Can SQL3 Be Simplified?

Author: ***David Beech***

Status: ***SQL3 discussion paper***

Introduction

The body of this paper is submitted below in electronic form. For the published form, together with illustrations, please see the January 1997 issue of Database Programming and Design.

Since the simplification of SQL3 was not finally completed in Madrid, some further discussion of the issues seems to be desirable. In particular, I am concerned that if there is undue redundancy in the type system, this hurts users not only because of the complexity but also because product support will be patchy. Vendors will not quickly implement all features of such a vast language and will choose different subsets. As far as I am aware, most current effort is going into supporting named row types, because their instances are referenceable and can be stored as rows of tables. Therefore it may be wise, as in SQL86, to remove features on which there is not such broad consensus.

Can SQL3 be Simplified?

David Beech

SQL is of vital importance to the database community, and has for some time been acknowledged as "intergalactic dataspeak" by Michael Stonebraker (while wistfully recalling the beauties of the former Ingres language QUEL). With the emergence of object-relational systems as commercial realities, the future direction of SQL is a topical concern, and the language has just reached a critical juncture: the Committee Draft (CD) version of the proposed new standard, popularly known as SQL3, has been circulated for international ballot.

Some of the extensions, such as triggers and authorization roles, have already been widely implemented in close approximation to the CD specifications, so that most of the excitement surrounds the addition of user-defined types, and in particular those that define "objects" and collections of objects.

Jim Melton's entertaining article "A Shift in the Landscape" (in the August 1996 *Database Programming & Design*) summarized the story of the SQL3 type system prior to the decision to begin a CD ballot. At press time, the balloting will have been completed by the individual countries of the Data Base Languages (DBL) rapporteur group in ISO. At the DBL meeting in Madrid, during the last week of January and first week of February 1997, the national ballots will be reviewed, and the DBL will attempt to resolve any negative ballots. Once they reach something close to unanimity, the DBL will regard the draft to have passed, and SQL3 functionality will be settled.

There was a time when revisions of widely-used standards received intense public scrutiny, but committees have learned from the experience of a COBOL committee that was reputed to have received more than 700 sets of comments during a U.S. ballot. The U.S. position on the SQL3 CD appears to have been determined within the ANSI committee, which has also done most of the detailed work on the revisions, as Jim Melton has described.

COMPLEXITY OF SQL3

In Melton's short article his history was necessarily selective, and concentrated on events leading towards the conclusion that an "admirable solution" had been reached. This is a characteristically generous assessment, and one that reflects some well-earned relief at having reached a breathing point in the process of language design by committee, following *Robert's Rules of Order*.

Standing back a little from the action, and filling in other parts of the history, we can arrive at a different, although no less optimistic, perspective on the past and future of SQL3. The single most pressing question when looking at the whole picture is whether the language has become unnecessarily complex.

Even the physical size of the SQL documents has grown enormously. While SQL-92 totaled about 600 pages, the equivalent parts of SQL3 currently run to over 1100 pages. Allowing for some duplication of material due to the splitting into parts, this is still a formidable increase.

Oracle, like other vendors, supports standards, and also has to ship products that users want. The products must not only be well implemented, but have to solve real problems, and be simple enough to be usable. Relational technology in its origins was a deliberate simplification compared to hierarchical and network database technology, with impressive results for a wide range of applications.

Object-relational technology is admittedly making a trade-off of simplicity against other benefits, extending the relational model to provide more convenient and more efficient support for applications that do not fit easily within the relational discipline. But this must still be done as simply as possible. My personal rule of thumb is that it is seven to ten times harder to implement a

language feature in a database system than in a programming language compiler. If we want a standard fully implemented within five years (even with a head start on some of the features), considerable economy of language design must be practiced.

Supposing that the same set of features is widely available in different implementations, will the standards be well enough understood by users who are not programming wizards? I'm not implying that application developers will all use SQL directly. Even if higher-level tools conceal the syntax of the language, users must clearly understand the data model or type system of the manipulated information. The model cannot be as simple as the relational model alone—indeed, it is a proper superset of the relational model—but how closely can it approach the same ideals of economy and elegance?

At least one of the CD comments being framed at the time of this writing calls for simplification of the SQL3 type system. This system forms the heart of the extensions, and Jim Melton described some salient parts of it. In this article, I'll give a description of other parts of the proposed type system, and we will discover that there is considerable overlap among the parts.

We could also observe that currently SQL3 tends to resemble a union of features from different vendors, rather than the intersection that was necessary to reach agreement on the first SQL standard in 1986.

TWO MAJOR ADVANCES

In adding "objects" to SQL, the standards committees are making two major advances. Although I'll have to concentrate on just one, the extension of the type system to define individual objects, we should note the other advance because of its importance and encouraging nature. This advance is the treatment of *collections* of objects, such as sets, multisets, and lists. An effort is under way jointly with ODMG representatives to achieve convergence between SQL3 and OQL for collection queries. This has additional significance since the OMG CORBA Query Service adopted SQL-92 and OQL, while expressing strong hopes that they would converge in SQL3.

An "SQL3 Concepts" section, contributed by the United Kingdom national body, defines collections in a way that is consistent with mathematical practice and with the ODMG model. A slightly modified form of SELECT, derived from OQL, can then be used to query such collections without having to treat them as tables. At the same time, a few problems with today's SELECT can be avoided, and this workhorse of SQL emerges refreshed for many more years of service. This is the other half of the simplification story, but it is too long a tale to tell here.

THE PANOPLY OF SQL3 TYPES

The original SQL type system was very simple. The only explicit types were the built-in numeric and character types, usable as column types. The date and time types were added in SQL-92, with some elaboration of the character types to support national character sets.

From a programming language point of view, we'd say that the table, as the one composite entity permitted in the relational model, deserved a type, but that was never made explicit. If one wanted to have two tables of similar structure, one could use the LIKE construct, but this was weaker than a type concept—it was more like a text editor facility that allowed one table definition to be copied to form all or part of another one.

Yet in the "Concepts" section of SQL-92, one reads "A table is a multiset of rows." (A multiset is a collection that may contain duplicates, whereas a strict set may not.) The concept of a table type could well have been formalized by having multiset as a collection type, and a tuple or record type for the rows that are the elements of the collection.

SQL-92 also introduced what it called *domains*, as a shorthand for types optionally decorated with constraints. Thus one can write

3

```
CREATE DOMAIN Dollars AS INTEGER;
CREATE DOMAIN Miles AS INTEGER;
```

and use Dollars and Miles in place of data types for columns to provide a better visual indication of what the integers represent. But the domain names have no teeth. They are like names introduced by a typedef in the C language, being replaced by their definitions before any checking is performed, so that errors like assigning miles into a dollars column would not be caught.

The choice of terminology was probably unfortunate, because the use of *domain* in relational theory appears to correspond much more to a strongly checked type concept, where values from domains with different names could not be interchanged without explicit coercion. However, given the sacred nature of eternal compatibility of standards, the SQL-92 use of DOMAIN is probably here to stay, although it would be technically possible to introduce an alternative syntax and officially "deprecate" DOMAIN, which could then be withdrawn 5 years later in another revision of the standard. One might, for example, prefer

```
CREATE WEAK TYPE Dollars AS INTEGER;
```

The reason for paying so much attention to this minor feature of SQL-92 is that the underlying distinction between weak and strong typing will prove crucial to an understanding of the history and current status of the SQL3 type system.

SQL3, standing on the shoulders of SQL-92, includes all of the above type and domain facilities, plus many more. Before giving examples, we will first summarize the SQL3 extensions (still incomplete) to see the big picture:

- Unnamed row types

- Abstract data types (ADTs), with
    1. inheritance
    2. encapsulation
    3. polymorphism
    4. **no identity**

- Collection type generators (table, multiset, set, list)

- Distinct types

User-defined type generators (templates) have been specified, but are deferred until SQL4. (We use *type generator* to mean a concept that is not a complete type in itself, but needs additional parameters to generate an individual type, for example *multiset* is a generator of types like *multiset(integer)*, or *multiset(Employee)* where Employee is an abstract data type).

It is also possible to create tables (with unnamed row types) that inherit columns from supertables, but there is no inheritance between the corresponding unnamed row types.

In the separate "SQL/Object" part of SQL3 described by Jim Melton, there are also:

- Named row types (let's call these NRTs)
    1. polymorphism
    2. identity
    3. **no inheritance**
    4. **no encapsulation**

- REF type generators

SOME EXAMPLES

Let's look at some examples, beginning with unnamed row types such as

```
ROW (emp_no INTEGER,
     name CHARACTER VARYING(60),
     dept_no INTEGER)
```

which obviously make explicit what was implicit in SQL-92, allowing the familiar

```
CREATE TABLE Emp (
     emp_no INTEGER,
     name CHARACTER VARYING(60),
     dept_no INTEGER)
```

to be treated as creating an entity of type

```
MULTISET(ROW (
     emp_no INTEGER,
     name CHARACTER VARYING(60),
     dept_no INTEGER))
```

Unnamed row types can also be used quite generally as data types, for example as column data types, thus departing from First Normal Form, for which even some relational pundits seem to have lost enthusiasm. "Row types" seems to be a misnomer when one considers that these types can be used for entities other than rows, but U.S. attempts to have them named something like *record* or *struct* types failed in ISO.

Proceeding from one misnomer to the next, we come to abstract data types. At least to me, this phrase seems to have generally been used in connection with an intermediate generation of extensible languages such as Ada, before the full-fledged object models took shape (or, perhaps one should say, before the Simula tradition was revived). Yet it was as an object type that the abstract data type (ADT) was introduced into SQL3, and we will try to shed more light on how the ADT has been stripped of its essential objecthood. At the object "core" is the concept of identity, which has been acquired instead by the fledgling "named row type."

As a simple illustration of an ADT, suppose we want to create something rather like a row type in a familiar Employee table, but modified by adding an age function computed from an encapsulated birthdate:

```
CREATE TYPE Emp_type (
    emp_no INTEGER,
    name CHARACTER VARYING(60),
    dept_no INTEGER,
  FUNCTION age(e Emp_type) RETURNS INTEGER .....;
  PRIVATE
    birthdate DATE
)
```

HISTORICAL BACKGROUND

The 1991 papers that Melton referred to as "the earliest papers" were in fact preceded by other significant papers, for example from Phil Shaw when at IBM in developing "user-defined types" not unlike what ADTs have now become, and from Cetin Ozbutun and myself in a paper "Object Databases as Generalizations of Relational Databases." The latter was presented first at a database workshop, then in December 1990 to the ANSI SQL committee at a pleasant location (in the French Quarter of New Orleans), and was later published in the *International Journal of Computer Standards and Interfaces*. It showed in some detail how the relational model of SQL could be extended into an object-relational one, but stopped short of making a "detailed change proposal" to add SQL3 text to the base document, since much work remained to be done first to complete what became SQL-92.

Therefore the first detailed change proposal from the Digital Equipment Corporation authors came as a surprise, and introduced an element of friendly rivalry into the process. It was a good paper, close enough to the Oracle approach in essentials to gain our support, and it also contained some pointers to further work needed to strengthen the data manipulation language—ideas that still deserve to be followed up!

The main difference between the approaches was in the treatment of attributes and encapsulation, where the DEC authors favored treating attributes as necessarily private, much like Smalltalk instance variables, and offering only routines in the public interface. Our approach was more akin to that of C++, since the implicit row type of an SQL-92 table could be seen as an unnamed *struct* type as in C. By adding inheritance, methods, encapsulation and identity, one would have the equivalent of the C++ *struct* (that is, the *class* except that the initial default is for members to be public rather than private). In fact, we adopted the C++ public/private/protected mechanism directly rather than inventing a new one. Thus attributes (as well as routines) may be exposed or not, as desired. We reached a happy accommodation for SQL3 when we combined the best of both approaches.

All seemed to be going well with SQL3 in 1992 until the United Kingdom national body launched an all-out attack, basing their case on a bug in the specification of encapsulation applied to rows of tables. To avert World War III, the U.S. decided to concede defeat, although no example of a failure of encapsulation was ever produced after the bug was fixed! The U.K. claim was that removal of tables with objects as their "rows" would avoid delay in the completion of SQL3. Ironically, the language seems further from completion now than it was four years ago.

During 1993 and 1994, there were many divisive efforts to separate ADTs into two kinds, usually labeled VALUE and OBJECT, although without total agreement as to what the differences were, apart from VALUE instances lacking identity and therefore referenceability. Oracle for simplicity tried to keep a single kind of ADT, and to associate with each ADT such as Person a corresponding reference type REF(Person).

Space does not permit a full account of these issues. As an important historical footnote it should be noted that the U.K. submitted a paper in 1995 that thoroughly surveyed previous papers and magnanimously expressed regret at their earlier opposition to the unified type system and treatment

of REF types. The only point where they held fast was in continued opposition to the "table of ADT" concept, for apparently philosophical reasons rather than any technical flaw.

To explain their philosophical position, we need to move on to the third SQL3 extension to the type system, the collection type generators multiset(T), set(T), and list(T) for any type T. Thus if T is an ADT Emp_type, it is quite acceptable to define a multiset(Emp_type). However, a *table* is defined as a *multiset* whose element type is a *row type*, that is, a simple *struct*, and therefore the table has the familiar columns. An important implication follows from this definition, if we remember that tables are still the only top-level named entities that can be stored persistently in an SQL3 database. Hence a persistent instance of any other data type such as an ADT can exist only within a column of a table, and relational theory continues to reign supreme, although having quietly shed the burden of First Normal Form which forbade anything composite in a column.

This politically correct update of relational theory might be called *nouvelle relationelle*. As a fashionable cuisine, it is unusual in that it has proceeded from the extreme low-fat diet of pure relational to much more sumptuous fare. The choicest of the new culinary delights takes an arbitrarily rich object and encases it in the crust of a one-row one-column table—this is the *Object Wellington*. Unfortunately, there is not much relational theory applicable to such dishes. They may not be partitioned horizontally or vertically, so that with only relational utensils, they would not be edible in polite society.

The remaining type definition in the main body of SQL3 ("SQL/Foundation") is the *distinct* type, so-called to emphasize that such types are distinct from other types if their names are distinct, as in

```
CREATE DISTINCT TYPE Dollars AS INTEGER;
CREATE DISTINCT TYPE Miles AS INTEGER;
```

If we define Dollars and Miles this way instead of with CREATE DOMAIN, we get strong checking, and values of the two types cannot be interchanged, even though they are both defined over the same underlying type INTEGER.

SQL/OBJECT EXAMPLES

To illustrate the two type extensions in "SQL/Object," named row types and REF types, we need look no further than an example of Jim Melton's that combines them:

```
CREATE ROW TYPE account_type(
  acct_no     INTEGER,
  cust        REF(customer_type),
  type        CHARACTER(1),
  opened      DATE,
  rate        DOUBLE PRECISION,
  balance     DOUBLE PRECISION);
```

where customer_type has been previously defined as a named row type.

REF is a type generator that must be parameterized by the name of a named row type NRT to produce a specific type whose values are references to instances of NRT.

A surprising aspect of named row types at the moment is that they are weakly checked, whereas their corresponding REF types are strongly checked, that is NRT1 and NRT2 may be matching types if the types of their fields match in order, despite the fact that the names NRT1 and NRT2 differ; but REF(*NRT1*) and REF(*NRT2*) can never match. We will try to suggest below how this confusing situation can be avoided.

OCCAM′S RAZOR

William of Occam would be sharpening his razor at this point. Celebrated since the fourteenth century for his maxim "It is vain to do with more what can be done with fewer," he would surely ask three questions:

- Why cannot ADTs, named row types, and distinct types be combined into a single strongly checked named type concept? This could be specified by the form

  CREATE TYPE <type name> [AS] <type expression>

- Why cannot domains and weakly checked named row types be combined into a single weakly checked named type concept? This could be specified by the form

  CREATE WEAK TYPE <type name> [AS] <type expression>

  (Nameless row types are of course weakly checked, since they have no names to check strongly.)

- Why cannot the table hierarchy and the type hierarchy concepts be integrated by associating a row of a subtable with a subtype in a type hierarchy?

It would be hard to know how to answer Occam. For example, if one were contemplating defining an Employee type in today's SQL3, how could one decide whether to choose an ADT or a named row type? If one wants to use inheritance from a Person type, or have any encapsulation, it would have to be an ADT. But then there is no identity, and no REF(Employee) type is available. One might need to define an Emp_ADT and an Emp_NRT for different usages, and write routines to cast between them.

But if the plan of work outlined by Jim Melton is carried out for SQL3, and NRTs acquire inheritance and encapsulation, we have a simpler choice based upon our need for identity and referenceability. If yes, choose the NRT; otherwise choose the ADT. This is virtually the old OBJECT vs. VALUE type distinction, revived with a different syntax and much duplication in the language. Even this distinction forces the user to make a decision that may be regretted later. By choosing an ADT, one rules out forever the possibility of having instances that can be referenced directly. For example, an Address_ADT may be defined, expecting it to be used as an attribute data type within a Person_NRT. Later, it seems a good idea in another application, say, to keep a table of addresses since the addresses are often shared. Bad luck! A different Address_NRT would have to be defined, and if the applications wanted to share data, casting routines would have to be written between the types, with ensuing complications, for example on update.

So it would be sufficient to just have the NRT in the language, once it had been given the full functionality of the ADT plus identity and referenceability, and users would never have to make decisions between NRTs and ADTs. The ADT could be dropped, the NRT could be folded back into the main body of SQL3 instead of being in a separate SQL/Object part, and with strong typing of NRTs as well as REF(NRT)s, a simple and easily understood type system would have been restored. You'll recognize this as the alternate optimistic solution promised at the beginning of this article. It represents one possible fulfillment of a continuous thread of historical development in SQL3.

Will the traditional Just In Time wisdom of the committee system choose this path for SQL3? Will the single greatest virtue of the relational model, its simplicity, be the guiding principle in extending it to an object/relational model? These questions are about to be decided in Madrid.