Collections of Objects in SQL3

David Beech

Oracle Corporation
500, Oracle Parkway, Redwood Shores, CA 94065
dbeech@oracle.com

Abstract

SQL3 generalizes the relational model into an object model offering abstract data types, multiple inheritance, and dynamic polymorphism. Tables may then contain collections (multisets) of objects, and sets and lists are defined as closely related collection types. By specifying an SQL_Table type template to correspond to the existing Table concept, it is possible to treat sets and lists as subtypes of tables that inherit the behavior (and SQL syntax) for tables, while adding their own specializations. The SQL set-at-a-time data manipulation language can then be applied to collections of objects, i.e. tables in which each row is an object.

1. Introduction

This paper discusses some of the major considerations in introducing collection types such as Set and List into SQL3, which is the informal name for the language defined in the working draft for the next revision of the ISO and ANSI SQL standard [1]. The extensions described in this paper have been adopted by ANSI, and are under consideration by ISO at the time of writing, but of course many further changes may occur before the work on SQL3 is completed.

Although SQL3 contains a number of extensions that are still relational in character, most of the work on SQL3 since December 1990 has been devoted to generalization of the type system of the relational model into an object type system offering abstract data types, multiple inheritance, and dynamic polymorphism [2]. The extensions are completely upwards compatible from the relational language, and subsume the relational model rather than juxtaposing an object model. To emphasize that the SQL3 model retains the relational model at its heart as a special case, it may be called a *circumrelational* model.

Since an object type system requires the definition of functions, SQL3 has been provided with procedural language extensions as one means, although not the only means, of specifying the function bodies. Thus SQL3 is no longer merely a database sublanguage, but is a computationally complete programming language, with an emphasis on database applications. It does not aim to be a universal application language, but rather aspires to be a very convenient way of implementing database procedures and functions to define the semantics of the data in the database.

The SQL revision previously known informally as SQL2 was completed and formally approved by both ISO and ANSI in 1992, and has thus become SQL-92. Previous versions of the standard were SQL-86 and SQL-89. This aggressive pace is unlikely to be maintained indefinitely, and SQL3 will probably be finally approved in 1996 or 1997. Features such as the object extensions, for which there are strong user requirements, are of course likely to be implemented ahead of formal standardization. This has already happened with other features such as triggers and stored procedures, which were not included in SQL-92 but have draft definitions in SQL3.

Another reason for urgency in stabilizing the essential features of SQL3 is that there is strong interest internationally in developing SQL3 type library definitions for various functional areas, so that objects can be stored in databases together with the functions defined on them, and will then be accessible to applications written in a variety of programming languages. (SQL-92 defines seven language bindings to standard programming languages, and SQL3 aims to strengthen this ability to work effectively with many languages by taking advantage of the increasing similarity of its type system to those of programming languages, defining higher-level bindings involving whole objects - which may themselves be collections of objects). An emerging new ISO project on SQL Multi-Media already

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the 19th VLDB Conference, Dublin, Ireland, 1993

has a base proposal for an SQL3 Text type to address the Full Text area, and proposed SQL3 type definitions for the Spatial Data area are expected shortly.

Rather than attempt a complete overview of SQL3, this paper will go into some depth in examining the implications of the type system for the treatment of collections of objects. To set the stage, section 2 gives a brief outline of the SQL3 type system. In section 3, the relational collection type Table is generalized into a table of objects. Although this functionality has now been built directly into the language, it will be shown that the table concept is no longer primitive in SQL3, but is also expressible by means of an SQL3 type template (parameterized type). Section 4 discusses the functionality of Sets and Lists, and shows how they may be defined economically as subtypes using the SQL_Table template. Section 5 summarizes current status and open questions.

2. Outline of SQL3 Type System

The main feature of an object model, by comparison with the relational model, is the richness of its type system. Given some predefined base types, the relational model imposes a strict (and often healthy) discipline in allowing only one kind of composite entity to be constructed from these, namely the table, a collection of first-normal-form rows formed from elements of the base types (or simple domains defined over the base types). A row often represents a real-world object like an employee or department, and is indeed a simple case of an object in the current software sense.

2.1 Types and Tables

Since SQL3 needs to enrich the type system, and the type of a row is implicit in a relational table definition, a good place to begin is by making it possible to create a named type separately from a table, although with almost identical syntax to **Create table**:

```
Create type Address
  ( number char(6),
    street char(30),
    aptno integer,
    city char(30),
    state char(2),
    zip integer );
```

One use of such a type is as a column data type:

```
Create table People
  ( name char(30),
    address Address,
    birthdate date );
```

The table People is our first non-relational example - it has an attribute of a composite type (by any reasonable definition of "composite").

Now it is easy to see how to extend the type system with the features necessary for it to qualify as an object system:

```
Create type Person
  ( name char(30),
    address Address,
    function age (p Person)
                    returns interval day;
             return ... /* flattering expression */ ;
    function set_age(p Person,
             d interval day) returns Person;
       begin set p.birthdate = current_date - d;
             return p;
       end ;
    private
     birthdate date );
```

Some (or all) of the attributes can be encapsulated, as birthdate is here, by being preceded by **private**. This means that the birthdate is directly accessible only internally to this type definition. (Subsidiary functions called only from function bodies in the type can also be encapsulated by specifying them as private.) **Protected** is also specifiable, as in C++, to allow access both within the type definition and within its subtype definitions. A **public** or **private** tag can precede any attribute or function of the type definition, and is effective until the next tag. The default on the first member is **public**.

Create type also has an **operators** clause to allow the identification of "friend" functions that are given the same accessibility rights as functions specified within the type definition.

The similarity of a SQL3 type definition to a simplified C++ class definition [10] is obvious and intentional, since mapping between these languages is likely to be of fundamental importance as the decade progresses.

The word "type" is used in SQL3 in preference to "class", which in the database arena has been overloaded to refer to a type, or a collection of instances of a type, or both. In fact, these user-defined types are commonly referred to in SQL3 as abstract data types (ADTs), although they go beyond the basic ADT concepts in supporting inheritance and polymorphism.

Inheritance is provided by an **under** clause:

```
Create type Employee under Person
        ( emp_no char(10),
          dept Department );
```

Multiple inheritance is supported, with a conservative approach deemed wise in database schemas - name conflicts must be resolved by renaming inherited elements as part of the **under** clause. Dynamic polymorphism is applied as part of overloading rules that unify the treatment of specialization in the type hierarchy with overloading of a routine name across multiple arguments. Space precludes fuller discussion of these features here, since they are not crucial to what follows.

2.2 Object identifiers

By default, ADTs are defined **with oid**, i.e. their instances have system-managed object identifiers, generated implicitly on creation of the instance. Formally, an instance of such a type is a pair <oid, value>, where only the value part participates in comparisons and assignments.

In the example above, where the dept attribute of an Employee had type Department, this would imply that the value of the attribute was a reference to a Department instance, i.e. a copy of its oid (assuming that Department was a normal ADT with oid). In SQL3, reference values are constrained pointers, working very much like foreign keys, with the usual SQL referential integrity checking applicable to them when used for persistent objects.

Two optimizations of object identity are provided in creating a type.

First, if a type has an oid for internal references within the database, it may still be created **with oid not visible**, meaning that the oid cannot be extracted outside the database, so that the DBMS need not manufacture an oid that it will guarantee indefinitely, but can use something like a rowid. The default is **with oid not visible** for a top-level type, and subtypes inherit the "strongest" specification from their supertypes, or may strengthen what they inherit.

Second, if a type need never be used as a reference type, it may be specified to be **without oid**. This has another important implication. For example, if the type Address had been specified by

```
Create type Address without oid
  ( number char(6),
    street char(30),
    aptno integer,
    city char(30),
    state char(2),
    zip integer );
```

then in

```
Create table People
  ( name char(30),
    address Address,
    birthdate date );
```

a value in the address column would be an actual Address instance (much like an **expanded** instance in Eiffel [11]) rather than a references to an Address.

2.3 Type templates, collections, and distinct types

Type templates are supported, i.e. parameterized types, with similar syntax to Create type:

```
Create type template MySet(T type)
    ( ...
    );

Create table People
  ( name char(30),
    addresses MySet(Address),
    birthdate date
    );
```

The use of MySet(Address) creates a type from the template by substituting the type Address for T throughout the body of the template. Multiple parameters are permitted, and are not restricted to being type parameters.

Collection types like this are a common use of type templates, as will be illustrated later in this paper.

Distinct types are strongly typed typedefs:

```
Create distinct type Kilometres as integer;
Create distinct type Kilogrammes as integer;
```

The strong typing implies that Kilometres and Kilogrammes cannot be assigned to each other.

2.4 Attributes

Attributes may be either stored or virtual.

Stored attributes (like columns in tables) do double duty:

- They provide a neater notation than pairs of get and set functions
- They define both interface and representation

To retain the neat notation, and still have encapsulation (representation-independence in the interface), SQL3 also offers **virtual** attributes:

- The attribute notation now makes no commitment to representation
- Users of the interface are unaffected by change of representation

For example, "age" can be inserted as a virtual attribute into our previous definition of the Person type:

```
   Create type Person
( name char(30),
  address Address,
   age virtual,
  function age (p Person)
                    returns interval day;
             return ... /* flattering expression */ ;
  function set_age(p Person,
            d interval day) returns Person;
     begin set p.birthdate = current_date - d;
              return p;
      end ;
  private
   birthdate date );
```

This uses the default naming convention. In full, we could write

```
age updatable virtual get with foo set with bar,
```

where foo and bar are any functions defined in the type with appropriate parameter and return types. It is also possible to define a **read only virtual** attribute having a **get with** function only.

With the introduction of virtual attributes, attributes do not break encapsulation. Of course, one could always make all attributes private and only get the neat notation inside method bodies. But now the attribute notation can be used in the public interface orthogonally to the representation.

3. Tables and Type Templates

In order to capture the functionality of an SQL-92 table in a type template SQL_Table(T type), the type parameter may be defined to be a simple type with attribute names and types corresponding to the column names and types of the table.

The main challenge is to define the SQL_Table template with functions that provide equivalent power to the existing SQL Data Manipulation Language (DML) statements on tables - **Select, Insert, Update,** and **Delete.** (There is generally room for some debate as to exactly which operations should be grouped with a given type or template, due to the asymmetry of the ADT model - a function with multiple parameters whose types are different ADTs is a candidate for being placed in any of those ADTs. For example, is **Grant select on** Employees **to** dbeech; an operation principally on the table, or on the grantee, or on a privilege manager object?)

The existing DML can then be regarded as syntactic sugar for invoking these functions. If a user were allowed to overload the definitions in certain ways and write new function bodies, the syntactic sugar would still be available, e.g. an **Update** statement on a complicated view could be given user-defined semantics to make it valid. Likewise, if SQL_Lists and SQL_Sets are definable as subtypes of SQL_Tables, they can inherit all the DML functionality and syntactic sugar currently applicable to tables.

The outline structure will be as follows:

```
Create type template  SQL_Table ( T type)
    ( equals none,
      less than none,

      constructor function SQL_Table ...,
      destructor function Remove_SQL_Table ...,

      function SQL_xxx ...,
      function SQL_yyy ...,
      ...
     );
```

The naming convention of the SQL_ prefix in SQL_Table and SQL_xxx allows names like xxx to be statement keywords without causing clashes with reserved words.

All of these functions are, by default, **public.**

Equals and **less than** are **none,** since SQL does not extend its builtin comparison operators to whole tables - their most general operands are <row value constructor>s.

The goal is to complete the signatures of the functions - it is not necessary to write out their bodies, since the semantics of these functions are already well defined in the standard, when they are invoked with the usual statement syntax.

3.1 SQL_Table constructor

The first thing to note about the constructor is that it is not equivalent to the more powerful **Create table,** which not only constructs a table object but makes it persistent and

gives it a name that will be recognized by the built-in name resolution of SQL. There is currently no way with the ADT mechanism to build this functionality out of anything else - **Create table** is the primitive for creating a persistent named data object, and this is the only kind of data object that SQL3, with its relational heritage, currently allows to be created at the top level.

A similar situation exists in the procedural language with **Declare** tvar Ttype; --- the naming and lifetime control of the variable are language primitives, and the constructor is invoked as part of the whole creation process. Thus the constructor is neutral as to whether the constructed instance is to be persistent or transient, and as to how it is named. This is an advantage, since the normal case is to construct (i.e. initialize) instances in the same way whether they are transient or persistent, and the same default constructor can be used. In exceptional cases, different constructors can be defined and invoked explicitly.

The initialization carried out by **Create table** is very simple (bearing in mind that it is using an already created type). It makes a table descriptor that references the type descriptor, and then creates an empty table. In the ADT model, a generated type SQL_Table(SomeType) will have its descriptor pointing to the descriptor for SomeType, and the constructor makes the empty instance.

A question that naturally arises is what stored attributes could be used to represent the state of an SQL_Table. Since the type template is defining an SQL table out of more primitive components, it cannot itself use a table. Fortunately, the reference concept is available as a primitive that supports the building of arbitrary collections, and a table can be constructed by chaining the rows together, in any of several familiar programming styles, e.g. using the following SQL_Chain template, where SQL_Chain and U are **with oid** and hence imply the use of references:

```
Create type template SQL_Chain (U type)
    ( next gen_type,
      item U);
```

(The symbol **gen_type** is SQL3's way of representing the type generated when the type template is provided with an actual parameter.)

```
Create type template SQL_Table (T type)
    ( ... ,
      private
        rows SQL_Chain(T);
```

Since this is encapsulated in an abstract specification, an implementation is free to optimize! Yet for a persistent table, this use of a chain with indirection is not so far removed from a practical implementation using an index to disk blocks which may not be contiguous.

By default, the *rows* attribute will be initialized to **null**, so the constructor has an easy time. In fact, there is no need for an explicit constructor, since the implicit constructor assigns defaults.

If the type passed to the type parameter T has been defined (as an optimization) to be **without oid**, this removes the above possibility of constructing aggregations of instances of T out of other primitives, and the representation of such tables becomes primitive. However, it is interesting to have observed that when references are available, tables become non-primitive. References essentially provide a means of being able to link objects together by a join, without the objects needing to be in tables. The **private** representation of an SQL_Table will in any case not be used formally in defining the semantics of the functions corresponding to DML statements, so these functions will apply to all tables, regardless of whether the "row types" of the tables are **with** or **without oid**.

3.2 Remove_SQL_Table

Just as the constructor was only a part of the semantics of **Create table** (or of the creation of a transient table), so the destructor is only a part of the semantics of **Drop table** (or of the destruction of a transient table). The significant semantics include destroying the chain and all its referenced "items". As with the default constructor, the criterion for the functionality to be included in the default destructor is that it should apply to both persistent and transient tables.

3.3 DML functions

Deciding how to structure the functions corresponding to DML statements is much like designing a call level interface (CLI) to invoke SQL functionality. The same questions arise as to how to:

- pass expressions that need to be evaluated inside a function, especially when they may contain <variable name>s to be bound;
- determine the result type of a query (using templates, the result type will certainly be of the form SQL_Table(T), but what is T?);
- iterate over the individual rows of a table returned by a query.

Since these problems have already been solved in SQL-92 in defining Dynamic SQL (and closely corresponding call

interfaces have been implemented), there is no need to look further for one viable approach. This may not be the only useful set of functions, since they are designed for dealing with a fixed (although growing!) language, and they assume that the implementor of the functions will cheerfully parse SQL statements passed as strings. For purposes of extensibility by user-defined overloading of the functions, it may be more convenient to define some functions that accept arguments already structured into parse trees. Fortunately the two approaches are not mutually exclusive, since the type template could contain both sets of functions - there is no requirement that the functions provided should be a minimal set without any semantic overlap, so long as they are semantically consistent.

The solution offered here will take an intermediate course in the interests of simplicity, without precluding future extensions of the template for other purposes. The approach is to assume that the SQL statement has already been parsed so far as to recognize its initial keyword such as INSERT, so that a SQL_Insert function can correspond to this; and that a table expression has been evaluated so far as to determine a table to serve as the principal operand.

The existing descriptor mechanism for dynamic SQL will be employed. There is not even any need to define an SQL_Descriptor type, since it suffices to represent descriptor names as character strings and use the existing allocate, deallocate, get, and set statements on descriptor names, once the SQL_Table operation semantics have been defined (informally) to work with descriptors.

Note that although some of the mechanisms of dynamic SQL come in useful, this paper is not attempting to capture the full functionality of dynamic SQL in an ADT. Because the general structure of dynamic SQL handles any SQL statement except for some special treatment of **Select** and cursors, it might be conceptualized in terms of SQL_Source_Statement and SQL_Prepared_Statement types. Those prepared statements that turn out to be DML on tables would then, when executed, cause invocation of the functions defined on SQL_Tables.

3.3.1 Input values

Following the style of dynamic SQL and the accompanying treatment of input data, the SQL statement passed as a string to the operations SQL_Insert, etc., will contain question marks in the positions where the input values are to be substituted in order. The actual arguments are then provided as data values in a descriptor, whose name is passed as a character string argument to the DML operation.

3.3.2 Output values

SQL_Select on an SQL_Table(T) returns an SQL_Table(U), where U may be arbitrarily different from T, and indeed may be a type previously undefined. Here there is a very strong motivation to define SQL_Select to be a function returning an SQL_Table, since nested queries are then treated very naturally. Since the SQL_Table type template cannot know how to define the result type in terms of its T parameter (and not even a particular generated type can know, since SQL_Select can accept any SELECT statement as a string argument, and could return a table of any type), we can do no better than exploit Object as a supertype of all ADTs, and return an SQL_Table(Object). The actual type of table returned by a particular query will be specified in a descriptor, and this information can be fed into the semantics of further processing of the query result.

3.3.3 Iteration

The classical approach to iteration over a collection of objects, e.g. in Smalltalk-80, is to have an Iterate operation on the collection type, that takes a function as an argument and applies it to each element of the collection in turn (in some undefined order if the collection is unordered). It is as though the iteration is being carried on "inside" the collection object, using a call-back. Until SQL3 has function parameters, this approach is not available.

The alternative is to extract the elements in turn from the collection in order to operate on them "outside" the collection. This is the cursor approach, where **Open** materializes an SQL_Table and **Fetch** is the sequential extractor. Since the use of cursors is at the periphery of SQL where it interfaces to languages without set-at-a-time operations, this paper does not attempt to capture the cursor functionality by using ADTs. However, questions about the relationship between cursors and tables have received some thought. For example, are a cursor and a table best thought of as different specializations of an SQL_Relation type, in which different manipulative operations are provided to augment the definition inherited from SQL_Relation? The definition itself could take the general form of either the body of SQL_Select, or of SQL_Fetch -- - in mathematical terms, either of something akin to a membership predicate, or of a generator, where (over finite domains) one function can always be derived from the other, although optimizers have to worry about strategies for doing this. (There is some relevant discussion in [8].) This definition could be made **protected** in SQL_Relation, so that the subtypes can use it, but don't have to expose its functionality in their public interface if they don't want to --- a table would only want to expose it if it were SQL_Select, and a cursor if it were SQL_Fetch. A base table has a simple SQL_Fetch that runs along the chain of its stored representation.

3.4 SQL_Table Type Template

Many other functions applicable to tables are provided in SQL, either with a function or an operator syntax. Function signatures for these are rather straightforwardly included in the SQL_Table template. The template as currently specified is as follows:

```
Create type template SQL_Table
                              (Element_Type type)
(
  cast(gen_type as Element_Type
      with Table_to_Element);,
  cast (gen_type as  SQL_Set( Element_Type )
      with Distinct);,
  cast (gen_type as  SQL_List( Element_Type )
      with Table_to_List);,
  cast (SQL_Empty_Table as gen_type
      with Empty_Table_to_Table);,

  function SQL_Insert (tabref gen_type,
              stmt character varying(max_stmt_length),
      descr_name character varying(max_name_length));,
  function SQL_Select (tabref gen_type,
              stmt character varying(max_stmt_length),
      descr_name character varying(max_name_length))
      returns SQL_Table(Object); ,
  function SQL_Update (tabref gen_type,
              stmt character varying(max_stmt_length),
      descr_name character varying(max_name_length));,
  function SQL_Delete(tabref gen_type,
              stmt character varying(max_stmt_length),
      descr_name character varying(max_name_length));,
  function In(table gen_type, elem Element_Type)
      returns Boolean ;,
  function Exists(table gen_type)
      returns Boolean  ;,
  function Unique(table gen_type)
      returns Boolean  ;,
  function For_Some(table gen_type,
              pred character varying(max_pred_length),
      descr_name character varying(max_name_length))
                  returns Boolean ;,
  function For_All(table gen_type,
              pred character varying(max_pred_length),
      descr_name character varying(max_name_length))
                  returns Boolean ;,
  function Average(table gen_type)
      returns Element_Type ;,
  function Maximum(table gen_type)
      returns Element_Type ;,
  function Minimum(table gen_type)
      returns Element_Type ;,
  function Sum(table gen_type)
      returns Element_Type;,
  function Count(table gen_type)
      returns count_type ;,
  function Distinct(table gen_type)
      returns SQL_Set(Element_Type);,
  function Union(table gen_type, table2 gen_type)
      returns gen_type;,
  function Union_All(table gen_type,
              table2 gen_type) returns gen_type;,
  function Except(table gen_type, table2 gen_type)
      returns gen_type;,
  function Except_All(table gen_type,
              table2 gen_type) returns gen_type;,
  function Intersect(table gen_type,
              table2 gen_type) returns gen_type;,
  function Intersect_All(table gen_type,
              table2 gen_type) returns gen_type;,

  private:
  function Table_to_Element(table gen_type)
      returns Element_Type;,
  function Table_to_List(table gen_type)
      returns SQL_List( Element_Type );,
  function Empty_Table_to_Table(target gen_type)
      returns gen_type;
);
```

4. Sets and Lists

The groundwork has been laid now for using the type template approach to define functionality for sets and lists which stays very close to the existing language for tables (multisets). This has been an active area of SQL3 development during 1992 and 1993 [3]-[7], [9].

Already, the SQL_Table template type increases flexibility in the use of tables in SQL, since it is possible, given any type *T*, to use the type SQL_Table(*T*) as the data type of a column, attribute, variable or parameter. This is in addition to being able to create a top-level named persistent instance of an SQL_Table by using the familiar syntactic sugar **Create table** tab **of** *T* . The DML statements can be applied to instances of SQL_Tables, however they were created or nested.

What follows will make it possible to create and use sets and lists in both the above ways. Top-level sets and lists may be created by **Create set** and **Create list**, with syntax otherwise identical to **Create table**.

Although some of the technicalities of this paper may seem rather intricate, they are needed only in the bootstrapping process of building collection types by the extensibility

mechanisms in the language, so that very little actual language extension is needed. These mechanisms are encapsulated as far as the end user is concerned, and the resulting application language for sets and lists is similar to that for tables.

4.1 Sets

Sets are treated as differing from tables only in excluding duplicate members (rows). Working from the SQL_Table template, an SQL_Set template can be defined **under** it with an additional uniqueness constraint, otherwise inheriting all the behavior of tables.

```
Create type template SQL_Table
                    (Element_Type type)
(          ...,
  function SQL_Insert(...)...,
  function Unique(tab gen_type),
          /* the <unique predicate> */
  ...
);

Create type template SQL_Set
                    (Element_Type type)
          under SQL_Table(Element_Type)
( check(Unique(value)),
  function Subset(set gen_type,
          set2 gen_type) returns Boolean;
);
```

The SQL_Table template contains the function Unique to correspond to the <unique predicate>. This is then inherited by SQL_Set, and used in a **check** constraint. A Subset function has also been introduced.

The keyword **value**, as in domain constraint definitions, provides a way of naming, not the type, but the whole instance of the type or domain being referenced in an expression that is being evaluated (it is often called **self** or **this** in programming languages). When a particular attribute or column is referenced in the check constraint, the instance name is usually implicit; and when the expression occurs within a table definition, the table name is in any case the instance name, not a mere type name. However, in a constraint definition of an ADT (including a template, which generates ADTs), there is no instance name, and one is sometimes needed.

Since all the SQL DML statements are just syntactic sugar for the corresponding functions like SQL_Insert on an SQL_Table, and these functions are inherited by an SQL_Set, the DML statements are immediately applicable to sets. By analogy with **Create table**, **Create set** is introduced to make a named persistent instance of an SQL_Set:

```
Create set CityZip
  ( city char(40),
    zip integer);
```

Then proceed with existing DML:

```
Insert into CityZip values('Los Altos', 94022);
Insert into CityZip values('Menlo Park', 94025);
Insert into CityZip values('Atherton', 94025);
Insert into CityZip values('Palo Alto', 94301);
Insert into CityZip values('Palo Alto', 94302);

Select zip from CityZip where city='Palo Alto';
```

So far, all that this illustrates is a slightly neater syntax for creating a table with a uniqeness constraint across all columns (the weakest uniqueness constraint, implied by uniqueness across any subset of columns). The behavior is just the same, including the exception condition raised on attempting to insert a duplicate - *integrity constraint violation.*

However, a framework is now in place for systematic consideration of other functions that might be added to templates like SQL_Table and SQL_Set, for example to increase the power of search conditions in queries.

And already there is the payoff from the sugar-free syntax, e.g for a set-valued attribute:

```
Create set CityZips
  ( city char(40),
    zips SQL_Set(integer));

Insert into CityZips
    values('PaloAlto',
      set(94301,94302,94303,94304,94306));
```

The **set** of insert values in parentheses looks as though it is serving as a constructor of a set. However, existing constructors on ADTs have to take a fixed number of arguments, and even with overloading (and extreme tedium) there can only be a finite number of constructors to cater for argument lists of different lengths. Recursive use of something like a 2-place cons function quickly palls - cons(94301, cons(94302, cons(94303, cons(94304, cons(94306))))) - so some primitive syntax is needed for constructing a single collection from an arbitrary number of elements. The extension adopted is to prefix the parenthesized lists by a <table type> of **table**, **set**, or **list**, corresponding to the trio of statements **Create** <table type> T.

A single syntactic primitive that used just plain parentheses was also considered, although that construct is often used in SQL to specify an ordered tuple whose elements may have

different data types, whereas list elements have a common data type. Further investigation might show that such a tuple could be implicitly cast to a table, set, or list where the context required it. Even so, it seemed preferable to have a simple explicit way of indicating what is intended in the case of these fundamental collection types, as with the corresponding **Create** statements.

The notations **table()**, **set()**, and **list()** represent an empty table, set, and list. These are instances of newly defined ADTs SQL_Empty_Table, SQL_Empty_Set, and SQL_Empty_List. Casts are defined from these to empty instances of SQL_Table(*T*), SQL_Set(*T*), and SQL_List(*T*), respectively, for any type *T*. (These casts have to be defined in the type template that is the target of the cast, since SQL_Empty_Table, for example, does not know anything about *T* in SQL_Table(*T*). By placing the cast in the template, a separate cast is defined for each generated type using a particular *T*.)

A frequent use of sets is to make sets, not only of objects, but of references to objects, e.g. a SQL_Set(Person). Since reference types are base types, this again is a collection of a base type.

Further consideration is being given to the extended syntactic sugar for SELECT and UPDATE that might be offered for these nested collections (SQL_Tables as well as SQL_Sets), but that is beyond the scope of the present paper.

4.2 Lists

Lists are treated as differing from tables only in maintaining an ordering of the elements (rows) independent of their state. Lists lend themselves to a similar approach to that used for sets. However, instead of an additional constraint, overloading of the inherited functions is needed to make use of the ordering, together with some new functions:

```
Create type template SQL_List
                                (Element_Type type)
            under SQL_Table(Element_Type)
  ( function SQL_Insert(...)... /* point of insertion
                                (at end by default ) */
    function SQL_Select(...)... /* ordering of result */
    function SQL_Update(...)... /* change ordering */
    function SQL_Delete(...)... /* maintain order */
    function Element (list gen_type, pos integer)
            returns Element_Type ...
    function Position (list gen_type,
            elem Element_Type) returns integer ...
  );
```

As with sets, a keyword is introduced for creation of persistent named SQL_Lists, following the pattern of **Create table**:

```
Create list Incoming_calls
    ( line integer);
Insert into Incoming_calls values(4) /* at end of
                                        list */
    alias joe;
```

For general positioning, the Element and Position functions are provided:

```
Select line
from Incoming_calls ic
where position(Incoming_calls,ic)=1;

Declare line_no integer;
Set line_no = element(Incoming_calls,1);
```

If a query has a list in its **from** clause, its result will be a list computed in order from its elements. If there are multiple lists in the **from** clause, the order is that of the Cartesian product formed recursively by iterating most slowly over the leftmost list, e.g. **from** L1,L2 will begin by taking the first element of L1 with each element of L2, etc. Unordered tables and sets are taken in arbitrary order. An ordered set could be modeled by an SQL_Ordered_Set template, using multiple inheritance from SQL_Set and SQL_List (Fig. 1), but the value of this appears marginal.

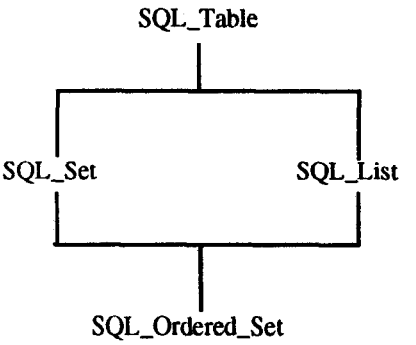

Figure 1. Possible collection hierarchy

For **Insert**, an option is provided to insert before or after a member, specified by a subquery returning a single row:

```
Insert into Incoming_calls values(1)
            before element where oid = joe;

Insert into Incoming_calls values(5)
            after element where line = 1;
```

The specified element must exist in the set --- except possibly when the set is empty, so that no special initialization logic is required, e.g. a loop around **Insert** ... **after** could begin by inserting after position 0.

With the form of **Insert** that contains a <query expression> producing multiple rows, this is treated, compatibly, as successively inserting single elements and not as inserting the collection as a single element. The first insertion takes place at the specified position, and the other elements follow in order.

Moving now to **Update**, the syntactic sugar needed to deal with nested lists as attribute values will be inherited from SQL_Table, once it has been defined there for collections in general. The additional refinement on a List is that a new kind of update is provided to reorder its elements:

```
Update Incoming_calls
        move before element where oid = joe
where sex(caller(line))='F';
```

3.3 Relationship to mathematical sets and lists

Mathematically-inspired treatments of sets and lists regard them as immutable objects with self-defining identity. There is only one empty set, and when two people independently talk about a set {1,2,3}, they are talking about the same set. Computation with these sets and lists is purely functional, i.e. assignment-free, and a function to insert an element into a set does not alter that set, but rather produces a new one (or, more strictly, the denotation of a set which may already exist, or may even have always existed in some Platonic sense). Typical functions are IsEmpty, Union, and Intersection.

A system like this may however be supplemented by an updatable memory, provided by a general assignment operator that does not allow partial mutation of its target, but completely replaces its value. Thus to insert element X into set S, instead of Insert(S,X) we have

```
set S = Union(S,set(X))
```

so then the optimization challenge is to avoid lots of temporary sets, if that is the only way this state change can be expressed.

Computing systems generally take a more relaxed approach to the points at which state changes can occur. This seems to be because the intuitive model of a memory, as something whose state can be partially changed while retaining its identity, is usually simpler to work with. This is especially true with a large and intricate memory such as a human brain, or the state of the universe, which are subject to many concurrent state changes. It applies also to relatively simple collections, such as a football team which has an identity even though the playing staff may change.

Thus for both conceptual and performance reasons, a hybrid situation is the norm --- functions are often designed to be free from side-effects, but they do not have to be, and indeed "procedures", expressible in SQL3 ADTs as functions with negligible returned values, are usually intended to effect state changes. A database in particular is designed to be selectively updatable, rather than having each change produce a complete new database. An SQL_Table has update operations such as SQL_Insert on it, besides having pure functions such as Exists (which means IsNotEmpty), Union and Intersect.

Functions like Union (and UnionAll) may be added to SQL_Table to correspond to existing functionality. Call this a Hybrid Multiset (HM), and let a Pure MultiSet (PM) have only the pure functions of the HM interface. (Hybrid is not entirely a derogatory word --- to a nurseryman, say --- or to Webster: "A blend of two diverse cultures or traditions".)

Then SQL_Set as defined above inherits both the pure and the impure functions from HM, and becomes the Hybrid Set (HS), rather than a Pure Set (PS). Similarly for SQL_List. There could be a corresponding hierarchy of pure collections. One way of relating the hierarchies by multiple inheritance is shown in Fig. 2.

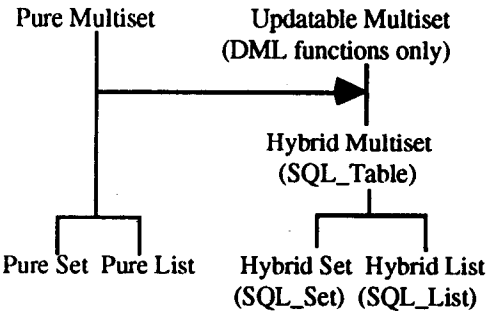

Figure 2. Pure and hybrid collection types

Is it worth introducing these other templates, and connecting them in this way? Of course, HM is really symmetrically descended from PM and UM, and the other hybrids could be derived from their pure counterparts, as illustrated in Fig. 3.

Is this "mixin" style preferable?

Although initially attractive, both these approaches lead to further complications, because one really wants the

combination of them to give true "is a" semantics - an HS is an HM in one picture, and an HS is a (specialization of a) PS in the other.

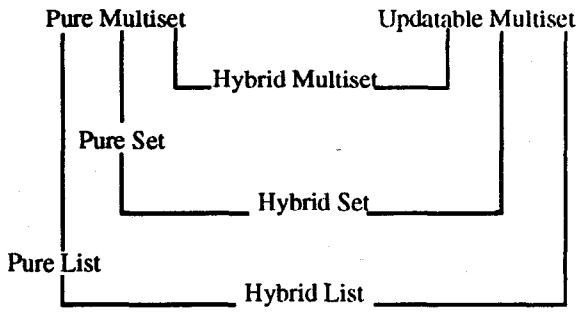

Figure 3. An alternative hierarchy

Therefore, the simpler path has been adopted, offering only the Hybrid Multiset, Hybrid Set, and Hybrid List, as in Fig. 1. The user can then choose whether to work with the the pure or impure functions or with both.

5. Conclusion

Besides the direct availability of the SQL_Table, SQL_Set and SQL_List type templates for generating collection types (now with syntax multiset(*T*), set(*T*), and list(*T*)) to be used as the data types of columns, attributes, variables and parameters, a few basic language extensions are visible to the SQL3 user:

- The **Create table** statement now becomes **Create {table | set | list}...**, with the remainder of its syntax unchanged.
- **{table | set | list}** may also be used as prefix to a parenthesized list of any number of elements separated by commas, to form a collection.

For sets in particular:

- The Subset function is introduced.

For lists in particular:

- Functions Element and Position are introduced to work with ordering in lists, together with Head, Tail, and Append for Lisp style of usage.
- **Insert** acquires an extra option to control position of insertion **before** or **after** an element specified by a query (default is at end of list).
- **Update** acquires a **move** option to update the ordering within a list.

The use of type templates for defining sets and lists provided formal justification for their inheriting the functionality of tables.

Open questions include how to provide improved syntactic sugar in the DML for operations on nested collections of objects, and how precisely to conceptualize views and cursors as type templates.

Acknowledgements

I am grateful to Ken Jacobs and Andy Mendelsohn for discussions clarifying the relationship of the table template functions to dynamic SQL and corresponding call interfaces.

References

[1] Jim Melton (ed): (ISO/ANSI Working Draft) Database Language SQL3, ANSI X3H2-93-091 and ISO DBL-YOK 003, February, 1993.

[2] David Beech and Cetin Ozbutun, "Object Databases as Generalizations of Relational Databases", ANSI X3H2-90-412, December, 1990.

[3] Amelia Carlson, "LIST, SET, and BAG as Type Templates", ANSI X3H2-92-055rev1 (ISO DBL OTT-9), January 2, 1992.

[4] Jonathan Bauer, Mike Kelley, Krishna Kulkarni, Jim Melton, "Sets, Multisets and Lists", ANSI X3H2-92-003rev1 (ISO DBL KAW-71), February 24, 1992.

[5] Len Gallagher, "SQL as Integrator of Non-SQL Repositories", ANSI X3H2-92-51 (ISO DBL KAW-98), December 19, 1991.

[6] David Beech, "Tables as Type Templates", ANSI X3H2-92-138rev1 (ISO DBL CBR-9), June 12, 1992.

[7] David Beech, "Sets and Lists", ANSI X3H2-92-220rev1 (ISO DBL CBR-56), October 15, 1992.

[8] David Beech, "Intensional Concepts in an Object Database Model", Proc. OOPSLA '88, ACM SIGPLAN Notices, 23(11), 1988.

[9] Amelia Carlson, Rafiul Ahad, Bill Kent, "SQL3 OO Features", ANSI X3H2-92-066 (ISO DBL YOK-35), January , 1993.

[10] Margaret Ellis and Bjarne Stroustrup, The Annotated C++ Reference Manual, Addison Wesley, Reading, 1990.

[11] Bertrand Meyer, Eiffel: The Language, Prentice Hall, New York, 1992.