations, Dot_Product and Cross_Product, each defined as an external procedure in programming language C. The Frame data type has only one operation, Rotate, defined in Ada. Integrity constraints are enforced on Frames by defining a new domain, Geometry, which enforces referential integrity of frame components with respect to existing parts in the database and requires a normalized orientation.

```
Create Data Type  Vector  as Struct
      (      [x-coord]   Real,
             [y-coord]   Real,
             [z-coord]   Real )
      Operation  Dot_Product  Language C
             Parameters  Vector, Vector
             Returns  Real
      Operation  Cross_Product  Language C
             Parameters  Vector, Vector
             Returns  Vector


Create Data Type  Frame  as Struct
      (      [Object]          Part_Id,
             [Orientation]     Vector,
             [Outline]         Sequence of Vector)
      Operation  Rotate  Language Ada
             Parameters  Frame, Rotation
             Returns  Frame


Create Domain  Geometry  As Frame
      Check (Value.Object References Parts)
      Check (Dot_Product(Value.Orientation, Value.Outline(1)) = 0)
```

The Dot_Product in the second Check clause of the Geometry domain is meant to enforce a constraint that the first vector of the outline always be perpendicular to the orientation vector.

4. Inheritance

Inheritance is a feature in object-oriented programming languages that allows hierarchical classification of data objects, with the ability for one class to acquire characteristics from another. A class may inherit attributes and operations from a more general class and may have its attributes and operations inherited by subclasses.

4.1 Domain Hierarchies

The domain specification in SQL2 provides only "weak type checking" in that all domains assume the data type of their underlying definition. With this definition, one can multiply "apples" and "oranges" provided they both have a numeric base. With optional "strong type checking", under consideration for SQL3, an application would be able to specify only those operations that are valid on a given type, or operations that are valid across types with the result being yet another type.

With this approach it is possible to have hierarchies of domain types with or without inheritance of operational capabilities. This technique is analogous to the concept of methods and method inheritance in object-oriented programming languages. Specifications can get quite complex, so a decision must be made as to the appropriate level of "strong typing" in the SQL3 specification. The existing SQL3 working

draft [12] specifies domain hierarchies, but it does not yet incorporate "weak" or "strong" typing concepts or any of the "constructor" data types mentioned in the previous section.

4.2 Table Hierarchies

A feature currently specified in preliminary form in the SQL3 document is a generalization hierarchy for tables and table definitions, with the ability for a subtable to inherit the columns, key attributes, and integrity constraints from a more general table definition, called the gentable. This makes it possible to define a single table to represent a general class of database objects with separate, heterogeneous subtables for subclasses of those objects. Updates, insertions, and deletions to any subtable or parent table effectively propagate consistent modifications to items in other tables. The advantage is that the logical ISA relationship, popular in semantic data models, is supported directly by database system software, with the implied cascade effects optimized internally to minimize physical data modification.

One way to visualize a subtable and its parent gentable is to picture the rows of the subtable as a subset of the rows of the gentable -- but with additional attributes, not found in the parent table, represented separately as follows:

Gentable

Primary Key Columns	Gentable Columns

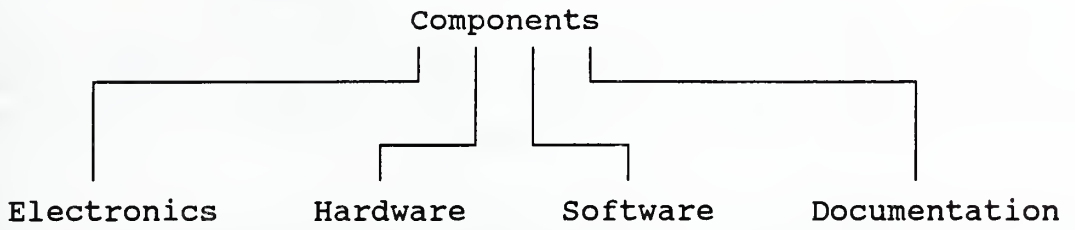
Subtable

Primary Key Columns	Inherited Columns	Other Columns

The primary key of the gentable becomes the primary key of the subtable. Each column of the gentable becomes an inherited column of the subtable. The primary key values and the inherited column values in the subtable are the same values as those in the underlying row of the gentable.

Consider, for example, a Components table that identifies and describes all potential components in an integrated manufacturing system, with separate, heterogeneous subtables for electronics, hardware, software, and documentation components. The Components table might contain common attributes: Part_Id (a primary key), Part_Name, and a textual description of the part. The subtables contain other attributes

specific to items in each subclass, possibly power requirements for electronic components, number of pages for documentation components, or license details for software components.



There is no requirement that subtables be disjoint, so one component could be classified both as an electronics component and as a hardware component and carry the additional attributes for each subtable. If the subtables represent mutually disjoint components, then additional constraint declarations could be specified to enforce this requirement.

Insertions

In the above example, an insertion into any component subtable would result in exactly one of the following:

1) insert a row into that subtable and propagate an insertion of a new row to the Components gentable, or

2) insert a row into that subtable and reclassify an existing row of Components as a member of the subtable, or

3) raise an error if some insert value does not match properly an inherited value from an existing Components row with the same primary key.

Case 1 occurs when a new primary key value, not already in the gentable, is inserted into the subtable. Case 2 or 3 occurs when the primary key value of the row inserted into the subtable is identical to an existing primary key value in the gentable.

An insertion into a gentable does not automatically generate any actions on its subtables. It is always possible in the above example to insert a row into the Component table without inserting one into any of the defined subtables, unless an explicit integrity constraint rules out this possibility.

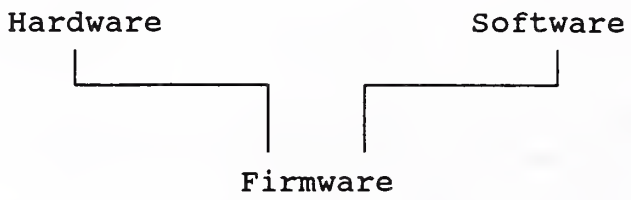
Updates

An update to an attribute of some item in a gentable effectively propagates to the corresponding inherited column in all subtables of that gentable, to whatever level of nesting. Conversely, an update to an inherited column in a subtable also propagates to an update of that same column in the corresponding row of the gentable and thereby propagates to all other subtables in the same subtable family. An update to a non-inherited column in a subtable is local only to that subtable (unless the subtable is also a gentable).

Multiple Inheritance

If a subtable is specified to depend upon more than one gentable, then it inherits all attributes and all constraints from each parent table, i.e. the subtable has a property commonly referred to as multiple

inheritance. As an example of multiple inheritance, consider a subtable Firmware defined as a subtable of both Hardware and Software as follows:



The SQL3 specification requires that each subtable have a unique maximal gentable. Thus Firmware could not be a subtable of both Hardware and Software unless they, in turn, were subtables of a common gentable, i.e. Components in this example. Firmware assumes the primary key value of the maximal gentable as its own entity identifier rather than introducing the complexity of alternative primary keys or construction of a compound entity identifier from the primary keys of unrelated parent gentables. Although Firmware inherits the primary key from both Hardware and Software, it is the same primary key and logically appears only once in the subtable. The same is true for all other columns in Hardware and Software inherited from Components, even if they have been renamed, differently, in the Hardware and Software definition. Alternatively, non-inherited columns in both Hardware and Software are inherited as separate columns by Firmware, even if they happen to have the same column name. A renaming facility allows disambiguation of inherited columns with the same column name.

Deletions

A deletion applied to any row of a gentable triggers a cascade delete of every dependent row in the subtable family. In the above examples, deletion of a row from the Components table with a given Part_Id would result in the deletion of any rows in the Electronics, Hardware, Software, Documentation, or Firmware subtables that have an identical value for the Part_Id column. Similarly, a deletion from Hardware would trigger deletion of any row with an identical Part_Id from Firmware, but not from Software or Components. In general, deletion of a row from a subtable has no direct effect on the rows of a parent gentable or sibling subtables. If an application requires that deletion of a row from a subtable also delete the corresponding parent row from a gentable, then an SQL3 trigger could be defined to achieve that effect; this might be the desired semantics if the component subtables defined above represent a partition of the Components table into mutually disjoint and collectively exhaustive subtables.

The subtable facility does not make SQL3 an object-oriented database system, but it does provide a tool to support object-oriented applications, a prime CALS requirement. With the subtable facility SQL supports the representation of complex data objects and interrelationships but retains the simplicity of tables as the underlying object of data manipulation. This simplicity is what allows implementation of a high-level, flexible query and update language, the prime advantage of the relational model.

5. Recursive Expressions

A perplexing concern that has challenged data management since the early days of file management systems is the "Bill-of-Materials" problem. The challenge is to determine an efficient method to answer queries like the following:

"Construct an expanded list of all parts needed to manufacture component X."

"Find all parts that contain part Y as a component."

The problem is the recursive nature of these queries. In the first query, a system might first retrieve parts that are direct components of X, then parts that are components of those direct components, etc. In the second query, the system might first retrieve all parts that directly contain the given component, then parts that contain those as sub-components, etc. In each case, it is the indefinite number of steps needed to produce the result that presents a difficulty for existing SQL language constructs.

A recursive union facility in SQL3 [18] provides a generic tool for specifying queries of this type. It begins with an initial query expression to determine a starting point for a result table, then an iteration expression that determines how new elements get added to the result. Optional clauses allow the user to specify whether the recursive search is to be depth first, breadth first, or arbitrary, how the search sequence should be recorded in the result table, how to detect and control cycles and report detection in the result table, and whether to stop execution or report an error if iteration exceeds a specified limit.

Recursion Example

Assuming a normalized database, the following three tables might be used to represent a parts database:

COMPONENTS (Part_Id, Part_Name, Description)

STRUCTURE (Part_Id, Used_In, Quantity)

INVENTORY (Part_Id, Availability, Cost)

The Components table would be comprehensive and would list every elementary part, every sub-assembly, and every complete assembly known to the environment. The Structure table identifies only those parts or sub-assemblies that are used in some other component. The Inventory table lists only those items that are normally stocked in the storeroom; sometimes complete assemblies are stocked and sometimes only component parts are stocked. If desired, Inventory could be defined as a subtable of Components. In that case, the join operations defined in the following example might execute more efficiently.

A user may wish to produce a report listing all parts required to produce a "Wheel_Assembly", taking advantage of any assembled components that are already available in the storeroom. A temporary result table could be created with the following format:

RESULT (Seq_Nbr, Part_Name, Part_Id, Used_In, Quantity, Availability, Cost, Cycle_Mark)

The Seq_Nbr column uses integer values to record the retrieval sequence for a depth-first or breadth-first search. The Cycle_Mark column carries a user-specified symbol to denote system detection of a cycle, where a cycle is determined by observing repeated values for a specified list of columns. The following query expression might be used to populate the Result table:

```
SELECT  0, Part_Name, Part_Id, null, 1, null, null, null
      FROM Components
      WHERE Part_Name = 'Wheel_Assembly'
RECURSIVE UNION
      Rslt (Seq_Nbr, Part_Name, Part_Id, Used_In, Quantity, Availability, Cost, Cycle_Mark)
SELECT  0, Part_Name, Part_Id, Used_In, Quantity, Availability, Cost, '<space>'
      FROM (Structure NATURAL LEFT OUTER JOIN Inventory) INNER JOIN Components
      USING (Part_Id)
      WHERE Structure.Used_In = Rslt.Part_Id AND NOT (Rslt.Availability = 'stocked')
SEARCH DEPTH FIRST BY Part_Name SET Seq_Nbr
CYCLE Part_Id SET Cycle_Mark TO '*'
RETURN LIMIT (100)
```

A database report writer facility might then order the Result table by the value of the Seq_Nbr column to produce the following output report. In this report, indentation of the Part_Name field is used instead of identifying parent/child relationships from the Used_In column:

Part_Name	Part_Id	Quantity	Availability	Cost	Cycle_Mark
Wheel_Assembly	#158	1			
Bearing	#259	2	stocked	$23.15	
Sleeve_Unit	#301	2			
Bearing	#259	4	stocked	$23.15	*
Washers	#021	8	stocked	$2.50	
Tire	#192	1	back_order	$85.75	
Wheel_Housing	#621	1	back_order	$215.00	
Axle	#135	1	stocked	$195.00	
Sleeve_Unit	#301	2			*
Wheel_Info	#411	1	stocked	$15.95	

The *null* value for Availability of the Sleeve_Unit was derived from the LEFT OUTER JOIN in the above FROM clause. It indicates that the Sleeve_Unit is not normally contained in the inventory and must always be assembled from its components. The "back_order" value for Availability of the Wheel_Housing indicates that the Wheel_Housing is normally stocked at the given price but is now on back_order, so the user must decide whether to wait for the order or assemble the unit from its component parts at a higher price. Likewise, the "back_order" Availability value for tire indicates that the tire is just plain not available because it has no component parts. The two Cycle_Marks on the second occurrence of Bearing and Sleeve_Unit result from the CYCLE clause of the above query expression and indicate that the system has encountered a cycle on Part_Id. Each of these components has been previously expanded, so futher expansion is suppressed at this point. Absence of the optional CYCLE clause would result in full expansion. The output table only contains 10 rows, less than the limit of 100 rows declared in the LIMIT clause to trigger an interruption, so the Result table represents the desired parts expansion.

This example gives a reasonable overview of the power and flexibility of the recursive union facility specified in the SQL3 document, and how it might be used to address CALS requirements.

6. Existential and Universal Quantifiers

The Exists predicate in SQL-1989 provides a method for testing whether or not a specified query expression results in an empty table. In particular, the expression

Exists (Select * From T Where <search condition>)

returns *true* if the Select statement results in a non-empty table and *false* if it results in an empty table. In SQL-1989, an Exists predicate can only return *true* or *false*; it can never return "maybe". In some cases the <search condition> may evaluate to *unknown*, but an *unknown* value is not a *true* value, so such rows are not included in the output of the Select statement. If the <search condition> is negated, the result is still *unknown*, so such rows are still left out of the result of the Select statement. Thus, in SQL-1989, there is no straight-forward way for a user to determine if there exist rows of T for which the <search condition> evaluates to *unknown*, thereby implying a "maybe" result for the Exists predicate.

One way to isolate *unknowns* is with the following query expression, using set difference and negation:

```
Select * From T
      Except
Select * From T Where <search condition> Or Not <search condition>
```

However, since set difference is not specified in SQL-1989, this option is not available in a standard way until SQL2 is adopted. Of course, if the <search condition> is very simple, e.g. Age > 50, then one could isolate the *unknowns* by direct application of the Null predicate, e.g. Age Is Null. This explicit testing of *null* values is less suitable as the <search condition> becomes more complicated.

SQL2 Tests for Truth Values

The above methods for isolating *unknown* values are too indirect. Instead, the SQL2 specification provides a straight-forward technique for transforming the 3-value logic of a <search condition> into the 2-value logic of the Where clause. SQL2 provides direct tests for *true*, *false*, and *unknown* that can be applied to a Boolean primary in a <search condition>.

In SQL2, the production for Boolean factor is enhanced to allow the following expressions:

<boolean primary> Is [Not] Unknown

<boolean primary> Is [Not] True

<boolean primary> Is [Not] False

These new tests allow a straight-forward user specification of complex retrievals. The "Is True" and "Is False" options duplicate SQL-1989 functionality, but are specifed in SQL2 for completeness, and because their negations add desirable new flexibility to the SQL language. For example, "Is Not False" is logically equivalent to "Maybe". With these tests, knowledgeable users can filter their own search conditions to achieve the desired effect.

True Quantified Predicates in SQL3

To enhance the functionality of the Exists predicate in SQL-1989, and to remove the clumsiness of the present use of the Exists predicate for quantification, SQL3 specifies new existential and universal predicates [17], which are upwardly compatible with the current Exists predicate. These new operators correspond more faithfully to the well known existential and universal quantifiers in 3-valued predicate logic.

These new predicates for quantification are of the form

Exists <table reference list> (Where <search condition>)

or

For All <table reference list> (Where <search condition>)

The <table reference list> is a list of table names, joined tables, or query expressions, and has exactly the same syntax as does the From clause in the usual SQL Select statement. If R is the result of the expression

Select * From <table reference list>

then the result of the Exists predicate is:

false - if R is empty, or if the <search condition> of the Exists predicate is *false* for every row of R

true - if there exists any row of R for which the <search condition> of the Exists predicate is *true*

unknown - otherwise

and the result of the For All predicate is:

true - if R is empty, or if the <search condition> of the For All predicate is *true* for every row of R

false - if there exists any row of R for which the <search condition> of the For All predicate is *false*

unknown - otherwise

The new SQL3 quantified predicates allow a user to analyse "what if" and "maybe" conditions that arise naturally in many CALS applications.

7. Roles and Data Security

The existing SQL-1989 standard uses a simple security model for declaring and managing database access control. The model consists of three main entities: Objects, Actions, and Users. Objects are things defined in a database schema definition, actions are operations on objects, and users are the entities that

invoke an action on an object. A privilege is an authorization of an action on an object to a user. A privilege is represented by an ordered tuple:

(Grantor, Grantee, Object, Action, Grantable)

The grantor is a user who is already authorized to perform the action on the object and who is authorized to grant that authorization to other users. The grantee is the user who receives the authorization from the grantor. Grantable is a yes/no variable indicating whether or not the grant authorization is passed from the grantor to the grantee.

When an object is created, some user is designated as the owner of the object. The owner is authorized to perform all actions on the owned object and to grant this authorization to other users. No user other than the owner of an object is authorized to perform any action on that object unless an explicit sequence of privileges can be traced from the owner to that user. The grantor of a privilege is also authorized to revoke that privilege from the grantee at a later time, thereby cascading a revoke action through the maze of subordinate privileges.

In SQL-1989, there is essentially only one object, the table, so all objects are identified by table name. In SQL2, database objects include domains and assertions, and in SQL3, also include triggers, user defined types, and external procedures. Each of these objects is characterized by object type and by object name since object names are unique within object type. Object names may be scoped by catalog name and schema name, or other multi-level naming conventions authorized by the appropriate SQL standard.

Actions on table objects include: Select, Insert, Update, Delete, and References. Insert, Update, and References actions may be qualified by a list of column names to indicate which columns of the table may be accessed. A References action means that the table or column may be referenced by an integrity constraint defined by some user other than the owner. Privileges on references are needed in SQL because integrity constraints, such as referential integrity and assertions, determine what subsequent actions are legal on the referenced table. Actions on other schema objects include references, so that definitions can be kept private, if desired, or shared with other users.

The main drawback of the security model described above is that it may get quite complex as the number of users increases and as users are moved from one role to another. A single user may have Personnel privileges for a time and then be switched to Payroll privileges. The management of such role changes is subject to human errors when the privileges are granted and revoked one at a time. In addition, the existing security model does not allow the same user to assume a different set of privileges for each role that user may play, e.g. Personnel clerk versus Payroll clerk. The privileges granted to a user are always cumulative, so that the sum total of privileges for Personnel clerk plus Payroll clerk may be greater than what is intended. The current solution to this problem is to require users to sign-on to the system separately for each role that they play. This solution is subject to misuse if users are required to remember multiple sign-on procedures.

Roles in SQL3

The SQL3 specification defines an enhanced facility for database security management that builds upon the existing Grant and Revoke mechanism. It extends the security model entities to include Roles in addition to Objects, Actions, and Users. A role is a named collection of authorized actions on objects. The existing Grant and Revoke mechanism for privileges is used to assign privileges to and from roles. A user identified as the role administrator is authorized to manage roles and to grant and revoke roles from individual users. In addition, roles may be granted to other roles so that role hierarchies, without cycles, may be established to facilitate groups of application privileges.

A role authorization is represented by an ordered tuple:

(Grantee, Role, Admin)

The grantee is the user, or role, receiving a role authorization from the role administrator. Role identifies the role by its role name, which is unique in the database environment. Admin is a yes/no variable indicating whether or not the role administration authorization is passed from the role administrator to the grantee. Role authorizations differ from privileges in that the grantor of the role authorization is not a persistent component of the privilege. Thus the role authorization remains intact even if the role administrator changes. A role authorization can only be revoked by a current administrator for that role. Roles are not revoked as the side-effect of some other action as privileges sometimes are.

Roles support the management of privileges needed to execute individual applications. Authorizing a user to run an application often involves many dozens of separate privilege authorizations. Instead of database administrators granting and revoking privileges on individual tables to individual users, the privileges for each application can be assigned to a role and the roles granted and revoked from individual users. In practice, the simplicity of role management should result in fewer human errors and thus more effective security management.

Roles also support the reassignment of responsibility for security management without complications. If user X is responsible for database security and is replaced by user Y, the current SQL security mechanism requires that all privileges granted by X be regranted again by Y before they can be revoked from X. One must be careful of side-effect actions that may inadvertently cause database objects such as views and integrity constraints to be dropped or privileges belonging to other users to be revoked. With roles, responsibility for security management can be reassigned by granting the role with the administration option to user Y and revoking the role from user X, thereby avoiding the side-effects.

Roles also avoid the problem of security breaches caused by one user receiving privileges to execute several different applications, having the cumulative effect of those privileges much greater than intended. The complexity is increased if a different security administrator is responsible for each application. With roles, only one role is enabled at any one time, and the definer of the role has complete control over the set of privileges available to the user at any one time. A user is allowed to change from one authorized role to another using a Set Role statement.

A key design goal for roles [15] is to support the administration of end-user security without the risk and complexity of side-effect cascading actions that can inadvertently disrupt service. To achieve this goal, privileges granted to roles may not be grantable. Users cannot obtain a privilege from a role that they can then grant to other users. Also, privileges obtained through roles cannot be used to enable the definition or operation of views, integrity constraints, or other database schema definitions. A role can thus be revoked from a user without cascaded effects and without destruction of other database objects.

Roles can be used to group together sets of privileges and sets of users so that the sets of privileges can be mapped to sets of users. For example, suppose A, B, and C are three important applications, each involving a complex set of privileges, and G1 and G2 are two groups of users where G1 requires access to applications A and B and G2 requires access to applications B and C. We could define roles as follows:

Role_A	--	Contains privileges for application A.
Role_B	--	Contains privileges for application B.
Role_C	--	Contains privileges for application C.
Role_G1	--	Contains authorization for Role_A and Role_B.
Role_G2	--	Conatins authorization for Role_B and Role_C.

Individual users could then be assigned to Role_G1 or Role_G2, or both, as appropriate, but no user would have access to the cumulative effects of the privileges for all three applications without creation of a new role explicitly for that purpose.

The most significant advantage of roles is that they maintain a mapping among database privileges, applications that access objects the privileges protect, and users who need to be authorized to use the applications. By grouping together the privileges needed by each application and assigning them to a role, one can readily tell which database privileges are used by an application and which users are authorized for each application. The role mechanism, although by itself quite simple, it sufficiently complete to define and maintain these relationships, thereby encouraging effective usage.

8. Savepoints and Subtransactions

In the current SQL-1989 standard there are provisions for two levels of rollback within a transaction: statement level rollback and transaction level rollback. A statement must either fully succeed or completely fail; thus, if an exception condition is raised during execution of a statement, the database is rolled back to its state immediately prior to statement execution. Likewise, a transaction must fully succeed or completely fail. The user invokes a Commit or Rollback statement to end a transaction and to achieve commitment or rollback of the data. These two levels of transaction management are sufficient to ensure safe and reliable execution of application programs.

The SQL3 proposal specifies additional capabilities to manage subtransactions. A subtransaction is a portion of a transaction that is marked for potential rollback without affecting the other parts of the transaction. A subtransaction is said to be "pre-committed" if it is no longer considered to be a candidate for rollback, i.e. if its markings are destroyed. Subtransactions in SQL3 are defined and managed through savepoints, which are points within a transaction to which the transaction can be rolled back. The releasing of a savepoint is equivalent to pre-committing the statements executed after the savepoint.

Savepoints can be very useful in interactive "what if" sessions. The following interactive example from [16] illustrates the properties of savepoints:

```
SAVEPOINT  a;                -- immediately creates a savepoint a
DELETE ... ;                 -- do some DML statement(s)
SAVEPOINT b;                 -- creates a savepoint b subordinate to a
INSERT INTO ... ;            -- do some more DML
SAVEPOINT c;                 -- creates a savepoint c subordinate to b
UPDATE ... ;                 -- do some more DML

-- at this point the user user realizes a mistake and rolls back
-- part of the transaction, then continues from that point.

ROLLBACK TO SAVEPOINT c;     -- Savepoint c remains defined, and Update is undone
ROLLBACK TO SAVEPOINT b;     -- Savepoint c is destroyed, and Insert is undone
UPDATE ... ;                 -- do some more DML
RELEASE SAVEPOINT b;         -- Subtransactions subordinate to b are pre-commited
                             -- Savepoint b and any savepoints subordinate to b are destroyed
COMMIT;                      -- Delete and 2nd Update commited and all
                             -- outstanding savepoints destroyed
```

Savepoints also have a natural use in embedded SQL programs. A procedure can establish a savepoint at the beginning of its logic. If an SQL statement within the procedure raises an exception condition, then the database could be rolled back to its state immediately before procedure invocation, without destroying any previous database work done. If all SQL statements within the procedure are successful, then the savepoint can be released, having the same effect as pre-commiting a subtransaction containing just that procedure. The following procedure example from [16] illustrates this use of savepoints:

```
procedure Do_Some_Work (Parm1, Parm2, ... , Completed_OK):
  declare   Savepoint_Id   integer;
  Completed_OK := false;
  EXEC SQL SAVEPOINT INTO :Savepoint_Id;
  EXEC SQL WHENEVER SQLERROR GOTO problems;
  .
  EXEC SQL INSERT ... ;
  EXEC SQL DELETE ... ;
  EXEC SQL UPDATE ... ;
  .
  Completed_OK := true;
  EXEC SQL RELEASE SAVEPOINT :Savepoint_Id;
  return;

problems:
  EXEC SQL ROLLBACK TO SAVEPOINT :Savepoint_Id;
  Completed_OK := false;
  return;
end;
```

In the above example, the Savepoint statement causes the SQL processor to establish a savepoint and to create an integer identifier as a "handle" for the savepoint, which is then saved in the Savepoint_Id variable. The Release Savepoint statement is executed only after the successful execution of all SQL data modification statements in the procedure and is effectively equivalent to a "pre-commit subtransaction" command. The Rollback Savepoint statement in the code segment labeled problems would also release any savepoints established after the identified savepoint.

One somewhat controversial issue in SQL standardization committees is the extent to which the SQL language should be extended in the area of transaction management, a topic not directly related to the relational data model. The situation at the present time is that other standardization committees, such as Transaction Processing (TP) and Commitment, Concurrency, and Recovery (CCR), are defining protocols for transaction management, but no existing standardization committee is defining syntax for application program interface (API). Thus if SQL requires this feature, it is necessary to define the syntax for SQL and to define the semantics in a way that maps correctly to existing or proposed protocols when used in Open Systems Interconnection (OSI). SQL must always remain compatible with the emerging Remote Database Access (RDA) standard, which defines protocols for remote access to other SQL environments and depends upon transaction processing services in the TP and CCR standards.

9. Distributed Database Management

Distributed database management means totally integrated distributed data under the control of a fully functional distributed database management system. Distributed database management implies capabilities for partitioning logical data structures into pieces that may be stored at different remote sites, for

replicating data at multiple sites with coordinated update and maintenence of logical consistency, and for distributed concurrency control and access management. True distributed database is still a research consideration. Commercial products are making some progress in this direction, but usually with many update and concurrency restrictions and often with severe performance penalties. True heterogeneous distributed database management, with different implementations at each remote site, is still some years away. A NIST publication, Guide to Distributed Database Management [10], defines and describes the characteristics and options of distributed database management.

In the near term distributed processing and interoperability are possible. Distributed processing implies client/server access to remote sites, full capabilities for data definition and data manipulation, and standardized transfer of data parameters and query responses across the communication line; however, management of multiple remote sites remains the responsibility of the client process. Distributed processing also means that all interchange protocols are standardized for "pairwise" connections, including two-phase commit protocols, but that coordinated management of multiple remote connections is non-standard. Interoperability means that database statements can be prepared at one site and executed at multiple other sites. This allows development of application programs that access and manipulate remote data, but may still require that distributed transaction management be the responsibility of the application itself. In some cases, one database management system may act as a client application to access remote data, and then present a view of that data to a local user as if it were local data. In this way, standardized distributed processing can be used to simulate true distributed database.

9.1 Remote Database Access

The emerging standard for Remote Database Access (RDA) provides protocols for establishing a remote connection between a database client and a database server. The RDA standard will enable distributed processing in a client/server SQL environment, i.e. an environment with standard-conforming RDA/SQL servers at each remote node in a communications network. RDA specifies a two-way connection between a client and a server, as well as transfer syntax and semantics for SQL database operations. The client is acting on behalf of an application program or remote process, while the server is interfacing to a process that controls data transfers to and from a database. The communications protocols are defined in terms of OSI standards for Association Control (ACSE), Remote Operations (RO), Transaction Processing (TP), and Commitment, Concurrency and Recovery (CCR). The goal is to promote distributed processing by standardizing the interconnection among SQL database applications at non-homogeneous sites.

The RDA specification is under development by Joint Technical Committee One (JTC 1) of the International Organization for Standardization (ISO) and the International Electrotechnical Commission (IEC). It is specified in two parts, a Generic RDA [3] for arbitrary database connection and an SQL Specialization [4] for connecting databases conforming to Database Language SQL. The initial specifications for both pieces have been completed, but active refinements are still underway. The formal review process began in early 1990 with final adoption expected in 1992.

There are no known existing implementations of these impending RDA standards, but many SQL vendors are planning to have conforming products available soon after final adoption. A consortium of SQL vendors known as SQL Access hopes to demonstrate interconnection of working prototypes in early 1991.

Proposed Generic RDA

The proposed Generic RDA standard [3] provides an RDA Service Interface and an RDA Communications Element that exist at both the client and server sites. The Generic RDA Service Interface consists of service elements for association control, for transfer of database operations and parameters from client to server, for transfer of resulting data from server to client, and for transaction management. Association control includes establishing an association between the client and server remote sites and managing

connections to specific databases at the server site. Transaction management includes capabilities for both one-phase and two-phase commit protocols.

The RDA Communication Element at the client site converts RDA service requests into the appropriate underlying RO, ACSE, and TP protocols as part of an open systems interconnection. The RDA Communication element at the server site converts received protocols into requests to its underlying database management system.

The Generic RDA service does not specify the syntax or semantics of database operations sent from client to server. Instead, the standard assumes the existence of a language specialization (e.g., IRDS or SQL) that specifies the exact transfer syntax for standard operations.

Proposed SQL Specialization

The RDA/SQL Specialization [4] complements the Generic RDA standard for use when a standard conforming SQL data manager is present at the server location. The client site may also have an SQL conforming data manager, but this is not required. The client processor transforms user requests into the appropriate standard protocols for transmission across the network to the server site. SQL data operations are sent as character strings conforming to the SQL language and are packaged in an RDA/ASN.1 envelope that allows for embedded parameter values to be sent with the data operation. The result of an SQL statement will contain status code and exception code parameters and may contain one or more rows of data in response to a query. The transfer syntax for all such data is specified in ASN.1 format as part of the SQL Specialization standard.

It is expected that the RDA/SQL Specialization will become the basis for all interconnection among SQL database management products from different vendors. Interconnection among database products from the same vendor will likely continue to use vendor specific communication and interchange forms.

9.2 Distributed Database Requirements

A totally integrated and fully-functional distributed database, with heterogeneous SQL systems at each local node, is not possible with existing or proposed SQL/RDA standards. Thus the CALS requirement for distributed database management cannot be met totally with existing standards efforts. RDA does not specify the syntax for invoking RDA services from an application program, i.e. an Application Program Interface (API), and RDA does not provide any tools at the SQL schema level for federating existing remote databases, partitioning an existing local database into distributed components, or replicating data at multiple sites. Also, RDA does not specify how a global transaction manager might handle multiple outstanding connections.

In spite of the current limits of the RDA specification, new standards for distributed database management, and distributed transaction processing [5], can be built on top of existing SQL and RDA standardization efforts. Therefore, the SQL and RDA efforts are vital building blocks for future distributed database management standards.

It is very important that future standardization efforts take into account CALS requirements for distributed database management. The schema statements that need to be specified include distributing a global schema into local components or combining previously defined local components into a global schema. Some of this work will be included in the SQL3 specification, but much more work needs to be done. New standards for truly distributed database management, built on top of distributed transaction processing (DTP) standards, will emerge later in CALS Phase III.

In addition to the specification and standardization of distributed database management, the architecture for using distributed database management effectively based on CALS requirements must also be specified.

Some of the architectural issues include the distribution of objects (hardware, software, data, and control), distribution transparency (the degree of visibility of the location of the data in the system), and the domain of controls for establishing and maintaining policies for accessing and updating the data (central, local, or hybrid).

10. Database Export and Import

Database export is the act of unloading a database definition and the data contents of an existing database into an external form, representable on various media, for the purpose of later automatic re-generation. Database import is the act of loading a database definition and contents from an external source. These activities are needed for the efficient management of any database, but are especially critical to support data administration functions such as backup, recovery, archiving, and database interchange in a heterogeneous distributed processing environment.

If database export/import occurs under the control of a single homogeneous database management system, then there is very little need for standardization of the external representation. All that would be required is that the vendor specific representation of database objects be structured from primitive elements (e.g., 8-bit bytes, ASCII characters, etc.) that are representable on the desired exchange medium (e.g. tape, disk, OSI). The meaning of the transfer syntax would be known at both source and target, so the only standardization requirement might be for command syntax (e.g LOAD, UNLOAD) to convey a user's request to the system. This is the current state of affairs in existing commercial SQL conforming systems as the SQL standard is silent on external representations. This approach is unacceptable for CALS Phase II because it would require a single vendor's product at each node of the distributed network.

Most existing SQL systems are able to unload database contents into ASCII format in human readable tables, with data elements represented in their "literal" form as character strings and with elements, rows, and tables separated by some user specified delimiters. A crude form of database interchange, marginally acceptable for CALS Phase I, can then be achieved. This is usually accomplished by interchanging schema definitions as character strings via the existing standard SQL schema definition language. Then, with human intervention to execute the proper load and unload commands, the database contents can be moved from source to target system in ASCII tabular format. This approach is also unacceptable for CALS Phase II because it requires either a non-standard global data manager or direct human intervention.

CALS Phase II has the goal that all data be interchangeable among multiple vendors and products. A necessary step toward this goal is the requirement that all data under the control of any standard conforming SQL data managers be readily accessible and mutually interchangeable. This requirement is partially addressed by the forthcoming RDA standard which provides standardized access to conforming SQL databases. The data exchange section of the RDA/SQL Specialization [4] is particularly important because it specifies an OSI/ASN.1 external exchange representation for sequences composed of any data type defined in the SQL standard. This enables the result of an SQL query to be exchanged among conforming systems. Unfortunately, this section of the RDA/SQL Specialization does not yet address the new data types defined in the forthcoming SQL2 standard. These deficiencies, and other CALS requirements, can be addressed in the near term as addenda to the RDA/SQL Specialization.

The RDA/SQL Specialization is not of itself sufficient for the unrestricted export/import of SQL databases. It does not provide for the exchange of whole schema definitions or multiple tables as one exchange object. What is needed in addition to RDA is a standard specification of a complete Export/Import facility, with syntax for invoking load and unload functions and with options to include or not include definitions for domains, assertions, or other integrity constraints associated with each exported object. It should be possible for a user to specify that a collection of tables and table definitions satisfying specific conditions

and including specific integrity constraints be exported from System A to System B and that specific data occurrences be included as part of the package. For example, it should be possible to specify that all schema objects and all data occurrences associated with a given project be exported from System A and imported into System B, all as part of a single SQL transaction.

The need for a standard Export/Import facility has already been recognized by ISO/IEC JTC1/SC21 with the approval of a new project titled "Data Management: Export/Import." In addition to its obvious support for data administration, it is likely that export/import facilities defined in this standard will provide the basis for specification of "fragmentation" or "replication" schemas in a subsequent distributed DBMS standard. Proposals to support federation and fragmentation of existing databases have already been considered by the SQL3 development committees, but no specification has yet been accepted into the SQL3 document.

11. Other Proposed Features

The following features have been proposed for SQL3 to enhance the general capabilities available in the language. In general these features are somewhat controversial. Proponents regard them as "fundamental tools" that can be used as building blocks for data abstraction, whereas opponents regard them as superficial and cite them as examples of "creeping featurism" in the language. A decision will be made by the SQL standardization committees over the next few years as to the appropriateness of these kinds of facilities.

Multiple Unknown Values

The SQL3 specification includes a simple specification for user-defined *null* values. This feature permits definition of an arbitrary number of application specific *null* values, such as "Unknown", "Missing", "Not Applicable", or "Pending", each with a different representation in the database so that they can be distinguished upon retrieval. No attempt is made to define logical or arithmetic operations over different *null* values; instead, all such operations default to the "general" *null* value as specified in the existing SQL standard. Although simple to implement, this feature allows a CALS application to specify semantic differences to missing information without dedicating specific data values and special tests from the underlying domain.

Enumeration Data Type

The new Enumeration data type in SQL3 is controversial because it provides yet another alternative, in addition to domains, for data definition. Proponents argue that it is necessary to ensure that all enumerations are defined the same way so that they can be generically optimized and mapped one-to-one to the Enumeration type in many programming languages. Using the Enumeration data type, the domain Month defined above might have, instead, been defined as:

```
Month (Jan,Feb,Mar,Apr,May,Jun,Jul,Aug,Sep,Oct,Nov,Dec)
```

Proponents of a distinct Enumeration type also point out that it has an implied ordering of its component parts, whereas a domain definition limited to those same values does not. Thus one can have meaningful inequality comparisons or orderings for Enumeration types that may not be possible on a domain defined over character strings.

The rules for an Enumeration data type require that the items in the enumerated list be character string identifiers. They are tested in predicates either by position location or by literals of the form Month::Apr and are passed to application programs by position number. Using the SQL2 Cast function to cast values from Enumeration types to Character types, and vice versa, a user can always control whether an Enumeration value is seen as an integer value, representing its position in the enumeration, or as its character string equivalent.

Cursor Sensitivity

In the SQL-1989 standard, it is implementor defined whether an Open cursor statement results in a base table or a viewed table. Thus, when the Declare cursor statement

> Declare C Cursor for Select * From T Where <search condition>
> Order By T.colx

is opened with an "Open C" statement, the implementation may choose to construct a new base table, separate from T, that contains all the rows of T that satisfy the <search condition>, or the implementation may choose to construct a view over T with the <search condition> as a filter. The choice of whether to construct a base table or a viewed table usually depends upon the physical organization of the database, and the "wrong" choice may have severe performance implications.

If a user executes an Update, Insert, or Delete statement on table T while Cursor C is open, then the choice of implementation method for the Open Cursor may determine whether or not the user sees the modifications to T through subsequent Fetch statements via Cursor C. The resulting uncertainty is called the "cursor sensitivity problem". It is a problem, because the obvious solution, i.e. for the standard to just pick one way of doing it, is not the correct technical solution in too many situations.

In SQL-1989, the user is responsible for constructing application program logic to avoid the cursor sensitivity problem. This includes self-imposed restrictions to not modify ordering columns for the cursor and to avoid any situations that may affect subsequent Fetch statements. The SQL2 specification allows an Insensitive option on the Declare Cursor statement, which requires that the effect of any other changes to the database, even by the same user in the same transaction, will not be visible through that cursor, unless it is closed and re-opened. An alternative Sensitive option, to require that all modifications by the same user in the same transaction be visible through the cursor, is under consideration for SQL3, but is very controversial. Other tools to ameliorate the cursor sensitivity problem are also under consideration for SQL3.

Partially Null Foreign Keys

In the SQL-1989 standard, it is possible to declare a foreign key in a table that references the primary key of another, possibly the same, table. The implementation is required to ensure that no rows in the foreign key table have a foreign key value different from some primary key value in the primary key table. However, in SQL-1989, this rule only applies if all of the foreign key values are non-*null*. If a foreign key is a multiple column key, and if any one of the columns in the foreign key has a *null* value, then the "matching primary key" rule is not enforced. This allows an application to insert new rows into the foreign key table with "missing" information for foreign key values and not have the row "connected" to a parent row until all of the foreign key values are non-*null*, thereby uniquely determining a parent row.

In SQL2, the Cascade Update and Cascade Delete options on a referential integrity constraint require the implementation to maintain consistency among a primary key row and its foreign key row during data modification on the primary key table. However, there is no requirement to maintain consistency for

partially *null* foreign keys. The SQL3 specification contains options to require cascade support for partially *null* keys, if desired.

Pendant

Pendant is a proposed enhancement to referential integrity constraints that would allow backward cascade effects from the foreign key table to the primary key table. If pendant is specified, then when the last foreign key dependent of a primary key row is deleted or otherwise removed from the relationship, the primary key row is also deleted. This capability is desirable in those applications where a primary key row is meaningless unless it has properties determined by its foreign key row.

An example of a pendant relationship may be a situation where one employee acts as an advisor to another group of employees. When the last dependent advisee "graduates" or leaves the program, then the advisor ceases to play that role. The pendant feature is controversial because most of its functionality can be achieved through the use of SQL triggers. Proponents claim that it is difficult for triggers to model all aspects of pendant and that it is sufficiently important to deserve a separate syntax to enable implementations to optimize the desired effects.

Nullary Keys

A nullary key [19] is a primary key or foreign key declaration that does not reference any columns. The effect is to specify a table that contains exactly one row. This concept is useful to represent explicitly in the database values that are implicit in the processing environment, for example, the name of the manufacturing plant in a manufacturing database. In the absence of nullary keys, it is difficult to represent the concept of a group of tables, all associated with a single topic, without propagating a value for that topic throughout the database.

For example, consider a company named ACME that has a manufacturing database with tables for personnel, parts, projects, etc., but no efficient way to indicate that they are all part of the same enterprise. The natural language query

"Find all ACME employees who work on project X"

may fail because the natural language processor has no knowledge of the company name ACME unless that name is somehow represented in the database. Proponents of nullary keys argue that occurrences of such situations are pervasive in natural language processing and that nullary keys are the appropriate tool to address the problem.

In the above example, the database administrator could define a link from the Personnel and Projects tables to the Company table as follows:

```
Company (Company_Id, Name, etc. )
        Primary Key ()

Personnel (Emp_Id, Name, etc. )
        Primary Key (Emp_Id)
        Foreign Key () References Company

Projects (Project_Id, Name, etc. )
        Primary Key (Project_Id)
        Foreign Key () References Company
```

The effect of the nullary primary key in the Company table is to require the existence of exactly one company. The nullary foreign keys then establish a one-to-many association from the single row of Company to every row of Personnel and every row of Projects. An alternative method for establishing this association is to declare Company_Id the primary key of the company table and have every other table in the database link to Company_Id in a foreign key declaration. However, this alternative has the undesirable effect of propagating Company_Id values throughout every row of every table in the company database.

Asynchronous DML

A proposed SQL3 feature is the ability to execute SQL statements asynchronously. An application would be able to execute a statement that is known to be lengthy, and then do other work while waiting for a result to come back. Any SQL statement executed asynchronously is assigned an identifying "handle" by the system, which can be used to monitor progress using the Test Completion statement. For example, the following statements are valid, where <statement list> is a list of "handles" assigned as identifiers for asynchronous statements.

Test Any <statement list> Completion

Test All <statement list> Completion

Wait Any <statement list> Completion

Wait All <statement list> Completion

If Wait Any is specified, then the database implementation waits until at least one outstanding asynchronous SQL statement in the statement list has completed that has not already been tested for successful completion. The Test Any statement could then be used to determine which statements have completed.

Synchronous and asynchronous execution of SQL statements may be intermixed, but the effect of the statements is the same as if they had all been executed synchronously in the order in which they were initiated. The main benefit of asynchronous execution is that it allows the database application to communicate with the application user in order to keep the user informed as to the progress of specific requests.

Software Engineering Tools

SQL may provide an avenue for the integration of Computer-Aided Systems Engineering (CASE) tools with DBMS (see [8]). Such integration is desirable in CALS Phase III and appropriate facilities can become part of SQL3 development.

- Bibliography -

[1] ANSI, "American National Standard - Database Language SQL with Integrity Enhancement," Number X3.135-1989, American National Standards Institute, 1989. Revision of ANSI Standard X3.135-1986.

[2] ANSI, "American National Standard - Database Language - Embedded SQL," Number X3.168-1989, American National Standards Institute, 1989.

[3] ISO, "Information Processing Systems - Open Systems Interconnection - Remote Database Access - Part 1: Generic Model, Service, and Protocol," ISO CD 9579-1, Document ISO/JTC1/SC21 N4282, March 1990.

[4] ISO, "Information Processing Systems - Open Systems Interconnection - Remote Database Access - Part 2: SQL Specialization," ISO CD 9579-2, Document ISO/JTC1/SC21 N4281, February 1990.

[5] X/Open, "Resource Manager Requirements for Distributed Transaction Processing," X/Open Transaction Processing Working Group, April 6, 1989.

[6] NIST, NISTIR 89-4140 "Working Implementation Agreements for Open Systems Interconnection Protocols," August 1989.

[7] NIST, "Use of the IRDS Standard in CALS," D.Jefferson and C.Furlani, April 3, 1989.

[8] Edelstein, Herb; "Tools to Database: Hey, Can We Talk," Special Section on DBMS: Case Tool Integration, Software Magazine, July 1989.

[9] FIPS, "Federal Information Processing Standard - Database Language SQL," FIPS PUB 127-1, U.S. Department of Commerce, National Institute of Standards and Technology, February 2, 1990.

[10] NIST, NBS Special Publication 500-154, Guide to Distributed Database Management, April 1988.

[11] ISO, "Database Language SQL2," ISO/IEC Committee Draft 9075 Revised, Document JTC1/SC21 N5215, 480 pages, July 1990. Proposed revision of ISO 9075:1989, the international equivalent of ANSI X3.135-1989 above. Available in the U.S. from Global Engineering (800-854-7179) as document BSR X3.194-199x.

[12] ISO, "Database Language SQL3," Working Draft, Committee document ISO JTC1/SC21/WG3 DBL SEL-3b, 790 pages, April 1990, also referenced as X3H2-90-181.

[13] ISO, "Common Language-Independent Datatypes", Working Draft #3, document JTC1/SC22/WG11 N162 or X3T2/90-087, March 1990.

[14] NIST, "Use of the SQL and RDA Standards in CALS", L. Gallagher, J. Sullivan, J. Collica, CALS Deliverable, July 31, 1989.

[15] Smith, Gordon and Ken Jacobs; "Roles for better security", document X3H2-89-343 or ISO DBL/SEL-9, November 17, 1989.

[16] Jacobs, Ken; "Savepoints: An extension to ROLLBACK", documents X3H2-89-170 and X3H2-89-205 or ISO DBL/SEL-10, November 12, 1989.

[17] Beech, David; "Exists as a true quantifier", documents X3H2-90-40 and X3H2-90-124 or ISO DBL/SEL-70, January 17, 1990.

[18] Shaw, Phil; "Recursive expressions", documents X3H2-87-330, X3H2-88-63, X3H2-88-93Rev, and X3H2-88-122 or ISO DBL/CPH-20 and 21, April 1988.

[19] Darwen, Hugh; "Justification for nullary primary and foreign keys", document X3H2-89-261 or ISO DBL/FIR-36.

ST 14A
EV -90)

U.S. DEPARTMENT OF COMMERCE
NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY

BIBLIOGRAPHIC DATA SHEET

1. PUBLICATION OR REPORT NUMBER
NISTIR 4494

2. PERFORMING ORGANIZATION REPORT NUMBER

3. PUBLICATION DATE
December 21, 1991

TI E AND SUBTITLE

QL Support for CALS Applications

AU IOR(S)

C ard Gallagher

PE ORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS)

DEPARTMENT OF COMMERCE
ONAL INSTITUTE OF STANDARDS AND TECHNOLOGY
HERSBURG, MD 20899

7. CONTRACT/GRANT NUMBER

8. TYPE OF REPORT AND PERIOD COVERED
NISTIR

SP NSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

SU LEMENTARY NOTES

AB RACT (A 200-WORD OR LESS FACTUAL SUMMARY OF MOST SIGNIFICANT INFORMATION. IF DOCUMENT INCLUDES A SIGNIFICANT BIBLIOGRAPHY OR LITERATURE SURVEY, MENTION IT HERE.)

ous reports to CALS have identified the importance of Database Language SQL in CALS
: II requirements. In particular, a July 1989 point paper on SQL and RDA identified
ires in the existing SQL standard and its near-term SQL2 replacement that are most
opriate to CALS data management concerns. This report focuses on SQL3, a follow-on
dardization project for major new SQL enhancements that is expected to be adopted by ANSI,
and as a FIPS in the mid 1990's. Many of the proposed SQL3 features are of particular
tance to the Standard for the Exchange of Product model data (STEP) because of that
dard's unique data modeling and data access requirements. Existing and planned features
QL3 may not satisfy all STEP requirements, but they should provide an appropriate base
which many requirements can be suitably addressed. Since features in SQL3 are just now
specified, they are open to modification and improvement to best suit CALS needs. This
t identifies the major enhancements under consideration by the ANSI and ISO SQL
dardization committees and relates them to known STEP requirements. It also discusses the
s of these features in the SQL3 specification and indicates opportunities available to
to influence further development.

KE ORDS (6 TO 12 ENTRIES; ALPHABETICAL ORDER; CAPITALIZE ONLY PROPER NAMES; AND SEPARATE KEY WORDS BY SEMICOLONS)

action; CALS; database; distributed database; encapsulation; inheritance; multi-value
; recursion; savepoints; SQL; standards; STEP

AV ABILITY

UNLIMITED

FOR OFFICIAL DISTRIBUTION. DO NOT RELEASE TO NATIONAL TECHNICAL INFORMATION SERVICE (NTIS).

ORDER FROM SUPERINTENDENT OF DOCUMENTS, U.S. GOVERNMENT PRINTING OFFICE, WASHINGTON, DC 20402.

ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.

14. NUMBER OF PRINTED PAGES
38

15. PRICE
A03

ONIC FORM