SQL:1999, formerly known as SQL3

Andrew Eisenberg
Sybase, Concord, MA 01742
andrew.eisenberg@sybase.com

Jim Melton
Sandy, UT 84093
jim.melton@acm.org

Background

For several years now, you've been hearing and reading about an emerging standard that everybody has been calling SQL3. Intended as a major enhancement of the current second generation SQL standard, commonly called SQL-92 because of the year it was published, SQL3 was originally planned to be issued in about 1996…but things didn't go as planned.

As you may be aware, SQL3 has been characterized as "object-oriented SQL" and is the foundation for several object-relational database management systems (including Oracle's ORACLE8, Informix' Universal Server, IBM's DB2 Universal Database, and Cloudscape's Cloudscape, among others). This is widely viewed as a "good thing", but it has had a downside, too: it took nearly 7 years to develop, instead of the planned 3 or 4.

As we shall show, SQL:1999 is much more than merely SQL-92 plus object technology. It involves additional features that we consider to fall into SQL's relational heritage, as well as a total restructuring of the standards documents themselves with an eye towards more effective standards progression in the future.

Standards Development Process

The two *de jure* organizations actively involved in SQL standardization, and therefore in the development of SQL:1999, are ANSI and ISO.

More specifically, the international community works through ISO/IEC JTC1 (Joint Technical Committee 1), a committee formed by the International Organization for Standardization in conjunction with the International Electrotechnical Commission. JTC1's responsibility is to develop and maintain standards related to Information Technology. Within JTC1, Subcommittee SC32, titled Data Management and Interchange, was recently formed to take over standardization of several standards related to database and metadata that had been developed by other organizations (such as the now-disbanded SC21). SC32, in turn, formed a number of Working Groups to actually do the technical work—WG3 (Database Languages) is responsible for the SQL standard, while WG4 is progressing the SQL/MM (SQL MultiMedia, a suite of standards that specify type libraries using SQL's object-oriented facilities).

In the United States, IT standards are handled by the American National Standards Institute's Accredited Standards Development Committee NCITS (National Committee for Information Technology Standardization, formerly known more simply as "X3"). NCITS Technical Committee H2 (formerly "X3H2") is responsible for several data management-related standards, including SQL and SQL/MM.

When the first generation of SQL was developed (SQL-86 and its minor enhancement SQL-89), much—perhaps most—of the development work was done in the USA by X3H2 and other nations participated largely in the mode of reviewing and critiquing the work ANSI proposed. By the time SQL-89 was published, the international community was becoming very active in writing proposals for the specification that became SQL-92; that has not changed while SQL:1999 was being developed, nor do we believe it's likely to change in the future—SQL is truly an international collaborative effort.

A work of explanation is in order about the informal names we're using for various editions of the SQL standard. The first versions of the standard are widely known as SQL-86 (or SQL-87, since the ISO version wasn't published until early 1987), SQL-89, and SQL-92, while the version just now being finalized should become known as SQL:1999. Why the difference…why not "SQL-99"? Well, simply because we have to start thinking about what the *next* generation will be called, and "SQL-02" seemed more likely to be confused with "SQL2" (which was the project name under which SQL-92 was developed)…not to mention the fact that "02" isn't really greater than "99". In other words, we don't want even the *name* of the SQL standard to suffer from the year 2000 problem!

Contents of SQL:1999

With that background under our belts, it's time to take a survey of the actual contents of SQL:1999.

While we recognize that most readers of this column will not know the precise contents of even SQL-92, space limitations prohibit our presenting the complete feature set of SQL:1999. Consequently, we're going to restrict our overview to just those features that are new to this most recent generation of the SQL standard.

The features of SQL:1999 can be crudely partitioned into its "relational features" and its "object-oriented features". We'll cover them in that sequence for convenience.

Relational Features

Although we call this category of features "relational", you'll quickly recognize that it's more appropriately categorized as "features that relate to SQL's traditional role and data model"…a somewhat less pithy phrase. The features here are not strictly limited to the relational model, but are also unrelated to object orientation.

These features are often divided into about five groups: new data types, new predicates, enhanced semantics, additional security, and active database. We'll look at each in turn.

New Data Types

SQL:1999 has four new data types (although one of them has some identifiable variants). The first of these types is the LARGE OBJECT, or LOB, type. This is the type with variants: CHARACTER LARGE OBJECT (CLOB) and BINARY LARGE OBJECT (BLOB). CLOBs behave a lot like character strings, but have restrictions that disallow their use in PRIMARY KEYs or UNIQUE predicates, in FOREIGN KEYs, and in comparisons other than purely equality or inequality tests. BLOBs have similar restrictions. (By implication, LOBs cannot be used in GROUP BY or ORDER BY clauses, either.) Applications would typically not transfer entire LOB values to and from the database (after initial storage, that is), but would manipulate LOB values through a special client-side type called a *LOB locator*. In SQL:1999, a locator is a unique binary value that acts as a sort of surrogate for a value held within the database; locators can be used in operations (such as SUBSTRING) without the overhead of transferring an entire LOB value across the client-server interface.

Another new data type is the BOOLEAN type, which allows SQL to directly record truth values *true*, *false*, and *unknown*. Complex combinations of predicates can also be expressed in ways that are somewhat more user-friendly than the usual sort of restructuring might make them:

```
WHERE COL1 > COL2 AND
      COL3 = COL4 OR
      UNIQUE(COL6) IS NOT FALSE
```

SQL:1999 also has two new *composite* types: ARRAY and ROW. The ARRAY type allows one to store collections of values directly in a column of a database table. For example:

```
WEEKDAYS VARCHAR(10) ARRAY[7]
```

would allow one to store the names of all seven weekdays directly in a single row in the database. Does this mean that SQL:1999 allows databases that do not satisfy first normal form? Indeed, it does, in the sense that it allows "repeating groups", which first normal form prohibits. (However, some have argued that SQL:1999's ARRAY type merely allows storage of information that can be decomposed, much as the SUBSTRING function can decompose character strings—and therefore doesn't truly violate the spirit of first normal form.)

The ROW type in SQL:1999 is an extension of the (anonymous) row type that SQL has always had and depended on having. It gives database designers the additional power of storing structured values in single columns of the database:

```
CREATE TABLE employee (
  EMP_ID      INTEGER,
  NAME        ROW (
    GIVEN       VARCHAR(30),
    FAMILY      VARCHAR(30) ),
  ADDRESS     ROW (
    STREET      VARCHAR(50),
    CITY        VARCHAR(30),
    STATE       CHAR(2) )
  SALARY      REAL )

SELECT E.NAME.FAMILY
FROM   employee E
```

While you might argue that this also violates first normal form, most observers recognize it as just another "decomposable" data type.

SQL:1999 adds yet another data type-related facility, called "distinct types". Recognizing that it's generally unlikely that an application would want, say to directly compare an employee's shoe size with his or her IQ, the language allows programmers to declare SHOE_SIZE and IQ to each be "based on" INTEGER, but prohibit direct mixing of those two types in expressions. Thus, an expression like:

```
WHERE MY_SHOE_SIZE > MY_IQ
```

(where the variable name implies its data type) would be recognized as a syntax error. Each of those two types may be "represented" as an INTEGER, but the SQL system doesn't allow them to be mixed in expressions—nor for either to be, say, multiplied by an INTEGER:

```
SET MY_IQ = MY_IQ * 2
```

Instead, programs have to explicitly express their deliberate intent when doing such mixing:

```
WHERE MY_SHOE_SIZE >
   CAST (MY_IQ AS SHOE_SIZE)
SET MY_IQ =
   MY_IQ * CAST(2 AS IQ)
```

In addition to these types, SQL:1999 has also introduced user-defined types, but they fall into the object-oriented feature list.

New Predicates

SQL:1999 has three new predicates, one of which we'll consider along with the object-oriented features. The other two are the SIMILAR predicate and the DISTINCT predicate.

Since the first version of the SQL standard, character string searching has been limited to very simple comparisons (like =, >, or <>) and the rather rudimentary pattern matching capabilities of the LIKE predicate:

```
WHERE NAME LIKE '%SMIT_'
```

which matches NAME values that have zero or more characters preceding the four characters 'SMIT' and exactly one character after them (such as SMITH or HAMMERSMITS).

Recognizing that applications often require more sophisticated capabilities that are still short of full text operations, SQL:1999 has introduced the SIMILAR predicate that gives programs UNIX-like regular expressions for use in pattern matching. For example:

```
   WHERE NAME SIMILAR TO
'(SQL-(86|89|92|99))|(SQL(1|2|3))'
```

(which would match the various names given to the SQL standard over the years). It's slightly unfortunate that the syntax of the regular expressions used in the SIMILAR predicate doesn't quite match the syntax of UNIX's regular expressions, but some of UNIX's characters were already in use for other purposes in SQL.

The other new predicate, DISTINCT, is very similar in operation to SQL's ordinary UNIQUE predicate; the important difference is that two null values are considered not equal to one another and would thus satisfy the UNIQUE predicate, but not all applications want that to be the case. The DISTINCT predicate considers two null values to be not distinct from one another (even though they are neither equal to nor not equal to one another) and thus those two null values would cause a DISTINCT predicate not to be satisfied.

SQL:1999's New Semantics

It's difficult to know exactly where to draw the line when talking about new semantics in SQL:1999, but we'll give a short list of what we believe to be the most important new behavioral aspects of the language.

A long-standing demand of application writers is the ability to update broader classes of views. Many environments use views heavily as a security mechanism and/or as a simplifier of an applications view of the database. However, if most views are not updatable, then those applications often have to "escape" from the view mechanism and rely on directly updating the underlying base tables; this is a most unsatisfactory situation.

SQL:1999 has significantly increased the range of views that can be updated directly, using only the facilities provided in the standard. It depends heavily on functional dependencies for determining what additional views can be updated, and how to make changes to the underlying base table data to effect those updates.

Another widely-decried shortcoming of SQL has been its inability to perform recursion for applications such as bill-of-material processing. Well, SQL:1999 has provided a facility called *recursive query* to satisfy just this sort of requirement. Writing a recursive query involves writing the query expression that you want to recurse and giving it a name, then using that name in an associated query expression:

```
WITH RECURSIVE
  Q1 AS SELECT...FROM...WHERE...,
  Q2 AS SELECT...FROM...WHERE...
SELECT...FROM Q1, Q2 WHERE...
```

We've already mentioned locators as a client-side value that can represent a LOB value stored on the server side. Locators can be used in the same way to represent ARRAY values, accepting the fact that (like LOBs) ARRAYs can often be too large to conveniently pass between an application and the database. Locators can also be used to represent user-defined type values—discussed later in this column—which also have the potential to be large and unwieldy.

Finally, SQL:1999 has added the notion of *savepoints*, widely implemented in products. A savepoint is a bit like a subtransaction in that an application can undo the actions performed after the beginning of a savepoint without undoing all of the actions of an entire transaction. SQL:1999 allows ROLLBACK TO SAVEPOINT and RELEASE SAVEPOINT, which acts a lot like committing the subtransaction.

Enhanced Security

SQL:1999's new security facility is found in its *role* capability. Privileges can be granted to roles just as they can be to individual authorization identifiers, and roles can be granted to authorization identifiers and to other roles. This nested structure can enormously simplify the often difficult job of managing security in a database environment.

Roles have been widely implemented by SQL products for several years (though occasionally under different names); the standard has finally caught up.

Active Database

SQL:1999 recognizes the notion of active database, albeit some years after implementations did. This facility is provided through a feature known as *triggers*. A trigger, as many readers know, is a facility that allows database designers to instruct the database system to perform certain operations each and every time an application performs specified operations on particular tables.

For example, triggers could be used to log all operations that change salaries in an employee table:

```
CREATE TRIGGER log_salupdate
  BEFORE UPDATE OF salary
    ON employees
  REFERENCING OLD ROW as oldrow
              NEW ROW as newrow
  FOR EACH ROW
  INSERT INTO log_table
    VALUES (CURRENT_USER,
            oldrow.salary,
            newrow.salary)
```

Triggers can be used for many purposes, not just logging. For example, you can write triggers that keep a budget balanced by reducing monies set aside for capital purchases whenever new employees are hired…and raising an exception if insufficient money is available to do so.

Object Orientation

In addition to the more traditional SQL features discussed so far, SQL:1999's development was focussed largely—some observers would say too much—on adding support for object-oriented concepts to the language.

Some of the features that fall into this category were first defined in the SQL/PSM standard published in late 1996—specifically, support for functions and procedures invocable from SQL statements. SQL:1999 enhances that capability, called SQL-invoked routines, by adding a third class of routine known as *methods*, which we'll get to shortly. We won't delve into SQL-invoked functions and procedures in this column, but refer you to an earlier issue of the SIGMOD Record [6].

Structured User-Defined Types

The most fundamental facility in SQL:1999 that supports object orientation is the structured user-defined type; the word "structured" distinguishes this feature from distinct types (which are also "user-defined" types, but are limited in SQL:1999 to being based on SQL's built-in types and thus don't have structure associated with them).

Structured types have a number of characteristics, the most important of which are:

– They may be defined to have one or more attributes, each of which can be any SQL type, including built-in types like INTEGER, collection types like ARRAY, or other structured types (nested as deeply as desired).
– All aspects of their behaviors are provided through methods, functions, and procedures.
– Their attributes are encapsulated through the use of system-generated observer and mutator functions ("get" and "set" functions) that provide the only access to their values. However, these system-generated functions cannot be overloaded; all other functions and methods can be overloaded.
– Comparisons of their values are done only through user-defined functions.
– They may participate in type hierarchies, in which more specialized types (subtypes) have all attributes of and use all routines associate with the more generalized types (supertypes), but may add new attributes and routines.

Let's look at an example of a structured type definition:

```
CREATE TYPE emp_type
  UNDER person_type
AS ( EMP_ID      INTEGER,
     SALARY      REAL )
INSTANTIABLE
NOT FINAL
REF ( EMP_ID )
INSTANCE METHOD
  GIVE_RAISE
    ( ABS_OR_PCT  BOOLEAN,
      AMOUNT      REAL )
    RETURNS REAL
```

This new type is a subtype of another structured type that might be used to describe persons in general, including such common attributes as name and address; the new `emp_type` attributes include things that "plain old persons" don't have, like an employee ID and a salary. We've declared this type to be instantiable and permitted it to have subtypes defined (`NOT FINAL`). In addition, we've said that any references to this type (see the discussion on REF types below) are derived from the employee ID value. Finally, we've defined a method (more on this later) that can be applied to instances of this type.

SQL:1999, after an extensive flirtation with multiple inheritance (in which subtypes were allowed to have more than one immediate supertype), now provides a type model closely aligned with Java's—single inheritance. Type definers are allowed to specify that a given type is either instantiable (in other words, values of that specific type can be created) or not instantiable (analogous to *abstract types* on other programming languages). And, naturally, any place—such as a column—where a value of some structured type is permitted, a value of any of its subtypes can appear; this provides exactly the sort of substitutability that object-oriented programs depend on.

By the way, some object-oriented programming languages, such as C++, allow type definers to specify the degree to which types are encapsulated: an encapsulation level of PUBLIC applied to an attribute means that any user of the type can access the attribute, PRIVATE means that no code other than that used to implement the type's methods can access the attribute, and PROTECTED means that only the type's methods and methods of any subtypes of the type can access the attribute. SQL:1999 does not have this mechanism, although attempts were made to define it; we anticipate it to be proposed for a future revision of the standard.

Functions *vs* Methods

SQL:1999 makes an important distinction between "ordinary" SQL-invoked functions and SQL-invoked methods. In brief, a method is a function with several restrictions and enhancements. Let's summarize the differences between the two types of routine:

– Methods are tightly bound to a single user-defined type; functions are not.
– The user-defined type to which a method is bound is the data type of a distinguished argument to the method (the first, undeclared argument); no argument of a function is distinguished in this sense.
– Functions may be polymorphic (overloaded), but a specific function is chosen at compile time by examining the declared types of each argument of a function invocation and choosing the "best match" among candidate functions (having the same name and number of parameters); methods may also be polymorphic, but the most specific type of their distinguished argument, determined at runtime, allows selection of the exact method to be invoked to be deferred until execution; all other arguments are resolved at compile time based on the arguments' declared types.
– Methods must be stored in the same schema in which the definition of their tightly-bound structured type is stored; functions are not limited to a specific schema.

Both functions and methods can be written in SQL (using SQL/PSM's computationally-complete statements) or in any of several more traditional programming languages, including Java.

Functional and Dot Notations

Access to the attributes of user-defined types can be done using either of two notations. In many situations, applications may seem more natural when they use "dot notation":

```
WHERE emp.salary > 10000
```

while in other situations, a functional notation may be more natural:

```
WHERE salary(emp) > 10000
```

SQL:1999 supports both notations; in fact, they are defined to be syntactic variations of the same thing—as long as "`emp`" is a storage entity (like a column or variable) whose declared type is some structured type with an attribute named "`salary`"…*or* there exists a function named "salary" with a single argument whose data type is the (appropriate) structured type of emp.

Methods are slightly less flexible than functions in this case: Only dot notation can be used for method invocations—at least for the purposes of specifying the distinguished argument. If salary were a method whose closely bound type were, say, employee, which was in turn the declared type of a column named emp, then that method could be invoked only using:

```
emp.salary
```

A different method, say give_raise, can combine dot notation and functional notation:

```
emp.give_raise(amount)
```

Objects…Finally

Careful readers will have observed that we have avoided the use of the word "object" so far in our description of structured types. That's because, in spite of certain characteristics like type hierarchies, encapsulation, and so forth, instances of SQL:1999's structured types are simply *values*, just like instances of the language's build-in types. Sure, an employee value is rather more complex (in appearance, as well as in behavior) than an instance of INTEGER, but it's still a value without any identity other than that provided by its value.

In order to gain the last little bit of characteristic that allows SQL to provide *objects*, there has to be some sense of identity that can be referenced in a variety of situations. In SQL:1999, that capability is supplied by allowing database designers to specify that certain tables are defined to be "typed tables"…that is, their column definitions are derived from the attributes of a structured type:

```
CREATE TABLE empls OF employee
```

Such tables have one column for each attribute of the underlying structured type. The functions, methods, and procedures defined to operate on instances of the type now operate on rows of the table! The rows of the table, then, are values of—or instances of—the type. Each row is given a unique identity that behaves just like a OID (object identifier) behaves…it is unique in space (that is, within the database) and time (the life of the database).

SQL:1999 provides a special type, called a REF type, whose values are those unique identifiers. A given REF type is always associated with a specified structured type. For example, if we were to define a table containing a column named "manager" whose values were references to rows in a typed table of employees, it would look something like this:

```
manager     REF(emp_type)
```

A value of a REF type either identifies a row in a typed table (of the specified structured type, of course) or it doesn't identify anything at all—which could mean that it's a "dangling reference" left over after the row that it once identified was deleted.

All REF types are "scoped" so that the table that they reference is known at compilation time. During the development of SQL:1999, there were efforts made to allow REF types to be more general than that (for example, any of several tables could be in the scope, or any table at all of the appropriate structured type would be in the scope even if the table were created after the REF type was created); however, several problems were encountered that could not be resolved without further delaying publication of the standard, so this restriction was adopted. One side effect of the restriction, possibly a beneficial effect, is that REF types now behave very much like referential integrity, possibly easing the task of implementing this facility in some products!

Using REF Types

You shouldn't be surprised to learn that REF types can be used in ways a little more sophisticated than merely storing and retrieving them.

SQL:1999 provides syntax for "following a reference" to access attributes of a structured type value:

```
SELECT emps.manager->last_name
```

The "pointer" notation (->) is applied to a value of some REF type and is then "followed" into the identified value of the associated structured type—which, of course, is really a row of the typed table that is the scope of the REF type. That structured type is *both* the type associated with the REF type of the manager column in the emps table and the type of that other table (whose name is neither required nor apparent in this expression). However, that structured type must have an attribute named last_name, and the typed table thus has a column of that name.

Schedules and Futures

SQL:1999 is not yet a standard, although it's well on its way to becoming one. Last year, what is called the Final Committee Draft (FCD) ballot was held for four parts of the SQL specifications (see references [1], [2], [4], and [5]). In November, 1998, the final round of the Editing Meeting was held for those parts. The changes to the specifications have been applied by the Editor (Jim Melton) and are now being reviewed by Editing Meeting participants. When those reviews are completed, those four specifications will be submitted for one last ballot (called a Final Draft International Standard, or FDIS, ballot), the result of which is either “approve and publish without change” or “disapprove and go back to FCD status”. All participants currently anticipate that the result will be to approve and publish, resulting in a revised standard sometime in mid-1999.

Another part of SQL, SQL/CLI (see reference [3]), is also being revised and has just undergone an FCD ballot. It is expected that it will be published later in 1999 as a revision of the CLI-95 standard.

It’s hard to know what the future will bring, but both the ANSI and ISO groups are committed to avoiding the lengthy process that resulted in SQL:1999. We all believe that 6 years is simply too long, especially with the world working in “web time” more and more. Instead, plans are being developed that will result in revisions being issued roughly every three years, even if the technical enhancements are somewhat more modest than those in SQL:1999.

In addition to evolving the principle parts of the SQL:1999 standard, additional parts of SQL are being developed to address such issues as temporal data, the relationship with Java (explored in the previous issue of the SIGMOD Record), and management of external data along with SQL data.

Recognition of Individual Contributors

There have been many, many people involved in the development of SQL:1999 over the years its development occupied. While we don’t have the space to mention everybody who participated in the committees during its development, we do think it appropriate to mention at least the names of people who wrote significant numbers of change proposals or simply wrote significant proposals.

– Mihnea Andrei (France; Sybase)
– Jon Bauer (USA; Digital and Oracle)
– David Beech (USA; Oracle)
– Ames Carlson (USA; HP and Sybase)
– Stephen Cannan (Netherlands; DCE Nederland and James Martin Consulting)
– Paul Cotton (Canada; IBM)
– Hugh Darwen (UK; IBM)
– Linda deMichael (USA; IBM)
– Judy Dillman (USA; CA)
– Rüdiger Eisele (Germany; Digital and independent consultant)
– Andrew Eisenberg (USA; Digital, Oracle, and Sybase)
– Chris Farrar (USA; Teradata and Compaq)
– Len Gallagher (USA; NIST)
– Luigi Giuri (Italy; Fondazione ugo Bordoni)
– Keith Hare (USA; JCC)
– Bruce Horowitz (USA; Bellcore)
– Bill Kelly (USA; UniSQL)
– Bill Kent (USA; HP)
– Krishna Kulkarni (USA; Tandem, Informix, and IBM)
– Nelson Mattos (USA; IBM)
– Jim Melton (USA; Digital and Sybase)
– Frank Pellow (Canada; IBM and USA; Microsoft)
– Baba Piprani (Canada; independent consultant)
– Peter Pistor (Germany; IBM)
– Mike Pizzo (USA; Microsoft)
– Jeff Richie (USA; Sybase and IBM)
– Phil Shaw (USA; IBM, Oracle, and Sybase)
– Kohji Shibano (Japan; Tokyo International University and Tokyo University of Foreign Studies)
– Masashi Tsuchida (Japan; Hitachi)
– Mike Ubell (USA; Digital, Illustra, and Informix)
– Murali Venkatrao (USA; Microsoft)
– Fred Zemke (USA; Oracle)

Each of these people contributed in some significant way. Some of them designed major aspects of the architecture of SQL:1999, others focussed on specific technologies like the call-level interface (SQL/CLI), while others worked on very focussed issues, such as security. They are all—as are other contributors not mentioned here—to be congratulated on a large job well done.

References

[1] ISO/IEC 9075:1999,*Information technology—Database languages—SQL—Part 1: Framework (SQL/Framework),* will be published in 1999.

[2] ISO/IEC 9075:1999,*Information technology—Database languages—SQL—Part 2: Foundation (SQL/Foundation),* will be published in 1999.

[3] ISO/IEC 9075:1999,*Information technology—Database languages—SQL—Part 3: Call-Level Interface (SQL/CLI),* will be published in 1999.

[4] ISO/IEC 9075:1999,*Information technology—Database languages—SQL—Part 4: Persistent Stored Modules (SQL/PSM)*, will be published in 1999.

[5] ISO/IEC 9075:1999,*Information technology—Database languages—SQL—Part 5: Host Language Bindings (SQL/Bindings),* will be published in 1999.
[6] *New Standard for Stored Procedures in SQL*, Andrew Eisenberg, SIGMOD Record, Dec.1996

The SQL specifications will be available in the United States from:

American National Standards Institute
Attn: Customer Service
11 West 42nd Street
New York, NY 10036
USA

Phone: +1.212.642.4980

It will be available internationally from the designated National Body in each country or from:

International Organization for Standardization
1, rue de Varembé
Case postale 56
CH-1211 Genève 20
Switzerland

Phone: +41.22.749.0111

Web References

American National Standards Institute (ANSI)
http://web.ansi.org

International Organization for Standardization (ISO)
http://www.iso.ch

JTC1 SC32 – Data Management and Interchange
http://bwonotes5.wdc.pnl.gov/SC32/JTC1SC32.nsf

National Committee for Information Technology Standards (NCITS)
http://www.ncits.org

NCITS H2 – Database
http://www.ncits.org/tc_home/h2.htm