Recent Design Trade-offs in SQL3

Nelson Mattos   Linda G. DeMichiel *

1 Introduction

SQL3 [1], the name given to the new draft of the SQL standard that is likely to become an international standard replacing SQL92 in 1996 or 1997, contains several object-oriented extensions [2]. When defining such extensions, X3H2 (the American Committee responsible for the specification of SQL3) and DBL (the International Committee for the same purpose) have had to make (and are still making) some of the same decisions made by the designers of other object-oriented languages. Among these decisions is the one described by Zdonik and Maier in [3]. Zdonik and Maier observed that it is not possible to combine more than three of the following four features in a single language:

1. Substitutability.
2. Static type checking.
3. Mutability
4. Specialization via constraints.

The question is, which three of these four features should be incorporated into SQL3? This paper discusses the options and suggests what the authors believe to be the most appropriate set for the SQL language. Section 2 of the paper reviews the four concepts (based on the definitions given in [3]) and shows why they cannot be combined in a single language. Section 3 provides some background on SQL3. Section 4 discusses the set of features that should be incorporated into SQL3 and illustrates how this is being accomplished. Finally, Section 5 summarizes the main ideas of the paper and describes ongoing work.

*Authors' addresses: Nelson Mattos, Database Technology Institute, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 (email: mattos@vnet.ibm.com); Linda G. DeMichiel, IBM Almaden Research Center, 650 Harry Road, San Jose, CA 95120 (email: lgd@almaden.ibm.com).

2 Four Desirable, But Incompatible Features

As has been observed by Zdonik and Maier, it is not possible to combine all of the following in a single type system: substitutability, static type checking, mutability of data types, and subtyping by means of constraints. The following definitions are taken from [3].

1. *Substitutability.*

   "To say that B is a subtype of A typically means that any context that is expecting an instance of type A must also accept an instance of type B. We call this the principle of *substitutability*."

2. *Static type checking.*

   "By *static type checking*, we mean that all reasoning based on information expressed on types is checkable at compile-time. In other words, there is no need to check type compatibility at run-time. This property is clearly desirable, since it means that there is no need to insert expensive run-time checks in the resulting code, and also that the coder can be assured that a certain class of errors can never occur."

3. *Mutability.*

   "An operation m is a mutator for type T if, for some instance x of T and for some r in O_r, it is possible to execute the following code fragment such that, at the end, a is not equal to b.

   $$a = r(x); m(x); b = r(x);$$

   That is, if m is to be a mutator, it must be possible to observe its effect on some object. A type system incorporates *mutability* if it is possible to construct a type T with some operation that is a mutator."

4. *Specialization via constraints.*

   "It is often useful to allow subtypes to be defined by restricting some aspect of the supertype. This is a reflection of the fact that some models use a class hierarchy to express application-specific constraints. If **RoadSegments** have directions that can be **one-way-SE**, **one-way-ES**, or **two-way**, we would like to define **OneWaySegment** as a subtype of **RoadSegment** with direction **one-way-SE** or **one-way-ES**."

2.1 Incompatibility

Let us assume a type `rectangle` and a subtype of `rectangle`, `square`, with the same definition as `rectangle` except that all four sides are constrained to have the same length. Consider a function `shorten_side` defined on `rectangle` that takes a rectangle and an integer value and mutates the rectangle by reducing the length of two opposite sides of the rectangle. What happens when we pass a square to this function? The function does not know that all sides of a square must have the same length. In other words, if we have mutability and substitutability, a square can be passed to `shorten_side` at run time and modified in such a way that it violates the constraint defined for `square`. Any of the following four approaches can be used to solve this problem.

1. We do not support mutability and, hence, a function that mutates its arguments, like `shorten_side`, cannot be defined.
2. We eliminate substitutability and the problem would disappear because squares could not be passed to functions defined on rectangles.
3. We do not do static type-checking and the execution of `shorten_side` raises a run-time error if it causes the constraint on squares to be violated.
4. We do not permit a subtype to be defined as a constrained variant of a supertype.

3 SQL3 Background

SQL3 extends the SQL language to include the concepts of abstract data types (ADTs). ADTs in SQL3 are user-defined types with an encapsulated internal structure. An ADT consists of a heterogenous sequence of named fields called *attributes*. When an ADT is defined, functions are automatically defined to:

- access the attributes of the ADT (so-called *observer* functions)
- modify the attributes of the ADT (so-called *mutator* functions)
- create instances of the ADT (so-called *constructor* functions)

For example, the following SQL3 type definitions would cause the automatic definition of the functions whose signatures are given below.[1]

```
CREATE TYPE person
 (name varchar(30),
  date_of_birth date,
  ssn integer,
  address address)

CREATE TYPE address
 (street varchar(30),
  city varchar(30),
  country varchar(30))
```

Constructors:

```
person() → person
address() → address
```

Observers:

```
name(person) → varchar(30)
date_of_birth(person) → date
ssn(person) → integer
address(person) → address
street(address) → varchar(30)
city(address) → varchar(30)
country(address) → varchar(30)
```

Mutators:

```
name(person, varchar(30)) → person
date_of_birth(person, date) → person
ssn(person, integer) → person
address(person, address) → person
street(address, varchar(30)) → address
city (address, varchar(30)) → address
country(address, varchar(30)) → address
```

SQL3 also provides a "dot notation" as a syntatic sugar for the invocation of functions on ADTs. The two assignment statements in the following piece of code are fully equivalent in SQL3.

[1] The syntax of the definitions has been simplified somewhat to avoid having to explain details of the SQL3 language that are not relevant for an understanding of this paper.

```
BEGIN
DECLARE var person;
DECLARE x varchar(30);
SET x = var..name;
SET x = name(var);
END;
```

The creation of instances of an ADT can only be accomplished by invoking the constructor function that is provided as part of the ADT definition.[2] When a constructor is invoked, it

1. Generates a new instance of the ADT.
2. Attaches a type tag to the newly created instance.
3. Sets each attributes of the new instance to a null value of the corresponding type.

For example,

```
BEGIN
DECLARE var person;
DECLARE x varchar(30);
SET var = person();
SET var..name = 'John Doe';
SET var..address = address();
SET var..address..city = 'Rio';
END;
```

The two declarations set the variables `var` and `x` to null values of types `person` and `varchar` respectively. (Note that a null instance is not the same as an instance all of whose attributes are null.) In the assignment statement `SET var = person()`, a new instance of `person` is created and assigned to `var`. At this point, `var` contains a non-null instance of `person` all of whose attributes are null. The assignment statement `SET var..name = 'John Doe'` replaces the value of the attribute `name` with `'John Doe'`. The next assignment statement creates a new instance of `address` and assigns it to the attribute `address` of `var`. This new address is a non-null instance of `address` all of whose attributes are null. Finally, the value of the attribute `city` of the `address` attribute of `var` is set to `'Rio'`.

An ADT can be defined to be a subtype of another ADT by providing the supertype name as part of the ADT definition:

[2] The user can of course define his or her own "constructor" functions or "constructor-initializer" functions to customize the initialization of ADTs. Such functions must first, however, invoke the system-defined constructor to create the ADT instance itself.

```
CREATE TYPE employee UNDER person
 (salary DECIMAL (9,2))
```

An ADT that is defined as a subtype of one or more other ADTs inherits structure and behavior from those ADTs. Structure is inherited in the sense that instances of the inheriting ADT have all the attributes that were defined for instances of its supertypes in addition to attributes that were defined for the inheriting type itself. Supertype operations are inherited in the sense that operations that are defined for a supertype are applicable to instances of its subtypes as well.

Thus, in the above example, the subtype `employee` has five attributes: the attribute `salary` that was defined by the type `employee` itself, and the attributes `name`, `date_of_birth`, `ssn`, and `address` that were defined for the type `person`.

The definition of the type `employee` causes the automatic creation of the constructor function `employee()` → `employee`, the observer function `salary(employee)` → `decimal`, and the mutator function `salary(employee, decimal)` → `employee`. In addition, any functions defined on the type `person` (i.e. any function that has a formal parameter of type `person`) is applicable to an instance of `employee` as well. Hence, the observer and mutator functions defined on type `person` are inherited by type `employee`.

SQL3 supports both function overloading and the dynamic dispatch of functions. Given a function invocation, the function instance that is chosen for execution depends on the types of the actual arguments to the function at run time. This function instance is selected from among the applicable function instances for the invocation, and of these will be the function instance that is the best match for the given run-time argument types.

4 Which Features Should Be Preserved in SQL3?

Of the four features listed in Section 2, three are directly related to the object-oriented extensions to SQL: mutability, substitutability, and specialization via constraints. Static type checking is orthogonal to these object-oriented extensions.

We will start our discussion with static type checking, because, of all of the features listed above, it is the most deeply rooted in the SQL language.

4.1 Why Static Type Checking Is Important

SQL has always been a statically-typed language. This tradition goes back to the first SQL standard, SQL86, and is assured by the fact that all SQL queries are type-checked before execution. Because SQL3 needs to be an upward compatible extension to SQL86, SQL89, and (especially) SQL92, preserving static type checking in SQL3 is essential. Retaining static type checking in SQL3 enables better performance (a constant goal of SQL implementors) and eliminates the possibility that applications that have been running for a long time or that are in production suddenly break with type errors when new subtypes are introduced.

4.2 Why Substitutability Is Important

One of the primary principles of object-orientation is that of substitutability: the ability to use an instance of a subtype in whatever context an instance of one of its supertypes can be used. In particular, this means the ability to pass instances of the subtype to functions defined on the supertype, to store instances of the subtype in variables or columns of the supertype, to return instances of the subtype as the results of any function whose formal result type is of the supertype, and so on.

Let us take a look at some examples showing the use of substitutability. Consider the following type and function definitions.

```
CREATE TYPE employee
 (id integer,
  name char(30),
  base_salary decimal(7,2));

CREATE TYPE manager UNDER employee
 (commission decimal(7,2));

CREATE FUNCTION salary (emp employee)
    RETURNS decimal(7,2)
 RETURN emp..base_salary;

CREATE FUNCTION salary (mgr manager)
    RETURNS decimal(7,2)
 RETURN mgr..base_salary + mgr..commission;
```

According to the above type definitions the type `manager` is a subtype of `employee`. Now consider the following SQL statement.

```
BEGIN
DECLARE emp employee;
        mgr manager;
        sal decimal (7,2);
SET emp = employee(123, 'John', 50000);
SET mgr = manager(456, 'Ann', 50000, 1000);
SET emp = mgr;
SET sal = salary(emp);
SET sal = base_salary(mgr);
END;
```

The first two assignment statements construct instances of `employee` and `manager` and assign these instance to `emp` and `mgr` respectively. The third assignment statement assigns the `manager` instance to a variable of type `employee`. The fourth invokes the `salary` function on this instance. Since the variable `emp` at this point contains an instance of `manager`, the `salary` function that is invoked will be that defined on `manager`, since it is the most specific applicable function for the given argument type. The last statement invokes the `base_salary` function defined on `employee` on the `manager` instance.

If substitutability were not supported in the language the last three assignment statements would be either invalid or produce a different result.

- `set emp = mgr`

  would be invalid because without substitutability an instance of a subtype cannot be assigned to a variable whose type is a supertype of the subtype.

- `set sal = salary(emp)`

  would always invoke the function on `employee` since `emp` could not contain an instance of `manager`.

- `set sal = base_salary(mgr)`

  the observer function `base_salary` could not be invoked with an instance of `manager` because it is defined on the type `employee`.

4.3 Why Mutability Is Important

In order to demonstrate why mutability is important, let us suppose a world in which ADTs are immutable and consider the consequences.

Let us first review in a simple example how ADTs can be mutated. Suppose we have the ADTs `person` and `address` defined previously. If `person` and `address` are mutable ADTs, we can invoke the system-supplied mutator function on `address` to change a person's street as follows: `street(address(p), new_street)` (Recall that this is equivalent to `p..address..street = new_street`.)

If `person` and `address` are not mutable, however, we must take a different approach. We must first create a new instance of the ADT `address` containing the same values for the `city` and `country` attributes, but with a new `street`, and then we must create a new `person` instance, containing the same values for the `name`, `date_of_birth`, and `ssn` attributes, but containing the new address. In other words, we need a function `street` on the type `address` that creates a new instance of `address`, in which all attribute values are copied from the old `address` instance except for the `street` attribute, which acquires the designated new value; and a function `address` on the type `person` that creates a new instance of `person`, in which all attribute values are copied from the old `person` instance except for the `address` attribute, which acquires the new address value. Note that such functions are needed as primitives to replace the primitive, system-generated mutator functions, since functions can no longer be mutators at all.

To change the `street` attribute in a `person` instance, we would then write

```
set p = address (p, street (address (p),
new_street));
```

Suppose now that `street` is an ADT consisting of a street number and a street name, and that we need to change the number but not the street name. Our example in the case of immutable ADTs then becomes

```
set p = address (p, street(address(p),
number(street(address (p)), new_number)))
```

If ADTs were mutable, this would simply be `number(street(address(p)), new_number)`

It should become clear that the operations on mutable ADTs are much simpler (not considering the possible performance penalty of having to construct new ADTs whenever a single attribute needs to be modified).

Consider now another example, this one involving subtypes. Suppose that we have the ADTs `employee` and `manager` that were previously defined and that ADTs are mutable. Suppose further that we have a function `raise` that increases an employee's salary by a specified value. We can use this function to update the `salary` of a manager as follows: `SET mgr = raise(mgr, 10000.00)`.

Suppose now that ADTs are not mutable, and suppose, as in our former example, that we have a function `raise` defined on the type `employee`.

In order to give a raise to an employee, this function needs to create a new `employee` instance, copying all the values of attributes from the old `employee` instance, but using the newly computed value for salary. What happens then when we try to use this function to give a raise to a manager? If we try to write the same statement as before, we get a type error, because the result type of the `raise` function is `employee`, not `manager`. In order for the above assignment to be valid, we must have a `raise` function on `manager`, returning an instance of `manager`. In other words, we lose substitutability! We are forced into overloading all supertype functions in order to avoid type errors. Therefore, SQL3 needs to support mutability if it wants to preserve sustitutability without forcing all supertype functions to be overloaded on the subtypes.

4.4 Can We Have Specialization Via Constraints As Well?

Consider again the example with the types `rectangle`, `square`, and the `shorten_side` function described in Section 2.1. The function `shorten_side` takes a rectangle and an integer value and reduces the length of two opposite sides of the rectangle. Assuming substitutability we could pass an instance of `square` to `shorten_side` at run time, in which case an error should be raised because we cannot modify the length of only two sides of a square because it would then no longer be a valid square. Therefore, if we want to support specialization via constraints we have to accept one of the following options, none of which we believe to be appropriate:

1. We do not perform type checking statically since functions that mutate their arguments can raise exceptions when subtypes are passed as arguments at run time (because of substitutability). This option is not acceptable because it changes a well-established tradition of the SQL language. Moreover, it may cause running applications to break with the addition of new subtypes in the database.
2. We eliminate substitutability, in which case an instance of `square` could not be passed to a function defined on the type `rectangle`. We believe, however, that substitutability is one of the most important object-oriented features and must therefore be preserved in SQL3.

3. We do not support mutability and consequently we cannot have operations that modify the state of an ADT instance without creating a new instance of that same type. However, as we described in Section 4.3, mutability is a requirement to provide substitutability in the SQL3 model of subtyping and inheritance. Therefore, mutability cannot be eliminated without removing the current support for substitutability.

4. We force the overloading of all functions defined on supertypes. That is, we change the current model of subtyping in SQL3. This option seems to be unacceptable because we believe that SQL3 users are not likely to define type hierarchies themselves, but to buy them as class libaries from third party vendors. It is an important requirement that users be able to define common subtypes of these type hierarchies in order to combine the functionality provided by each. If we force all functions to be overloaded, users will have to redefine every function provided by the class libraries whenever they need to combine them in defining subtypes. The current SQL3 model of subtyping and inheritance does not require such redefinition while allowing the functions that need to behave differently on subtypes to be overloaded.

For these reasons, we believe that the most appropriate trade-off for the SQL3 language is to not permit specialization via constraints.

5 Ongoing Work

We have discussed above why it is not possible to have substitutability, static type checking, mutability, and specialization via constraints all in a single language; and we have shown the reasons why SQL3 should ban specialization via constraints.

SQL3 does in fact seem to be moving towards the elimination of specialization via constraints, although there still is a lot of work to be completed. The latest draft of the SQL3 standard makes clear that there is true substitutability in the current SQL3 specification of ADTs. In the words of [1]: "If T_a is a subtype of T_b, then an instance of T_a can be used wherever an instance of T_b is expected. In particular, an instance of T_a can be assigned to a variable of type T_b, passed as an argument for an input parameter declared as type T_b, and returned from a function whose result type is declared to be T_b." It supports mutation as defined in [1]: "*Mutator operations*: Functions that given an ADT instance and a new value, change some part of the state of the ADT instance." Finally, the language has been extended while performing type checking statically.

We also believe that the approach currently taken in [1] is indeed correct in the sense that it separates the issue of mutation from that of object identity. Mutability and object identity should be orthogonal concepts. Objects without identity (so-called *value ADTs* in SQL3) and objects with identity (so-called *object ADTs*) should behave exactly the same except for object identity. This means that object ADTs can be shared, and value ADTs cannot.

While we believe that the SQL3 document implicitly assumes that ADTs are mutable, several issues related to the mutability of ADTs have not yet been clearly specified in the language. For example, there are places in SQL3 where we think mutability should not be allowed or should be allowed only in a controlled fashion—as in the `WHERE` clause of a `SELECT` statement. It is currently underspecified as to where mutability should not be permitted or what the semantics would otherwise be. While we believe that SQL3 is moving towards the elimination of specialization via constraints, the ANSI version of SQL3 defines sets, lists, and multisets as subtypes of a more general collection type. While sets and lists are subtypes of collection defined by means of constraints (i.e., sets are collections that cannot contain duplicated elements and lists are collections in which elements are ordered), it is not clear how this can be handled in SQL3. Similarly, ADTs can have constraints defined on their attributes, but it is not clear what this means in the context of substitutability, mutability, and static type checking.

These and other open issues are likely to be the subject of debate in upcoming meetings of the SQL3 committees.

References

[1] American National Standards Institute (ANSI) Database Committee (X3H2). *Database Language SQL3*. J. Melton, Editor. August 1994.

[2] L. Gallagher. Influencing Database Language Standards. *SIGMOD Record*, Vol. 23, no. 1, March 1994.

[3] S.B. Zdonik and D. Maier. *Readings in Object-Oriented Databases*. Morgan Kaufmann, 1989.