NISTIR 4902

Database Language SQL: Integrator of CALS Data Repositories

Leonard Gallagher
Joan Sullivan

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Information Systems Engineering Division
Computer Systems Laboratory
Gaithersburg, MD 20899

NIST

[illegible]
4902
1992

NISTIR 4902

c. 2

Database Language SQL: Integrator of CALS Data Repositories

Leonard Gallagher
Joan Sullivan

U.S. DEPARTMENT OF COMMERCE
Technology Administration
National Institute of Standards and Technology
Information Systems Engineering Division
Computer Systems Laboratory
Gaithersburg, MD 20899

September 1992

Database Language SQL: Integrator of CALS Data Repositories

Leonard Gallagher
Joan Sullivan

National Institute of Standards and Technology
Information Systems Engineering Division
Gaithersburg, MD 20899, USA

CALS Status Report on SQL and RDA

- Abstract -

The Computer-aided Acquisition and Logistic Support (CALS) program of the U.S. Department of Defense requires a logically integrated database of diverse data, (e.g., documents, graphics, alphanumeric records, complex objects, images, voice, video) stored in geographically separated data banks under the management and control of heterogeneous data management systems. An over-riding requirement is that these various data managers be able to communicate with each other and provide shared access to data and data operations and methods under appropriate security, integrity, and access control mechanisms. Previous reports to CALS have identified the importance of Database Language SQL and its distributed processing counterpart, Remote Database Access (RDA), for their ability to address a significant portion of CALS data management requirements. This report presents the new "Object SQL" facilities proposed for inclusion in SQL3, introduces SQL abstract data types (ADTs), discusses the benefits of "generic ADT packages" for management of application specific objects, and proposes a new external repository interface (ERI) that would allow integration of heterogenous, non-SQL data repositories. The appendices give the current status and applicability of national, international, and Federal standards for SQL and RDA, discuss the availability of conforming implementations, and present the status of NIST SQL and RDA validation testing.

Keywords: (ANSI; ASN.1; CALS; CLID; conformance; database; data model; FIPS; GIS; GOSIP; ISO; IWSDB; MILSTD; object; object-oriented; PDES; RDA; relational; standard; SQL; STEP; testing)

Note: This document was prepared as a draft report to the CALS Evaluation and Integration Office. Its publication by NIST does not imply CALS endorsement of the conclusions or recommendations presented.

Table of Contents

Database Language SQL: Integrator of CALS Data Repositories

Leonard Gallagher
Joan Sulliven

1. Introduction

Data management requirements for the U.S. Department of Defense's Computer-aided Acquisition and Logistic Support (CALS) program often exceed the capabilities of existing database management systems. CALS requires a logically integrated database of diverse data (e.g., documents, graphics, alphanumeric records, complex objects, images, voice, video) stored in geographically separated data banks under the management and control of heterogeneous data management systems. An over-riding requirement is that these various data managers be able to communicate with each other and provide shared access to data and data operations and methods under appropriate security, integrity, and access control mechanisms.

Previous reports to CALS [28,29,30] have identified the importance of Database Language SQL and its distributed processing counterpart, Remote Database Access (RDA), for their ability to address a significant portion of CALS data management requirements. SQL is particularly appropriate for the definition and management of data that is structured into repeated occurrences having common data structure definitions. SQL provides a high-level query and update language for set-at-a-time retrieval and update operations, as well as required database management functions for schema and view definition, integrity constraints, schema manipulation, and access control. SQL provides a data manipulation language that is mathematically sound and based on a first order predicate calculus. SQL is self-describing in the sense that all schema information is queryable through a set of catalog tables. Features of the existing SQL standard (see Appendix A) and of its near-term, late-1992 enhancement, often called SQL2, are discussed in a 1989 report to CALS [29]. Preliminary features of an emerging substantial SQL enhancement, often called SQL3, its applicability to the Standard for the Exchange of Product Model data (STEP), and the status of proposed standards for Remote Database Access (RDA) are contained in a 1990 report to CALS [30].

Early in 1991, technical committees for SQL standardization, operating under the procedures of the American National Standards Institute (ANSI) and the International Organization for Standardization (ISO), committed to enhancing SQL into a computationally complete language for the definition and management of persistent, complex objects. This includes the specification of abstract data types, object identifiers, methods, inheritance, polymorphism, encapsulation, and all of the other facilities normally associated with object data management. Preliminary specifications for these facilities are contained in the current SQL3 Working Draft [1].

This report focuses on the new object-oriented facilities proposed for inclusion in SQL3. These facilities replace the user-defined data type and inheritance sections of [30]. All of the other features discussed in [30] are still valid and need not be repeated here. In addition, this report places SQL and RDA in a simplified CALS architecture and points to new opportunities for future enhancements of SQL. In particular, Section 2 focuses on a three-schema data management architecture with an emphasis on using SQL and RDA to integrate local and remote data repositories, including both SQL and non-SQL data. Section 3 describes the preliminary status of object-oriented facilities proposed for SQL3; these facilities are subject to revision and improvement as they evolve over the next two or three years before final standardization. Section 4 looks at future opportunities and focuses on the potential benefits of defining a collection of standard "generic ADT packages" as the basis for SQL management of objects common to a number of application areas. Section 5 proposes a new SQL interface that would allow non-SQL data repositiories to make their data and special methods available, in a standard manner, to full-function SQL systems and to SQL applications. Section 6 draws some conclusions about the benefits of existing SQL and RDA standards as well as future opportunities.

Previous reports to CALS have focused on functionality of proposed standards rather than conformance, testing, and validation issues related to vendor implementations of existing standards. We include in this report a number of appendices that identify the exact status of standardization and validation testing for SQL and RDA products. Appendix A gives the status of ANSI, ISO, and Federal Information Processing Standards (FIPS) for SQL, and Appendix B does the same for RDA. They describe the general content of each standard and discuss its applicability, availability, completeness, maturity, stability, existing usage , and known limitations. Appendix C focuses on NIST validation testing and Appendix D on obtaining copies of specifications.

2. CALS data management architecture

In the Integrated Weapons System Database (IWSDB) described in previous CALS reports, we assume two or more data processing environments communicating with one another at the highest conceptual schema levels. This ensures that the communicating environments share a common understanding of business semantics of the data. This is a desirable goal that depends heavily on data management standards not yet fully developed (e.g., PDES).

At the other extreme, in the absence of any data management standards, two sites can only communicate at the very lowest levels. Each site may be able to access files of data or "bulletin-board" views of data at other sites, or it may be able to pass parameters to application processes at remote sites, but it is not possible to access the external schema logical views of the data without knowing the external data model employed at the remote site and a syntax or protocol for invoking operations on that data model. Unless there are standards for schemas, access to remote sites may be effectively limited to the use of very low-level external schemas, thereby limiting common understanding to very basic syntax and semantics. This low-level communication forces CALS application programs to perform many of the data structuring and re-structuring tasks that could be performed more effectively by a database management system.

Phased implementation of CALS requirements allows gradual adoption of existing and emerging data management standards to move from low-level communication, through logical operations at the external schema level, toward the desired IWSDB goal of high-level conceptual object interoperation.

2.1 Three schema architecture

Data management has traditionally employed a three-schema architecture to place itself in a data processing environment. A conceptual schema represents a high-level, enterprise-wide view of all data, data relationships (including rules restricting updates or cascading the effects of updates to related data), and the business processes that use and update the data. Generally an enterprise has only one conceptual schema. Changes in the details of computer implementation or the specific human users and application programs that access the data have no effect on the conceptual schema.

An external schema represents a logical view of the data as accessible to a set of human users and application programs; an enterprise may have many external schemas. An external schema is generally a small subset of the conceptual schema, and may have application-oriented views of the data defined in the conceptual schema. Changes in computer implementation have no effect on an external schema; this facilitates migration of the data to other hardware and software environments. An external schema may change to accommodate changes in the use of the data.

An internal schema represents a physical view of the data as stored on persistent storage devices. An enterprise may have many internal schemas to provide efficiency on a variety of hardware and software environments. Conceptual and external schemas are independent of the structures and access methods of any underlying file system; in contrast, an internal schema may be heavily dependent on file structures and access methods.

Each schema is constructed according to the rules of a data model, (e.g., SQL is based on the relational data model). The data model prescribes not only the rules for defining data structures, but also the rules for interpreting and manipulating the data structures.

The conceptual schema may consist of a very large collection of object types and their interrelationships; no single application program wll require access to all the objects described by the conceptual schema. In contrast, an external schema may consist of a simple "record-oriented" view of a single object type; a third generation programming language can easily process data described by such a schema. The conceptual schema itself may be so complex that it must be maintained by specialized software such as an Information Resource Dictionary System (IRDS). The IRDS may also be required to manage the mappings between the conceptual schema and the different external schemas, and the relationships among data and processes.

2.2 External schema communication

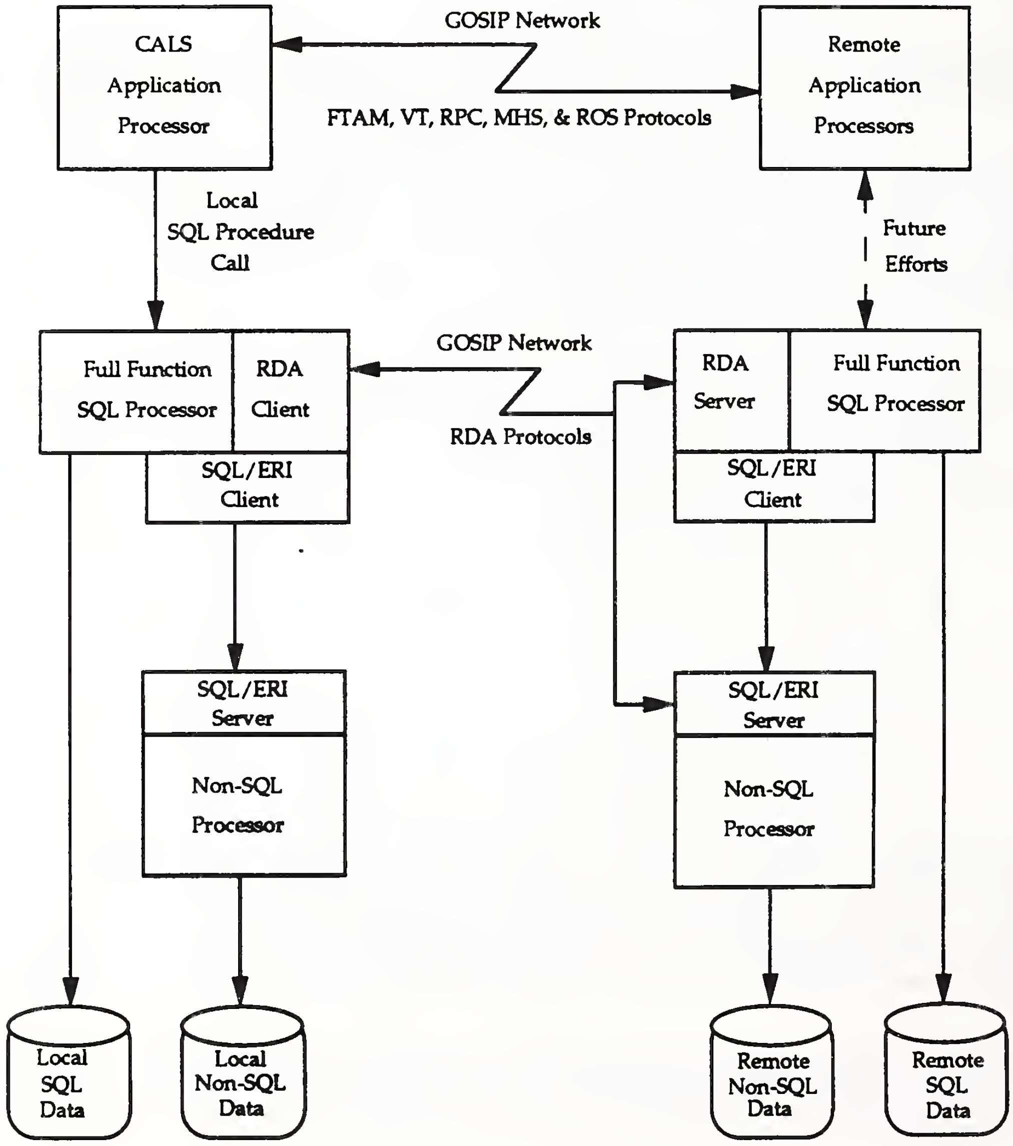
Standard communication among cooperating systems is possible at the present time using the government open systems interconnection profile (GOSIP). Currently, the application layer of GOSIP contains standards for association control (ACSE), file transfer (FTAM), virtual terminal (VT), and electronic mail message handling systems (MHS). The next version, GOSIP 3 [31], is expected in early 1993 and will likely contain extensions of these facilities as well as remote database access (RDA) and additional facilities for handling documents (ODA/ODIF), electronic data interchange (EDI), and remote operations (ROS). Extensions to MHS and ROS should make it possible for user-defined objects at various remote sites to communicate their existence and provide access to their methods to application processors. Objects at remote sites may be able to "show themselves" to users at local workstations by using the graphical user interfaces proposed for future versions of VT.

We believe that the RDA component of GOSIP will provide the basis of distributed access to external schema definitions and to object data instances. In particular, existing and very near-term SQL and RDA standards will make it possible for various non-homogeneous data management systems to describe themselves to external processors and to provide "standard" access to the data they manage. With implementation of an External Repository Interface (ERI), discussed in Section 5 below, it is possible for each heterogeneous external schema to be "self-describing" in the sense that it can construct a "tabular" view of its logical data structures that can then be accessed and manipulated by all other sites. With longer-term emerging data management standards that support object-oriented and knowledge-based features, an ERI interface can evolve into the desired IWSDB communication, with all sites communicating at the conceptual level and with "seamless" integration of complex, structured data and supporting application services.

We begin with a "CALS Application Processor" (see Figure on next page) that wishes to communicate with and access data at a number of different data repositories, some local and some remote. The application processor could use existing GOSIP protocols to connect to external processes or transfer files, but it would prefer not to have to manage its own communications links or worry about integrity, access control, remote transactions, or any number of different data manipulation functions; instead, it would rather communicate with a single, "familiar" data repository interface for both schema data and actual data occurrences. The "familiar" data repository could then connect itself to remote sites and access the desired data and data definitions, returning them to the accessing processor in a standard format. A remote object would still be able to use VT to "show itself" to the accessing process or use FTAM to transfer files containing objects or object definitions not under the control of the communicating data managers.

We assume the existence of any number of heterogeneous data repositories, some at the local site and some at distributed sites. We assume a full-fuction SQL processor at all sites, but not necessarily as the manager of the most important data. The non-SQL processors may control the documents, graphics images, or complex engineering structures that the application processor wishes to access. The local SQL processor conforms to Database Language SQL and has two integrated client components, one conforming to the RDA/SQL Specialization and one conforming to the SQL/ERI interface proposed in Section 5 of this report. Communications among the three SQL components are likely proprietary. The local site may have any number of non-SQL data repositories each controlled by a Non-SQL Processor having a component that conforms to the SQL/ERI interface. Communications among the internal components of the Non-SQL Processor are also proprietary. The local site has a proprietary local procedure calling mechanism and a proprietary local inter-process communications capability. Using these proprietary mechanisms the CALS Application Processor is able to issue standard SQL calls to the local full-function SQL processor, and the SQL/ERI Client component of the SQL processor is able to communicate, using an ERI specified subset of standard SQL, with the SQL/ERI Server of the Non-SQL Processor.

External Schema Communication



The local site is connected to one or more remote sites via a standard communications network (i.e., GOSIP) that is able to connect to remote processes (using ACSE) and allow "messages" or "calls" to be exchanged among processes (using ROS or MHS). Some messages may be sent directly from the CALS application processor to processes or file stores at the remote site, but idealy, some local repository manager makes a connection and sends messages on behalf of the application processor. The Generic RDA and RDA/SQL Specialization standards specify protocols that allow the RDA Client component of the local SQL processor to send SQL statements to the RDA Server component of a remote SQL processor, or the SQL/ERI Server component of any Non-SQL processor, and receive data in return. All protocols and data are defined in the RDA standards and are transmitted as ASN.1 (ISO 8825) packages. If the CALS Application Processor is operating, interactively, on behalf of a human user, then any of the data repositories may use a local graphical user interface (GUI), or non-local GOSIP VT protocols, to present status information or a "menu of choices" to the human user. In this way an interactive "browsing" or "navigational" capability is provided to the human user without losing the standard RDA/SQL protocol communications used by the non-human processors.

At the remote site there exists a full-function SQL Processor as well as any number of Non-SQL Processors. Components of the SQL Processor conform to the SQL and RDA standards, and satisfy the proposed SQL/ERI Client requirements. Each Non-SQL Processor has a component that conforms to the SQL/ERI Server specification. The remote site handles internal communications and procedure calls in the same proprietary manner as does the local site.

At the present time the RDA standard specifies interchange protocols for transmitting records of data from a server site to a client site, provided that the data items in the records are either numbers or character strings. Near term RDA follow-on specifications will extend the data types handled to all of those specified in the forthcoming SQL2 specification; i.e., fixed and variable length character strings, fixed and variable length bit strings, fixed and floating point numerics, dates, times, timestamps, and intervals. Later RDA follow-on specifications will provide interchange mechanisms, in terms of ASN.1 elements, for the user defined abstract data types (ADTs) specified in the emerging SQL3 standard. RDA protocols do not by themselves provide interchange mechanisms for other CALS data objects, so standards for IGES, CGM, SGML, EDI, ODA/ODIF, CCITT GROUP IV, and other PDES objects, will remain critical for transmitting agreed object definitions among various sites.

SQL and RDA provide the basis for standard external schema communication. An SQL external repository interface (SQL/ERI) makes it possible for non-SQL data repositories to share their data with CALS applications. With emerging SQL enhancements for object-oriented and knowledge-based data management and emerging RDA extensions for distributed database, the ERI can evolve to support the IWSDB goal of direct conceptual schema communication.

3. Object Management in SQL3

ANSI and ISO SQL standardization committees have for some time been adding features to the SQL3 specification that support object-oriented data management. Our 1990 CALS report [30] shows how Assertions, Triggers, and External Procedures support the notion of encapsulation and how Cast functions, Domains, and User-Defined data types support data abstraction. In each case the object concepts were partially supported without fully enforcing them as a discipline. In addition, a definition of subtables and gentables provided a table hierarchy that supported multiple inheritance based on rigorous enforcement of primary key occurrences, but that fell short of defining classes and subclasses for user-defined data types.

Early in 1991 both the ANSI and ISO SQL committees began discussing the requirement for true "objects" and "object identifiers" in SQL and concluded that SQL3 should provide full support for defining and managing persistent objects. As the first true object capabilities were added to the ISO specification in May, 1991, it became clear that there was a general commitment to let SQL evolve, in an upward compatible manner, into a computationally complete language for the definition and management of complex objects.

The proposed Database Language SQL3 specification [1] now has the capability to support user-defined abstract data types (ADTs), including methods [2], object identifiers and object management [3], subtypes and inheritance [4], polymorphism [5], and integration with existing facilities [6] and external languages [8]. More recent proposals have added control structures [7] and parameterized types [25, 26] to make SQL a computationally complete language for creating, managing, and querying persistent objects.

3.1 Abstract data types

As the result of work referenced above, the SQL3 specification now has facilities that provide the capability to define and manage persistent data type definitions, including structures and operations on those structures. A new ADT can be "constructed" from any existing data type, including previously defined abstract data types known to the current SQL environment.

An abbreviated version of the current syntax is as follows:

```
<ADT definition> ::=
          CREATE [ OBJECT | VALUE ] TYPE <ADT name>
                [ <subtype clause> ]
                [ ( <member defn> ... ) ]

<subtype clause> ::=  UNDER <supertype ADT name list>

<member defn> ::=
          <column definition>
        | <equals definition>
        | <less-than definition>
        | <cast definition>
        | <function definition>
        | <operator definition>

<column definition> ::=
          [ <encapsulation level> ]
          <column name> [ { <data type> | <domain name> } ]
          [ <other normal column defn subclauses> ]

<encapsulation level> ::= PRIVATE | PROTECTED | PUBLIC

<equals definition> ::=  EQUALS <equals function specification>

<less-than definition> ::=  LESS THAN <less-than function specification>

<cast definition> ::= CAST ( <datatype> AS <datatype> ) WITH <cast function specification>

<function definition> ::=  FUNCTION <function specification>

<operator definition> ::= OPERATORS <operator name> ...
```

If OBJECT is specified in an ADT definition, then a persistent object-identifier is created for each instance of the data type constructed; otherwise, if VALUE is specified, each instance represents itself just like values of primitive data types do. There is a continuing debate in the SQL standardization committees as to whether SQL should support both OBJECT and VALUE definitions, or if every new ADT definition should be assumed to carry a unique object identifier. The outcome of this debate will not affect the functionality of the new language, but it will greatly influence its appearance and style. A final decision on this philosophical point will be made within the next two or three meetings.

Specification of UNDER in a <subtype clause> permits a new ADT to be defined as a subtype of an existing ADT. A type can have more than one subtype and more than one supertype. Thus a subtype is a specialized type of one or more supertypes and a supertype is a generalized type of one or more subtypes. A supertype shall not have itself as a proper subtype and a subtype family shall have exactly one maximal supertype.

For every column definition in an ADT, an <encapsulation level> is specified as either PUBLIC, PRIVATE, or PROTECTED. Public components form the interface of the ADT and are visible to all authorized users of the ADT. Private components are totally encapsulated, and are visible only within the definition of the ADT that contains them. Protected components are partially encapsulated, being visible both within their own ADT and within the definitions of all subtypes of that ADT.

Public components of an ADT are accessible to authorized users through a special "attribute reference" operator (i.e., <ADT reference>.<column name>). If the ADT is an OBJECT ADT then the <ADT reference> will be to an object identifier of a specific ADT; if the ADT is a VALUE ADT then the <ADT reference> will be to the ADT itself. The attribute reference identifies a specific component of the ADT instance and permits the user to read or modify its value.

The EQUALS definition identifies a function that specifies conditions under which two instances of the defined data type are considered to be equal. The LESS THAN definition identifies a function that specifies a sequential ordering over data type instances. The definitions of equality and sequential ordering, taken together, specify the semantics to be used in the SQL comparison predicate when applied to ADTs. If the LESS THAN definition is not specified, then the ordering tests of the comparison predicate cannot return a true or false value.

The CAST definitions specify how an ADT may be mapped to other existing data types. For example, the DATETIME data type in the current SQL2 specification, having components for year, month, day, hour, second, and fractional-second, may be CAST to any one of the components or to a character string representation as specified by ISO standards for date and time. With the ability to include CAST specifications in any ADT definition, a data type definer can define mappings to specific external representations. In this way the internal representation may be kept PRIVATE and not directly accessible, thereby allowing efficient implementation.

The FUNCTION and OPERATOR definitions specify the data operations on the ADT and return either BOOLEAN, if the result is to be used as a truth value in a Boolean predicate, or a single value of a defined data type, if the result is to be used as a value specification. Functions may be SQL functions, completely defined in an SQL schema definition, or they may be external function calls to functions defined in standard programming languages. Special "constructor" and "destructor" functions are defined to make or remove instances of an OBJECT abstract data type. At the present time constructor and destructor functions are invoked implicitly through Insert or Delete operations on a table.

Abstract data type definitions and SQL or external function declarations are treated just like any other SQL objects as far as access control and other usage privileges are concerned. Access control is independent of encapsulation since encapsulation defines the structure of what is possible to see and access control determines who can see that structure. All names are qualified by the schema name of the containing schema and by the catalog name of the containing catalog. Each such object is "owned" by the authorization identifier of the schema in which it is defined and all privileges to use the object, to "see" its representation, or to modify its definition must be explicitly granted by the object owner. Privileges on existing ADTs may be GRANTed and REVOKEd and new ADTs or function declarations may be ADDed or DROPed from the schema as part of the SQL schema manipulation language.

The following subsections expand on some of the notions introduced here and discuss other facilities normally associated with object management.

3.2 Object identifiers

Object identity is that aspect of an object that never changes and that distinguishes the object from all other objects. Ideally, an object's identity is independent of its name, structure, or location. Object identity is therefore a unique identification of an object that is independent of the state of that object, and which persists over time. The identity of an object persists even after the object no longer exists (e.g., like a timestamp), so that it may never be confused with the identity of any other object. Other objects can use the identity of an object as a unique way of referencing it.

Object ADTs are subject to special "constructor" and "destructor" operations that either create a new instance of the ADT and make it part of the database or remove an instance of an ADT from the database. Since SQL is a "table based" language, SQL designers have to address issues concerning whether or not SQL objects may exist outside of table occurrences. If SQL objects are allowed to exist outside of tables, then new functions to create and destroy objects must be added to the language and new structures (e.g., the set of all objects of a given class) must become part of the language. Although these issues are still subject to debate and modification, the current status is to allow the existence of a "tabular" shell over an ADT class. In this way, constructor and destructor functions are automatically invoked when rows are inserted or deleted from the table, and the table itself is the collection of all object occurrences. SQL query and update statements may then be applied to the table without the need for any special language enhancements. If a requirement surfaces later to allow objects to exist independently of tables, this can be handled as an upward compatible language enhancement.

In order to accommodate this implicit invocation of constructor and destructor functions, minor enhancements are needed to the syntax and semantics of the CREATE TABLE and INSERT statements (see [6]) as follows:

```
CREATE TABLE <table name> OF <ADT name> [<table element list>]

CREATE TABLE <table name> OF NEW TYPE <ADT name> [ OBJECT ] <table element list>

INSERT INTO <table reference> <insert table values> [ ALIAS <target specification> ]
```

In the first alternative of CREATE TABLE; i.e., when "OF <ADT name>" is specified, the table definition creates a tablular envelope around the abstract data type specified by <ADT name>. All column definitions of the abstract data type become column definitions of the new table. The optional <table element list> can add additional columns to the table or add table constraints or referential integrity constraints.

In the second alternative of CREATE TABLE; i.e., when "OF NEW TYPE <ADT name>" is specified, the table definition is a combined new table definition and a new abstract data type definition. If OBJECT is specified, then the new abstract data type is an OBJECT type with each row having an object identifier (OID); otherwise, it is a VALUE type and rows do not have object identifiers.

With both CREATE TABLE alternatives, every insert statement applied to that table results in invocation of the underlying constructor function for the corresponding ADT and every delete statement applied to the table results in invocation of the underlying destructor function. If the table is defined as either a supertable or a subtable of some other table, then the constructor or destructor functions are applied recursively, as appropriate, to the other tables in the same subtable family.

The ALIAS option in the INSERT statement is simply a facility for returning the "handle" of any new object created by that insert statement. The object identifier of any newly created object is assigned to the <target specification>; syntax rules prohibit specification of an alias when the underlying ADT is not an OBJECT ADT.

3.3 Methods and functions

An abstract data type includes not only a collection of values or properties but also a set of operations (methods) on those values. Such operations are the procedures and functions that define the behavior of the abstract data type. Some of the operations associated with an ADT might be realized by means of data that is stored in the database, while other operations might be realized as executable code (functions). An implementation of an ADT is the stored data together with the data structures and code that implement the behavior of the ADT.

As seen in Section 3.1, "Abstract data types," methods may be encapsulated with the ADT definition. Specific methods for determining equality, and ordering when appropriate, are then usable in regular SQL comparison predicates. As we have seen, other methods can be defined as special operators on ADTs or as predicates that return truth values. A single value returned from a function call can be used any place in the SQL language that a single value is allowed. A truth value returned from a function call can be used as one of the terms in a boolean predicate.

Functions may be defined completely in SQL, or only their interface definition may be specified in SQL with the content of the function written in some programming language (e.g., Ada, C, or eventually C++). An SQL function may be "defined" as an independent schema element, as part of an ADT definition, or as part of a module definition. An external function may be "declared" in the same places.

The syntax of an SQL function is:

```
[CONSTRUCTOR | ACTOR | DESTRUCTOR] FUNCTION <function name>
<parameter declaration list>
RETURNS <data type>
<SQL statement> ;
END FUNCTION
```

Only constructor or destructor functions may create or destroy new ADT instances; they have already been discussed above. An actor function is any other function that reads or updates components of an ADT instance or accesses any other parameter declared in the <parameter declaration list>. A parameter in the parameter list consists of a parameter name and an SQL data type. The RETURNS clause specifies the SQL data type of the result returned. Since all data types in an SQL function are SQL types, it is not necessary to worry about "indicator" parameters to convey Null values.

The <SQL statement> may be any SQL statement, including compound statements and control statements. Of particular importance here are the following:

- A NEW statement that allows creation of a new OBJECT ADT instance; it is only allowed in a CONSTRUCTOR function.

- A DESTROY statement that destroys the existence of an OBJECT ADT instance; it is only allowed in a DESTRUCTOR function.

- An ASSIGNMENT statement that allows the result of an SQL value expression to be assigned to a free standing local variable, a column, or an attribute of an ADT.

- A CALL statement that allows invocation of an SQL procedure.

- A RETURN statement that allows the result of an SQL value expression to be returned as the RETURNS value of the SQL function.

Other SQL control statements allowed in an SQL function are discussed in Section 3.6, "Control structures."

The syntax of an external function declaration is:

```
DECLARE EXTERNAL <external function name>
        <formal parameter list>
        RETURNS <result data type>
                [ CAST AS <cast data type> ]
        LANGUAGE <language name>
```

The <formal parameter list> is a list of SQL data types. If a data type in the parameter list is supported in the programming language identified by the LANGUAGE clause, then the corresponding programming language routine has two parameters for that data type; the second parameter is the "indicator" parameter to convey Null values. If a data type in the parameter list is an ADT not supported in the programming language identified by the LANGUAGE clause, then the corresponding programming language routine has two parameters for each base type in the ADT definition, recursively. Again, the second parameter in each case is an "indicator" parameter. The actual mapping from the <formal parameter list> in the external function declaration to the parameter list of the programming language routine can become quite complex, but is completely specified in the SQL3 draft.

The CAST AS clause is a convenience to allow "encapsulated" casting from a programming language data type to an SQL data type. For example, an SQL DATETIME data type that appears in the formal parameter list is automatically cast to its character string literal representation before it is passed to a programming language routine, e.g., FORTRAN. The CAST AS clause could also automatically cause the character string RESULT to be re-cast into an SQL DATETIME value. Since every SQL data type has a defined CAST operation to and from character string representations, it is possible to pass any SQL data type to any programming language that supports character strings.

3.4 Subtypes and inheritance

Inheritance is an abstraction mechanism that adds to the power of data abstraction by allowing classes of objects to be related hierarchically. Inheritance allows classes to share definitions with other classes, thereby supporting newer, more specialized, data definitions without losing the existing properties and operations of the superclass.

Through inheritance, new types can be built over older, less specialized types rather than having to rewrite properties from scratch. Inheritance makes it possible to build a hierarchy of related ADTs; i.e., a "type hierarchy," that share the same interface, and possibly the same representation and implementations. As we move up in the inheritance hierarchy, types become more generalized; as we move down types become more specialized. These generalization/specialization capabilities allow more accurate and succinct modeling of applications.

The SQL implementation of a type hierarchy (see [4]) requires that an instance of a subtype is also an instance of all of its supertypes. Every instance is associated with a "most specific type" that corresponds to the lowest subtype assigned to the instance. At any given time, an instance must have exactly one most specific type. Note that the most specific type of an instance need not correspond to a leaf type in the type hierarchy. For example, a type hierarchy might consist of a maximal supertype PERSON that has STUDENT and EMPLOYEE as two subtypes, and STUDENT may have two subtypes GRAD and UNDERGRAD. An instance in this hierarchy might be created with a most specific type of STUDENT, e.g., a special, non-degree student, even though STUDENT is not a leaf in the type hierarchy.

As we saw above, every column definition in an ADT has an encapsulation level specified as either PUBLIC, PRIVATE, or PROTECTED. Public and protected components are visible to the definitions of all subtypes of that ADT, but private components are not.

A subtype can define constructor, actor, and destructor operations just like any other ADT. All operations of the supertype are invocable from the subtype, so there is a high potential for name conflicts when the subtype defines more specialized operations. Name resolution rules, described in the following section, ameliorate this problem.

In the current specification, if a supertype is a VALUE ADT, then all of its subtypes must be VALUE ADTs, and conversely. Similarly, if a supertype is an OBJECT ADT, then all of its subtypes must be OBJECT ADTs, and conversely. There is no inherent requirement for these restrictions and they could be relaxed if the need arises.

Since an instance must have at most one "most specific type" associated with it, a given instance cannot have two sibling types simultaneously as its most specific type. For example, in the above student type hierarchy, an instance of PERSON may not be an instance of both STUDENT and EMPLOYEE simultaneously. However, real world examples require that we have some method for modeling a person as both STUDENT and EMPLOYEE, for this can certainly be the case. To handle these situations, SQL provides "multiple inheritance;" i.e., a subtype can have more than one direct supertype. With multiple inheritance, we can define a new type STUDENT-EMP which is a subtype of both STUDENT and EMPLOYEE. A person who is both a student and an employee can be modeled as an instance of the STUDENT-EMP type. In this way an instance will satisfy the requirement to always have a "most specific type."

Multiple inheritance could lead to ambiguous inheritance of components from its supertypes, so SQL provides some disambiguity rules as follows:

- If a representation component in more than one supertype is inherited from a common supertype higher in the hierarchy, then only the "first occurrence" of the component is inherited. This puts some order dependency into subtype definition and complicates the later deletion of components from supertypes.

- If representation components with the same name in each of the supertypes are not inherited from a common supertype higher in the hierarchy, then the type definition is invalid unless the type definer renames the inherited components to remove the name clash.

These rules, and other related issues, are subject to improvement and evolution as the SQL ADT facility stabilizes over the next two or three years.

3.5 Polymorphic functions

Polymorphism is the ability to invoke an operation on any of several different objects and have that object determine what to do at run-time. A polymorphic function is one that can be applied in the same way to a variety of data objects. Support for polymorphism involves technical decisions concerning early or late binding among objects and the procedures that invoke their methods. To help address some of these technical decisions, a number of techniques have evolved (see [5]), such as:

Overloading	The ability to assign the same name to more than one function or procedure - - name resolution is then determined by a set of rules, thereby allowing a

	processor to distinguish among different functions of the same name by examining the "type" of the input data.
Coercion	The ability to omit semantically needed type coversions -- we have already seen that SQL uses this technique in some of its parameter passing to external functions.
Inclusion	The ability to manipulate objects of a subtype "as if" they were objects of a supertype -- possibly with a function of the same name that calls different routines.
Generalizing	The ability to specify that a parameter should "take on" the type of some supertype during processing of a specific function call.

The resolution rules to support polymorphism are derived from one basic concept; i.e., that for any particular function invocation, there must exist a single "best match" from the candidate functions that are "in scope." When a function call is executed, the unique "most specific type" of the various input parameters is used to help define the "best match" according to the following rules, many of which are derived from those used by C++:

- Begin with the set of all functions that are "in scope" for a particular function call; i.e., those that are defined or declared with the calling function name in the statement, procedure, module, or schema associated with the function call.

- For each argument, determine the set of functions that is a "best match" for that argument, then take the intersection of these sets. Unless this intersection has exactly one member, the call is illegal. That is, the function selected must be a "strictly better match" for at least one argument than every other possible function, but not necessarily the same argument for each function.

- To decide which functions are the best match for each argument, agree: an "exact match" is better than one based on type coercion (i.e., CASTing), an implicit conversion to the "closest" supertype is better than SQL or user-defined type coercion, and an implicit SQL-defined CAST is better than an implicit user-defined CAST.

Consider an example (from [5]) where the following three functions are defined:

```
FUNCTION F(:p1 X, :p2 INTEGER)
FUNCTION F(:p1 X, :p2 REAL)
FUNCTION F(:p1 Y, :p2 REAL)
```

Suppose that X and Y are abstract types defined in the same schema and that X has a user-defined CAST clause that defines a conversion from INTEGER to X. A function call F(1,1), where both 1's are integer literals would result in the following analysis:

- For the first argument, there is no exact match, and no implicit SQL conversion, so the user-defined conversion from INTEGER to X is used. The first two function definitions are in the set of "best matches."

- For the second argument, the INTEGER type of the literal is an exact match to the INTEGER parameter. Only the first function is in the set of "best matches."

Based on this analysis, the first function is the "best match" so it is the one invoked.

As a second example (also from [5]), suppose you have a type hierarchy in which A is a supertype, B and C are subtypes of A, and D is a subtype of both B and C.

```
  A
 / \
B   C
 \ /
  D
```

In this case, suppose that three functions F are defined as:

```
F(:p A) RETURNS A
F(:p B) RETURNS B
F(:p C) RETURNS C
```

Based only on the compile time rules described so far, a function call F(d), where the value d is of type D, would be ambiguous because, with an implicit conversion to the supertypes B and C of D, both the second and third functions would be in the set of "best matches." A function call F(x), where the value x is of type A, would also be ambiguous at compile-time because the x might really be of type D at run-time.

As currently specified, the SQL Syntax Rules require that an implementation consider all possible cases that might occur at run-time. For each case; i.e., for each of the four possible "most specific types" that x might have at run-time in the above example, the above rules must hold and identify for each case, a unique function. In addition, the set of identified functions must specify a RETURNS data type such that they all share a "common" supertype. This common supertype is the one returned in all cases.

There are a number of issues associated with polymorphism. In some cases the rules for resolving function calls are arbitrary and not always the best choice for every application scenario. Other issues concern the run-time overhead associated with the "late" binding required to support inheritance of properties to all subtypes of a given type. All of these issues will be addressed during the next two years as the SQL3 specification finds its way through the standardization process.

3.6 Control structures

Support for SQL functions is discussed in Section 3.3 above. We saw there that it was necessary to introduce several "control" statements into the SQL language, e.g., ASSIGNMENT, CALL, and RETURN. The obvious next step is to consider if more control statements and other "programming language" facilities should be added to SQL. In particular, we need to consider the appropriateness of the following additional facilities (see [7]):

- sequences of SQL statements in a procedure instead of the single SQL statement allowed in the current SQL standard,
- flow of control statements, such as looping, branching, etc.,
- exception handling, so that when an exception is raised, the SQL function or procedure can resolve the issue internally, or propagate the exception to the next outermost exception handler, rather than always returning control to the main calling routine.

These 3GL programming language facilities are valuable because they allow procedural encapsulation and they allow complete behavior to be specified within an ADT definition without the need to escape to a procedure written in some other language. Complex behavior can be made available to the host application program via a single call. This offers benefits in both cost and control. In SQL2, all procedures are single SQL statements so multiple calls must be made to address complex problems. All temporary state and flow of control belong to the host language application, thereby adding complexity to the application program that could be encapsulated in the SQL procedure. The following facilities have been adopted by ANSI and are under consideration within ISO (see [1]).

Compound statement

A compound statement is a statement that allows a collection of SQL statements to be grouped together into a "block." A compound statement may declare its own local variables and specify exception handling for an exception that occurs during execution of any statement in the group. Its syntax is as follows:

```
[ <beginning label>: ]
[ <variable declaration list> ]
BEGIN
        [ <SQL statement list> ]
        [ <exception handler> ]
END [ <ending label> ]

<exception handler> ::=
        EXCEPTION [ {WHEN <condition> THEN <SQL statement list>}... ]

<condition> ::= <exception name list> | OTHER
```

An <exception name> is unique within a <module> and may be declared with an <exception declaration>.

Exception handling

An exception declaration establishes a one-to-one correspondence between an SQLSTATE error condition and a user-defined exception name. It's syntax is as follows:

```
DECLARE <exception name> EXCEPTION FOR SQLSTATE <SQLSTATE literal>
```

The exception handling mechanism under consideration for SQL3 is based very strongly on the mechanism defined in Ada. Each compound statement is assumed to have an exception handler; if one is not explicitly defined, then a default handler is provided by the system. When the execution of a statement results in an active exception condition, then the containing exception handler is immediately given control. Ultimately, the exception handler terminates with one of the following behaviors:

- The compound statement terminates with the active exception condition still active, or
- The compound statement terminates with a new active exception condition, or
- The compound statement terminates successfully, as though no exception occurred, and there is no outstanding active exception condition.

The exception handler may execute SIGNAL or RESIGNAL statements to identify a new exception name or to pass on the existing exception name. If an exception condition occurs in the exception handler itself, then the compound statement is terminated and that exception condition becomes the "active" exception condition.

Flow of control statements

The following program flow of control statements are currently specified in the draft SQL3 document:

- A CASE statement to allow selection of an execution path based on alternative choices. A <value expression> is executed and, depending on the result, control is transferred to the appropriate block of statements.

- An IF statement with THEN, ELSE, and ELSEIF alternatives to allow selection of an execution path based on the truth value of one or more conditions.

- A LOOP statement, with a WHILE clause, to allow repeated execution of a block of SQL statements based on the continued true result of the <search condition> in the WHILE clause. A LOOP statement is also allowed to have a statement label.

- A LEAVE statement to provide a graceful exit from a block or loop statement.

3.7 Stored procedures

In the existing SQL standard, and in its 1992 replacement, a module is a persistent object created by the Module Language. It is a named package of procedures that can be called from an application program, where each procedure consists of exactly one SQL statement. However, there is no requirement that an implementation be able to execute Module Language (the alternative is Embedded SQL) and the resulting persistent module is not stored as part of the SQL schema, is not reflected in the information schema tables, and cannot be passed across an RDA connection to a remote site.

In the emerging SQL3 specification, standardization committees have recognized the requirement for some "standard" capability to define persistent modules that "live" in the SQL schema and whose procedures may be called from any SQL statement in the same processing environment. In SQL3 the CREATE MODULE statement has the same status as any other schema definition statement. The result of execution is a module that is managed by SQL rather than by the proprietary facilities of the host operating environment. Module definitions are reflected in the Information Schema just like any other schema object and they are subject to ownership and access control declarations.

The primary benefit of supporting stored procedures is that implementations are able to optimize groups of statements rather than just individual statements. Entire packages of SQL procedures can be sent across a wire to a remote SQL conforming site, be optimized at that site, and then be executed with a single call when needed. The ISO RDA standardization committee is now considering RDA enhancements that would support database stored procedures that persist beyond the end of an RDA dialogue.

3.8 Parameterized types

The ability to define abstract data types does not by itself provide the capability to define "parameterized" types. A parameterized type is really a "type family" with a new data type for each value of an input parameter. For example, an ADT definition for VECTOR(N) can be thought of as a family of data types, one for each positive integer value of N. This idea is not new as we already have parameterized, predefined types in the existing SQL standard, e.g., CHARACTER(N) and DECIMAL(P,S), and it is a common feature in programming languages that support user-defined types. Reference [25] adds the ability to specify parameterized ADT definitions in SQL; it has been adopted by ANSI and is currently under consideration by ISO.

We may think of a "parameter" as any value of a data type known to the SQL environment, e.g., an integer value in the examples above. We may also think of a "parameter" as a reference to an exisitng data type, rather than a value of that type. For example, we may wish to specify that VECTOR(N) is really a vector of integers, or reals, or decimals with fixed precision and scale. This can be achieved by passing a data type name to the ADT definition instead of just a data type value.

The syntax for specifying a parameterized type in SQL3 is very similar to that for specifying a regular ADT, namely:

```
CREATE [ VALUE | OBJECT ] TYPE TEMPLATE <template name>
        ( { <template parameter declaration> }... )
        <abstract data type body>

<template parameter declaration> ::=
        <template parameter name> { <data type> | TYPE }
```

The keyword TEMPLATE indicates that the specification is for a paramaterized ADT rather than a regular ADT. The keyword TYPE indicates that a parameter is a data type name rather than a data type value. The <abstract data type body> is analogous to the body of a regular ADT definition.

A parameterized type is referenced by specifying the type template name and an actual parameter list. Each actual parameter must be a value, or a data type, that can be determined at syntax evaluation time; i.e., usually a literal or a data type name. If the actual parameter is a data type name, then the formal template parameter must specify TYPE.

Note that nesting of data types is allowed in the definition of a parameterized type. For example, one might specify two type templates as

```
POINT(::coord_type)
SEQUENCE(::object_type)
```

and generate a new type as SEQUENCE(POINT(FLOAT)).

You are allowed to define more than one type template with the same name, just as you may define more than one <SQL function> with the same name. For example, it is legal to define two POINT data types, one for 2-dimensional points and one for 3-dimensional points. The rules for matching a parameterized type reference to a parameterized type definition are the same as the rules for matching overloaded functions, see Section 3.5, "Polymorphic functions."

Distinct types

Sometimes it is desirable to be able to distinguish between table or ADT attributes that have the same underlying ADT definition. For example, table T1 might have a column named Cartesian_Coordinate that is defined to have the data type POINT and table T2 might have a column named Polar_Coordinate that is also defined to have the data type POINT. The POINT data type may have a DISTANCE function defined to calculate the distance between any two points, but clearly the calculation

```
DISTANCE(T1.Cartesian_Coordinate,T2.Polar_Coordinate)
```

may not be a meaningful calculation.

The SQL3 draft provides a facility (from [26]) for the user to declare that two otherwise equivalent ADT declarations are to be treated as "distinct" data types. The keyword DISTINCT used in an ADT declaration indicates that the resulting type is to be treated as "distinct" from any other declaration of the same ADT. In the above example, if two new types are declared

```
CREATE DISTINCT TYPE CARTESIAN_POINT AS POINT

CREATE DISTINCT TYPE POLAR_POINT AS POINT
```

and if Cartesian_Coordinate and Polar_Coordinate are declared to have the data types CARTESIAN_POINT and POLAR_POINT respectively, then both kinds of coordiante points would have all the methods and operations for POINT, but any attempt to apply the DISTANCE function between them would result in an error.

The DISTINCT facility in SQL3 is currently only applicable to abstract data types, not to pre-defined data types. For example, it is not legal to declare the following:

```
CREATE DISTINCT TYPE PART_NBR AS INTEGER

CREATE DISTINCT TYPE EMP_ID AS INTEGER
```

Certainly, it is possible to extend the definition for "distinct" types from abstract types to pre-defined types. This is an issue that will be addressed in the near term.

CLID generator types

The emerging Common Language Independent Datatype (CLID) specification [9], under development in ISO JTC1/SC22/WG11, also specifies facilities for parameterized and distinct data types. However the syntax is slightly different. CLID uses the keyword GENERATOR instead of TEMPLATE and NEW instead of DISTINCT, but the effect is essentially identical.

Some of the following sections in this paper were written before parameterized types were added to the SQL3 draft, so they are written using the CLID syntax. Thus in the following sections

```
CREATE GENERATOR TYPE  <==>  CREATE TYPE TEMPLATE
```

and

```
CREATE NewType = NEW OldType  <==>  CREATE DISTINCT TYPE NewType AS OldType
```

3.9 Complex types

At the present time SQL3 only defines a limited number of data types, including: fixed and variable length character strings, fixed and variable length bit strings, fixed and floating point numerics, dates, times, timestamps, intervals, Boolean, and enumerations. The components of an ADT must therefore be defined as one of these base data types or as a previously defined ADT.

There is a need to extend the pre-defined, base data types to include "generator" data types such as those specified in the emerging ISO common language-independent datatype (CLID) specification [9]. The CLID specification includes, among others, the following generator types:

```
ARRAY {[<lower>..<upper>]}... OF <base type>

LIST OF <base type>

SET OF <base type>

CHOICE ({<identifier>:<base type>}...)

RECORD ({<identifier>:<base type>}...)

RANGE Subtype Generator

SIZE Subtype Generator

EXTEND Supertype Generator
```

Declared Generator

ARRAY creates a new data type whose values are fixed-length sequences of values from the <base type>. Values in the sequence are in a one-to-one correspondence with a value in the product space of the <lower> to <upper> limits for each index component. Operations defined for an array are:

Equal	Equal is a Boolean predicate on two arrays that returns true if their corresponding components are pairwise equal, and returns false otherwise.
Select	Select operates on an array and on an element of the index product space to return the appropriate value from the <base type>.
Replace	Replace operates on an array, an element of the index product space, and a value from the <base type> to produce a new array with the given value substituted into the appropriate position.

LIST creates a new data type whose values are ordered sequences of values from the <base type>, including the empty sequence. The following operations are defined for lists:

Equal	Equal is a Boolean predicate on two lists that returns true iff the two lists have the same length and all components are pairwise equal.
IsEmpty	IsEmpty is a Boolean predicate on a single list that returns true iff the sequence is empty.
Head	Head operates on a list to return the first element from the sequence.
Tail	Tail operates on a list to return a new list consisting of all elements except the first.
Append	Append operates on a list and a single value from the <base type> to produce a new list with the value as the last element of the sequence.
Empty	Empty is a niladic operation yielding the empty sequence.

SET creates a new data type whose values are taken from the power set (i.e., the set of all subsets) of the <base type>, with operations appropriate to the mathematical set algebra. In order to ensure uniqueness of representation, the <base type> is required to be discrete, meaning it cannot have a distance function defined that yields any limit points that are elements of the <base type>. Operations on sets consist of the following:

Equal	Equal is a Boolean predicate that returns true iff two sets are equal.
IsIn	IsIn is a Boolean predicate that operates on an element of the <base type> and a set to return true iff the element is a member of the set.
Subset	Subset is a Boolean predicate that operates on two sets and returns true iff the first is a subset of the second.
Union	Union operates on two sets to return their set union.
Intersection	Intersection operates on two sets to return their set intersection.
Complement	Complement operates on a set to return its set complement.

SetOf	SetOf operates on a single value of the <base type> to return the singleton set consisting of just that element.
Empty	Empty is a niladic operation that returns the empty set.
Universe	Universe is a niladic operation that returns the set of all values from the <base type>.
Select	Select operates on a set to return an arbitrary single value from that set.

CHOICE creates a new data type each of whose values is a single value from one of a set of alternative data types, logically labelled by the name of the alternative selected. Each alternative data type is labelled by an identifier that is unique within a given Choice-type, but there is no requirement that the <base type>s themselves be unique within a Choice-type definition. Operations on instances of a Choice-type consist of the following:

Equal	Equal is a Boolean Predicate on two Choice-type instances that returns true iff the two instances have the same alternative <identifier> and they are equal under the equal operation for the specified <base type>.
IsField	IsField is a Boolean predicate that operates on a given <identifier> and a given Choice-type instance to return true iff the identifier of the instance is identical to the given <identifier>.
Tag	Tag operates on an <identifier> and a single value of the <base type> associated with that identifier to produce a single Choice-type instance with that value and that identifier.
Cast	Cast operates on an <identifier> and a single instance of a Choice-type to return, if possible, an instance of the <base type> associated with that identifier, otherwise an exception.

RECORD creates a new data type whose values are aggregations of named values from a collection of named components. Each component consists of an <identifier> and an associated <base type>. Each aggregation has a named value for each component. Operations on Record-type are as follows:

Equal	Equal is a Boolean predicate that operates on two instances of the same Record-type to return true iff component values are pairwise equal.
Aggregate	Aggregate operates on a sequence of values of the appropriate data types and returns a single aggregate value of the specified Record-type.
FieldSelect	FieldSelect operates on an instance of Record-type and on an <identifier> of one of the components of that Record-type to return the appropriate value from the associated <base type>.

RANGE is a subtype generator that acts on any ordered data type to produce a subtype consisting of those values that satisfy the range constraints. The syntax is as follows:

<base type> : RANGE (<lowerbound>..<upperbound>)

All values from the <base type> that are greater than or equal to <lowerbound> and less than or equal to <upperbound> are included in the space of values for the subtype.

SIZE is a subtype generator that acts on a LIST or SET generator to produce a subtype by specifying bounds on the number of elements contained in a single instance of that data type. The syntax is as follows:

```
SET OF <base type> : SIZE (<min>, <max>)

LIST OF <base type> : SIZE (<min>, <max>)
```

For sets, the size declaration specifies constraints on the minimum and maximum cardinality of sets that are allowed as part of the subtype. For lists, the size declaration specifies constraints on the minimum and maximum length of the sequence of elements that determines the list.

EXTEND is a supertype generator that acts on a <base type> to create a new data type that has the <base type> as a subtype. The syntax is as follows:

```
<base type> : EXTEND (<value list>)
```

The value space of the extended data type consists of all values in the <base type> plus those additional values specified in the <value list>. The extend generator can be used with ENUMERATION or other data types to extend the value space to include the values in the value list. The operations defined for the <base type> are not automatically extended.

Working papers are under discussion for including "generator" types in the SQL specification, but as of this time there is no complete proposal.

4. Generic ADT packages

The SQL3 specification provides facilities for defining abstract data types, but it does not at present specify standard packages of specific abstract data types for various application areas. We believe that it makes sense to standardize packages that have general scientific or engineering applicability, document management applicability, or tools for the management of multimedia objects such as image, voice, and video. We are pursuing with ANSI and ISO SQL standardization committees the proper mechanism to achieve standardization of popular ADT packages. In addition, or if these efforts are not successful, different CALS groups might specify the functionality of CALS specific ADT packages in the same manner that they now specify document type definitions (DTDs) for use with the ISO Standard Generalized Markup Language (SGML).

Some packages, e.g., geographic information structures, have broad appeal across different application areas and could benefit from "generic" standardization. Other packages, e.g., military logistics objects, have limited appeal within a single CALS application area and need not be standardized internationally. The difficult part, and the most important, is for user groups to agree on the desired types and methods that are most useful in a specific application area. This is a task that CALS working groups should be very good at. Once an application package is well-defined and accepted by a significant user population, implementations will follow rapidly in response to procurements.

In the cases that qualify for standardization as "SQL generic ADT packages," the packages would be optional specifications that need not be supported in order to claim conformance to SQL. To emphasize this latter point, the packages could be processed in a separate specification under a separate project identifier. The main intent of standardization is to allow applications to use the same ADTs across different application areas, thereby promoting interoperability and the sharing of data, and encouraging performance optimization over a manageable collection of types.

The purpose of this Section is to give examples of a few generic ADT packages that have value in various application areas. If others agree that it makes sense to define named ADTs and syntax for accessing them in a standard specification, then we can pursue the proper mechanism (e.g., a new project proposal) to make that standardization happen.

The capabilities discussed in Section 3 above need to be available before one can define robust "generic" application packages with "well-chosen" definitions. Since Abstract Data Type facilities are already in the emerging SQL3 specification, we assume that the needed additional data types and data type generators (see Section 3.9, "Complex types") will either also be included as part of SQL3 or as the "base requirement" of a generic ADT packages standard.

In the subsections that follow, we discuss potential packages of ADT definitions that have general applicability to a wide range of application areas. It makes sense to consider standardization of packages like these as optional components of a multi-part "companion standard" to the SQL3 specification.

It is best if all of these packages are developed under some sort of coordinated effort, in order to avoid incompatible duplication of the data types that get used in a number of different application areas. It is too early to begin a detailed SQL-specification of the desired data types, because the SQL3 specification may evolve substantially in the coming two years. However, it is not too early to reach agreement on the desired semantics for each ADT package, including a list of the appropriate structures and operations. It is not too early to approve a new project and get people thinking seriously about how to address the problem. A "standing" working paper, much like the contents of this Section, that gets periodically updated as the result of group decisions, may be the most appropriate mechanism for further discussion and development.

In the following subsections we assume that all SQL-defined types are able to accommodate null values. All operations must be able to handle input and output parameters that may have null values. In some cases we will want the aggregate type (e.g., an array) to handle null types in the components and sometimes we won't. This will be a decision that needs to be made for each newly defined type. To accommodate null values, we must sometimes replace the CLID BOOLEAN data type by an SQL data type that recognizes true, false, and unknown. We achieve this by using the CLID data type STATE(true, false, unknown) wherever a truth value is expected in SQL.

4.1 Vector spaces

We know from linear algebra [12, 13] that any finite-dimensional vector space defined over the field of real or complex numbers is isomorphic to a real or complex product space. In addition, if a finite-dimensional vector space has an inner product defined, then that inner product space is isomorphic to the real or complex product space with the well-known scalar product for vectors of real or complex numbers. Many geometric properties of any inner product space are closely related to geometric properties on real or complex product spaces. For these reasons, the finite-dimensionsal real and complex product spaces play an important practical role in many engineering and physical science applications. It would be an important contribution to these applications if the base operations were standardized and incorporated into all "scientific" database management systems.

We assume the existence of a COMPLEX data type

```
TYPE COMPLEX = RECORD (real:REAL, imag:REAL)
```

with operations, using the notation of [9], defined as follows:

```
Zero():COMPLEX
One():COMPLEX
Add(u:COMPLEX,v:COMPLEX):COMPLEX
AddInv(u:COMPLEX):COMPLEX
```

Mult(u:COMPLEX,v:COMPLEX):COMPLEX
MultInv(u:COMPLEX):COMPLEX where u not equal to ZERO()
Conjugate(u:COMPLEX):COMPLEX

where Zero and One are the additive and multiplicative identities, Add and Mult are the addition and multiplication operations for complex numbers, AddInv is the additive inverse, and MultInv is the multiplicative inverse. Under these operations the non-null values of COMPLEX satisfy the axioms of a mathematical field.

The following additional COMPLEX operations are defined in terms of the base operations for Record-type:

Equal(u:COMPLEX,v:COMPLEX):STATE(true, false, unknown)
Aggregate(x:REAL,y:REAL):COMPLEX Note: x or y null implies result is null
RealPart(u:COMPLEX):REAL
ImagPart(u:COMPLEX):REAL

Product Spaces

Let FIELD be any numeric data type that has the operations of a mathematical field; i.e., addition, subtraction, multiplication, and division. We define the following general product space to represent vectors defined over FIELD:

CREATE GENERATOR TYPE VECTOR(N) OF FIELD = NEW ARRAY [1..N] OF FIELD

We are particularly interested in the cases where FIELD is either REAL or COMPLEX, so define the following:

CREATE GENERATOR TYPE REALVECTOR(N) = VECTOR(N) OF REAL

CREATE GENERATOR TYPE COMPLEXVECTOR(N) = VECTOR(N) OF COMPLEX

Let VECTOR(N) be a shorthand notation for VECTOR(N) OF FIELD and define the following operations for VECTOR(N):

Zero():VECTOR(N)
Add(x:VECTOR(N),y:VECTOR(N)):VECTOR(N)
AddInv(x:VECTOR(N)):VECTOR(N)
ScalarMult(a:FIELD,x:VECTOR(N)):VECTOR(N)
ScalarProd(x:VECTOR(N),y:VECTOR(N)):FIELD
Norm(x:VECTOR(N)):REAL
Dist(x:VECTOR(N),y:VECTOR(N)):REAL

With Zero as the zero vector in N-space, Add as vector addition, AddInv as the additive inverse for vectors, ScalarMult as multiplication of a vector by a scalar, and ScalarProd as the scalar product of two vectors, VECTOR(N) OF FIELD will satisfy the axioms of an inner product space over the base field [12, 13]. In addition, if Norm(x) is defined to be the positive square root of ScalarProd(x,x), and if Dist(x,y) is defined as Norm(Add(x,AddInv(y))), then VECTOR(N) OF FIELD becomes a complete metric space in the norm topology.

The following operations, analogous to Record-type operations, allow equality comparison of vectors, construction of vectors from base components, and reference to individual vector components:

Equal(x:VECTOR(N),y:VECTOR(N)):STATE(true, false, unknown)
Aggregate(a1:FIELD,...,aN:FIELD):VECTOR(N)

Project(x:VECTOR(N),i:(INTEGER:RANGE(1..N))):FIELD

There is no LESS THAN operation defined for vectors, so it is not possible to test for order relationships in comparison predicates.

Cross Products in Real 2-Space and 3-Space

We note that REALVECTOR(2) and REALVECTOR(3) have important vector applications in many engineering and physics problems. In each of these data types, it makes sense to define an additional operation, the vector cross product, as follows:

CrossProd(x:REALVECTOR(2),y:REALVECTOR(2)):REALVECTOR(3)
CrossProd(x:REALVECTOR(3),y:REALVECTOR(3)):REALVECTOR(3)

We could consider generalizing the CrossProd to be a vector product on VECTOR(N), but the definition becomes very complex. It might be better to wait until after matrix algebra operations have been defined to represent linear transformations of vectors.

Also consider adding other vector operations, e.g., Gradients and Curl, etc., that are important in physical science applications.

4.2 Matrix algebra

Many practical applications can be modeled in terms of linear transformations on finite-dimensional real or complex vector spaces. Such linear transformations can be represented as two-dimensional matrices over real or complex fields. Addition and scalar multiplication of matrices is defined to coincide with addition and scalar multiplication of linear transformations, and multiplication of matrices is defined to coincide with composition of linear transformations. Standardization of matrices and matrix operations as SQL data types would be an important contribution to many areas of applied mathematics.

Let RING be any numeric data type that has the operations of a mathematical ring; i.e., addition, subtraction, and multiplication, but not necessarily division. We define the following general product space to represent matrices defined over RING:

CREATE GENERATOR TYPE MATRIX(M,N) OF RING = NEW ARRAY [1..M][1..N] OF RING

We are particularly interested in the cases where RING is either REAL or COMPLEX, so define the following:

CREATE GENERATOR TYPE REALMATRIX(M,N) = MATRIX(M,N) OF REAL

CREATE GENERATOR TYPE COMPLEXMATRIX(M,N) = MATRIX(M,N) OF COMPLEX

Let VECTOR(N) be a shorthand notation for VECTOR(N) OF FIELD and let MATRIX(M,N) be a shorthand notation for MATRIX(M,N) of RING. Then define the following operations for MATRIX(M,N):

Zero():MATRIX(M,N)
Identity():MATRIX(M,N)
Add(x:MATRIX(M,N),y:MATRIX(M,N)):MATRIX(M,N)
AddInv(x:MATRIX(M,N)):MATRIX(M,N)
ScalarMult(a:RING,x:MATRIX(M,N)):MATRIX(M,N)
MatrixProd(x:MATRIX(M,S),y:MATRIX(S,N)):MATRIX(M,N)
Transpose(x:MATRIX(M,N)):MATRIX(N,M)

```
ConjugateTranspose(x:MATRIX(M,N)):MATRIX(N,M)
Trace(x:MATRIX(N,N)):RING
Determinant(x:MATRIX(N,N)):FIELD
Adjoint(x:MATRIX(N,N)):MATRIX(N,N)
MatrixInv(x:MATRIX(N,N)):MATRIX(N,N)
Rank(x:MATRIX(M,N)):(INTEGER:RANGE(1..Min(M,N)))
Reduce(x:MATRIX(M,N)):MATRIX(M,N)
```

With Zero as the all-zero matrix, Add as matrix addition, AddInv as the additive inverse for matrices, and ScalarMult as multiplication of a matrix by a scalar, MATRIX(M,N) satisfies the axioms of a vector space over the base field. In addition, with Identity as the diagonal identity matrix, MatrixProd as the product of two product-compatible matrices, and MatrixInv as the inverse of square, non-singular matrices, REALMATRIX(N,N) and COMPLEXMATRIX(N,N) are linear algebras; i.e., they satisfy the properties of a non-commutative mathematical ring with identity.

The operations Transpose, Conjugate, Trace, Determinant, and Adjoint (see [12]) all provide helpful tools when matrices are used to analyze linear transformations. The ConjugateTranspose is intended only for COMPLEXMATRIX(M,N), but it is well-defined for all MATRIX(M,N). Rank and Reduce are important operations whenever an MxN-Matrix is used to represent a system of M linear equations in N variables.

The following operations, analogous to Record-type operations, allow equality comparison of matrices, construction of matrices from base components, and reference to individual matrix components:

```
Equal(x:MATRIX(M,N),y:MATRIX(M,N)):STATE(true, false, unknown)
RowAggregate(x1:VECTOR(N),...,xM:VECTOR(N)):MATRIX(M,N)
ColumnAggregate(x1:VECTOR(M),...,xN:VECTOR(M)):MATRIX(M,N)
Project(x:MATRIX(M,N),(i,j):(RECORD
        (rowid:(INTEGER:RANGE(1..M)),colid:(INTEGER:RANGE(1..N)))):RING
ProjectRow(x:MATRIX(M,N),i:(INTEGER:RANGE(1..M)):VECTOR(N)
ProjectColumn(x:MATRIX(M,N),j:(INTEGER:RANGE(1..N)):VECTOR(M)
```

4.3 Euclidean geometry

Any instantiation of the values of REALVECTOR(N) is said to be a Euclidean space. Every Euclidean space can support the structures and operations of Euclidean geometry. Such structures include, but are not limited to, point, line, segment, polysegment, polygon, convex polygon, angle, angular measure, distance from point to line, distance from point to polygon, distance between two disjoint polygons, planes, hyperplanes, etc. The notion of convex hull and optimization of linear functions defined on convex sets is important in operations research and applied economics. It would be an important contribution to these fields if certain fundamental structures were standardized in all "applied economics" database systems.

```
CREATE GENERATOR TYPE EUCLIDEAN(N) UNDER REALVECTOR(N)
```

All of the vector operations from REALVECTOR(N) would be inherited by EUCLIDEAN(N) and new structures and operations are defined as follows:

```
CREATE TYPE POINT UNDER EUCLIDEAN(N)

CREATE TYPE LINE = NEW SET OF POINT : SIZE(2,2)

CREATE TYPE HYPERPLANE = NEW ARRAY [1..N+1] OF REAL
```

CREATE TYPE HALFSPACE = NEW ARRAY [1..N+1] OF REAL

CREATE TYPE SEGMENT = NEW RECORD(start:POINT,stop:POINT)

CREATE TYPE POLYSEGMENT = NEW LIST OF POINT

CREATE TYPE CONVEX_POLYHEDRON = NEW SET OF HALFSPACE : SIZE(1,M)

Note: M represents the number of "faces" of the convex polyhedron.

Define CAST operations so that a convex polyhedron can be represented as a set of M linear inequalities in N variables; i.e., $AX <= B$ where A is an MxN matrix and X and B are elements of VECTOR(M). Also be able to CAST a conve polyhedron as an Mx(N+1) matrix so that reduction algorithms can be applied to "solve" the inequalities.

CREATE TYPE LINEAR_FUNCTION = NEW VECTOR(N) OF REAL

Note: A linear function is of the form $f(X) = AX$ where A is a 1xN vector and X is an Nx1 vector variable.

The following operations might be the start of a long list of operations of interest to operations research professionsals:

```
Distance(p:POINT,l:LINE):REAL
Distance(p:POINT,h:HYPERPLANE):REAL
Maximum(f:LINEAR_FUNCTION,cp:CONVEX_POLYHEDRON):REAL
Minimum(f:LINEAR_FUNCTION,cp:CONVEX_POLYHEDRON):REAL
etc.
```

Euclidean Plane

Of particular importance in many engineering and cartographic applications is the two-dimensional Euclidean plane and the three_dimensional Euclidean space. These instantiations of EUCLIDEAN(2) and EUCLIDEAN(3) deserve special notation for they are likely candidates for optimization in database systems catering to these application areas.

CREATE TYPE EUCLIDEAN_PLANE = EUCLIDEAN(2)

CREATE TYPE EUCLIDEAN_SPACE = EUCLIDEAN(3)

In the EUCLIDEAN_PLANE, it also makes sense to define the following new structures and operations:

```
CREATE DOMAIN POLYGON AS POLYSEGMENT
        CHECK(Equal(Head(VALUE),Last(VALUE))

CREATE DOMAIN CONVEX_POLYGON AS POLYGON
        CHECK( ScalarProd of all successive points is ...)
```

[Continue with needed functions -- especially those dealing with distances, conic sections, surveying, navigating, map making, etc.]

4.4 Geographic regions on Earth surface

Regions on the surface of the Earth could be defined as part of a "geographic" package as follows:

```
CREATE TYPE DEGREE = NEW INTEGER:RANGE(-360..360)

CREATE TYPE LONG_DEGREE = DEGREE:RANGE(-179..180)

CREATE TYPE LAT_DEGREE = DEGREE:RANGE(-90..90)

CREATE TYPE MINUTE = NEW INTEGER:RANGE(0..59)

CREATE TYPE SECOND = NEW INTEGER:RANGE(0..59)

CREATE TYPE PRECISION = NEW INTEGER:RANGE(0..9999)
```

Note: This definition of precision is biased toward a decimal representation. As an alternative, one might define precision as a positive integer, p, coded as a bit string of length n, to represent the fraction p/2**n.

```
CREATE TYPE LATITUDE = RECORD
        (deg:LAT_DEGREE,min:MINUTE,sec:SECOND,prec:PRECISION)

CREATE TYPE LONGITUDE = RECORD
        (deg:LONG_DEGREE,min:MINUTE,sec:SECOND,prec:PRECISION)

CREATE TYPE MEASURE = RECORD
        (deg:DEGREE,min:MINUTE,sec:SECOND,prec:PRECISION)
```

With these definitions the normal field extraction operations on RECORD can be used to extract "degree," "minute," "second," and "precision" from LATITUDE and LONGITUDE even if values of these types are stored in vendor specific formats. As currently defined, PRECISION is a partition of SECOND into 10,000 subunits. Reference [10] recommends this level of precision as adequate (i.e., within 1/8-th inch) for identifying any point on the surface of the Earth.

Other functions can be defined on LATITUDE and LONGITUDE to extract Universal Transverse Mercator System (UTM) units or other units for accepted ways to identify points on the surface of the Earth (see [10]).

Define Addition/Subtraction operations so that LATITUDE, LONGITUDE and MEASURE are additive groups.

```
CREATE TYPE LOCATION = NEW RECORD(lat:LATITUDE,long:LONGITUDE)
```

Determine if it makes sense to define arithmetic operations on LOCATION. Define distance between LOCATIONs via great arcs on the surface of the earth (or use other accepted GIS distance measurements).

```
CREATE TYPE REGION = NEW SET OF LOCATION

Define Earth_Surface = Universe()
```

Note that Earth_Surface has a "finite" domain (i.e., approximately 2**70 elements). All instances of REGION that represent contiguous geographic entities (e.g., countries, cities, bodies of water) can be represented efficiently and the set operations of Union, Intersection, Complement can be optimized using the methods of [11].

Define CAST operation from LOCATION to REGION.

Define QUADRANT as RECORD(base:LOCATION,lat:MEASURE,long:MEASURE)

Define CAST operation from QUADRANT to REGION.

A "quadrant" represents a "rectangular" region on the surface of the earth; i.e., all locations within the rectangle determined by the "base" and the "lat" and "long" measures.

Define CAST operations between QUADRANT and RECTANGLE in EUCLIDEAN space.

Import the Definitions of Euclidean Geometry for 2-space, especially segment, polysegment, polygon, and convex polygon.

Define CAST operation from POLYGON to REGION to be the set of all locations that: 1) fall on the boundary or 2) fall in the interior of the polygon. Note that the definition of interior may be tricky unless the polygon is a convex polygon.

Define other important transformations between Euclidean 2-space and the Earth_Surface.

With the above data structures to represent Earth_Surface geography, many simple queries can be answered quite easily. For example, the query "find the closest international airport to Chesapeake Bay" or the query "find the locations of all hotels within 3 miles of Interstate 95 between Washington and New York" can be answered by treating all such geographic objects as REGIONs. The answer to the first query is the distance between the set of all locations of international airports and the set of locations comprising the region Chesapeake Bay, and the answer to the second query is the intersection of the three regions: 1) set of all hotel locations, 2) 3-mile BUFFER around Interstate 95, and 3) Washington-NewYork-Corridor.

Another typical geographic query is to ask if Region A lies to the NorthWest of Point B. This is easily modeled by defining NorthWest of Point B as a Quadrant and then determining if Region A is contained in the Region determined that Quadrant.

4.5 Spatial data types

To be completed -- using Geographic data types and Euclidean data types as the base. Consider the proper integration with Euclidean 3-space so that topographic features, mineral deposits, clouds and weather, etc. can be related to locations on, above, and below the Earth's surface.

Add new abstract data types to model appropriate operations for spatial data management, with application to topography, satellite tracking, geodetics, etc.

4.6 Text and document data types

ISO/IEC JTC1/SC21 has already approved a new Question on suppport for Full-Text data manipulation [14]. This question has been assigned to WG3 and the first international meeting was held November 13-15, 1991, in Japan. This meeting is recommending to ISO how best to address the requirements for text manipulation in SQL. Canada has already submitted a new project proposal to create a new ISO project based on the SFQL approach [15]. A Japanese position is outlined in [16].

It is conceivable that the requirements of [15] and [16] can be satisfied by using the new ADT facilities in SQL3 to specify appropriate document and/or text handling facilities. If so, then new text ADT's could be specified as one part of a new multi-part "SQL generic ADT package" standard.

Also see the definitions of "chunk" expressions in HyperTalk; i.e., distances between words, sentences, paragraphs, etc.

4.7 Computer graphics

To be completed -- take full advantage of the GKS, PHIGS, and Computer Graphics Metafile (CGM) standards to define the appropriate structures to be stored in a database and the appropriate retrieval commands. Since CGM is already a list of graphics commands, it may be appropriate to define a single abstract type for CGM, with commands for casting it to and from character strings. Additional operations might be to treat it as a list with the ability to use Head, Tail, and Append operations to retrieve CGM commands from it or add CGM commands to it.

4.8 Pictures

Take full advantage of CCITT, ODA/ODIF, and Computer Graphics standards for representing images. Provide data types that allow vendor specific internal representations with Cast functions to cast occurrences to and from the standard formats. SQL data types need only capture the highest level abstractions of an image; more specialized image management systems will provide the sophisticated operations for color enhancement, shadowing, animation, etc. In the future, these more sophisticated tools should be able to use an SQL database to store images and image components in the same way that they now use file systems. The difference is that the database management system will provide standardized query capabilities, shared access, and transaction management lacking in standardized file systems.

Consider the following as a starting point:

```
CREATE GENERATOR TYPE PIXEL(P) = NEW ARRAY [0..P-1] OF BIT

CREATE GENERATOR TYPE PICTURE_ARRAY(M,N,P) = NEW
        ARRAY [0..M-1][0..N-1] OF PIXEL(P)
```

There are many different representations of colored PIXELs in commercial products. Some, like Macintosh [27], use Red-Green-Blue (RGB) as the basic color components, with each color represented by an 8, 16, or 32 bit positive integer; others use color models based on Cyan-Magenta-Yellow (CMY), or Hue-Saturation-Value (HSV), or Hue-Lightness-Saturation (HLS). I've seen references to YIQ and YUV representations, but without definitions. Each implementation will use a color model tailored for its hardware and software environment. It is important that SQL not attempt to specify the color model to be used; instead, SQL should provide basic ADT's like PIXEL and PICTURE above, so that applications can manipulate the components of a picture, or store and retrieve components from the database, even if the exact visualization of that picture is implementation specific.

This approach is consistent with how SQL handles character strings. The SQL specification provides concatenation, substring, index, and length operations for character strings even though the actual bit representation of a character is implementation-defined. Like with character strings, SQL could provide the language constructs necessary for labeling pixels as conforming to a particular color model. Similarly, SQL can provide trimming, cutting, and pasting operations for pictures without necessarily defining the semantics of a PIXEL representation.

Most hardware/video implementations provide transformations between and among the most popular color representation models. For example, Macintosh [27], provides mappings to and from the RGB, CMY, HSL, and HSV color representation of each PIXEL. These mappings can be incorporated as methods on implementation specific subtypes of the PIXEL and PICTURE data types.

Sets of Picture elements

Sometimes it is inconvenient to think of a picture as a full array of pixels. Instead, it may be more convenient to think of a picture as a set of picture elements, where a picture element consists of a pixel value and a location in the array. This is particularly true for "sparse" pictures consisting of just a few brush strokes on a canvass, but it may also apply to pictures that have large contiguous blocks of picture elements with the same pixel value. For example, a flock of birds, the outline of a figure, or large patches of blue sky or ocean or grass may fall into these categories.

Consider the following as a starting point for the handling of sets of picture elements:

```
CREATE TYPE HORIZONTAL_LOC = NEW SMALLINT

CREATE TYPE VERTICAL_LOC = NEW SMALLINT

CREATE GENERATOR TYPE PEL_LOCATION(M,N) = NEW
      RECORD      (      hor:HORIZONTAL_LOC:RANGE(0..M-1),
                         vert:VERTICAL_LOC:RANGE(0..N-1)
                  )

CREATE GENERATOR TYPE PICTURE_ELEMENT(M,N,P) = NEW
      RECORD      (      pixel:PIXEL(P),
                         loc:PEL_LOCATION(M,N)
                  )

CREATE GENERATOR TYPE PICTURE_SET(M,N,P) = NEW
      SET OF PICTURE_ELEMENT(M,N,P)
```

Note that picture sets comprised mainly of contiguous pixels of the same pixel value can be represented internally quite efficiently as a "labeled tree" of pixels using the methods of [11].

Operations

Consider PICTURE_ARRAY and PICTURE_SET to be two different logical representations of the same entity.

```
CREATE GENERATOR TYPE PICTURE(M,N,P) = NEW
      CHOICE(array:PICTURE_ARRAY(M,N,P),set:PICTURE_SET(M,N,P))
```

Define CAST operations for casting instances of PICTURE to either a PICTURE_ARRAY or PICTURE_SET logical representation. Define operations for Cutting, Pasting, and Copying pictures to and from each other. Define how to build "aggregate" pictures from smaller pieces. Define a notion of "layering" so that pictures can be placed on top of one another. Possibly incorporate some of the emerging Computer Graphics image processing operations [24] as standardized methods on picture types. Don't get too fancy -- leave the real subtle color and photographic enhancements to "professional" software packages -- but provide these application packages with the links they need to pictures or picture components stored in the database.

Import/Export

Define export and import functions to cast pictures to and from the common standard (ISO/IEC and CCITT) image representations, e.g., ISO ODIF images, ISO JPEG compression, and/or CCITT Group 4 Facsimile standards. Reference [18] discusses the most common de facto standards for bit-mapped graphics, including PICT sponsored by Apple, TIFF (Tagged Image File Format) sponsored by Microsoft, and GIF sponsored by CompuServe.

Markers

At times it is important to be able to "mark" a certain position in a PICTURE data type or to identify a "segment" of the picture data for special treatment. Markers and Segments could be used for animation of certain marked portions of a picture or for defining "hot spots" in HyperMedia applications. The following data types are candidates for discussion:

```
CREATE TYPE PICTURE_MARKER = NEW
        RECORD      (       pictureID:(REFERENCE to some PICTURE instance),
                            hor:HORIZONTAL_LOC,
                            vert:VERTICAL_LOC,
                    CHECK   (hor   <=   HOR_SIZE(pictureID)   AND   vert   <=
                    VERT_SIZE(pictureID))
                    )

CREATE TYPE PICTURE_QUADRANT = NEW
        RECORD      (       pictureID:(REFERENCE to some PICTURE instance),
                            begin_hor:HORIZONTAL_LOC,
                            end_hor:HORIZONTAL_LOC,
                            begin_vert:VERTICAL_LOC,
                            end_vert:VERTICAL_LOC,
                    CHECK ( begin_hor <= end_hor AND begin_vert <= end_vert AND
                    end_hor   <=   HOR_SIZE(pictureID)   AND   end_vert   <=
                    VERT_SIZE(pictureID))
                    )

CREATE TYPE PICTURE_SUBSET = NEW
        RECORD      (       pictureID:(REFERENCE to some PICTURE instance),
                            elements:(SET OF PICTURE_LOCATION(M,N)),
                    CHECK   (M   =   HOR_SIZE(PictureID)   AND   N   =
                    VERT_SIZE(PictureID))
                    )
```

4.9 Audio

Take full advantage of recognized existing standards for digitial audio representations. The Audio Engineering Society (AES) has established standards for two-channel serial transmission [20], multichannel interface [21], and synchronization of equipment [22]. Each of these standards is consistent with accepted CCIR, CCITT, and IEC standards for audio and electronic sound equipment. The International MIDI Association has established a Musical Instrument Digital Interface (MIDI) standard [23]. IEC Pub 908, "Compact disc digital audio system", and IEC Pub 958, "Digital audio interface", are also applicable.

The following paragraphs are taken from Reference [18]:

> The SND resource format is very popular for sampled (i.e., recorded and digitized) sound on Macintosh computers. It is supported by sound-editing programs such as SoundEdit and by

authoring programs such as MacroMind Director and HyperCard. It supports 22-KHz stereo and monaural sound, as well as lower sampling rates. For CD-quality (i.e., 44.1-KHz) sound on a Macintosh, the Audio Interchange File Format (AIFF) is a popular choice. Another often-used format is SoundDesigner by Digidesign, to support its AudioMedia board.

There is no standard for sound on the IBM PC, but a product growing in popularity is the SoundBlaster card from Creative Labs which offers 44.1-KHz sound. The file formats for this card include .VOC (voice), .ORG (organ), .CMF (creative music), .CMS (creative music stereo), and .MID (MIDI).

In the worlds of professional and amateur music, MIDI (musical instrument digital interface) is the standard [23] for connecting synthesizers, keyboards and other musical instruments to computers. Unlike most sound formats, MIDI is a series of instructions, not a description of a waveform, and MIDI instructions can be triggered by means of a script.

One might use the following as a starting point for an Audio ADT package:

```
CREATE GENERATOR TYPE SAMPLE(S) = NEW ARRAY [0..S-1] OF BIT
```

Note: S is the size in bits of the sample, see AIFF [19]. In AIFF, each sample point is a linear, 2's complement value from 1 to 32 bits wide. AES [20] defines a positive audio sample to be the positive analog voltages at the input of the analog-to-digital converter (ADC). AES restricts bit sizes to be either less than 20 bits or 24 bits. If less than 20 bits are used, then those bits determine the "audio sample". If 24 bits are used, then the first 4 bits define a "preamble" and the next 20 bits determine the "audio sample".

It is important that SQL not attempt to specify the semantics of a sample sound representation; instead, SQL should provide basic ADT's so that applications can manipulate segements of sound and store and retrieve audio components from the database, even if the exact bit string value of a "sample" is implementation specific.

We might define multichannel audio architectures as follows:

```
CREATE TYPE CHANNEL = NEW ENUMERATION(mono, stereo, 3-channel, quad, 4-
channel, 6-channel)

CREATE TYPE CHANNEL_ID = NEW INTEGER:RANGE(1..6)
```

Note: For multichannels, AIFF [19] recommends the following mapping from Channel type to channel_id's:

Channel_ID

	1	2	3	4	5	6
Stereo	left	right				
3-Channel	left	right	center			
Quad	front left	front right	rear left	rear right		
4-Channel	left	center	right	surround		

6-Channel	left	left center	center	right center	right	surround

```
CREATE GENERATOR TYPE AUDIO_FRAME(C,S) = NEW ARRAY [1..C] OF SAMPLE(S)
```

Note: C identifies the number of channels, so each frame assigns one sample to each channel.

```
CREATE GENERATOR TYPE SOUND(C,S) = NEW LIST OF AUDIO_FRAME(C,S)

CREATE TYPE AUDIO_RATE = NEW INTEGER:RANGE(x..y)
```

Note: The rate defines the number of frames per second. Are there certain "standardized" rates? Yes -- in [20], AES only recognizes sampling frequencies of 32-kHz, 44.1-kHz, or 48-kHz. What's the relationship between 32-KHz or 44.1-KHz and the rate value in frames/second? Defined in [20]? No!

Note: We assume in the following discussion an "exact" integer number of frames per second. Changes are needed if this is later found to be a false assumption.

```
CREATE TYPE AES_STATUS = NEW ARRAY [1..24] OF OCTET
```

Note: AES_Status is a 24 octet field specified as AESChannelStatusData in the AES digital sudio specification [20]. Some of this information duplicates the explicit fields defined above. Of additional interest might be bits 2,3, and 4 of byte 0, which describe recording emphasis.

```
CREATE GENERATOR TYPE AUDIO(C,S,R) = NEW
    RECORD      (       number_channels:(SMALLINT:RANGE(1..6)),
                        sample_size:(SMALLINT:RANGE(1..32)),
                        audio_rate:AUDIO_RATE,
                        channel_distribution:CHOICE(NULL,CHANNEL),
                        AES_status:CHOICE(NULL,AES_STATUS),
                        sound:SOUND(C,S),
                CHECK (C=number_channels AND S=sample_size AND R=audio_rate)
                )
```

Note: Can AES_status be different for each channel? If so, then AES_status should be ARRAY [1..C] OF CHOICE(NULL,AES_STATUS).

Markers

At times it is important to be able to "mark" a certain position in an Audio data type or to identify a "segment" of the sound data for special treatment. Markers and Segments could be used for repetition of certain marked sequences of frames, for synchronization of various input sources, or for Audio HyperMedia applications. The following data types are candidates for discussion:

```
CREATE TYPE AUDIO_POSITION = NEW INTEGER:RANGE(0..MAX)
```

Note: In AIFF [19], a position in sound data marks a location that conceptually falls between two sample frames. A location that falls before the first sample frame in the sound data is at position zero, while a location that falls between the n-th and n+1-st sample frames in the sound data is at position n. A position larger than the length in frames of a sound data element could return an error or could just default to the end of the list of frames.

Note: For very long sound bites (e.g., several hours) and for very high sampling rates (e.g., 48-kHz) the integer position may overflow the integer data type in some implementations. To avoid this problem, it may be necessary to structure a POSITION in terms of Hours, Minutes, Seconds, and Frame position within a second.

```
CREATE TYPE AUDIO_MARKER = NEW
        RECORD      (       audioID:(REFERENCE to some AUDIO instance),
                            position:AUDIO_POSITION
                    )

CREATE TYPE AUDIO_SEGMENT = NEW
        RECORD      (       audioID:(REFERENCE to some AUDIO instance),
                            begin:AUDIO_POSITION,
                            end:AUDIO_POSITION,
                    CHECK ( begin <= end)
                    )
```

4.10 Video

Take full advantage of recognized existing standards for digitial video representations. Unfortunately, there is more than one candidate and they are not necessarily mutually compatible.

The following paragraphs are taken from Reference [18]:

> Analog video signals can be incorporated into a digital presentation, and a digital image can be output to videotape in analog format. The relevant standard for analog video in the United States and Japan is called NTSC (National Television Standards Committee). Its counterpart in Great Britain and most of Europe, South America, and Australia is PAL (Phased Alternate Line).
>
> NTSC describes a signal that displays color images, or frames, at the rate of 30 frames per second. Each frame has 525 lines of image information divided into two fields (even-numbered and odd-numbered lines) for interlaced scanning. PAL uses a 625-line frame. By contrast, computer displays usually offer 480 lines of non-interlaced video.
>
> Voice is synchronized by using SMPTE (Society of Motion Picture and Television Engineers) time code, which is a signal recorded on the tape. SMPTE is the recognized standard in professional video circles. It measures time in minutes, seconds, and frames.

Storage requirements and data transfer rates for real-time video surpass the capabilities of many traditional data processing systems. For this reason, video compression algorithms are very important. According to [18], ISO's MPEG (Motion Picture Experts Group) is a standard video-compression algorithm that is two years away from final adoption in ISO and is already endorsed by JVC, IBM, Phillips, and Matsushita. The MPEG algorithm allows for compressing multiple frames and recording only changes from frame to frame. For small pictures; i.e., video in a window, MPEG can be used in real time from storage media having 1 to 1.5 Mbps data trasfer rates; whereas full screen broadcast-quality motion video requires data transfer rates in the range of 16 to 24 Mbps, which is too fast for today's LANs and storage devices. Improvements to the MPEG algorithm could bring its full-screen transfer rate requirement down to about 4 to 6 Mbps, which is in the range of the next generation of digital storage products for the desktop [18].

Video, like Audio, can be represented as a sequence of frames, where each frame is defined as in Section 4.8, "Pictures." It is important that SQL not attempt to specify the semantics of a video-frame representation; instead, SQL should provide basic ADT's so that applications can manipulate segements of video and store and retrieve video components from the database, even if the exact bit string value of a "video line" or "video pixel" is implementation specific.

Consider the following as a starting point for video ADT's:

```
   CREATE GENERATOR TYPE MOTION_PICTURE(M,N,P) = NEW LIST OF PICTURE(M,N,P)
```

It may or may not make sense to define the following subtypes, because many television representations are more interested in "lines" than in "picture elements" or "pixels." And lines are often interleaved so that the odd lines are transmitted first and then the even lines, or *vice versa*. However, whatever the internal representation, an implementation should be able to produce the "picture elements" of a specific "video frame," so the above logical definition of a MOTION_PICTURE data type seems appropriate.

```
        CREATE TYPE TELEVIDEO UNDER MOTION_PICTURE(480,N,24) where N=?

        CREATE TYPE NTSC_VIDEO UNDER MOTION_PICTURE(525,N,P) where N=? and P=?

        CREATE TYPE PAL_VIDEO UNDER MOTION_PICTURE(625,N,P) where N=? and P=?

        CREATE TYPE HD_VIDEO UNDER MOTION_PICTURE(M,N,P) where M=?, N=?, P=?
```

Also, it may make sense to decompose PIXEL into three parts for Red, Blue, and Green primary colors? Is that where 24-bits in a pixel for TELEVIDEO comes from; i.e., one OCTET for each primary color intensity? See the discusssion under Section 4.8, "Pictures," for reasons why the various color models for picture elements may not be good candidates for standardization in SQL.

```
        CREATE TYPE VIDEO_RATE = NEW INTEGER:RANGE(x..y)
```

Note: The rate defines the number of frames per second. Are there certain "standardized" rates? Yes -- but video rates seem to be considerably more complicated than Audio rates. It may make more sense to "name" the different video rates that are popular in the industry. I've seen references to a fractional number of frames per second!

Note: We assume in the following discussion an "exact" integer number of frames per second. This may not be the case. However, the SMPTE time code and synchronization standard mentioned above identifies each frame by minute and second, and frame within second. But the number of frames per second may not always be constant. Sometimes a frame gets replayed or dropped to maintain synchronization.

```
    CREATE GENERATOR TYPE VIDEO(M,N,P,R) = NEW
            RECORD     (       lines_per_frame:SMALLINT,
                               pels_per_line:SMALLINT,
                               pixel_size:SMALLINT,
                               frame_rate:VIDEO_RATE,
                               film:MOTION_PICTURE(M,N,P),
                    CHECK (M=lines_per_frame AND N=pels_per_line AND P=pixel_size
                    AND R=frame_rate)
                    )
```

Markers

At times it is important to be able to "mark" a certain position in a VIDEO data type or to identify a "segment" of the video data for special treatment. Markers and Segments could be used for repetition of certain marked sequences of frames, for synchronization of various input sources, or for Video HyperMedia applications. The following data types are candidates for discussion:

```
CREATE TYPE VIDEO_POSITION = NEW INTEGER:RANGE(0..MAX)
```

Note: A position in motion picture data marks a location that conceptually falls between two motion picture frames. A location that falls before the first frame in the motion picture data is at position zero, while a location that falls between the n-th and n+1-st sample frames is at position n.

Note: It may be necessary to modify this definition to bring it into alignment with the SMPTE standard referenced above, which numbers frames by minute, second, and frame within second.

```
CREATE TYPE VIDEO_MARKER = NEW
        RECORD      (       videoID:(REFERENCE to some VIDEO instance),
                            position:VIDEO_POSITION
                    )

CREATE TYPE VIDEO_SEGMENT = NEW
        RECORD      (       videoID:(REFERENCE to some VIDEO instance),
                            begin:VIDEO_POSITION,
                            end:VIDEO_POSITION,
                    CHECK ( begin <= end)
                    )
```

4.11 HyperMedia data types

It is sometimes desirable to be able to combine text, graphics, still images, audio, and video in a single document so that complete multimedia packages can be interchanged among interested parties. Digital Equipment Corporation is developing a compound document format that combines text, graphics, and sound in a single document and Microsoft and IBM have proposed RIFF (Resource Interchange File Format) as a general specification for text, graphics, animations and sounds [18]. Various Document Type Definitions (DTD) for SGML in CALS applications have the same goal. Each of these specifications is defined as an umbrella data type that incorporates other standard formats for the underlying components.

In addition, it is often desirable to be able to embed cross-references among various multi-media pieces into the same document. If this is achieved, then Hypertext/Hypermedia documents can be interchanged among hypermedia information processing systems. A recently emerging file format for interchanging hypermedia documents is HIFF (Hypermedia Interchange File Format) sponsored by the HyperMedia Group of Emeryville, CA. This format has been used in a commercial product that converts HyperCard stacks on the Macintosh to ToolBook documents on the PC. Another emerging specification is HyTime

(Hypermedia/Time-based Document Representation) sponsored by ANSI X3V1 in the United States and as a committee draft proposal in ISO/IEC JTC1/SC18. It is an SGML-based specification that supports "links" and "synchronization." The new Apple and IBM alliance announced in a press release that standards for multimedia datatypes are one of their high priorities.

A number of vendors are developing protocols for operational interface to video and audio equipment, including camcorders, cameras, VCRs, videotape recorders, laser videodisc players, and CD-ROM drives. Microsoft, along with IBM, recently has proposed the Media Control Interface (MCI) as a set of protocols for controlling a variety of multimedia devices. According to [18], MCI has a good chance of becoming the de facto standard interface for all media devices for the Microsoft Windows environment. Apple Computer has proposed a similar set of protocols, called QuickTime, for interfacing audio/visual equipment to the Macintosh. Sony has proposed ViSCA and LANC control protocols for Sony products.

With a collection of standardized multimedia data types and standardized interface protocols, SQL3 application programers should be able to use SQL queries to retrieve appropriate multimedia pieces from the database and feed them to supporting peripherals for playback. In many cases, the peripherals can be defined as SQL user-defined, abstract data types with the standardized operational protocols as methods.

4.12 ASN.1 data type

The Abstract Syntax Notation (ASN.1) standards (ISO 8824 and ISO 8825) define a number of data types along with an underlying representation for each of those data types as an Octet String; i.e., LIST OF OCTET where OCTET = ARRAY [0..7] OF BIT. We should make sure that all SQL data types can be cast to some ASN.1 data type (preferably a predetermined one), and that all ASN.1 data types are representable in SQL. With the appropriate cast operations to Octet Strings, the RDA standard can be used for interoperability, even if the communicating systems don't fully support one another's data types.

5. External repository interface (ERI)

CALS applications require access to multiple data repositories, not all of which are managed by SQL conforming processors; in fact, most are probably managed by non-SQL processors. It is not unusual for CALS applications to require data from the operating system, from graphics repositories, from CD-ROM's, from CAD/CAM databases, or from libraries of cataloged data. We need to consider a new conformance alternative in SQL so that such non-SQL data repositories can make their data available, in simplified but standard form, to SQL systems or SQL applications. NIST representatives have proposed development of such new interface specifications to ANSI and ISO standardization committees. The next year will see if we can persuade others to see the benefits.

From a user's perspective, it is unrealistic to expect every data repository to be able to handle even the lowest "Entry Level" SQL queries. For example, who would expect an electronic mail system to handle SQL joins and subqueries over its message headers? Yet, every e-mail system is a data repository with information that applications sometimes require. Sometimes it may make sense for an e-mail system to use an SQL conforming DBMS as its persistent data repository, but more often an e-mail system will totally manage its own data -- even to the point of using encrypted files in the file system of the underlying operating system. What we need is an interface specification that enables a non-SQL data repository to make certain external views available to SQL processors and for those SQL processors to, in turn, allow the full power and flexibility of the SQL query language over those views to the end user.

This "consciousness-raising" discussion is intended to heighten awareness to user requirements for better integration among heterogeneous data repositories. It may make sense to specify a "client" and a "server" interface to external repositories so that non-SQL systems can act as servers to SQL requests for data. It may make sense to propose a new standardization project to develop the conformance requirements needed for non-SQL systems to provide SQL views of their data and for SQL systems to provide fully functional views over that data to SQL users. This interface might be specified as part of the emerging SQL3 standard [1], or processed as a completely separate standard, or even defined and processed as a CALS/PDES specification. In any case, it would repackage functionality from the SQL and RDA standards and give new conformance requirements for both SQL and non-SQL systems.

If we label this new interface as the SQL external repository interface (SQL/ERI), then it might consist of a "server" part and a "client" part. Non-SQL systems could claim conformance to the "server" part and SQL systems could claim conformance to the "client" part. A wide range of non-SQL products and services might be able to claim conformance as SQL/ERI Servers. They could provide high level abstract data types with application specific methods and operations. They might even be required to provide an SQL schema definition to describe data that will be externally available. In addition, SQL/ERI Servers would be required to evaluate "simple" SQL queries over individual tables defined in the schema. The exact meaning of "simple" can be specified in the ERI definition -- possibly at different levels of service. The SQL processor can then think of the external repository as an SQL_CATALOG that can be CONNECTed to, but that can only respond to whatever SQL statements are specified for that level of service.

In turn, SQL systems might be able to claim conformance as SQL/ERI Clients. If an SQL system claims conformance as an SQL/ERI Client, then it agrees to provide SQL functionality, at whatever level of the SQL standard it conforms to, over any table provided by an SQL/ERI Server. This may require that the SQL system automatically create a temporary table whenever the external view is referenced in a query, and then populate that table using the limited capabilities provided by the "server" interface so that it can guarantee the ability to perform nested queries, or searched updates and deletes, or recursive queries, or whatever is requested by its application.

With the SQL/ERI "client" and "server" definitions, non-SQL systems would be able to provide services to SQL-based applications even though they might not be able to provide the expected query flexibility, access control, concurrency control, or updatability required of a full-function SQL data manager. Full-function SQL processors could provide these expected data management facilities and, in addition, provide user access to data repositories not otherwise accessible via the SQL language. Section 2.2, "External schema communication," describes how the SQL/ERI definitions might be used to provide uniform CALS application access to both SQL and non-SQL data at local and remote sites.

The SQL/ERI standard might provide several different conformance packages for non-SQL systems. Certainly one kind of conformance package might coincide with the services provided by read-only tables on CD-ROM, e.g., the data repositories discussed in Sections 5.3 and 5.4 below. Data on the CD-ROM would "conform" to this SQL/ERI server package if it includes a data manager kernel that is executable on a wide range of workstations and responds correctly to an SQL CONNECT statement using the call interface provided by that workstation. Another kind of conformance package might coincide with the services provided by online databases, such as those discussed in Sections 5.2 and 5.5 below. Such systems would be required to respond to RDA requests as if they were remote SQL servers conforming to the RDA/SQL-Specialization. Other conformance packages might correspond to the services required in a multi-vendor environment, such as that discussed in Section 5.6 below. Some of the tables might be updatable and there may be some requirements to "read" SQL data in other remote catalogues.

In each of these cases, a conforming SQL/ERI server would be required to be "self-describing" as if it were a separate SQL catalog. It would be required to supply an SQL Information_Schema describing all available tables and the equivalent SQL data types for its columns. In particular, the following tables from the Information_Schema would probably be required, although some might be empty: TABLES, COLUMNS, DOMAINS, VIEWS, and SCHEMATA. If the ERI Server provides new abstract data types not defined in the SQL standard, then it would also be required to provide an SQL ADT interface definition as specified in the emerging SQL3 standard.

The following sections provide some examples of situations that could benefit from a new "external repository interface" (ERI) standard.

5.1 Electronic mail repository

Suppose an electronic mail system maintains persistent information about all of the messages it handles. Also suppose that the special "Mail" directories at each user's node in the file system are accessible only by the e-mail system itself and thus form part of the e-mail system's data repository.

The e-mail system might choose to provide the following tabular views to external applications:

Mail (MailID, Owner, Title, Subject, Content)

Mail_Sent (MailId, Owner, Title, To_Address, Reply_To, TimeStamp, Length, Confirmation)

Mail_Received (MailId, Owner, Title, From_Address, TimeStamp, Length)

It is reasonable for the e-mail system to maintain the Mail_Sent and Mail_Received tables indexed on "Owner" and to be able to access directories in the user's file space for the Mail table. The "Owner" of a sent message is the user who sent it and the "Owner" of a received message is the user who received it. If a message is sent in "reply" to another message, then the "Reply_To" column contains the MailId of the message replied to, otherwise it is null. Upon receiving a message, a user could "file" it in a "Subject" directory using a facility common to many e-mail systems. It should then be relatively easy for the e-mail system to respond to simplified SQL queries; i.e., those with only a single table mentioned in the FROM clause (e.g., no joins or derived tables) and only simple AND conditions in the FROM clause, and having no GROUP BY or HAVING clauses. Queries on the Mail table would automatically be qualified in the WHERE clause with an additional term of the form "AND Owner = USER." Except for the special user "E-mail Administrator," queries on the other two tables would be similarly qualified.

A full-function SQL system conforming to the "client" part of the ERI interface would be able to CONNECT to the e-mail system, attach itself to and "read" ERI tables. It could then provide "views" over each of the tables to each Owner. The effect would be a view in each Owner's schema of each of the three tables with only the instances relevant to that Owner available through the view. Once these views are established, the Owner could control privileges on them by using the usual SQL GRANT and REVOKE statements.

With this approach, an external application would have access to whatever e-mail information it was granted privileges to by the mail Owner, yet the e-mail system itself need not be concerned with complex queries or privilege granting mechanisms; instead, it need only respond to each request from an Owner Auth_Id to provide specific data relevant only to that Owner.

For example, suppose we want to determine if Phil sent any messages to Jim on the topic of "DIS Ballot" during August 1991 that Jim received and properly catalogued but forgot to reply to. Assuming that both Jim and Phil have granted privileges to me on a view that includes the "DIS Ballot" subject, we could determine the answer to my request as follows:

```
Select * From Phil.Mail_Sent PMS
      Where  TimeStamp >= DATE '1991-8-1 0:0:0'
             AND   TimeStamp < DATE '1991-9-1 0:0:0'
```

```
        AND   To_Address LIKE '%jim@digital.com'
        AND   Confirmation = 'yes'
        AND   EXISTS
              ( Select * From Jim.Mail_Received JMR
                    Where JMR.Title = PMS.Title
                          AND   JMR.From_Address LIKE '%phil@ibm.com'
                          AND   JMR.TimeStamp > PMS.TimeStamp
                          AND   JMR.TimeStamp < PMS.TimeStamp + INTERVAL '3'
                                DAY
                          AND   EXISTS
                                ( Select * From Jim.Mail JM
                                      Where JM.MailId = JMR.MailId
                                            AND   JM.Subject = 'DIS Ballot' )
                          AND   NOT EXISTS
                                ( Select * From Jim.Mail_Sent JMS
                                      Where JMS.Reply_To = JMR.MailId ) )
```

The exact correctness of the above query is not as important as the fact that it is likely to be quite complex and probably will involve remote database access (RDA). An e-mail system shouldn't be expected to have to deal with this type of complexity, or remote access, but DBMS's conforming to both SQL and RDA standards should be able to handle both the complexity and the remote access routinely. Additional complexity would follow if a user wanted to trace "recursively" the strings of replies that begin with some message. Such recursive capability is part of the emerging SQL3 specification [1], but is not likely to be part of any e-mail system.

What is needed to make the above scenario feasible is an SQL/ERI standard specification. The e-mail system could present external views to each Owner through an intermediate SQL "client" processor. The SQL client could create a temporary table for each view and allow the full power of whatever level of SQL it supports to be applied.

The SQL processor should also make it possible for the Owner to execute searched UPDATE and DELETE statements against the Mail table, but not the Mail_Sent and Mail_Received tables because they are maintained by the e-mail system. For example, we should be able to execute the following:

```
Update Len.Mail Set Subject = "2nd DIS Ballot"
      Where Subject = "DIS Ballot"
            AND   TimeStamp > Date '1992-1-30'

Delete From Len.Mail
      Where TimeStamp < Date '1991-1-1 0:0:0'
            AND   Subject NOT LIKE '%Ballot%'
```

It would be the responsibility of my SQL processor to use whatever facilities are provided by the SQL/ERI interface to CONNECT to the e-mail system and update or delete the appropriate messages. If the ERI interface only allowed a positioned delete over the files in Len.Mail, then the SQL processor would have to set up the appropriate cursor, check the predicate, and issue the appropriate Updates or Deletes.

A special e-mail user, the "E-mail Administrator," should be able to execute the following:

```
Delete From Mail_Received Where TimeStamp < Date '1988-1-1 0:0:0'
```

If there were Referential Integrity constraints from the Mail table to the Mail_Received table, then the "E-mail Administrator" would be able to query the DIAGNOSTICS information in SQL to obtain a list of all Owners that were keeping old mail in their files (maybe against the rules!). Or a CASCADE DELETE declaration could specify that old mail in individual directories is automatically deleted, or set Null, or some other desired result. Each of these alternatives are features provided routinely by conforming SQL systems that would be unreasonable to expect of the average e-mail system.

None of this is possible unless an SQL/ERI standard exists that specifies the syntax and semantics required of both the SQL "client" and the external repository "server."

5.2 Archival data storage

The U.S. government maintains a number of large data repositories for the mutual benefit of federal agencies and interested commercial concerns. The following is a representative sampling of such repositories:

USGS Earthquake Data -- The U.S. Geological Survey (USGS) of the Department of the Interior is using optical disk systems to store and share natural and man-made earth tremor data. The intent is to make seismic data readily accessible throughout the world. The optical disk system will let USGS preserve the data, without periodic rewriting, for up to 100 years. Right now they use a 12-inch, single-drive Aquidneck Systems optical archiving system over a cluster of ten Digital MicroVax's with the support of about 5 gigabytes of on-line disk storage. (See Govt Computer News 10/14/91).

Civil War Database -- The personal histories of more than 3.5 million Civil War soldiers will be available on line through the National Park Service's Civil War Soldiers System (CWSS) within the next few years. CWSS will link a soldier's name to his regiment and cite the battles in which that regiment fought. All told, the database will hold about 5.5 million records. By 1994, historians and other interested parties will be able to trace a soldier's military history and burial records through the comprehensive database. Remote users will be able to access the database via modem links to a Unix-based Aviion 6200 at NPS headquarters in Washington. CD-ROM versions of the data will be available at 28 Civil War parks across the nation. Volunteers from the Federation of Genealogical Societies will enter most of the data, thereby saving the government about $4.5 million in development costs. (See Govt Computer News 10/14/91).

USIA Drug Data --Librarians at USIA's Washington headquarters soon will begin using CD-ROM Hypertext systems to develop and maintain an International Drug Library. The library will be a collection of full-text materials on aspects of the illegal drug issue, from international interdiction and eradication efforts to prevention, treatment, arrests, and drug laws. The Hypertext systems to be applied will use Standard Generalized Markup Language (SGML) to automatically create tables of contents, indexes for searching, and hypertext links to other SGML documents, so users can find exactly what they need. The Hypertext systems will allow USIA librarians to mark up raw text, dividing the information into titles, chapters, section headings, etc., all in SGML. The Hypertext systems will retain the SGML markup within the documents, but additionally use it to create tables of contents, fields for searching, and hypertext cross-references. Users in libraries overseas will be able to access the data, searching by title, chapter, or section, or if desired, directly access the raw text with system provided search functions. (See Govt Computer News 10/14/91).

NGS Ocean Floor Analysis Tools -- Geophysicists at the National Geodetic Survey (NGS) of the National Oceanic and Atmospheric Administration (NOAA) have a requirement for specialized data access to ocean floor survey data. They need special data analysis tools to help chart the sea floor in parts of southern oceans that are poorly mapped from surface shipboard surveys. Such tools will display isosurfaces, threshold surfaces, and arbitrary cut planes in 3-D, as well as combine 3-D gridding methods for interpolation from 3-D scattered data points, like temperature and pressure, into a 3-D gridded data structure. (See Govt Computer News 9/30/91).

All of these examples came from just a couple of issues of recent newspapers, and thereby show the large extent of non-SQL data repositories in the federal government. For whatever reason, these systems are being implemented with specialized data management tools, not general purpose SQL database management systems. Yet, user applications, like the NOAA/NGS analysis tools above, need "standard" access to these data repositories.

What is needed, is an SQL/ERI standard, so that these non-SQL data repositories can provide a simple, external interface, accessible from full-function SQL systems. Sophisticated applications can then be built without the need to "understand" the non-standard data access methods unique to each repository. Instead, full-function SQL systems could be used as intermediaries. The SQL "client" could CONNECT itself to the non-SQL "server" using the standard SQL/ERI interface; the application could then use the full power and flexibility of the SQL data manipulation language, as well as the system provided special access methods, to select and mange the data as if it were maintained in an SQL database.

5.3 Structured Full-Text Query Language (SFQL)

Within the airline industry, standards for manufacturers' technical data are developed and maintained under the auspices of the Air Transport Association (ATA) in conjunction with the Aerospace Industries Association (AIA). The emphasis of subcommittee 89-9C is "requirements and standards for advanced retrieval systems for manuals." This group has produced a draft specification [15] for a structured full-text query language (SFQL) that is "similar" to SQL. The intent is to provide all maintenance manuals and other information for aircraft engines on CD-ROM. The CD-ROM would also contain database engine programs, executable on a number of different workstations, that would respond to high-level SFQL queries. The intended result is that users could use their favorite presentation tools to access the data on the CD-ROM without having to be aware of specific data access methods; instead, they would use "standard" SFQL to specify the desired access.

Earlier versions of the SFQL language have been criticized by various SQL committees as having the flavor of SQL, without abiding by the semantics. In some cases the original SFQL used SQL "join" syntax in the FROM clause to specify something completely alien to SQL. The newest version of SFQL [15] addresses this criticism by severely restricting the scope of the language to make it more compatible with SQL. It is no longer possible to specify more than one table in a FROM clause, or to specify subqueries, or GROUP BY or HAVING clauses. Instead, SFQL provides very simple SQL access, but with a number of powerful additional "methods" applicable to TEXT objects. It may now be possible to view SFQL as providing a simple external interface to a document repository, with the addition of special ATA abstract data types having their own methods, "relevance functions," distance functions, etc.

If an SQL/ERI standard existed, we believe it would be possible for a new version of SFQL to be defined that "conforms" to the specification of an ERI Server. This would be of benefit to both SQL vendors and the airline producers of CD-ROM based technical manuals. A full-function SQL system residing on the host computer could CONNECT itself to the SFQL engine and provide SQL views to the end user's application programs. The SFQL engine would provide simplified external views of tables of contents, tables of figures, tables of diagrams, parts lists, price sheets, etc., without having to worry about outer joins, derived tables in the from clause, recursive queries, or other advanced SQL features. The SFQL engine could also provide what general purpose SQL systems cannot provide; i.e., specialized methods and functions on ATA_TEXT data types. The end user's application programs could use the full power and flexibility of SQL to retrieve the desired information and then "present" it to the user in the form desired by that user. The user gets exactly what is desired without the SFQL engine, or the SQL processor, needing to be aware of "presentation" preferences, and without the general purpose SQL system needing to be aware of special "ATA" data types and methods.

As an example of the kinds of restrictions that might be appropriate for some level of an ERI "server" interface, we list some of the restrictions to SQL assumed by the SFQL specification in [15]:

- No ability to process schema definition or schema manipulation statements; i.e., no CREATE, DROP, or ALTER statements. The SFQL schema is fixed for each SFQL "repository."

- No ability to update data or add new data to the database; i.e., no UPDATE or INSERT statements. The data is read-only for each SFQL "repository," as might be expected for the intended CD-ROM distribution method.

- No Transaction Management capabilities; i.e., no COMMIT or ROLLBACK statements. Since the data is read-only, there is no possibility of encountering "dirty" or "phantom" reads.

- No Join capability; i.e., no ability to combine data residing in two or more SFQL tables. This is what led the developers of SFQL to extend the SQL tabular data model in ways that SQL development committees consider to be "inappropriate." This may not be a problem if a full-function SQL system is part of the application environment at execution time.

- No View definition capability; i.e., no ability to specify pre-defined queries to provide appropriate external views to each user. Again, this may not be a problem if a full-function SQL system is available for this purpose.

- No Subquery capability; i.e., no ability to specify derived tables in the FROM clause or to reference other tables in the WHERE clause. It does seem that SFQL provides some capability for "intermediate" results, so that the output of one SELECT statement can be used as the basis for the evaluation of subsequent SELECT statements.

- No Module Language or Procedure definition capabilities; i.e., no capability for a user to specify desired queries that can then be "optimized" by the system for efficient "call" from application programs.

- No Embedded SFQL capability; i.e., no constructs to embed SFQL statements directly into third-generation programming language. Instead, the application must use the special "client/server" interface provided as part of the SFQL specification (based on a Microsoft Windows 3.0 spec).

- No Integrity constraints; i.e., no constructs to specify ranges of values or other constraints on values. This follows directly from the non-existence of a Schema definition language.

- No Access control; i.e., no ability to restrict access to "privileged" users.

- No features from emerging SQL standards (i.e., SQL2 and SQL3); no Dynamic SQL, no Date/Time data types, no concatenation, substring, or index functions, no recursive query capabilities, etc.

The SFQL specification also adds some new functions to its TEXT data types as follows:

- A "contains" predicate to determine if a block of text contains any of a list of given words.

- A "proximity" predicate to determine if a block of text contains any words from one list of words within a given distance of any word in a second list of words. Distance is measured in characters, words, sentences, or paragraphs.

- A "relevance" function for each server to determine, in an implementation defined manner, the relevance of each item of a result set.

- A "match method" for character string matching that can be any of: match exactcase, match anycase, match fuzzy, or default.

- A "hits" function that returns the number of hits to a given query specification.
- A "thesaurus" function that returns a list of synonyms for a given word. The thesaurus function also has "word broaden" and "word narrow" options to return a larger or smaller list of synonyms.

It seems that SFQL is a perfect candidate to be the first specification to attempt conformance to an SQL external repository interface (SQL/ERI) standard. The SFQL intent is to provide simple SQL query access to a fixed repository of non-updatable data, and to provide text management extensibility for specialized data types. This is exactly the intent that should be satisfied by an ERI "server" specification.

5.4 CD-ROM databases

The preceding section discusses one particular attempt to use CD-ROM technology to distribute databases of information. This is not an isolated attempt. A recent issue of American Scientist contains a whole list of "CD-ROM databases." Each of them could benefit from an SQL/ERI specification so that they could make their data available, in a standard manner, to full-function SQL systems. The SQL systems could, in turn, make the data available to any application program through the existing SQL standards for language bindings.

The following CD-ROM databases are listed in the September/October, 1991, issue of American Scientist:

- Biological Abstracts: Abstracts, reports, reviews, meetings, etc. from over 9,000 serials and other publications. Equivalent to the current file of BIOSIS's online database.
- Toxline Plus: An extensive collection of toxical information from Chemical Abstracts Service, BIOSIS, the American Society of Hospital Pharmacists, and the National Library of Medicine.
- USDA Research Information: Provides extensive information on agricultural and food research projects in the United States and Canada. Updated annually.
- International Nuclear Information: Provides information on the peaceful uses of nuclear science and technology, and includes information gathered since 1976 from journals, research reports, monographs, proceedings, theses, conference papers, and patents. Updated quarterly.
- ICONDA Civil Engineering Database: Contains records from over 800 civil engineering journals dating from 1976. Updated semiannually.
- Institute for Scientific Information: The Institute for Scientific Information has released a "Current contents on diskette with Abstracts" in four editions: life sciences, agriculture/biology and environmental sciences, physical/chemical and earth sciences, and clinical medicine. Customers receive weekly updates.
- CompendexPlus Database: The Ei/Engineering Information, Inc. has released two subsets of its CompendexPlus database, the Ei EEDisc for electrical and computer engineering, including sources from journals, conferences, colloquia, proceedings, and other publications, and Ei Energy/Environment Disc, an expanded versions of its Energy Abstracts. Updated quarterly.

A legitimate concern of providers of information on compact disk is to preserve their proprietary investment in the data collection and packaging. An SQL/ERI standard shouldn't add any concerns.

The ERI Server interface that they conform to would simply make it easier for many different applications to access the data, and thus increase the market for selling the information on compact disks. The value added, in terms of "specialized" abstract data types and "convenient" methods defined on these types, would still only be available through the proprietary engine supplied with the data. Even special methods for graphical display of the data could be part of the proprietary engine; the SQL system would simply provide a powerful query language and capabilities for invoking the proprietary methods. The SQL system, and the end user, would still be bound by existing legal requirements concerning re-packaging or re-selling the data.

5.5 Library information services

None of the above sections have even mentioned the extensive services provided by online library information systems. Since we are not very knowledgeable about these systems, we will not attempt to list them. These systems are clearly non-SQL systems that might be willing to provide an SQL/ERI Server interface to their clients.

These systems would still be able to charge a subscription fee to any user that CONNECTs to them, but the user would now be able to use standard SQL queries to retrieve desired information, instead of being required to use the unique human interface, or unique protocols, provided by each such service. This would make library information available to a whole host of new applications. If every library repository were to provide an SQL/ERI external schema definition for its Subject and Author catalogs, and for library specific ADTs, then any SQL application in the world could connect to a remote library using existing RDA communication protocols (see Section 2.2, "External schema communication"), browse the Subject and Author catalogs, and access selected document contents using the specialized methods provided for each document type.

Library information service providers have the same proprietary concerns as do CD-ROM suppliers (discussed in the preceding section), but end users would still be bound by existing legal requirements concerning re-packaging or re-selling the data. The "value-added" provided by the information service would be legally protected.

5.6 Manufacturing engineering data

In a manufacturing environment, activities are hierarchically arranged into plant, workstation, and cell modules. A cell may consist of a single robot or other manufacturing tool, a workstation may consist of a group of tools, and a plant may consist of a group of workstations. Each tool at the cell or workstation level may produce data that is of interest to other components of the manufacturing environment, but the resources available at the cell or workstation may be unable to support a full-function SQL database management system. Instead, it may be more reasonable for the cell and the workstation to provide conforming SQL/ERI "server" facilities to make available their data in limited form for access or update by external sources.

For example, if a cell represents a single robot, then the cell may maintain only three tables as follows:

Activities (ActivityId, SampleId, OwnerId, Activity_Name, TimeStamp)

Requests (RequestId, ProcedureName, SampleId, Timestamp)

Sensors (Line_Pressure, Cutting_Temperature, Active_Procedure, TimeStamp)

The Requests table would be updatable from the parent workstation so that a new request to perform a Procedure on a specific Sample could be made at any time. The Activities table would be read-only; it keeps a log of all actions performed by the robot since the last time this table was read by the parent workstation. The Sensors table would be read-only; it reads all of its available sensors at specific time intervals and stores the results in this table for access by the parent workstation. Triggers and assertions in the Sensors table could send appropriate warning messages if safety constraints were exceeded. A request would be deleted from the Requests table after it is completed by the robot. After every read of the Activities or Sensors tables by the parent workstation, a trigger would empty them so that the robot's storage requirements would never exceed the limited capacity available.

It is reasonable for the manufacturer of a robot to specify as part of its claim of conformance to an SQL/ERI specification, the tables, abstract data types, and methods that will be supported. In addition, a cell may assume it has CONNECT capabilities via Remote Database Access (RDA) to other tools in a manufacturing environment. In that way, the parent workstation software can assume it is communicating with a standard SQL server, even though the robot only provides limited access to three tables. In turn, the workstation can provide standard SQL/ERI "server" views to its parent manufacturing plant. A workstation may be able to "pass on" any views provided by the cells, as well as provide specific status tables of its own. Maybe the manufacturing plant parent will have the only full-function SQL database management system in the plant environment.

Using the above facilities, a manufacturing plant can be configured from any number of cells and workstations, each procured from a separate vendor. Each workstation will be able to communicate, using SQL language, with each of its component cells, even though neither the cells nor the workstation have the ability to function as a full-function SQL database management system.

6. Conclusions

Database Language SQL provides standardized facilities for defining, managing, and protecting data. With implementations available on all sizes and makes of computing equipment, the SQL standard is leading the way toward unprecedented portability of database applications. The emerging SQL3 specification includes object management capabilities over abstract, user-defined data types, thereby making SQL3 a complete language for creating, managing, and querying persistent objects. The emerging RDA standard promises to complete the link among SQL products from different vendors, leading to true open systems interconnection and interoperability among conforming SQL systems. Emerging specifications for future revisions of SQL and RDA promise enhanced facilities to support intelligent objects and knowledge-based applications in a distributed processing environment.

CALS requirements for integration of SQL and non-SQL data repositories can be met with the new user-defined data types in SQL and emphasis on a new common external repository interface (ERI) linking SQL to non-SQL data managers. An SQL/ERI standard, based on an appropriate subset of SQL and RDA capabilities, will provide new opportunities for integration of heterogeneous data repositories into CALS applications. Non-SQL data repositories would be able to use the object-oriented ADT definition facilities in SQL to present a standardized, yet functionally complete, external schema to CALS applications. With support from full-function SQL processors on one side of the ERI interface, and standardized access to data and data operations on the other side, applications can take full advantage of high-level data structures and operations provided by specialized, non-SQL processors while at the same time depend on the availability of full-function SQL statements to access and manage the data.

Armed with full-function SQL and RDA implementations at each remote site, and ERI access to specialized tools and data repositories, CALS applications will be able to specify, via an SQL statement, what data is to be analyzed, and will be able to direct that data to a chosen application tool, analyze the data through the eyes of that tool (e.g., sophisticated design analysis tools), and specify where the result should be directed for further access by other tools. The interoperability capabilities provided by SQL and RDA allow integration of data and applications from various processing sites. With these data management standards, and with other capabilities provided by emerging STEP and PDES specifications, the "seamless" interoperability goals of IWSDB are within reach.

Bibliography

1. Database Language SQL3, Jim Melton, Editor, document ISO/IEC JTC1/SC21 N6931, June/July 1992.

2. J.M. Kerridge, "A proposal to add User Defined Datatypes to SQL," document X3H2-89-319 or ISO DBL FIR-37, September 1989.

3. Jim Melton, Jonathan Bauer, Krishna Kulkarni, "Object ADTs (with improvements for value ADTs)," document X3H2-91-83 or ISO DBL ARL-29, 30 March 1991.

4. K. Kulkarni, et al, "Inheritance for ADT's," document X3H2-91-133 or ISO DBL KAW-6, June 1991.

5. J. Bauer, et al, "Polymorphic Functions," document X3H2-91-135 or ISO DBL KAW-7, June 1991.

6. David Beech and Hasan Rizvi, "Tables and subtables as sets of objects," document X3H2-91-142rev1 or ISO DBL KAW-8, June 1991.

7. A. Eisenberg and B. Johnston, "Additional control statements for SQL," document X3H2-91-179 or ISO DBL KAW-14, June 1991.

8. Phil Shaw, "External functions and ADTs," document X3H2-91-172 or ISO DBL KAW-10, June 1991.

9. ISO/IEC CD 11404, "Common Language-Independent Datatypes (CLID)," Working Draft #5, document JTC1/SC22/WG11 N233, May 1991. First CD Ballot closes 30 August 1991.

10. ANSI, "American National Standard for representation of geographic point locations for information interchange," ANSI X3.61-1986, approved 23 June 1986.

11. L. Gallagher, "Computer implementation of a discrete set algebra," NISTIR 4637, July 1991.

12. Kenneth Hoffman and Ray Kunze, Linear Algebra 2nd Ed, Prentice-Hall, 1971.

13. Andre Lichnerowicz, Linear Algebra and Analysis, Holden-Day, 1967.

14. ISO/IEC, "Support for full text manipulation in structured data," JTC1/SC21/WG3 Question #3, documents SC21 N5141 and N5467.

15. ATA, "CD-ROM interchangeability standard - Advanced Retrieval Standard - Structured Full-Text Query Language (SFQL)," document ATA 89-9C.SFQL2, version 2, revision 2, July 19, 1991, Air Transport Association, Washington DC.

16. Kohji Shibano, "Outline of a document query language (DQL)," document SC21/WG3 N1200 or DBL ARL-83, May 1991.

17. Leon Cooper and David Steinberg, Methods and applications of linear programming, W.B. Saunders Co., 1974.

18. Tony Bove and Cheryl Rhodes, "A standards primer for new media," Special Report in NewMedia Age, Vol 1 No 2, pages 34-38, April 1991.

19. AIFF, Audio Interchange File Format, A standard for sampled sound files, version 1.2, Apple Computer, Inc., June 17, 1988.

20. AES, Recommended practice for digital audio engineering -Serial transmission format for two-channel linearly represented digital audio data, AES3-199x, draft revision of AES3-1985 (ANSI S4.40-1985), Audio Engineering Society, 60 East 42nd St, New York, NY 10165.

21. AES, Recommended practice for digital audio engineering -Serial multichannel audio digital interface (MADI), AES10-1991, Audio Engineering Society, 60 East 42nd St, New York, NY 10165.

22. AES, Recommended practice for digital audio engineering -Synchronization of digital audio equipment in studio operations, AES11-1991, Audio Engineering Society, 60 East 42nd St, New York, NY 10165.

23. MIDI, Musical Instrument Digital Interface, Specification 1.0, the International MIDI Association.

24. ISO/IEC, "Computer Graphics - Image Processing and Interchange," ISO/IEC JTC1/SC24 WG draft documents IM-100 (Issues), IM-101 (Architecture), IM-102 (Application Program Interface), IM-103 (Interchange Format), 1991.

25. J. Bauer, et al, "Parameterized types," document X3H2-91-274 and ISO DBL KAW-41, October 1991.

26. J. Bauer, et al, "Distinct (derived) types," document X3H2-91-280 and ISO DBL KAW-42, October 1991.

27. Apple Computer, Inside Macintosh - Volume V, Addison-Wesley, 1986.

28. D. Jefferson and C. Furlani, "Use of the IRDS standard in CALS," NIST report to CALS, April 3, 1989.

29. L. Gallagher, J. Sullivan, J. Collica, "Use of the SQL and RDA standards in CALS," NIST report to CALS, July 31, 1989.

30. L. Gallagher, "SQL3 support for CALS applications," NISTIR 4494, NIST report to CALS, December 1990.

31. Tim Boland, Editor, "Working implementation agreements for OSI protocols," GOSIP 3 reference document, NIST OSI implementor's workshop, NIST and IEEE Computer Society, December 9, 1991.

Appendix A -- Database Language SQL

Existing Specifications:	ISO 9075:1989
	ANSI X3.135-1989 ANSI X3.168-1989
	FIPS 127-1 February 2, 1990
Emerging Specifications:	ISO/IEC DIS 9075:199x, SQL Revision ANSI dpANS X3.135-199x, SQL Revision (Expected late 1992)
Sponsoring Organizations:	ISO/IEC JTC1, ANSI X3, and NIST

Description

Database Language SQL specifies data definition, data manipulation, integrity checking, and other associated facilities of the relational data model. In addition, SQL specifies components that support access control, programming language interface, and data administration. The basic structure of the relational model is a table, consisting of rows and columns. Data definition includes declaring the name of each table to be included in a database, the names and data types of all columns of each table, constraints on the values in and among columns, and the granting of table manipulation privileges to prospective users. Tables can be accessed by inserting new rows, deleting or updating existing rows, or selecting rows that satisfy a given search condition for output. Tables can be manipulated to produce new tables by cartesian products, unions, intersections, joins on matching columns, or projections on given columns.

SQL data manipulation operations may be invoked through a cursor or through a general query specification. The language includes all arithmetic operations, predicates for comparison and string matching, universal and existential quantifiers, summary operations for max/min or count/sum, and GROUP BY and HAVING clause to partition tables by groups. Transaction management is achieved through COMMIT and ROLLBACK statements.

The standard provides language facilities for defining application specific views of the data. Each view is the specification of database operations that would produce a desired table. The viewed table is then materialized at application execution time.

The SQL standard provides a Module Language for interface from other languages. Each SQL statement may be packaged as a procedure that can be called and have parameters passed to it from an external language. A cursor mechanism provides row-at-a-time access from third generation programming languages since they can only handle one row of a table at one time.

Access control is provided by GRANT and REVOKE statements. Each prospective user must be explicitly granted the privilege to access a specific table or view using a specific statement.

The optional SQL Integrity Enhancement facility offers additional tools for referential integrity, CHECK constraint clauses, and DEFAULT clauses. Referential integrity allows specification of primary and foreign keys with the requirement that no foreign key row may be inserted or updated unless a matching primary key row exists. Check clauses allow specification of inter-column constraints to be maintained by the database system. Default clauses provide optional default values for missing data.

An Embedded SQL specification (ANSI X3.168) provides the SQL interface to programming languages, specifically Ada, C, COBOL, FORTRAN, Pascal, and PL/I, by specifying the effect of syntactically embedding SQL statements into otherwise conforming application programs. Applications may thereby integrate program control structures with SQL data manipulation capabilities. The Embedded SQL syntax is just a shorthand for an explicit SQL Module accessed from a standard conforming programming language.

Applicability

The purpose of a database language standard is to provide portability and interoperability of database definitions and database application programs among conforming implementations. Database Language SQL is appropriate for all database applications where data will be shared with other applications, where the life of the application is longer than the life of current equipment, or where the application is to be understood and maintained by programmers other than the original ones. The SQL standard is particularly appropriate for database applications requiring flexibility in the data structures and access paths of the database. It is desirable both for applications under production control and when there is a substantial need for ad hoc data manipulation.

Use of the SQL standard is appropriate in all cases where there is to be some interchange of database information between systems. The schema definition language may be used to interchange database definitions and application specific views. The data manipulation language provides the data operations that make it possible to interchange complete application programs. Used with the RDA remote database access standard or with a generic data interchange standard such as ASN.1, data occurrences may also be transferred in a standard manner.

Level of consensus

The ANSI, ISO, and FIPS specifications are essentially identical. ISO has not yet formally adopted Embedded SQL (ANSI X3.168) but it is forthcoming in the next ISO version (DIS 9075:199x). FIPS requires ANSI Level 2 conformance, with programming language bindings to one or more FIPS programming languages, and requires a FIPS Flagger to flag extensions. SQL has been adopted as the database management component of X/Open, OSF, SQL Access, and other vendor consortia.

Product availability

Numerous implementations of the existing SQL standard exist on all levels and brands of equipment. Vendors are vigorously implementing additional features from the proposed 1992 revision. An SQL validated processor list is available from NIST (see Conformance testing).

Completeness

The existing SQL standard specifies data definition, data manipulation, access control, and limited integrity constraints. The emerging, upward compatible revision expected in 1992 will include substantial additional features for schema manipulation, dynamic SQL, exception handling, enhanced integrity constraints, transaction management, and data administration.

Maturity

The SQL data model is based on the relational model first published in 1969. The first commercial systems were available in 1979 and the first SQL standard was published in 1986. The basic elements of the language are very mature and will not change.

Stability

The SQL language has firm mathematical foundations in the first order predicate calculus. Standards groups and vendors are firmly committed to upward compatibility in revisions and future extensions. Existing features are expected to remain stable for the foreseeable future.

De facto usage

The SQL language is the de facto industry standard for interface to relational database management systems. Implementations are ubiquitous and are available on all sizes and types of information processing equipment. The existing standard has been used successfully in many procurements.

Limitations

The existing standard is specified for standalone, single environment databases. Specifications for access to remote heterogeneous sites are under development in an emerging ISO Remote Database Access (RDA) specification (see Appendix B). Specifications for distributed database management are in the very early stages of development. Tools for the support of object-oriented data management such as triggers, assertions, user defined datatypes, domain and table hierarchies, and stored procedures are under active consideration as follow-on enhancements to the SQL standard.

Conformance testing

A formal SQL test service was instituted by NIST in April, 1990, and Version 3.0 of the NIST SQL test suite has been publicly available since December, 1991. The SQL test suite measures conformance to both required and optional features of FIPS 127-1. A NIST Validated Processor List (VPL) identifies SQL implementations that have undergone "witnessed" evaluations using the NIST SQL test suite (see Appendix C).

Future plans

An emerging SQL2 specification, with features identified in the Completeness paragraph above, is expected to be adopted by ANSI and ISO in late 1992. Further enhancements, including those identified in the Limitations paragraph above, are expected in the 1995 time frame. The following two paragraphs give a more detailed description of emerging SQL2 and SQL3 specifications.

Emerging SQL2 Specification

A substantial upward compatible enhancement of the existing SQL standard, 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 has been completed by the relevant technical committees; only formal approval by ANSI and ISO remains before publication. SQL2 is intended to be a superset of SQL-1989 that replaces the existing SQL standard. Review copies of SQL-199x are available from GLOBAL Engineering (reference dpANS X3.135-199x, April 1991) or from OMNICOM (reference ISO/IEC DIS 9075:199x, SC21 N5739, April 1991).

SQL2 significantly increases the size of the SQL language to include a schema manipulation language for modifying or altering schemas, schema information tables to make schema definitions accessable to users, new facilities for dynamic creation of SQL statements, and new data types and domains. Other new SQL2 features include outer join, cascade update and delete referential actions, set algebra on tables, transaction consistency levels, scrolled cursors, deferred constraint checking, and greatly expanded exception reporting. SQL2 also removes a number of restrictions in order to make the language more flexible and orthogonal.

SQL2 was registered as an ISO/IEC Draft International Standard (DIS) and as an ANSI dpANS in early 1991 and is currently undergoing an ISO/IEC national body ballot for final adoption as an international standard. Parallel processing to adopt the identical specification as an American National Standard is taking place in the United States. Formal ANSI and ISO adoption is expected by late-1992 and Federal adoption as a revised FIPS PUB 127 is expected early in calendar year 1993.

Emerging SQL3 Specification

A second substantial SQL enhancement, 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. It includes generalization and specialization hierarchies, multiple inheritance, abstract data types and methods, object identifiers, polymorphism, 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. Proposals are under consideration that would make SQL a computationally complete language with stored procedures, looping and branching control structures, and Ada-like exception handling. These features are preliminary and subject to significant modification and improvement before final adoption. An early draft of these specifications [1] should be available from OMNICOM after July 1992 as a Working Draft SQL Revision.

Alternative specifications: None.

Appendix B -- Remote Database Access (RDA)

Existing Specifications:	None.	
Emerging Specifications:	ISO/IEC CD 9579-Part1 (Expected late 1992)	Generic RDA
	ISO/IEC CD 9579-Part2 (Expected late 1992)	SQL Specialization
Sponsoring Organization:	ISO/IEC JTC1/SC21	

Description

Remote Database Access (RDA) provides standard protocols for establishing a remote connection between a database client and a database server. This emerging RDA standard will standardize 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 specified in two parts, a Generic RDA for arbitrary database connection and an SQL Specialization for connecting databases conforming to Database Language SQL. Both parts were registered as ISO/IEC Draft International Standard (DIS) in 1991 and are currently undergoing ISO/IEC national body ballots for final adoption as international standards. Formal ISO adoption is expected in late 1992; Federal adoption as part of GOSIP 3 will likely follow in early 1993.

Generic RDA

The proposed Generic RDA standard 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 Communications 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.

SQL Specialization

The RDA/SQL Specialization 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 send 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 as part of the SQL Specialization standard.

Applicability

Used to establish a remote connection between a database client, acting on behalf of an application program, and a database server, interfacing to a process that controls data transfers to and from a database. The goal is to promote interconnection of database applications among heterogeneous environments, with emphasis on an SQL server interface. 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.

Level of consensus

The ISO/IEC RDA specification is currently undergoing Draft International Standard (DIS) Ballot in JTC1/SC21. The specification is in two parts: Part 1 - Generic RDA and Part 2 -SQL Specialization. Review copies should be available from OMNICOM as ISO/IEC DIS 9579-1:199x (SC21 N6375, August 1991) and DIS 9579-2:199x (SC21 N6376, August 1991). RDA has been adopted as the remote database access component of SQL Access, X/Open, and other vendor consortia. RDA is also a working task group of the NIST OSI Implementor's Workshop and is expected to become part of GOSIP 3 early in 1993.

Product availability

There are no known existing commercial RDA implementations, but many SQL vendors are planning to have conforming client and server products available before final adoption as a standard in the 1992 time frame. Vendor consortia, such as SQL Access and X/Open, have working prototypes to demonstrate interoperability among different SQL servers. A public demonstration by the SQL Access Group to show interoperability among multiple SQL implementations took place during July 1991 in New York City.

Completeness

RDA services consist of association control, data transfer, and transaction management between a single client and a single server. Association control includes making a connection to a specific database at the server site. SQL statements are sent as character strings with a separate list of input parameters, and resulting data or exception conditions are returned. Transaction management includes capabilities for both one-phase (Basic Context) and two-phase commit (TP Context) protocols, but only the Basic Context is required to claim conformance to the standard. The existing specification does not consider multiple simultaneous connections, so distributed database management is the concern of the client process. Extensions for true distributed database management among different SQL implementations are under consideration.

Maturity

Methods for establishing communications links between client and server sites are well known, but agreements on non-proprietary communications protocols are very new. There is every reason to believe that the RDA protocols will be successful.

Stability

The client/server architecture is just one of several architectures used for implementing distributed systems and there is no final conclusion as to which is best. The stability of RDA depends upon the stability of the client/server architecture. At the present time the client/server architecture seems to be the architecture of choice for heterogeneous open systems interconnection.

De facto usage

Since there are no existing implementations, procurements have not yet used the RDA specification as a required component. Until delivery dates after mid-1993, proposals will likely be based on the Draft International Standard, ISO DIS 9579 Parts 1 & 2, and emerging RDA implementations provided by individual vendors or consortia.

Limitations

A common mistake is to assume that RDA is a distributed database specification. Although distributed extensions are under consideration, existing RDA is only a two-way service and protocol specification. In addition, although a two-phase (TP-Context) commitment protocol is specified as an option, it is not required in order to claim conformance.

Conformance testing

RDA will likely become a future part of conformance testing for GOSIP, since the specification is scheduled to become part of GOSIP 3 in early 1993. A the present time, RDA can be tested indirectly using the NIST SQL test suite, with application programs at the client site and data at the server site.

Future plans

The existing RDA/SQL Specialization is aligned with the 1989 SQL standard. Enhancement projects to extend RDA to align with the emerging 1992 SQL standard have already been approved by ISO JTC1. Since the existing SQL Specialization only handles the SQL data types defined in SQL-1989, these follow-on specifications will extend the data types handled to all of those specified in the forthcoming SQL2 specification; i.e., fixed and variable length character strings, fixed and variable length bit strings, fixed and floating point numerics, dates, times, timestamps, and intervals.
Later RDA follow-on specifications will provide interchange mechanisms for the user defined abstract data types (ADTs) specified in the emerging SQL3 standard. In addition, enhancement projects for distributed database and stored database procedures have already been proposed to ISO.

RDA protocols do not by themselves provide interchange mechanisms for other application-specific data objects, so standards for IGES, PDES/STEP, CGM, SGML, EDI, ODA/ODIF, CCITT, etc. will remain critical for transmitting agreed object definitions among various sites.

Alternative specifications

Vendor agreements reached by SQL Access, X/Open, and other vendor consortia are finding their way into the RDA standard, but some agreements will always precede the formal standard.

Appendix C -- NIST validation testing for SQL and RDA

CSL assists federal agencies in procuring conforming products by providing various test suites and test services. Each FIPS has a different history in the development of the test suite(s), the availability of testing services, the policies for certification, the treatment of nonconformities, and the reporting of test results.

In the area of data management standards, the Software Standard Validation Group at CSL provides testing for Database Language SQL as well as the programming languages COBOL, FORTRAN, Ada, and Pascal. Test suites for programming language C and the IRDS are currently under development by CSL.

The testing programs for the programming languages COBOL, FORTRAN, Ada, and Pascal have been in place for some time. Test results are published in the quarterly Validated Processor List (formerly the Validated Compiler List). Certificates of validation are issued for tested products which meet the published criteria. For Ada, certificates are issued only for products without deficiencies; whereas for the other programming languages, certificates are awarded even for products "with nonconformities," which is duly noted on the certificate.

The compiler testing program is very successful. Vendors routinely schedule their compilers for annual testing to maintain their certification. Federal agencies, as well as private industry and the international community, buy "off the list."

The testing program for SQL began in April 1990 and is in a "Trial Use Period" until January 1993. During this period, testing procedures and policies are under review and may change. Also, during this period, the list of tested products is growing. With each release of the Validated Processor List, additional SQL implementations are shown conforming to the standard. This puts pressure on the remaining SQL implementations to be tested in order to compete for federal procurements.

As a testing laboratory, CSL will validate both production software and pre-release software. The pre-release software, which may be offered for procurements when it attains production status, is tested early so that SQL vendors can comply with RFP requirements for testing well in advance of delivery. It is also helpful for federal agencies planning an RFP to know about product availability in order to formulate a strategy for getting conforming SQL implementations as early as possible at off-the-shelf prices.

C.1 SQL Testing

A suite of automated validation tests for SQL implementations was developed in-house by CSL and is currently available from CSL. The original Version 1.1 of the NIST SQL Test Suite was offered in August 1988. Version 1.2 was released in May 1989, followed by Version 2.0 in December 1989. Version 3.0 was released in January 1992 and will be used throughout all of 1992. Each release increases the test suite's coverage of the standard.

Version 2.0 contains eight programming language test suite types: embedded (preprocessor) COBOL, embedded FORTRAN, embedded C, embedded Pascal, module language COBOL, module language FORTRAN, module language C, and module language Pascal. An Interactive SQL test suite is also included. For each test suite type there is an optional testing module which evaluates conformance to the Integrity Enhancement Feature. Version 3.0 contains additional test suites for embedded Ada and module language Ada.

Over 50 organizations hold licenses for Version 2.0 of the test suite. SQL implementors purchase a copy of the NIST SQL Test Suite to assist in evaluating the conformance of their products to FIPS 127-1. After in-house testing, and hopefully after correcting deficiencies, SQL implementors may request CSL to perform a formal on-site validation.

Typically, an SQL implementation will be tested in all the programming language interfaces which are supported. If the Integrity Enhancement Feature is supported, the optional testing modules for that feature will be executed. The decision of which testing modules to execute is based on the vendor's claim of conformance. The vendor completes a "Supplier's Declaration of Conformance" to identify the SQL interfaces to be tested and the hardware/software environment in which the testing takes place. This information will be published in the Validated Processor List.

The vendor may also complete a "Derived Validation" declaration to identify other hardware/software environments where the vendor certifies that identical test results will be obtained. This claim must be based on testing (actual execution of the test suite or on other testing) and is restricted to processors with binary level compatibility. This claim, if accepted, will be published in the Validated Processor List under the category of "other environments."

The decision to validate is often a marketing decision; although the technical work to conform may have begun years before. Since it requires considerable resources for a requestor to prepare for a formal validation, it is usually not performed until required by a Request for Proposal (RFP) or until the advertising benefits of a validation can be justified. Although it is possible for an SQL product to conform but not to have been formally tested, formal testing is the user's best assurance of a vendor commitment and conformance to the required standards.

Certificates for SQL conformance are not offered during the Trial Use Period. Instead, a Registered Validation Summary Report is produced to document the results of the testing. At the end of the trial use period, criteria for certification will be published and certificates will be offered. In the interim, federal agencies are advised (1) to require a Registered Validation Summary Report, (2) to review the reported deficiencies for impact on current programs, and (3) to specify timeframes for correction of the deficiencies.

Given that there is high interest in conformance among SQL vendors, most of the major SQL products have been evaluated in-house. It is reasonable to assume that vendors know how well their products perform on the NIST SQL Test Suite, even though the products have not been validated by CSL. In-house test results may be shown by vendors to customers requiring a demonstration of conformance to FIPS 127-1. These results are valid, providing that testing has been conducted according to approved procedures and that the reporting does not contain any misrepresentations.

Clearly, there are benefits to customers in validation by a third party laboratory trained in the specific test method to be used. However, formal on-site validation is expensive and time consuming. It is simply not feasible for CSL to validate every release of every DBMS product on every hardware/software platform offered for sale to the federal government. For SQL testing, CSL is investigating whether some combination of periodic CSL validation and interim vendor self testing might be the most effective way to assist federal agencies in buying conforming products.

C.2 RDA Testing

Testing for RDA is premature, since RDA is only a Draft International Standard. It is expected that identical ANSI and ISO standards for RDA will be published in late 1992, so a federal standard could follow soon thereafter. If NIST is convinced that RDA products could be available by mid-1993, then the GOSIP 3 specification will be expanded to include RDA.

Most likely, GOSIP 3 will contain specifications for the "RDA Basic Application Context" and will defer "RDA Transaction Processing Application Context" for GOSIP 4. The former consists of RDA and ACSE. (One-phase commit transaction management is provided by the RDA Service.) The latter consists of RDA, TP, ACSE, and optionally CCR. (When required, two-phase commit transaction management is provided by the TP service.)

There is considerable interest in RDA testing in the private sector. One vendor consortium, SQL Access Group, seeking to promote rapid standardization of RDA has already demonstrated limited interoperability of heterogeneous database management systems. This demonstration used prototype software based on the RDA specification. It is probable that this vendor consortium will pool resources to develop an RDA test suite.

RDA testing will be different from SQL testing. It may be desirable to have a laboratory with a "standard" server to assist in testing the RDA Client Interface of client software and a "standard" client to assist in testing the RDA Server Interface of server software. In order to test the SQL Specialization of RDA, another test module for SQL must be created to validate those features of the SQL2 standard which are required for RDA but are not included in the SQL-1989 standard and hence are not tested in the current NIST SQL Test Suite.

It is not yet clear whether RDA testing should fall under the scope of GOSIP testing and operate under NVLAP accreditation procedures.

Appendix D -- Organizations and Standards Groups Referenced

ISO	International Organization for Standardization, recently entered into cooperative agreement with IEC via Joint Technical Committee 1 (JTC1) to jointly publish information technology standards. All ISO publications are available through ANSI. See OMNICOM for availability of early drafts.
IEC	International Electrotechnical Commission, recently entered into cooperative agreement with ISO via Joint Technical Committee 1 (JTC1) to jointly publish information technology standards.
ANSI	American National Standards Institute, 1430 Broadway, New York, NY 10018, publishes all American National Standards, Sales phone: 212-642-4900, International: 212-642-4986.
X3	An ANSI accredited committee for the development of Information Processing Standards in the United States. X3 standards are published through ANSI. See GLOBAL for availability of early drafts.
NIST	National Institute of Standards and Technology, develops federal information processing standards and guidelines (FIPS) for the United States government for sale through NTIS or GPO.
NTIS	National Technical Information Service, Springfield, VA 22161, publishes all FIPS documents, Sales phone: 703-487-4650.
GPO	Government Printing Office, Washington, DC 20402, publishes FIPS and NIST SP500 series documents, Sales phone: 202-783-3238.
GLOBAL	Global Engineering, Inc. A commercial firm that has a special relationship with X3 to provide copies of draft proposed ANSI/X3 standards before formal publication, phone: 800-854-7179.
OMNICOM	OMNICOM, Inc. A commercial firm that has a special relationship with ISO/IEC JTC1 to provide copies of draft proposed International Standards before formal publication, phone: 800-666-4266.

NIST-114A (REV. 3-90) U.S. DEPARTMENT OF COMMERCE NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY **BIBLIOGRAPHIC DATA SHEET**	1. PUBLICATION OR REPORT NUMBER NISTIR 4902
	2. PERFORMING ORGANIZATION REPORT NUMBER
	3. PUBLICATION DATE SEPTEMBER 1992

4. TITLE AND SUBTITLE

Database Language SQL: Integrator of CALS Data Repositories

5. AUTHOR(S)

Leonard Gallagher and Joan Sullivan

6. PERFORMING ORGANIZATION (IF JOINT OR OTHER THAN NIST, SEE INSTRUCTIONS) U.S. DEPARTMENT OF COMMERCE NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY GAITHERSBURG, MD 20899	7. CONTRACT/GRANT NUMBER
	8. TYPE OF REPORT AND PERIOD COVERED

9. SPONSORING ORGANIZATION NAME AND COMPLETE ADDRESS (STREET, CITY, STATE, ZIP)

OASD P&L/PR/CALS
Dept. of Defense
Washington, DC 20301-8000

10. SUPPLEMENTARY NOTES

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

Computer-aided Acquisition and Logistic Support (CALS) requires a logically integrated database of diverse data, (e.g., documents, graphics, alphanumeric records, complex objects, images, voice, video) stored in geographically separated data banks under the management and control of heterogeneous data management systems. An over-riding requirement is that these various data managers be able to communicate with each other and provide shared access to data and data operations and methods under appropriate security, integrity, and access control mechanisms. Previous reports to CALS have identified the importance of Database Language SQL and its distributed processing counterpart, Remote Database Access (RDA), for their ability to address a significant portion of CALS data management requirements. This report presents the new "Object SQL" facilities proposed for inclusion in SQL3, introduces SQL abstract data types (ADTs), discusses the benefits of "generic ADT packages" for management of application specific objects, and proposes a new external repository interface (ERI) that would allow integration of heterogenous, non-SQL data repositories. The appendices give the current status and applicability of ANSI, ISO, and FIPS standards for SQL and RDA, discuss the availability of conforming implementations, and present the status of NIST SQL and RDA validation testing.

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

ANSI; ASN.1; CALS; CLID; conformance; database; database; data model; FIPS; GIS; GOSIP; ISO; IWSDB; MILSTD; object; object-oriented; PDES; RDA; relational; standard; SQL; STEP; testing

13. AVAILABILITY		14. NUMBER OF PRINTED PAGES
X	UNLIMITED	69
	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.	15. PRICE
X	ORDER FROM NATIONAL TECHNICAL INFORMATION SERVICE (NTIS), SPRINGFIELD, VA 22161.	A04

ELECTRONIC FORM