An SQL3 Snapshot

Jim Melton
Sybase, Inc.

Abstract

The third major generation of the standard for the SQL database language, known as SQL3, is nearing completion. Many significant new features have been added to SQL, both in areas traditionally addressed by SQL and in pursuit of adding object technology to the language. The standard has been partitioned into a number of distinct parts, each of which may progress at its own rate. Publication of SQL3 as a replacement for the current version of the standard, SQL-92, is expected no sooner than 1998.

1. Introduction

A great many new capabilities have been added to the emerging next generation of the SQL standard (frequently known by its project name: SQL3). The best-known of the efforts to add new features is the object-oriented feature extensions to SQL, but there are many other categories of enhancements to the language.

SQL3 has been divided into a number of *parts*, some of which are expected to progress independently of the others. The parts of SQL today include:

- SQL/Framework [1] — Describes how the other parts are to be used, specifies an overall conformance policy
- SQL/Foundation [2] — Contains the "main body" of the SQL language
- SQL/CLI (Call-Level Interface) [3] — Provides a call-level interface to SQL databases
- SQL/PSM (Persistent Stored Modules) [4] — Specifies a computationally-complete language in SQL, as well as a powerful routine-invocation interface
- SQL/Bindings [5] — Details the host language bindings for SQL and the dynamic SQL interface
- SQL/Transaction [6] — Allows SQL database systems to interface with global transaction managers through the popular XA interface
- SQL/Temporal [7] — Supports time-series and other temporal data in SQL databases

Other parts of the SQL standard might be created in the future as needs arise.

2. Features not related to ADTs

While the majority of public visibility on the SQL3 progression has involved abstract data type (ADT) specification, most of the new features in SQL are of a more traditional nature. The principle categories of these new changes are:

- Data types
- Predicates
- Security
- Cursors
- Transactions
- Triggers
- Schema definition and maniuplation
- Routine invocation
- Condition handling
- Query processing

We look at each of those categories briefly.

2.1. New Data Types

Several new built-in data types have been added to SQL3: Boolean, rows, large objects and large object locators, and collection types.

The Boolean data type provides the values `TRUE` and `FALSE`, as expected.

The row data type permits applications to specify columns, variables, and so forth as *tuples* with structure; they can be arbitrarily nested.

There are two large object types: character large objects (CLOBs) and binary large objects (BLOBs). Each of them can take on (potentially) very large values and are generally manipulated in the same way that ordinary

1063-6382/96 $5.00 © 1996 IEEE

character and bit strings are. The principle difference is that LOBs are manipulated through *locators*. Entire LOB values cannot be retrieved at once into host programs, so substrings must be retrieved.

Large object locators permit the manipulation of LOBs without having to pass the actual LOB values back and forth between the application program and the database system. They act as a sort of handle to the LOB value within the database environment.

There are now three data types that provide *collections* in SQL: SET, MULTISET, and LIST. For example,

```
SET(INTEGER)
```

contains only integers without duplicates, while

```
MULTISET
  (ROW (A INTEGER, B CHARACTER(10)))
```

contains rows without prohibiting duplicates (closely resembling a table!), and

```
LIST (SET (INTEGER))
```

contains a sequence of sets.

In addition to these data types provided in the "core" of SQL3, a new part of the SQL standard has been initiated to provide support for temporal data going beyond SQL-92's datetime and interval data types (*e.g.*, time-series data). SQL/Temporal will be progressed on a schedule independent of the main body of SQL3, but is unlikely to be published as a standard earlier than that main body.

SQL3 also provides a "distinct type" capability that allows users to define a new type (say, MONEY) that is identical to an existing type (say, DECIMAL(7,2)), but that is not *comparable* to that existing type — that is, operations on the two types cannot be compared or combined in operations. Given one column, COL1, of type MONEY and a second column, COL2, of type DECIMAL(7,2), it is not possible to compare them, add them together, or average them. However, the CAST operator can convert items of type MONEY to a new value of type DECIMAL(7,2)...or to a new value of type REAL.

2.2. Predicates

A number of new predicates have been added to SQL3, including: SIMILAR, quantified, function invocation, octet LIKE, TYPE, and DISTINCT. In addition, new syntax to make predicate checking less confusing has been provided.

The SIMILAR predicate permits the application of a UNIX®-like regular expression to determine whether one character string is sufficiently similar to another, based on the criteria given.

The quantified predicate allows applications to test whether a particular search condition is true for any or all of a derived table. For example:

```
FOR ANY table-name
  (col1 < 5 AND col2 = 'Joe')
```

The function invocation predicate provides applications with the ability to invoke a Boolean-valued function and use the returned value as it would any predicate.

A new variation of the LIKE predicate that uses binary strings instead of character strings allows powerful bit matching capabilities that can be used for identifying rows in tables with specific bit patterns in specified columns.

The TYPE predicate permits applications to interrogate instances of an abstract data type to determine its most specific data type.

The DISTINCT predicate provides the ability to distinguish between values that are *equal, but distinct*, such as two character strings that differ only in the number of trailing blanks.

New syntax is provided to make complex combinations of predicates easier to write and comprehend:

```
...WHERE ((C1 = 5) AND C2 <> 'Joe'
      AND
         C3 LIKE 'F%X') IS NOT NULL
```

This sort of syntax is much easier to use in such situations than the traditional predicate syntax of SQL.

2.3. Security

In addition to the new privileges added to SQL3 to govern use of new capabilities like abstract data types and triggers, SQL3 has one significant new security feature: roles. (It must be noted that the SQL committees have not finally decided whether roles will be in SQL3 or in some future version of the standard.)

A role is a surrogate for an ordinary SQL authorization identifier. Privileges are granted to roles, which can in turn be granted to individual authorization identifiers as operational requirements evolve.

2.4. Cursors

Applications will benefit significantly from new cursor-related features in SQL3, which include: *holdable* cursors, *sensitive* cursors, and the ability to update through a cursor without having to redundantly specify the table name.

A *holdable cursor* is one that is kept open when the transaction in which it was opened is committed (unless, of course, the application has explicitly closed it). Cursor position is maintained across such commitments, supporting application requirements to frequently commit actions while processing large quantities of data.

A *sensitive cursor* is one that is guaranteed to "see" all changes made to its set of rows by other SQL state-

ments executed in the same transaction. (This is, of course, not related to the effects of other transactions, which is controlled by transaction isolation characteristics.)

Cursor declarations identify the underlying table or tables from which cursor rows are derived, and only those cursors having exactly one underlying table are updatable in SQL-92 and SQL3. Therefore, it is redundant for a statement performing updates of the rows of a cursor to specify the table being updated, though SQL-92 required it to do so. SQL3 permits this redundant information to be omitted by the application:

```
UPDATE SET SAL = SAL * 1.03
 WHERE CURRENT OF CUR
```

2.5. Transactions

Several improvements have been made to the transaction facilities in SQL3. They include an explicit START TRANSACTION statement, chained transactions, the ability to set transaction characteristics for an entire session, the ability to set transaction characteristics only for branches of a distributed transaction, and savepoints.

Until SQL3, every transaction in an SQL application was begun only when the first statement requiring the context of a transaction was executed. This had two unfortunate effects: humans reading application code were unable to easily note when transactions were started, and application designers were unable to control the time at which the program encountered the performance "hit" of transaction initiation. Giving applications the ability to control the time of transaction initiation will resolve both problems.

In a distributed environment, such as a client-server situation, it is desirable to start one transaction immediately upon terminating the preceding transaction (this can save network message traffic, for example). *Chained* transactions provide this capability for applications needing it; applications choose when terminating one transaction whether or not to immediately initiate a succeeding transaction with identical characteristics.

In SQL-92, a `SET TRANSACTION` statement allowed an application to set the characteristics of the immediately-following transaction. However, all subsequent transactions reverted to the SQL standard's default transaction characteristics. SQL3 provides a `SET SESSION CHARACTERISTICS` statement that permits an application to set the attributes of all subsequent transactions for the duration of a session.

SQL3 permits transactions that span SQL servers without requiring that they be managed by a global transaction manager ("TP Lite" some call it). It is desirable to permit some branches of such transactions to have different attributes that other branches or of the overall transaction. For example, one branch of an updating transaction might be read-only, permitting various protocol optimizations. SQL3 provides a `SET LOCAL TRANSACTION` statement to support this requirement.

Applications often require the ability to tentatively commit work they've already performed, without making a final decision regarding ultimate disposition. A fully-generalized subtransaction model would answer this requirement (and much more), but at substantial implementation cost. A "simply nested" subtransaction model satisfies this requirement at reasonable cost; in SQL3, this facility is provided by means of *savepoints*, which are "named marks" in the flow of a transaction that can be "undone" (rolled back as far as a mark) or "committed" (the mark deleted).

Yet another new part of SQL is being developed to provide SQL support for the widely-accepted XA interface between global transaction managers and resource manages like SQL. Publication of this part is likely to occur in the 1997 or 1998 timeframe.

2.6. Triggers

Triggers are active rules in databases, allowing database designers to specify "auxiliary" actions to be performed when certain SQL statements are executed by applications. Triggers have been widely implemented in SQL products for some years, but are just now being specified in SQL3.

SQL3's triggers allow a sequence of (one or more) SQL statements to be executed whenever a trigger is "fired". The triggering statements can be INSERT, UPDATE, or DELETE statements and the triggered statements can be executed either before or after the triggering statement. Furthermore, the triggered statements can be executed either once per triggering statement invocation or once per row affected by the triggering statement (the "trigger granularity").

Further enhancements are being considered for a future revision of the standard. They include: allow triggered statements to be executed *instead of* the triggering statement; fire a trigger only when certain columns of a table are updated; and support multiple triggers on the same combination of triggering statement, trigger timing, and granularity.

2.7. Schema definition and maniuplation

Quite a number of new capabilities added to SQL3 involve "database objects" or "schema objects".

SQL3 provides a new ALTER SCHEMA statement, allowing database designers to change certain attributes of their schemas. Among such schema-wide attributes are the user-defined operators described later and a "path" used to identify candidate routines to be used in routine invocations.

SQL3 provides the ability to define one table to be "just like" another table, using:

```
CREATE TABLE T1 LIKE T2
```

This facility makes it especially easy in a dynamic situation to make copies of tables, as the data from one table is easily copied to the other.

In the area of referential integrity, SQL3 provides an explicit RESTRICT option, making even temporary violations of referential integrity (during the execution of a single SQL statement) invalid. It also improves the support for implicitly identifying referenced columns, as well as providing for referencing and referenced columns to have *comparable* data types instead of identical types — *e.g.*, a column of type CHARACTER(20) can now reference a column of type CHARACTER VARYING(50).

A number of new "temporary" schema objects have been provided to supplement SQL-92's temporary tables: temporary views, temporary cursors, and temporary abstract data types. Each of these, including temporary tables, can be limited in scope to a BEGIN...END block in the SQL/PSM context.

SQL3 currently contains a feature called "subtables" that permits a table hierarchy to be defined; subtables have additional columns that their supertables do not have. This feature is partially redundant with the type hierarchy described later and may not be part of the published standard.

2.8. Routine invocation

In an SQL-92 environment, all arguments to SQL procedures were required to be atomic variables. In SQL3, enhancements permit the use of expressions as arguments to procedures and functions. This additional expressive power brings SQL into line with other modern programming languages.

In SQL3, routines (functions and procedures) are *polymorphic* — that is, many routines may share a name, with specific routines being distinguished by their parameter list. Because routines in SQL can be created in many schemas (that are, in turn, in many catalogs), it is very desirable to apply some boundaries on the scope across which routines are identified as candidate routines for a specific routine invocation. The notion of a *path* is provided in SQL3 to specify exactly those schemas that are to be used in locating candidate routines for a specific infocation. Statements to manage that path are of course provided as well. In addition to the path for a specific session, there is also a path for SQL statements in embedded SQL programs and their corresponding module language programs.

2.9. Condition handling

While SQL/PSM provides an extensive condition handling facility for use in stored modules and SQL's new computationally-complete language, there is still a need for improving the condition handling capabilities in ordinary embedded SQL.

SQL3 provides extensions to SQL-92's WHENEVER clause to permit applications to specify the behavior when a specific SQLSTATE value is encountered, when a specific constraint is violated, or when an SQLERROR is specifically an SQLWARNING or an SQLEXCEPTION.

In addition, the diagnostics area in SQL3 is enhanced to provide still more information to the application program, allowing quicker and more accurate determination of fault conditions.

2.10. Query processing

Several enhancements to SQL's query processing capabilities will appear in SQL3. Among these are the ability to update views (and other virtual tables, like cursors) under a greater variety of circumstances. Through the use of functional dependency analysis, many views that were not updatable in SQL-92 are updatable in SQL3, including some views formed with joins. (In a future version of SQL, there may be the ability to define "instead of" triggers that allow database designers to provide triggered statements to accomplish view updates that cannot be derived automatically.)

SQL3 also provides an operation called *recursive union* that directly supports recursive processing of tables. The use of the recursive union operator allows both linear (single-parent, or tree) recursion and non-linear (multi-parent, or general directed graph) recursion. This feature will allow easy solutions to bill-of-material and similar applications.

Finally, applications that find themselves repeatedly specifying subqueries within an outer query (in a single SQL statement) can now factor out those repeated subqueries, give them names, and use the names in their place in the body of the query. This facility, called "query naming", provides a sort of macro capability that reduces coding errors and improves maintenance without requiring significant costly database system implementation effort.

3. Object-Oriented Extensions

The most widely discussed category of additions in SQL3 has been those oriented towards object orientation. These include:

- Abstract data types (ADTs)
- Type hierarchies
- Multiple inheritance
- Encapsulation of implementation
- Multiple levels of encapsulation
- Polymorphic routine
- User-defined operators
- Multiple comparison capabilities

We will address each of these areas individually.

3.1. Abstract data types (ADTs)

SQL3 provides user-defined data type capabilities that allow applications to provide new, complex data types that have many of the characteristics of objects: type hierarchies, encapsulation and a functional interface, polymorphism, user-defined operators, *etc.*

Columns in tables can be defined to have as their data type any abstract data type (as well as any built-in or distinct type), as can SQL variables in SQL/PSM.

Value ADTs behave in many ways like any other value in SQL (such as integers). They correspond to what many object models call "literals" (though that term in SQL is reserved for the source text construction that represents a value). When a column of a table has a value ADT as its type, an actual complex value *state* is (effectively) stored in that column for each row. Encapsulated value ADT instances are "passed" into SQL routines in the same manner as any SQL value.

In addition to value ADTs, SQL3 provides *object* ADTs. The principle difference in SQL3 between a value ADT and an object ADT is that object ADTs have unique identities (represented by an object identifier, or OID) through which they can be referenced in routine invocations.

There is currently some controversy in SQL3 over the issue of "extents" of object ADTs.

- Some participants in the standards process believe that objects (instances of an object type) should "live" in their own extents — disjoint from SQL's tables, columns, and rows (there is additional controversy over the issue of whether these extents should be named or anonymous). They believe that all cells (column/row intersections) in tables should contain only OIDs identifying objects in some extent.
- Other participants believe that objects should be able to be placed *in* any row of a column declared to be "type X instance", while other columns, declared to be "type X reference", would contain references to the objects stored in the "instance" columns.

This remains a difficult subject under constant debate. Unfortunately, some participants in the process believe that SQL should not have any object support at all — that value ADT support is sufficient and object support is undesirable.

3.2. Type hierarchies

There is no disagreement in SQL3 standardization over the question of whether abstract data types should be able to participate in type hierarchies. SQL3 does provide type hierarchies and inheritance of attributes by subtypes from their supertypes.

SQL3 provides *multiple inheritance*, in which a subtype may have multiple immediate ancestors ("parents") from which it inherits attributes (including both encapsulated representation and methods).

When a subtype inherits attributes from a supertype, the attribute retains the name that it had in the supertype. Because of the encapsulation of state into pairs of functions ("get" and "set" functions), every "stored attribute" in the state of an ADT is accessible only through a functional interface. This means that, when the polymorphism for functions comes into play, no stored attribute of a supertype need ever be renamed when inherited by a subtype — even when multiple supertypes of the subtype have attributes of the same name. Of course, the same is true for all "methods" of multiple supertypes.

3.3. Multiple inheritance

Unlike several popular object models, SQL3 provides multiple inheritance — the ability to inherit attributes from multiple supertypes. This allows applications, for example, to define an "intern" to be simultaneously a "student" and an "employee", or a "computer" to be both a "network node" and "capital equipment".

3.4. Encapsulation of implementation

SQL3 provides complete encapsulation of all aspects of an ADT. Every portion of the state ("stored attributes") of an ADT are represented by a pair of functions, commonly called the "get" and "set" functions. The get function takes a single argument:

```
ATTR_NAME(ARG)
```

while the set function takes two arguments:

```
ATTR_NAME(ARG, VALUE)
```

The get function returns the value of the attribute identified by its name (ATTR_NAME), using the single argument (ARG, whose data type is the abstract data type of which that attribute is a part) to identify a specific instance of the ADT.

The set function sets that attrbiute, identified in the same way, to the value specified by the second argument (VALUE).

SQL3 provides for the "automatic" generation of these two functions for every stored attribute in the definition of an ADT (value or object). Type designers can, of course, define additional single- or double-argument functions, but the routine resolution policies of SQL3 demand that all routines with the same name (within the scope of the relevant path of schemas) be distinguishable by the number of arguments and the data types of those arguments. Therefore, two single-argument functions named SALARY, where the data type of the parameter for each function is EMPLOYEE, are not distinguishable. However, if EMPLOYEE has a subtype, TRAINEE, then a function named SALARY with a parameter data type of EMPLOYEE and a function named SALARY with a parameter data type of TRAINEE are distinguishable at runtime, even if the function *invocation* specifies an argument of EMPLOYEE.

Routines used to provide the behaviors of SQL3 ADTs can be written in SQL's computationally-complete language (part of SQL/PSM) or in one of several more common programming languages (the 7 to which bindings with SQL are defined: Ada, C, COBOL, Fortran, MUMPS, Pascal, and PL/I). The implementations of the routines (that is, the code of the routines themselves) are encapsulated in that no SQL application has access to them through the type definition with which they are associated.

3.5. Multiple levels of encapsulation

SQL3 provides type definers with three levels of encapsulation: PUBLIC, PROTECTED, and PRIVATE. This permits a type definer to reserve certain methods (either explicit functions or the implicit get/set functions associated with stored attributes) to be accessible only within the definition of the type itself (PRIVATE), only by subtypes of the type (PROTECTED), or by any user permitted to use the type at all (PUBLIC) — which is itself governed by the availability of the USAGE privilege on the type itself. (In addition, the type definer can further control access to certain methods by various SQL users by selectively choosing not to grant the EXECUTE privilege on those methods.)

3.6. Polymorphic routines

As stated earlier, SQL3 provides *polymorphic* routine capabilities. This facility makes it possible for many different routines to be invoked using the same name, with "the system" differentiating among the routines based on factors such as the number of arguments and the data types of those arguments. (SQL's naming policies generally require that every database object in a given schema have a unique name; this policy is satisfied for routines by giving them a "specific name" that is unique within a schema, as well as a "routine name" that need not be unique.)

When all arguments of invocations of routines sharing a routine name (call them "candidate routines" of a given routine invocation) are built-in data types, then the determination of the specific routine can be made at compile time. However, when one or more arguments of an invocation of candidate routines is an abstract data type for which subtypes (not necessarily *instances* of those subtypes) exist and two or more of the candidate routines are have "corresponding parameters" that are the specified abstract data type and one or more of its subtypes, then the final determination of the specific routine has to be left until runtime.

SQL3's routine resolution policies seem quite intricate, but they are in fact quite elegant when viewed in detail. The rules may sometime seem arcane; however, the ability to overload routine names for routines providing related functions over different data types is extremely powerful and useful.

SQL's policies provide a "generalized" routine invocation model, contrasted with C++'s "classic" model. C++ uses a *distinguished* argument to resolve polymorphism, while SQL uses all arguments for routine selection. One benefit of the generalized model is that one can define special routines that apply only to newly-added subtypes without having to modify existing methods that apply to supertypes.

3.7. User-defined operators

Many abstract data type systems provide a functional interface that applications can use to perform operations on instances of those ADTs. However, functional notation is not always the most intuitive notation for application programmers to use.

Consider a type library that supports complex numbers. Addition of two complex numbers using functional notation:

```
ADD (VAR1,VAR2)
```

is not nearly as direct and friendly as:

```
VAR1 + VAR2
```

SQL3 provides a mechanism by which type definers can supply prefix, infix, and postfix operators using a variety of symbols and identifiers, at a number of different precedence levels. While not really integrating seamlessly into SQL's ordinary operator use, it comes reasonably close without the extreme overhead that many extensible languages encounter. The operators are automatically "translated" by the compiler into a special sort of SQL identifier (called "delimited identifiers") and then used in a re-ordering of the operators and operands into the functional notation. Therefore, the SQL engine need deal only with the functional notation, while the application programmer gets to use a more friendly approach.

It is worth noting that user-defined operators can be applied to SQL's built-in datatypes as well as to ADTs.

3.8. Multiple comparison capabilities

Comparison of abstract data type instances is often a tricky issue. Most observers would agree that objects (having a unique identity) are reasonably compared only for identity or lack thereof, while values can be compared for equality, inequality, and an ordering relationship (less than, greater than).

However, not every observer would necessarily agree that comparison of two complex structures is done in exactly the same way. Some might want to compare representations (the "cheap way") on a bit-by-bit basis, while others might want to do a logical-value comparison while "walking the tree" of the complex value's structure.

SQL3 supports any form of comparison by providing default comparisons (identity for objects, equality of corresponding structure components for values) in addition to the ability for type definers to supply functions for other comparison semantics. Among the more interesting alternatives is a HASH comparison oriented specifically towards building hash values (through a function provided by the type definer) corresponding to the ADT instance that can be used, if desired, for identifying database pages or other storage locations.

4. Schedule

The partitioning of SQL3 into multiple parts was done for two principle reasons. First, it removed the problems associated with having a single, monolithic document that was, frankly, more than a little intimidating to hold and access. Second, it raised the possibility of different parts progressing on different schedules.

Indeed, two parts of the partitioned SQL have been progressed at an accelerated rate. A version of SQL/CLI [3] (part 3) oriented towards SQL-92 was approved in July, 1995, as an International Standard to be published as ISO/IEC 9075-3:1995. A version of SQL/PSM [4] (part 4), also oriented towards SQL-92, was advanced to Draft International Standard status in mid-1995 and will be published as an International Standard in late 1996 (ISO/IEC 9075-4:1996). Parts 1, 2, and 5 (SQL/Framework [1], SQL/Foundation [2], and SQL/Bindings [5]) have not yet been advanced to Committee Draft status, though that is expected in early- to mid-1996; a reasonable expectation is that publication as *de jure* standards might occur in 1998. Other parts are progressing on schedules that are slightly behind those three parts.

5. Summary

Continuing progress is being made by the SQL standards community on a major enhancement to the 1992 SQL standard. While the "theme" of these enhancements has long been viewed as the addition of object technology to SQL, there are a great many important enhancements that are independent of that theme.

The process of adding objects to SQL has proved to be extremely difficult and contentious, but the participants continue to work together to find stable, viable, and technically correct solutions to the problems that have been encountered. The resulting standard will set the agenda for the database system implementors for several years after its publication.

References

1. ISO/IEC JTC1/SC21 N9462, *(Working Draft) Information technology — Database languages — SQL — Part 1: Framework*
2. ISO/IEC JTC1/SC21 N9463, *(Working Draft) Information technology — Database languages — SQL — Part 2: Foundation*
3. ISO/IEC JTC1/SC21 N9464, *(Working Draft) Information technology — Database languages — SQL — Part 3: Call-Level Interface*
4. ISO/IEC JTC1/SC21 N9465, *(Working Draft) Information technology — Database languages — SQL — Part 4: Persistent Stored Modules*
5. ISO/IEC JTC1/SC21 N9466, *(Working Draft) Information technology — Database languages — SQL — Part 5: Host Language Bindings*
6. ISO/IEC JTC1/SC21 N9467, *(Working Draft) Information technology — Database languages — SQL — Part 6: Global Transactions (XA Specialization)*
7. ISO/IEC JTC1/SC21/WG3 DBL:LHR-008 (a/k/a X3H2-95-373), *(Working Draft) Information technology — Database languages — SQL — Part 7: Temporal*