Bulletin of the Technical Committee on

Data Engineering

December, 1994 Vol. 17 No. 4

Letters

Letter from the Editor-in-Chief . . . . . *David Lomet* 1
Letter from the Special Issue Editor . . . . . *Eliot Moss* 2

Special Issue on Emerging Object Query Standards

ODMG-93: The Object Database Standard . . . . . *Francois Bancilhon and Guy Ferran* 3
Object Technology and SQL: Adding Objects to a Relational Language . . . . . *Jim Melton* 15
A Comparison of Object Models in ODBMS-Related Standards . . . . . *Frank Manola and Gail Mitchell* 27
Evaluation of the Object Query Service Submissions to the OMG . . . . . *David L. Wells and Craig W. Thompson* 36

Conference and Journal Notices

1995 Data Engineering Conference Notice 46
Transactions on Knowledge and Data Engineering 47
Ride 95 Workshop Notice . . . . . back cover

Editorial Board

Editor-in-Chief
David B. Lomet
Microsoft Corporation
One Microsoft Way, Bldg. 9
Redmond WA 98052-6399
lomet@microsoft.com

Associate Editors
Shahram Ghandeharizadeh
Computer Science Department
University of Southern California
Los Angeles, CA 90089

Goetz Graefe
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052-6399

Meichun Hsu
EDS Management Consulting Services
3945 Freedom Circle
Santa Clara CA 95054

J. Eliot Moss
Department of Computer Science
University of Massachusetts
Amherst, MA 01003

Jennifer Widom
Department of Computer Science
Stanford University
Palo Alto, CA 94305

The Bulletin of the Technical Committee on Data Engineering is published quarterly and is distributed to all TC members. Its scope includes the design, implementation, modelling, theory and application of database systems and their technology.

Letters, conference information, and news should be sent to the Editor-in-Chief. Papers for each issue are solicited by and should be sent to the Associate Editor responsible for the issue.

Opinions expressed in contributions are those of the authors and do not necessarily reflect the positions of the TC on Data Engineering, the IEEE Computer Society, or the authors' organizations.

Membership in the TC on Data Engineering is open to all current members of the IEEE Computer Society who are interested in database systems.

TC Executive Committee

Chair
Rakesh Agrawal
IBM Almaden Research Center
650 Harry Road
San Jose, CA 95120
ragrawal@almaden.ibm.com

Vice-Chair
Nick J. Cercone
Assoc. VP Research, Dean of Graduate Studies
University of Regina
Regina, Saskatchewan S4S 0A2
Canada

Secretry/Treasurer
Amit Sheth
Department of Computer Science
University of Georgia
415 Graduate Studies Research Center
Athens GA 30602-7404

Conferences Co-ordinator
Benjamin W. Wah
University of Illinois
Coordinated Science Laboratory
1308 West Main Street
Urbana, IL 61801

Geographic Co-ordinators
Shojiro Nishio (**Asia**)
Dept. of Information Systems Engineering
Osaka University
2-1 Yamadaoka, Suita
Osaka 565, Japan

Ron Sacks-Davis (**Australia**)
CITRI
723 Swanston Street
Carlton, Victoria, Australia 3053

Erich J. Neuhold (**Europe**)
Director, GMD-IPSI
Dolivostrasse 15
P.O. Box 10 43 26
6100 Darmstadt, Germany

Distribution
IEEE Computer Society
1730 Massachusetts Avenue
Washington, D.C. 20036-1903
(202) 371-1012

Letter from the Editor-in-Chief

My letter in this issue has to serve multiple purposes. I want to briefly introduce and comment on the issue, as I usually do. But much has happened with respect to the editorial staff of the Bulletin, at least one aspect of which impacts Technical Committee members. I will begin with that.

Changes in Bulletin Procedures

An inspection of the back inside cover will reveal that I have a new employer. As of January 1, 1995, I am employed by Microsoft and work in Redmond, Washington. That means that all communication about editorial matters for the Bulletin and all problem reporting should be sent to me at my new email address-

lomet@microsoft.com

Mark Brown, Digital's Cambridge Research Lab director has very generously agreed to permit CRL to be the continuing source for electronic distribution of the Bulletin until a more permanent arrangement can be made. So, please note that requests for copies of the Bulletin should continue to be sent to

tcdata@crl.dec.com

Electronic membership applications will also continue to be handled via this CRL address.

Digital, and CRL in particular, have played a very significant role in the successful launching of electronic distribution for the Bulletin. CRL lab members have helped with setting up the latex formattting, establishing the Bulletin email server, and providing internet connnectivity and ongoing systems support. I want to thank them for this very generous contribution to the success of the Bulletin.

Other Editorial Staff Changes

Mine was not the only change of employer among our editors. Mei Hsu has left Digital to join EDS, and Goetz Graefe has gone from Portland State to join Microsoft in Redmond, Washington. I have enjoyed interacting with Mei while she was at Digital, and look forward to working with Goetz at Microsoft. I wish both Mei and Goetz well in their new jobs. Our technical community world seems very small at times like this.

About This Issue

The world of software is increasingly one in which products of different vendors are expected to interoperate relatively seamlessly. It is a world in which vendors are expected to conform to standards so that user applications do not break when switched from one system to another. These twin expectations can only be effectively realized via increased use of standards, be the standards the result of a standards body(de jure) or by the ubiquity of some vendor's offering (de facto).

Increasingly, it is object technology that provides the framework in which standards for interoperation and application portability are expressed. This is the focus of the current issue. Eliot Moss has brought together articles on the major standardization efforts, SQL3, OMG, and ODMG. To my thinking, this could not be more timely or important. These are *in progress* efforts that deserve scrutiny and comment from our community before they are set in concrete. The results of these efforts will surely impact the way in which we work in the future. So I urge you all to read this issue carefully. And to express your views to the appropriate authors.

David Lomet
DEC Cambridge Research Lab
lomet@microsoft.com

Letter from the Special Issue Editor

In putting together this issue of Data Engineering I aimed to find a topic of interest to practitioners, but also relevant to academics and researchers. Originally I set out to "survey" the state of the art in object database systems. What actually happened is probably more interesting, given that there have been a number of surveys of OODBs, both commercial and research, over the last five years. Such a survey might still be interesting, but the emerging standards in object data definition and querying are in some sense "where the action is" today. The collection of articles here, invited from people actively participating in the standards process, read almost like (technical) news articles (though we hope they do not date *too* rapidly!).

The first two articles describe two proposed OODB standards. One is from the Object Data Management Group (ODMG), a consortium of OODB vendors growing out of the Object Management Group's (OMG) broader efforts with respect to object standards and systems. The other is the evolving ANSI SQL standard proposal, generally known as SQL3. The third article compares and contrasts these two proposed standards, indicating where work is needed to bring them into conformance with each other (which is a goal both groups are now working towards quite actively). The final article discusses the OMG (not ODMG!) efforts to standardize Object Query Services, and critiques the various proposals OMG received, some of which rely on the ODMG or SQL proposals. Taken together, these articles discuss the technical challenges (and the benefits) of interoperability and conformance between all three efforts. It seems that we are in for some lively debates in the coming months and years!

In closing I thank the authors for their successful efforts to explain technical issues concisely and in a timely way, to the benefit of all of us.

Eliot Moss
University of Massachusetts at Amherst

ODMG-93: The Object Database Standard

Francois Bancilhon and Guy Ferran
O_2 Technology
2685 Marine Way-Suite 1220
Mountain View, California 94043
ferran@o2tech.o2tech.fr

Research on object databases started at the beginning of the 1980s and became very active in the mid 1980s. At the end of the 1980s, a number of start-up companies were created. As a result, a wide variety of products are now commercially available. The products have quickly matured after several years of market presence. Production applications are being deployed in various areas: CAD (computer aided design), software engineering, geographic information systems, financial applications, medical applications, telecommunications, multimedia, and MIS (management information systems).

As more and more applications were being developed and deployed, and as more vendors appeared on the market, the user community voiced a clear concern about the risk of divergence of the products, and expressed their need for convergence and standards. The response from the vendor community was both implicit and explicit. Implicit, because as the vendors understood more and more the applications and the user needs, the systems architecture started to converge and the systems to look alike. Explicit, because the vendors got together to define and promote a standard.

The Object Database Management Group (ODMG) was created in 1991 by five object database vendors (O_2, Object Design, Objectivity, Ontos, and Versant) under the chairmanship of Rick Cattell. It published an initial version of a standard interface, was joined by 2 more vendors (Poet and Servio) at the end of 1993, and produced a final revision in early 1994 [Catt94]. Thus, all object database vendors are active members of the group and are totally committed to comply to the standard in future releases of their products. For instance, the current release of O_2 [O_2] fully complies with the ODMG OQL query language, and the next release will comply with the ODMG C++ binding.

This standard is a major event and a clear signal for the object database market. It is clearly as important for object databases as SQL was for relational databases.

The ODMG standard is a *portability standard*, i.e., it guarantees that a compliant application written on top of compliant system X can be easily ported on top of compliant system Y. This is in contrast to an *interoperability standard*, such as CORBA (the Common Object Request Broker Architecture), which allows an application to run on top of compliant systems X and Y at the same time. Portability was chosen because it was the first demand of the OODB user community.

Because object databases cover more ground than relational databases, the standard covers a larger area than SQL does. An object database schema defines not only the data structure and types of the objects of the database, but also the *methods* associated with the objects. Therefore a programming language must be used to write these methods.

Instead of inventing "yet another programming language", ODMG strongly believes that it is more realistic and attractive to use existing object oriented programming languages. Moreover, during the last few years a huge effort has been carried out by the programming language community and by the OMG organization to define and adopt a commonly accepted *object model*. The ODMG contribution started from this state and proposed an object model which is a simple extension of the OMG object model.

This paper gives an informal presentation of the ODMG standard by presenting the data model, the query language, and the language bindings. The rest of this paper is organized as follows: Section 1 describes the data model, Section 2 introduces the C++ binding, and Section 3 presents OQL, the query language.

1 Defining an Object Database Schema

In ODMG-93, a database schema can be defined either by using the Object Definition Language (ODL), a direct extension of the OMG Interface Definition Language (IDL), or by using Smalltalk or C++. In this presentation, we use C++ as our object definition language. We first briefly review the OMG object model, then describe the ODMG extensions.

1.1 The OMG Object Model

The ODMG object model, following the OMG object model, supports the notion of class, of objects with attributes and methods, and of inheritance and specialization. It also offers the classical types to deal with dates, times, and character strings. To illustrate this, let us first define the elementary objects of our schema without establishing connections between them.

```
class Person{
   String name;
   Date   birthdate;

// Methods:
   Person();             // Constructor: a new Person is born
   int age();            // Returns an atomic type
};

class Employee: Person{ // A subclass of Person
   float salary;
};

class Student: Person{  // A subclass of Person
   String grade;
};

class Address{
   int number;
   String street;
};

class Building{          // A complex value (Address),
   Address address;      // embedded in this object
};

class Apartment{
   int number;
};
```

Let us now turn to the extensions brought by ODMG to the OMG data model: *relationships* and *collections*.

1.2 One-to-One Relationships

An object refers to another object through a *Ref*. A `Ref` behaves like a C++ pointer, but with more semantics. First, a `Ref` is a persistent pointer. However, a `Ref` allows referential integrity to be expressed in the schema and maintained by the system. This is done by declaring the relationship as symmetric. For instance, we can say that a `Person` *lives in* an `Apartment`, and that the `Apartment` *is used by* the `Person`, in the following way:

```
class Person{
   Ref<Apartment> lives_in   inverse is_used_by;
};

class Apartment{
   Ref<Person>    is_used_by inverse lives_in;
};
```

The keyword `inverse` is the only ODMG extension to the standard C++ class definitions. It is, of course, optional. It ensures the referential integrity constraint: if a `Person` moves to another `Apartment`, the attribute `is_used_by` is automatically reset to `NULL` until a new `Person` takes this `Apartment` again. Moreover, if an `Apartment` object is deleted, the corresponding `lives_in` attribute is automatically reset to `NULL`, thereby avoiding dangling references.

1.3 Collections

The efficient management of very large collections of data is a fundamental database feature. Thus, ODMG-93 introduces a set of predefined generic classes for this purpose: `Set<T>`, `Bag<T>` (a multi-set, i.e., a set allowing duplicates), `Varray<T>` (variable size array), `List<T>` (variable size and insertable array).

A collection is a container of elements of the same class. As usual, polymorphism is obtained through the class hierarchy. For instance a `Set<Ref<Person>>` may contain `Persons` as well as `Employees`, if the class `Employee` is a subclass of the class `Person`.

In a database schema, collections may be used to record extents of classes. (An *extent* is the set of all instances of the class.) In the ODMG-93 C++ binding, the extent of a class is not automatically maintained, and the application itself creates and maintains explicitly a collection whenever an extent is really needed. This management can be easily encapsulated in creation methods, and furthermore, the application can define as many collections as needed to group objects which match some logical property. For instance, the following collections can be defined:

```
Set<  Ref<Person> >    Persons;      // The Person class extent.
Set<  Ref<Apartment> > Apartments;   // The Apartment class extent.
Set<  Ref<Apartment> > Vacancy;      // The set of vacant apartments.
List< Ref<Apartment> > Directory;    // The list of apartments.
                                     // ordered by their number of rooms.
```

1.4 Multiple Relationships

Very often, an object is related to more than one object through a relationship. Therefore, the notion of 1-1 relationship defined previously has to be extended to 1-n and n-m relationships, with the same guarantee of referential integrity.

For example, a `Person` has two parents and possibly several children; in a `Building`, there are many `Apartments`:

```
class Person{
   Set  < Ref<Person> > parents  inverse children; // 2 parents.
   List < Ref<Person> > children inverse parents;  // Ordered by birthday
};

class Building{
   List< <Ref<Apartment> > apartments inverse building;
   // Ordered by apartment number
};

class Apartment{
   int number;
   Ref<Building> building inverse apartments;
};
```

1.5 Naming

ODMG-93 enables explicit names to be given to any object or collection. From a name, an application can directly retrieve the named object and then operate on it or navigate to other objects following the relationship links.

A name in the schema plays the role of a variable in a program. Names are entry points in the database. From these entry points, other objects (in most cases unnamed objects) can be reached through associative queries or navigation. In general, explicit extents of classes are named.

1.6 The Sample Schema

Let us now define our example schema completely:

```
class Person{
   String name;
   Date   birthdate;
   Set  < Ref<Person> > parents  inverse children;
   List < Ref<Person> > children inverse parents;

   Ref<Apartment> lives_in      inverse is_used_by;

// Methods:
   Person();                          // Constructor: a new Person is born
   int age();                         // Returns an atomic type
   void marriage(Ref<Person> spouse); // This Person gets a spouse
   void birth(Ref<Person> child);     // This Person gets a child
   Set< Ref<Person> > ancestors;      // Set of ancestors of this Person
   virtual Set<String> activities();  // A redefinable method
};

class Employee: Person{               // A subclass of Person
```

```
   float salary;
// Method
   virtual Set<String> activities();  // The method is redefined
};

class Student: Person{                // A subclass of Person
   String grade;
// Method
   virtual Set<String> activities();  // The method is redefined
};

class Address{
   int number;
   String street;
};

class Building{
   Address address;  // A complex value (Address) embedded in this object
   List< <Ref<Apartment> > apartments inverse building;
// Method
   Ref<Apartment> less_expensive();
};

class Apartment{
   int number;
   Ref<Building> building;
   Ref<Person> is_used_by inverse lives_in;
};


Set<  Ref<Person> >    Persons;      // All persons and employees
Set<  Ref<Apartment> > Apartments;   // The Apartment class extent
Set<  Ref<Apartment> > Vacancy;      // The set of vacant apartments
List< Ref<Apartment> > Directory;    // The list of apartments
                                     // ordered by their number of rooms
```

2 C++ Binding

To implement the schema defined above, we write the body of each method. These bodies can be easily written using C++. In fact, because `Ref<T>` is equivalent to a pointer (`T*`), manipulating persistent objects through `Refs` is done in exactly the same way as through normal pointers.

To run applications on a database instantiating such a schema, ODMG-93 provides classes to deal with `Databases` (with `open` and `close` methods) and `Transactions` (with `start`, `commit`, and `abort` methods). When an application creates an object, it can create a transient object which will disappear at the end of the program, or a persistent object which will survive when the program ends and can be shared by many other programs possibly running at the same time. Here is an example of a program to create a new persistent

apartment and let “john” move into it:

```
Transaction move;
move.begin();
   Ref< Apartment > home = new(database) Apartment;
   Ref< Person >    john = database->lookup_object("john");
                                    // Retrieve a  named object
   Apartments.insert_element(home); // Put the new apartment
                                    //   in the class extent
   john->lives_in = home;           // Persistent objects are handled
                                    //   like standard C++ objects
move.commit();
```

3 Object Query Language, OQL

ODMG-93 introduces a query language, OQL. OQL is a superset of the query part of SQL’92 (entry-level). It allows easy access to objects. We just presented an object definition language (using C++) and a C++ binding. We strongly believe that these two languages are not sufficient for writing database applications and that many situations require a query language:

Interactive ad hoc queries: A database user should not be forced to write, compile, link edit, and debug a C++ program just to get the answer to simple queries. OQL can be used directly as a stand alone query interpreter. Its syntax is simple and flexible. For someone familiar with SQL, OQL can be learned in a few hours.

Simplify programming by embedded queries: Embedded in a programming language like C++, OQL dramatically reduces the amount of C++ code to be written. OQL is powerful enough to express in one statement a long C++ program.

Besides, OQL directly supports the ODMG model. Therefore, OQL has the same type system as C++ and is able to query objects and collections computed by C++ and passed to OQL as parameters. OQL then delivers a result which is put directly into a C++ variable with no conversion. This definitively solves the well known “impedance mismatch” which makes embedded SQL so difficult to use, since SQL deals with “tables”, not supported by the programming language type system.

Let the system optimize your queries: Well known optimization techniques inspired from relational technology and extended to the object case can be used to evaluate an OQL query by virtue of its declarative style. For instance, the OQL optimizer of O_2 finds the most appropriate indexes to reduce the amount of data to be filtered. It factors the common subexpressions, finds expressions that can be computed once outside an iteration, pushes up the selections before starting an inner iteration, etc.

Logical/physical independence: OQL differs from standard programming languages in that the execution of an OQL query can be dramatically improved without modifying the query itself, simply by declaring new physical data structures or new indexing or clustering strategies. The optimizer can benefit from these changes and then reduce the response time.

Doing such a change in a purely imperative language such as C++ requires an algorithm to be completely rewritten because it cannot avoid making explicit use of physical structures.

Higher level constructs: OQL, like SQL, provides very high level operators which enables the user to sort, group, or aggregate objects, or to do statistics, all of which would require a lot of tedious C++ programming.

Dynamicity: C++ is a compiled programming language which requires heavy compiling and link editing. This generally precludes having a function dynamically generated and executed by an application at runtime. OQL does not suffer from this constraint since a query can be dynamically computed immediately.

Object server architecture: The new generation of Object Database Systems have a very efficient architecture. For instance, O_2 has a page server which minimizes the bottleneck in multi-user environment and exploits all the cpu and memory resources available on the client side, which holds visited objects in its own memory.

This architecture suits local area network applications very well. For wide area network and/or more loosely coupled database applications, this architecture can be supplemented by an OQL server, where the client sends a query to the server to be executed entirely on the server side. Without a query language, this architecture is impossible.

SQL'92 compliant: Object database systems must propose the equivalent of relational systems, i.e., a query language like SQL. Whenever possible, OQL is SQL. This facilitates the learning of OQL and its broader acceptance.[1]

Obviously, OQL offers more than SQL in order to accomodate the ODMG data model: path expressions, complex objects, method invocation, etc.

Support for advanced features (views, integrity constraints, triggers): Finally, without OQL it would be impossible to offer advanced services in object database systems. Features such as views, triggers and integrity constraints need a declarative language.

Let us now turn to an example based presentation of OQL. We use the database described in the previous section, and instead of trying to be exhaustive, we give an overview of the most relevant features.

3.1 Path Expressions

As explained above, one can enter a database through a named object, but more generally as soon as one gets an object (which comes, for instance, from a C++ expression), one needs a way to "navigate" from it and reach the right data one needs. To do this in OQL, we use the "`.`" (or equivalently "`->`") notation, which enables us to go inside complex objects, as well as to follow simple relationships. For instance, if we have a `Person p` and we want to know the name of the street where this person lives, the OQL query is:

```
p.lives_in.building.address.street
```

This query starts from a `Person`, traverses an `Apartment`, arrives in a `Building`, and goes inside the complex attribute of type `Address` to get the street name.

This example treated a 1-1 relationship; let us now look at 1-n relationships. Assume we want the names of the children of the `Person p`. We cannot write `p.children.name` because `children` is a `List` of references, so the interpretation of the result of this query would be undefined. Intuitively, the result should be a collection of names, but we need an unambiguous notation to traverse such a multiple relationship. We use the select-from-where clause to handle collections just as in SQL:

[1] ODMG voted recently a change of the first version of OQL in order to accept the SQL'92 syntax, without changing the power and simplicity of OQL. This new version is available in ODMG-93, 1.2

```
select c.name
from p.children c
```

The result of this query is a value of type `Bag<String>`. If we want to get a `Set`, we simply drop duplicates, like in SQL, by using the `distinct` keyword:

```
select distinct c.name
from p.children c
```

Now we have a means to navigate from an object to any object following any relationship and entering any complex subvalues of an object.

For instance, assume we want the set of addresses of the children of each `Person` of the database. We know the collection named `Persons` contains all the persons of the database. We have now to traverse two collections: `Persons` and `Person::children`. Like in SQL, the select-from operator allows us to query more than one collection. These collections then appear in the `from` part. In OQL, a collection in the `from` part can be derived from a previous one by following a path which starts from it. Our solution is:

```
select c.lives_in.building.address
from Persons p,
     p.children c
```

This query inspects all children of all persons. Its result is a value whose type is `Bag<Address>`.

Predicate

Of course, the `where` clause can be used to define any predicate, which then serves to select only the data matching the predicate. For instance, suppose we want to restrict the previous query to the people living on `Main Street`, and having at least two children. Moreover, we are interested only in the addresses of the children who do not live in the same apartment as their parents. The resulting query is:

```
select c.lives_in.building.address
from Persons p,
     p.children c
where p.lives_in.building.address.street = "Main Street" and
      count(p.children) >= 2 and
      c.lives_in != p.lives_in
```

Join

In the `from` clause, collections which are not directly related can also be declared. As in SQL, this allows us to compute *joins* between these collections. For instance, to get the people living in a street that have the same name as the street, we do the following. The `Building` extent is not defined in the schema, so we have to compute it from the `Apartment` extent. To compute this intermediate result, we need a select-from operator again. This shows that in a query where a collection is expected, the collection can be computed recursively by a select-from-where operator, without any restriction. So the join is done as follows:

```
select p
from Persons p,
     (select distinct a.building from a in Apartments) as b
where p.name = b.address.street
```

This query highlights the need for an optimizer. In this case, the inner select subquery should be computed once and not for each person!

3.2 Complex Data Manipulation

A major difference between OQL and SQL is that object query languages must manipulate complex values. OQL can therefore create any complex value as a final result, or inside the query as an intermediate calculation.

To build a complex value, OQL uses the constructors `struct`, `set`, `bag`, `list`, and `array`. For example, to obtain the addresses of the children of each person, along with the address of the person, we use the following query:

```
select struct (
         me: p.name
         my_address: p.lives_in.building.address,
         my_children: (select struct (
                                  name: c.name,
                                  address: c.lives_in.building.address
                                  )
                       from p.children c)
         )
from Persons p
```

This gives for each person the name, the address, and the name and address of each child, and the type of the resulting value is:

```
struct result_type {
   String me;
   Address my_address;
   Bag<struct{String name; Address address}> my_children;
}
```

OQL can also create complex objects. For this purpose, it uses the name of a class as a constructor. Attributes of the object of this class can be initialized explicitly by any valid expression.

For instance, to create a new building with 2 apartments, if there is a type name in the schema `List_apart` defined by:

```
typedef List<Ref<Apartment>> List_apart;
```

the query is:

```
Building(address: struct(number: 10, street: ''Main street''),
         apartments: List_apart(Apartment(number: 1),
                                Apartment(number: 2)))
```

3.3 Method Invocation

OQL allows us to call a method, with or without parameters, anywhere the result type of the method matches the expected type in the query. In the case where the method has no parameters, the notation for calling the method is exactly the same as for accessing an attribute or traversing a relationship. If the method has parameters, these are given between parentheses.

This flexible syntax frees the user from knowing whether the property is stored (an attribute) or computed (a method). For instance, to get the age of the oldest child of the person “Paul”, we write the following query:

```
select max(select c.age from p.children c)
from Persons p,
where p.name = "Paul"
```

Of course, a method can return a complex object or a collection, and then its call can be embedded in a complex path expression. For instance, suppose that inside a `Building b`, we want to know who inhabits those least expensive apartment. The following path expression gives the answer:

```
b.less_expensive.is_used_by.name
```

Although `less_expensive` is a method, we "traverse" it as if it were a relationship.

3.4 Polymorphism

A major contribution of object orientation is the possibility of manipulating polymorphic collections, and thanks to the *late binding* mechanism, to carry out generic actions on the elements of these collections.

For instance, suppose the set `Persons` contains objects of class `Person`, `Employee`, and `Student`. So far, all the queries against the `Persons` extent dealt with all three possible classes in the collection. If one wants to restrict a query to a subclass of `Person`, either one provides an extent for this subclass which can then be queried directly, or else the superclass extent can be filtered to select only the objects of the subclass, as shown in the example below with the "class indicator".

A query is an expression whose operators operate on typed operands. A query is correct if the type of operands matches those required by the operators. In this sense, OQL is a typed query language. This is a necessary condition for an efficient query optimizer.

When a polymorphic collection is filtered (for instance `Persons`), its elements are statically known to be of that class (for instance `Person`). This means that a property of a subclass (attribute or method) cannot be applied to such an element, except in two important cases: late binding to a method, or explicit class indication.

Late Binding

The following query gives the activities of each person:

```
select p.activities
from Persons p
```

Here, `activities` is a method which has three incarnations. Depending on the kind of person of the current `p`, the right incarnation is called.

Class Indicator

To go down the class hierarchy, a user may explicitly declare the class of an object that cannot be inferred statically. The interpreter then has to check at run time that this object actually belongs to the indicated class (or one of its subclasses).

For example, assuming we know that only `Students` spend their time in following a course of study, we can select those persons and get their grade. We explicitly indicate in the query that these persons are students:

```
select (Student)p.grade
from   Persons p
where "course of study" in p.activities
```

3.5 Operator Composition

OQL is a purely functional language: all operators can be composed freely as long as the type system is respected. This is why the language is so simple and its manual so short.

This philosophy is different from SQL, which is an ad hoc language whose composition rules are not orthogonal. Adopting complete orthogonality allows us not to restrict expressiveness and makes the language easier to learn without losing the SQL style for the simplest queries.

Among the operators offered by OQL but not yet introduced, we can mention the set operators (`union`, `intersect`, `except`), the universal (`for all`) and existential quantifiers (`exists`), the `sort` and `group by` operators, and the aggregation operators (`count`, `sum`, `min`, `max`, and `avg`).

To illustrate this free composition of operators, let us write a rather complex query. We want to know the name of the street where employees live and have the smallest salary on average, compared to employees living in other streets. We proceed step by step and then do it all at once. We can use the `define` OQL construct to evaluate temporary results.

1. Build the extent of class `Employee` (not supported directly by the schema):

   ```
   define Employees as
          select (Employee) p from Persons p
          where "has a job" in p.activities
   ```

2. Group the employees by street and compute the average salary in each street:

   ```
   define salary_map as
          select street,
                 average_salary: avg(select e.salary from partition)
          from   Employees e
          group by street: e.lives_in.building.address.street
   ```

   The `group by` operator splits the employees into partitions, according to the criterion (the name of the street where this person lives). The `select` clause computes, in each partition, the average of the salaries of the employees belonging to this partition.

   The result of the query is of type:

   ```
   Set<struct{String street; float average_salary;}>
   ```

3. Sort this set by salary:

   ```
   define sorted_salary_map as
          select s from salary_map s
          order by s.average_salary
   ```

   The result is now of type:

   ```
   List<struct{String street; float average_salary;}>
   ```

4. Now get the smallest salary (the first in the list) and extract the corresponding street name. This is the final result:

```
sorted_salary_map[0].street
```

5. In a single query we could have written:

```
(select street, average_salary: avg(select e.salary from partition)
 from (select (Employee) p from p in Persons
                 where "has a job" in p.activities) as e
 group by street: e.lives_in.building.address.street
 order by average_salary
)[0].street
```

3.6 C++ Embedding

An `oql` function is provided as part of the ODMG-93 C++ binding. This function allows one to run any OQL query. Input parameters can be passed to the query. A parameter is any C++ expression. Inside the sentence, a parameter is referred to by the notation:

`$<position><type>`

where `position` gives the ordinal position of the parameter in the parameter list and `type` is a tag indicating the kind of the parameter (`o` means object, `c` means collection, `i` integer, etc.). Let us now write as an example the code of the `ancestors` method of class `Person`. This example shows how easy it is to write a recursive query that gives OQL the power of a recursive query language.

Recursive Query

```
Set < Ref<Person> > Person::ancestors(){
   Set < Ref<Person> > result;
   oql(result, "flatten(select distinct a->ancestors from $1c as a)   \
                union $1c", parents);
   return result;
};
```

The `$1c` notation refers to the first parameter, i.e, the `Set parents`. In the select clause, we compute the set of ancestors of each parent. We get therefore a set of sets. The `flatten` operator converts this set of sets into a simple set. Then, we take the union of this set with the parent set. The recursion stops when the parents set is empty. In this case, the select part is not executed and the result is the empty set.

4 Conclusion

ODMG-93 provides a complete framework within which one can design an object database, write portable applications in C++ or Smalltalk, and query the database with a simple and very powerful query language. Based on the OMG, SQL, C++, and Smalltalk standards, available today in industrial products such as O_2, it is supported by all the actors of the object database world.

References

[Catt94] Rick Cattell, *et al*. The Object Database Standard: ODMG-93, release 1.1. Morgan Kaufmann, 1994.

[O_2] O_2 Technology. The O_2 User Manual, release 4.5.

Jim Melton
Senior Architect, Standards
Sybase, Inc.
1930 Viscounti Drive
Sandy, UT 84093
jim.melton@sybase.com

Abstract

The SQL standardization community is hard at work adding object technology to the SQL relational database language. Are these experts merely engaged in a Quixotic waste of their organizations' resources, or are they breaking important new ground that will produce the *database language for the introduction of the 21st century? By examining the motivations behind the work, reviewing some of the history behind it, and inspecting in some detail the technical aspects, the reader can form his or her own opinions.*

1 Introduction

SQL was invented at IBM in the early 1970s as an interface to their relational database prototype, System R. The "father" of SQL, Don Chamberlin, designed features for the language based primarily on Dr. E. F. Codd's relational model of data. One of the primary features of this model—and language—was the absence of the "navigational" facilities that characterized other database models, such as hierarchical and navigational models.

In the mid-1980s, a new model of data management began gaining significant attention. The *object-oriented* view of the world identifies each "thing" being modeled by a unique identifier ... and related "things" are linked by means of their OIDs, or object identifiers. This makes some think in terms of the relational model's primary keys and join operations, while it makes others think of following pointers for extremely fast access. Who's right? Perhaps both views are appropriate.

2 Why Add Object Technology To SQL?

There are many possible reasons, falling into about three categories, for adding object technology to SQL. First, there are the cynical reasons (perhaps I should be more charitable and call this category "market-driven" reasons). From a vendor's point of view, the decision to add object technology to SQL can be justified by the perception that his or her competitors are doing it; to remain competitive, his or her product must add the same facilities. Another reason is that customers seem to be demanding the ability to manage objects with their relational database systems. (It's not entirely clear to me that there are all that many customers sophisticated enough to know all the implications of this, but I'm quite certain that the number is increasing rapidly.)

Perhaps the most cynical reason of all is "there's money to be made by doing it". Of course, that's also the most compelling reason. After all, if companies (and individuals) don't make money, then there won't be any product set at all! The relational vendors clearly believe that there is a vast potential market for systems that combine relational and object technology; I happen to share that belief. Accompanying that third reason is one a little less savory: protect and increase the market share of relational systems by preventing "pure ODBMS" vendors from taking it away. I don't mean to imply in any way that the relational vendors are engaging in any activities likely to gain the attention of the antitrust forces; it's merely a reality that they will each try to protect and increase their market share by adding technology they view as requested by that market.

There are also several technical reasons for pursuing the merger of relational and object technology. There are increasing numbers of applications that need to access both their existing, "traditional" data *and* their newer

“non-traditional” data simultaneously—not merely in the same transaction, but in the same query (consider a multimedia application demand like “Retrieve the photograph, voice, and town map for all heads of state entering office between 1960 and 1990 who govern countries with gross domestic products greater than Norway’s”). Furthermore, it’s clear that many applications are being written in various object-oriented programming languages (OOPLs), such as C++, CLOS, Smalltalk, and even OO-COBOL. These applications often need to share persistent objects, so an object repository is required. Finally, there are genuine advantages, especially in the increasingly-popular client-server environment, to having objects (and other data) manipulated *inside* the DBMS, rather than merely storing them there.

Then there is the issue of adding the notion of objects to the SQL *standard* [SQL-92]. This step is required for the same reason that standardizing SQL in the first place is required—to avoid having at least as many language extensions as there are vendors doing them. It will ultimately be enormously helpful to users of these merged relational-object systems if all vendors agree on and implement the same object model. Not at all coincidentally, it will benefit the vendors, too. I believe that it’s important to integrate the object facilities gracefully into SQL’s relational heritage—not an easy task. It is also extremely important to create *bindings* between the various OOPLs (as well as traditional 3GLs (3rd Generation Languages)) and the object-enhanced SQL. Working in the standards arena has its negative aspects, but it has the very positive benefit of tapping the resources of a variety of companies and individual experts to work on the really hard questions.

3 A Little History

The concept of adding objects to SQL has been around almost as long as the concept of object orientation (OO). Very early on, many people seemed to recognize the need to provide persistence for objects and the benefits that some of OO’s features could provide for relational applications. However, little was done in a formal way until early 1988, when a proposal by T. Murata (Japan) added the notion of subtables and supertables to a future version of the SQL standard (by now informally called *SQL3*) [SQL3, FRAME, CLI]. A second step occurred later that same year when John Kerridge (The University, Sheffield, United Kingdom) proposed a feature called “User-Defined Types” (or UDTs) for inclusion in SQL3. This second step provided only a sort of record structure that could be the data type of a column in an SQL table ... useful, but limited.

The first real acknowledgement of the benefits of the OO model came in early 1991, when several researchers and engineers at Digital Equipment Corporation (Jim Melton, Jonathan Bauer, Krishna Kulkarni, Mike Kelley, and Umesh Dayal) proposed to revisit UDTs by adding many of the concepts from the OO world and then further enhance them by allowing the notion of unique identity. In this first proposal, the goal was to establish an object model that permitted both objects (more accurately, implicit object references) and “complex values” (UDTs with structure) to be the data type of columns of tables. This fundamental OO-SQL paper served, among other things, as a wake-up call to members of the SQL standardization community who had yet to recognize the increasing importance of object technology.

Digital was quickly joined by Oracle Corporation, initially in the person of David Beech, who provided very significant new material for SQL3’s emerging object capabilities. Beech’s major goal was the unification of the relational and object models and he wrote many substantial proposals pursuing it. In the end, however, this laudable and important effort was hindered by the number and difficulty of the problems that emerged. Several of us remain convinced that this unification is achievable, but the approach used initially may have been too ambitious.

As the committees recognized the importance of the OO-SQL effort, other vendors began sending their object experts to participate in the work. IBM’s Phil Shaw continued his long history of significant contributions even after he joined Oracle; his replacement at IBM, Nelson Mattos, has made major enhancements to SQL3’s object facilities. Amelia Carlson, first at HP and now at Sybase, has also been a significant contributor to the object model and to specific features. Krishna Kulkarni moved on to Tandem and has continued to be active in

writing proposals and in guiding the development of the overall model. The need to support object behaviors in the form of user-written functions was met by several solutions, notably procedural language extensions initiated by Andrew Eisenberg of Digital and subsequently the focus of contributions by many in the community.

For much too long, these major object-oriented SQL extensions (sometimes called "the MOOSE") were virtually ignored by the "pure ODBMS" vendors, although Mary Loomis stated her intent to join while still at Versant. In early 1994, UniSQL joined the ANSI X3H2 technical committee responsible for SQL3 and their representative, Bill Kelley, began producing proposals. Those proposals have focussed on taking SQL3's object facilities in a direction significantly different from that originally envisioned and have met with only limited success. Still, the community as a whole will benefit from the efforts of this extended-relational vendor.

4 SQL3's Object Model

It often seems that there are as many ways of describing an object model as there are object models. Because many readers may be familiar with the ODMG's description of their object model [ODMG], I will describe SQL3's in similar terms. At the highest level, the models are very similar:

- The basic modeling primitive is the *object*. (This is true of the OO facilities in SQL3; the basic modeling primitive of SQL3 as a whole is the *table*.)
- Objects can be categorized into *types*. All objects of a given type exhibit common behavior and a common range of states. (Note that "type" is used by ODMG and SQL3, while some other models use "class" synonymously.)
- The behavior of objects is defined by a set of *operations* that can be executed on an object of the type, *e.g.,* you can "format" an object of type Document.
- The state of objects is defined by the values they carry for a set of *properties*. These properties may be either *attributes* of the object itself or *relationships* between the object and one or more other objects. (In SQL3, the relationships are not identified using that term, but appear implicitly by references from one object to another using the object identifier of the referenced object.)

In SQL3, the notion of an *interface* to a type and an *implementation* of a type are distinguishable. However, the two are (currently) specified together and no provision is (yet) made for multiple concurrent implementations of a type. SQL3 does implicitly allow the implementation of a type to be changed without affecting the interface.

In SQL3, types are not objects and do not have properties as such. Instead, types are built-in concepts of the language that are created, manipulated, and destroyed using the SML (Schema Manipulation Language) aspects of SQL. This obviously derives from the relational heritage of SQL and the desires to maintain that cultural aspect. There are few—if any—disadvantages to not treating types as objects in SQL3.

SQL3 defines the notion of inheritance and type hierarchies—the ability to define subtypes of a type. SQL3's type hierarchies support multiple inheritance (the ability of a type to have more than one supertype), so "hierarchy" is technically not an accurate characterization; the type relationships form a directed graph that can be more complex than a tree structure. *Inheritance* means that all the characteristics of a supertype are also characteristics of all of its subtypes. One implication is that any instance of a subtype can be treated as though it were an instance of any of its supertypes. All types in SQL3 are *instantiable*, meaning that there can be instances of every type. The SQL3 object model does not have any inherent "root type".

The stickiest issue currently being addressed by the SQL standards community is the notion of *extents* for a type. The natural extent for any data in SQL is the table (strictly: a column of a row in a table), but that proves more elusive for *objects* with their own unique identity. This is the source of UniSQL's efforts to define a type as a table and a row as an instance of a type ... an object. Other possibilities currently being explored are: 1)

Placing objects in specially-designated columns in rows of tables, 2) Placing objects in the only column of rows of specially-designated tables (“extent tables”), and 3) Creating a new storage paradigm solely for use as extents for objects, which would then appear only as their object identifiers in ordinary table/row/column locations.

In SQL3, the behavior of objects is defined by operations (informally called *methods*) that are specified as functions, which may be written in SQL (extended with procedural language facilities) or in one of several of the more traditional 3GLs; future plans call for adding various OOPLs to this list. Methods are identified in two ways: by their *signature* (comprising the function’s name, the name and type of any arguments, and the type of the returned value); and by a special name (called the *specific name*) that is used for SML operations. SQL3 supports the notion of operation overloading, called *polymorphism*, which permits many functions to share a name; the functions are distinguished by the additional details of their signatures and the appropriate specific function is invoked based on the values of the arguments provided. When subtypes and supertypes are involved, the final determination of specific function is often deferred until execution; in many other situations, determination can be made during compilation.

4.1 What is ... and what isn’t ... an object in SQL

SQL3 has many identifiable “things”, having names, formal or informal types, and so forth: tables, views, constraints, data types, literals, authorization identifiers, schemas, catalogs, sessions, *etc*. None of these are objects! In SQL3, only instances of an Abstract Data Type (ADT) declared to be an Object ADT, are objects. This inherently puts an identifiable dividing line between “traditional” SQL data and “non-traditional” object data.

This distinction makes it very easy for an SQL vendor to choose to continue to support good old relational SQL and not implement the MOOSE. It also makes it quite easy for the vendors’ customers to continue to write their applications in traditional SQL without ever worrying about the new-fangled OO world until they’re ready to tackle it. There is also the possibility that after-market suppliers can provide customers with “add-in” MOOSE modules for SQL products, though this hasn’t been explored very much.

The down-side to this partitioning is that application writers must be able to deal with two paradigms: the relational mode of data management, and the procedural mode of object management. The SQL standard community believes that this dichotomy is inherent in the need to deal with different sorts of data and does not find this a disadvantage. Your mileage may vary...

4.2 Objects: unique identity or complex data structure?

Many participants in the SQL3 work believe that the notion of unique identity—separate from all of the attributes associated with a datum—is inherent in the OO paradigm and that this notion must be provided in order to realistically claim that SQL3 has OO support. Others, like UniSQL and Chris Date, assert that the relational model’s concept of *primary key*, coupled with unique table names, is more than adequate to uniquely identify any collection of data describing any entity. The debate goes something like this:

1. It’s fatuous to insist on “unique identity” unrelated to known attributes of an entity. Two cans of beans in the supermarket have the same product inside, the same dimensions, the same weight, the same words and colors on the labels, the same markings, the same expiration date, *etc.* Who cares about distinguishing between them? If they had serial numbers, like stereo systems or cars, then that would be the primary key and that would distinguish them.

2. My husband has an identical brother who was accidentally issued the same social security number as my husband. Therefore, as far as the bank can tell, they’re identical in every respect ... they even have the same initials. But I really would rather not have my brother-in-law writing checks on my account, even though it’s OK for my husband to do that. They’re *not the same person*.

I conclude that both parties are right ... but they're talking different languages. I don't deny that many entities in the real world are indistinguishable and are best described by values alone ... by their attributes. The millions of rivets and screws and bolts that go into an airliner don't need to be treated with unique identity. However, the airplane itself *may* be treated with unique identity, different from its "tail number" or its air-frame number *if the application demands it*. The lesson I draw from all this is that application needs differ. In some applications, the fact that you and I are both 185 cm tall, both weigh 90 kg, both have brown eyes and black hair and are Jewish might just mean that there are two slightly overweight, medium height, Jews available for employment as actors. In other situations, the need to distinguish between us may be vitally important ... without having to find some distinguishing attribute to record.

4.3 Type hierarchies—limits and characteristics

A lot of theoretical work has been done on the notion of supertyping. SQL3's designers have paid close attention to that work and have learned from it. In SQL3, supertypes and subtypes have a relationship defined by these characteristics:

Substitutability Anywhere an instance of a supertype can appear, an instance of any of its subtypes can also appear. That is, in any column of a table where one can store a Vehicle, one can just as well store an Automobile or a Truck. That's because any operation on an instance of a supertype must also be an operation on the subtype (noting that the *behavior* of the operations may differ between a supertype and a subtype).

Instance inclusion Every instance of a subtype is simultaneously an instance of every one of its supertypes. That is, one can treat a subtype instance as though it were a supertype instance because it *is* a supertype instance. SQL3 goes further: it provides a special operator (TREAT) to force the system to treat a subtype instance *exactly* like a supertype instance, even to the point of using the specific function implementing the supertype's behavior.

Attribute inclusion Subtypes have all the attributes of all of their supertypes. That is, if a supertype has some piece of stored data or some method, then all of its subtypes will have the same thing.

These three characteristics are intimately related, and are actually somewhat difficult to separate intuitively. Together, they offer a very powerful type facility that can model a lot of the real world.

4.4 Encapsulation

A key characteristic of most object models is that the interface of a type and the implementation of that type are separate, permitting the interface to remain constant (therefore not perturbing applications) while the implementation changes. By *hiding*, or *obscuring*, the implementation details from applications, the model gains protection for those applications and freedom for the implementors.

There may be many ways to provide this obscurity, but the one chosen by SQL3 (and by most object models) is *encapsulation*. In SQL3, type definitions are encapsulated so that applications are unable to distinguish between stored data (sometimes called the *state*) and methods in many cases. Every piece of stored data is represented ... and accessed ... in two equivalent ways: through "dot notation" and through a "functional notation". Any method whose signature has specific characteristics can also be invoked through dot notation or functional notation equivalently.

Dot notation may be more comfortable for some applications and looks like this:

```
typename..attributename
```

By contrast, functional notation is more intuitive in some situations and looks like this:

attributename(typename)

Any method of an ADT that has a single parameter whose data type is that ADT and that returns a value—for example, the weight of a person—can be invoked through either notation: weight(person) or person..weight. Therefore, an application is unable to determine whether the value returned is retrieved from a stored value or from a function that performs some computation to derive the value from other stored values. Similarly, any method of an ADT that has two parameters, the first of which has the ADT as its data type and the second of which is some other type—for example set_weight(person,185)—can be invoked through function notation or through an assignment sort of notation, such as person..weight=185. The converse is equally true, of course.

SQL3 provides three levels of encapsulation. PUBLIC attributes(including methods) are usable by any user of the type (possibly subject to SQL's privilege determination). PRIVATE attributes are usable only within the definition of the type itself—for example, methods in the type definition might need access to stored data. PROTECTED attributes are usable within the definition of the type itself and in the definition of any subtypes, but not by the "general public". SQL3 also provides something like C++'s "friend" functions that can access PRIVATE attributes; these functions are not defined internally in the type definition, but their specific names are included in the type definition.

5 Integrating Object Technology with Relational

Because SQL is a relational language at its heart, the first goal of adding object technology to the language has to be the ability to use objects (or references to them) as data in columns of rows of tables. Recall, though, that object types in SQL3 are merely special cases of the more general class of ADT. This leads to a tension in the design of SQL3.

- First, the desire for economy of specification leads toward specifying "complex values" and "uniquely identifiable objects" in a common way, such as Abstract Data Types, distinguishing between them only where required.
- However, SQL is today a value-based language and "complex values" are nothing more than values with identifiable internal structure. Maintenance of the culture of SQL leads towards specifying value ADTs in a manner similar to the way that other values are specified and specifying objects in a separate manner (possibly in a separate document or section of a document).

It is finding a resolution to this tension that gives the greatest challenge in the design of the language. Let's explore some of the issues that have been raised.

6 Some Hard Questions ... and a Few Answers

6.1 Stored attributes *vs* virtual attributes

In SQL3's object model, like most other object models, type designers specify a number of attributes that provide *stored data* to hold the *state* of each object of that type. For example, in an object type for Persons, one might have stored data containing the Name attribute, the Address attribute, and perhaps the Age attribute. Persistent objects (*i.e.*, those whose existence persists after termination of the application that created them) must somehow maintain their state—the values of these attributes—until changed or deleted, and the natural way to do this is through stored data.

However, the Age attribute could easily be computed instead of being stored. If the Person type has a stored attribute of Birthdate, Age can be computed from SQL's CURRENT_DATE function minus the Birthdate attribute. The Age attribute can then be considered a *virtual attribute* because it can be "seen" by applications, but isn't actually stored in the state of the object.

SQL3 had virtual attributes as part of its definition for about a year. However, as its designers began to clarify the encapsulation model, they realized that *all* attributes of an object must be accessible using a functional notation ... and that functions having certain characteristics (described earlier) must be accessible using dot notation. An additional benefit of this paradigm emerged: we realized that virtual attributes were an unnecessary complication, as the capability could be provided by type designers' writing additional functions (such as Age(Person) in this example) and allowing applications to reference the function either through the functional notation or the dot notation (Person..Age). After this realization, the notion of virtual attributes was removed from SQL3.

6.2 Methods—in what languages?

When the first object papers were being considered by the SQL standards community, there was no good answer to the burning question: In what programming languages will methods be written? Uncovering the answer to that question turned out to be surprisingly contentious.

It was obvious to everyone that we had to permit methods to be written in one or more of the "traditional" programming languages, such as Fortran, Pascal, or (perhaps even especially) C. There were a number of strong reasons for this: the commercial availability of vast libraries of functions written in languages like Fortran; the probability that application developers would have private libraries of functions written in C and other languages; and the desire to promote re-use of code and of object definitions and sharing those with OOPLs. It therefore was not difficult to conclude that SQL3 should permit methods to be written in any of the seven languages for which bindings with SQL already existed: Ada, C, COBOL, Fortran, MUMPS (now named M), Pascal, and PL/I.

However, one issue was repeatedly raised: In order to maximally promote writing of portable applications, we have to allow type designers to write their methods in some language that is known to be available everywhere SQL is available. Some claimed that C was that language, but several important counterexamples were found. After rancorous debate, we concluded that the only language known to run everywhere SQL runs is ... SQL itself. From that conclusion, we made the decision to enhance SQL by adding *computational completeness*, manifested by a set of *flow of control statements*.

It didn't take long to realize that this decision had other benefits, most notably that it enabled more powerful client-server operations by allowing application designers to offload additional program logic to the database server, thereby reducing communication between client and server. Once this final realization was reached, the controversy disappeared and all the primary database vendors are actively implementing the procedural aspects of SQL. Furthermore, a separate *Part* of the SQL standard is under development. Known as SQL/PSM (Persistent Stored Modules) [PSM], this standard will be applicable to ordinary, relational SQL as well as to the MOOSE.

6.3 Value ADTs, object ADTs, or both?

One question that arose early in the MOOSE discussions was: How far do we want to go? There were several participants who were quite interested in having "complex values", or "structured values" available for use in SQL applications and databases. (For example, international telephone numbers might be decomposed into CountryCode, CityCode or AreaCode, and LocalNumber; similarly, Address might be decomposed into BuildingNumber, StreetName, City, State or Province or Region, PostalCode, and Country.) This requirement was adequately satisfied by UDTs, but some participants wanted to enhance UDTs with method capabilities.

The majority of participants recognized that this was not an adequate answer to the requirement for object technology support with its notion of unique identity and sharing of objects by referencing them. The result is that SQL3 presently has both *value ADTs* and *object ADTs*. However, the debates haven't completely died down. There are proponents of eliminating the notion of objects entirely and replacing it with strictly relational model notions (see "Type hierarchy *vs* table hierarchy" below). While it's too early to be confident of the outcome of this on-going debate, I remain hopeful that SQL3 will have true support for object technology.

6.4 Multiple inheritance, single root type, and single most-specific type

If one were to draw a graph of a hierarchy, one would get the sort of graph known as a *tree*. The distinguishing feature of this sort of graph is that any entity (any *node*) can lead to multiple other entities, but can come from only one entity. In an inheritance hierarchy, like a type hierarchy, one would say that a supertype can have an arbitrary number of subtypes, but no subtype can have more than one supertype.

There are significant applications that can be implemented with this restriction. But there are many applications that cannot be built under this paradigm. Consider the classic example of a university: There are many People known to a university. Some of them are Students and others are Instructors. However, many universities have the notion of Interns ... People who are simultaneously Students and Instructors. We may assume that People known to the university have attributes like Name, Address, and Birthdate. Similarly, we may assume that Students have all the attributes of People, and additionally have attributes like Major and GradePointAverage, while Instructors have all People attributes and additional ones like Department and Salary. What are we to make of Interns? As Students, they have all Student attributes, but as Instructors they must also have all Instructor attributes.

That is, they have *inherited* attributes from multiple supertypes. This is called *multiple inheritance* and generalizes the graph from a hierarchy to a directed acyclic graph. Multiple inheritance introduces a new set of problems to be addressed. One of the most notorious involves encapsulation: In our example, Students and Instructors all have Names, but their Names are inherited from a common source: People. Suppose that Students have an attribute called ThesisAdvisor, which identifies the Instructor that serves as the guiding force for the Student's thesis efforts; Instructors, on the other hand, have an attribute called ThesisAdvisor, which flags ("Yes" or "No") whether or not the Instructor receives additional income for acting as an advisor for one or more students. One now must answer the question: Does an Intern have an attribute called ThesisAdvisor and, if so, what purpose does it serve?

There appears to be no correct answer to this question.

- If Interns do not have the ThesisAdvisor attribute at all, then inheritance has clearly broken down and one might call into question whether Intern is actually a subtype of either Student or Instructor.
- If Interns have only one ThesisAdvisor attribute, then we (or the system) must choose which and inheritance has again broken down, at least from the viewpoint of the attribute not chosen.
- If Interns have two ThesisAdvisor attributes, then we have no way to identify either of them, as SQL's name-resolution rules require unique names for individual members of any collection of peer entities. Using an approach of qualifying the name of the ThesisAdvisor attributes with the name of the type from which it was inherited is flawed because it violates encapsulation by requiring that we know details of the definition of the Intern type.
- If Interns have two ThesisAdvisor attributes, but one of them has been given a different name (such as ThesisAdvisorYesOrNo), then inheritance and encapsulation have broken down.
- If we refuse to allow Interns because of this conflict, then the model breaks down entirely. One might argue that this last alternative is preferable because, after all, we can just be more careful when designing

our Student and Instructor types. While this solution may sound initially attractive, consider the desire to create types based on two different shrink-wrapped type libraries purchased from different vendors! We may have no control at all on the names of attributes in the supertypes we're forced to use.

This is one of those problems for which no good solutions are likely to be found, but for which compromises will be required. But this isn't the only issue related to multiple inheritance in SQL3.

Some object models, including SQL3's, require that any type family (that is, all types in an inheritance graph) have exactly one maximal supertype. There are major advantages to this restriction, especially in specifying the rules associates with resolving attribute names and function selection in the face of polymorphism. However, the restriction may force type designers into situations that are artificially complex.

Suppose a company has modeled its CapitalEquipment in an object-oriented paradigm and then later decides to model its corporate computer network in the same paradigm. Clearly, some NetworkNode objects will also be represented as CapitalMachines. It's quite undesirable to have separate application databases with two separate type instances representing the mainframe running the payroll system. Therefore, it's likely that the company would choose to merge the two type systems and model that mainframe as a BigComputer that is simultaneously a CapitalMachine and a NetworkNode. But, if the type system requires a single maximal supertype, it's far from clear what the common supertype of NetworkNode and CapitalMachine might be! Some object models simply invent a non-instantiable type called "Thing" (or"Object") and use it as the maximal supertype, but that seems artificial and awkward. SQL3 is in this position at the moment and it is not yet clear if there is sufficient sympathy with the problem to remove it.

Similarly, some object models require that any specific type be an instance of exactly one most-specific type. Therefore, if one wants to allow some Students to earn money by also acting as Instructors, one must specifically define an Intern type. This doesn't sound especially problematic, and it's not ... in small type systems. However, when one begins to have type systems with thousands of types, the problems associated with creating the "common subtypes" required by this restriction really begin to get onerous. SQL3 has shed this restriction and has demonstrated that there are few, if any, problems associated with losing it.

6.5 Operator notation for method invocation

In many environments, functional notation is simply not the most natural notation for expressing operations on data. Instead, an operator sort of notation would be preferable. Imagine having to write expressions such as the following:

```
ADD(Salary,ADD(Bonus,SUBTRACT(Commission,Expenses)))
```

when you really wanted to do this:

```
Salary+Bonus+Commission-Expenses
```

With apologies to LISP programmers, that's hardly the most natural way of coding arithmetic expressions. The same arguments can be applied to the operations (methods) defined for use with Abstract Data Types in SQL.

The SQL standardization community has been quite sympathetic to the problem and two groups independently proposed (slightly different) solutions for inclusion in SQL3.The groups were able to work out their differences and ended up with a joint proposal that was happily accepted into SQL3. This proposal allows a type designer to provide an operator notation for functions with one and two arguments and to make the operators prefix, infix, or postfix in nature. The hierarchy of the operators is also specifiable by the type designer.

There are clear difficulties with designing a language in which a sublanguage can be defined, and some of those difficulties are probably not worth overcoming in the short term. Consequently, the operator notation facility in SQL3 is fairly limited and won't satisfy everybody's wildest requirements. Still, it will no doubt prove to be quite useful and popular in certain applications.

6.6 Type hierarchy *vs* table hierarchy

Earlier, I mentioned that SQL3 has had the notion of a table hierarchy (supertables and subtables) for several years. Now that it also has a type hierarchy, one must ask the question: Why have two hierarchies with similar functions? Just the notion of economy of specification would suggest that the two should be combined somehow or one of them should be jettisoned.

However, lengthy and concerted efforts to combine the notion of type and table in SQL3 proved much more difficult than the participants had expected. After a year-and-a-half of work, the community backed away from this unification in favor of distinct concepts.

Later, some interest was expressed (by standards participants and by the database community at large) in eliminating one of the hierarchies and many concluded that the type hierarchy was more fundamental and necessary than the table hierarchy. However, efforts to propose the removal of the table hierarchy have not moved forward, largely due to the strong requirements from POSC (the Petrotechnical Open Software Corporation, a consortium in the oil and petrochemical industry), whose application architecture depends heavily on SQL having a table hierarchy. This is a strong sign that users may need one feature for near-term use that conflicts with long-term strategic requirement.

6.7 Where do objects "live"?

Undoubtedly, the most important feature of SQL3's object model for which a satisfactory solution has not yet been achieved is the issue of object extents—where objects "live". In many object-oriented programming system environments, objects are stored in a "heap": an in-memory (or on-disk) area without much in the way of internal organization. Periodically, some piece of code (often called a "garbage collector") scans the heap looking for objects that have been "lost" ... for which there are no references. Such objects are deleted and the storage returned to the heap for use by other objects.

That implementation model works well in a workstation or other localized environment. But in a massively distributed network, which may never be completely connected at any instant in its history, the notion of heap allocation and garbage collection is problematic. As a result of recognizing the difficulties (especially performance!) of the heap model, the SQL committees have explored other alternatives.

One alternative that has been explored was to allow the application writer to specify that some "locations" where objects appear were to be "instance locations" and others "reference locations". That would permit an instance of an object ADT to appear in any row of a column whose data type was that ADT. However, it also unfortunately allowed objects to "live" in temporary variables, in parameters to methods, and other areas offering less than satisfactory semantics (persistence, for example).

Another alternative currently under consideration is to designate certain tables as "object extent tables", require them to have exactly one column whose data type is the ADT, and require all other object "locations" to contain only a reference to the object (the object identifier).

There is a continuing debate over whether this is the right approach, but it is broadly recognized that we must have a finite, known number of locations where an object can actually "live" and those locations must be such that they can (individually, if not as a group) be traversed by some sort of iterator (such as SQL's cursor) to allow an application or a database administrator track down "lost" objects.

7 SQL3's Current Whereabouts

After all this, it will come as no surprise that the current draft of SQL3 runs to over 800 pages, more than 1100 when the procedural language document is added in—significantly larger than SQL-92. Why is SQL3 so large? Clearly, the addition of value ADTs and objects have had a pervasive effect on the language ... virtually every nook and cranny of SQL was affected. Another reason is the fact that there have been so *many* change proposals

and quite a lot of "churn" in the model and the details. Every time something is proposed and put into the language, and later removed, there are remnants that are harder to remove than they were to put in. As SQL3 moves towards the later stages of standardization, there will be concerted efforts to remove this detritus, but the document will remain long simply because the language it defines is large.

These "later stages of standardization" are, in the international arena, called "DIS" (Draft International Standard) and "IS" (International Standard). SQL3 is expected to enter the first formal stage, called "CD" (Committee Draft) in mid-1995, which suggests that it may advance to DIS in mid-1996 and to IS in mid- to late-1997. It's certainly premature to start calling it "SQL-97", but the recent decisions to move most of the less stable features into a still-later version (informally called "SQL4") will go a long way towards improving its chances. SQL4 is not likely to reach even CD status until at least 1998, which suggests that a published standard won't be possible before 2000; my opinion is that 2001 is more likely.

None of this is to say that SQL3 is universally accepted as a "good thing". There are a lot of skeptics who believe that relational databases ought to stick to their roots and implement only tables (to be sure, many of that camp support the notion of complex values, like UDTs). But there are lots of converts, too ... people who have accepted that the market place wants this to happen and would rather be part of the process than oppose it. And the MOOSE has its fair share of true believers, the individuals and companies who invest significant resources into making it happen.

A perceptive reader will ask: what does all this have to do with other OODBMS standardization activities, like the ODMG? There is not—yet—a formal relationship between the ODMG's work and the SQL3 efforts. In fact, the organizational nature of the standards process, even in the USA, makes it difficult for there to be a formal relationship with a pseudo-consortium like ODMG. Nonetheless, the members of the organizations talk to one another, and there is an explicit goal to minimize (to zero, if possible) the differences in the object models of the two languages. There is a less firm goal of using SQL3 as the data manipulation language in the ODMG effort, but the model is the more urgent undertaking.

There has also been a lot of discussion among the ODBMS proponents of providing an SQL interface to those products in the form of a class (type) library that provides SQL semantics, if not SQL syntax. This approach has as many skeptics as the MOOSE, and I am unaware of significant work in the public domain in this area.

8 Summary and Conclusions

The work to add object technology to the relational database language SQL is well under way. It is being driven by object-knowledgeable people working for all of the principle relational database vendors. The commitment of those vendors to this direction is illustrated by Oracle's indications that the next major version of its product (ORACLE8) will have support for much of this material, as well as by IBM's recent hints of features forthcoming in its DB2 systems. Other vendors are well-known to be actively pursuing the addition of object technology to their SQL product lines.

There remains significant disagreement about several important issues. The two most important are: whether object instances in an extent should be represented by rows in relational tables; and precisely what model of extent should be chosen for objects. It is no coincidence that these two controversies are so closely related.

It does seem clear, though, that the SQL standards bodies and the ODMG will work together to minimize or eliminate differences in their object models. The Object Management Group (OMG) has recently issued a request for proposals on a query service for distributed objects and SQL3 has been proposed by at least one respondent. Gradually, painfully the various camps will understand one anothers' needs and positions. And the result, with any luck at all, will be a family of object models and languages that can communicate usefully for real applications.

References

[SQL-92] ISO/IEC9075:1992(E), *Information technology — Database Languages — SQL*, and ANSI X3.135:1992, *American National Standard for Information Systems — Database Language — SQL*, both published in 1992.

[FRAME] ANSI document number X3H2-94-338 and ISO document number DBL:RIO-003, *Working Draft Database Language SQL/Framework*, Jim Melton (Ed.), August, 1994.

[SQL3] ANSI document number X3H2-94-329 and ISO document number DBL:RIO-004, *Working Draft Database Language SQL3*, Jim Melton (Ed.), August, 1994.

[CLI] ANSI document number X3H2-94-330 and ISO document number DBL:RIO-005, *Working Draft Database Language SQL/CLI*, Jim Melton (Ed.), August, 1994.

[PSM] ANSI document number X3H2-94-331 and ISO document number DBL:RIO-006, *Working Draft Database Language SQL/PSM*, Jim Melton (Ed.), August, 1994.

[ODMG] *The Object Database Standard: ODMG-93*, Release 1.1, R. G. G. Cattell (Ed.), 1994, Morgan Kaufmann.

Frank Manola and Gail Mitchell
GTE Laboratories Incorporated
40 Sylvan Road
Waltham, MA 02254
{fm02,gmitchell}@gte.com

1 Introduction

Three important standards are being developed that are crucial to Object DBMSs (ODBMSs) and interoperability. The ODMG-93 specifications [Cat94], developed by the Object Database Management Group (ODMG), define proposed Object DBMS standards that are scheduled to be supported by most ODBMS vendors by 1995. The SQL3 specifications [Mel94], currently being developed by the ANSI X3H2 committee, include object extensions to the current SQL standard database language. These specifications will critically influence the future development of relational DBMSs. Varying estimates, from 1996 to 1998, have been given for completion of this standard. Both the ODMG-93 and SQL3 standards have already been discussed in companion papers.

A third important standard is the Object Management Architecture (OMA) [OMG92] and associated specifications of the Object Management Group (OMG). The OMA defines a Reference Model identifying and characterizing the components, interfaces, and protocols that compose a distributed object architecture. Of particular importance is the CORBA IDL (Common Object Request Broker Architecture—Interface Definition Language) specified in [OMG91], since object interfaces must be specified in this language. The OMG OMA, CORBA, and Object Service specifications provide the potential basis both for the development of ODBMS implementations, and also for the development of the distributed object architectures in which ODBMSs would likely be used. OMG specifications have already had great influence in defining the course of object standards, and commercial products based on them have been produced. While these products are not necessarily fully mature, and much work remains to be done, the OMG specifications represent an approach around which many software vendors are rallying.

These standards clearly have great individual importance, due to the number and importance of systems they potentially affect. In addition, it is imperative that systems based on these standards be capable of interoperating, and that there should be as few differences between these standards as possible. Commercial realities will create the need for most ODBMSs to support the current SQL standard (whatever it is and whatever its facilities are) or to provide efficient gateways to other DBMSs that do support it (independently of any other facilities they might support). At the same time, commercial realities will also create the need for DBMSs in general, either ODBMSs or relational DBMSs, to be easily integrated into distributed object architectures, most likely based on OMG specifications. ODMG-93 already provides one illustration of how database concepts can be integrated into the OMG framework. If these commercial realities are not addressed by actually harmonizing these specifications, they will be addressed by providing tight language mappings or bindings between them.

This paper briefly compares and contrasts these standards, with emphasis on their object models, for the purpose of illustrating both the commonalities that may facilitate a harmonization of these standards, and the differences that may present the key technical issues in performing that harmonization.

Current ODBMS products already illustrate varying degrees of support for object query facilities that also incorporate SQL-like relational capabilities. For example, Objectivity/DB's SQL++ query facilities fully support the current SQL standard and provide extensions for object facilities. On the other hand, the Illustra Object/Relational DBMS supports an earlier version of the SQL3 facilities (with a commitment to comply with the finalized SQL3 standard). These products illustrate the merger of object and relational query facilities in actual products, and demonstrate some of the technical problems that must be overcome as ODBMSs move toward supporting full relational query facilities and relational DBMSs move toward supporting object facilities.

While the OMG specifications are clearly important in addressing interoperability among object systems,

the full relevance of the OMG specifications to a discussion of ODBMSs might not be immediately obvious. OMG's OMA and CORBA can be used to extend the way applications interact with an ODBMS by providing connectivity between the applications and the object DBMSs connected to the network. The Object Request Broker is used as an object-oriented messaging component external to the ODBMSs and client applications. Application objects have access not only to objects managed by ODBMSs to which they are directly connected, but also to objects managed by any ODBMSs which they can reach via the Object Request Broker. Both the CORBA specifications and ODMG-93 explicitly discuss a specialized DBMS adapter to connect DBMSs within CORBA.

In addition, the OMG specifications are relevant in this context because the collection of objects within a distributed object architecture *can be considered as an object database* to be managed in the same way as the objects in an ODBMS database are managed. Managing such a collection of objects is effectively "object database management". As a result, the specifications being developed by OMG could be considered as candidate ODBMS standards. OMG specifications define an Interface Definition Language (IDL) for defining the interfaces of objects in the architecture. IDL effectively acts as a DDL for objects (ODMG-93 ODL is designed as a superset of CORBA IDL). In addition, the OMG OMA includes the concept of *Object Services* that provide basic facilities required to support applications in a distributed object environment. These services include concurrency, versioning, transaction, query, security, relationship, and other services similar to those provided by an ODBMS. Event Notification, Lifecycle, and Naming Services are specified in [OMG94], and a Persistence Service specification (which defines a mapping to ODMG-93) has also been approved. Candidate specifications for other services are currently being evaluated by OMG. The Texas Instruments Open OODB prototype [TI93] is an example of the use of an OMA-like architecture to construct an ODBMS as a collection of object services connected by an ORB-like messaging backplane.

This paper is essentially a condensation of [MM94], which compares and contrasts the object models of these standards in more detail.[2] In some cases, the explicit comparisons involve only ODMG-93 and SQL3. This is so because the ODMG-93 object model is designed as a strict superset of the CORBA IDL object model, with additional facilities added that are intended for use by Object DBMSs. Where there is a significant difference between ODMG-93 and IDL, this is noted (throughout the rest of this paper, "IDL" refers specifically to OMG's CORBA IDL).

2 Comparison of the Object Models and Languages

2.1 Objects and Values

The ODMG-93 object model provides a rich set of type constructors for defining both immutable objects ("literals" or values) and mutable objects ("objects") with identity. Both immutable and mutable types can be either atomic or structured. The result is that application entities can be modeled either using literal or object types. For example, a collection of application entities could be modeled as a table (collection) of rows (just as in the relational model), or as a collection of objects. In addition, mutable and immutable types can be freely mixed (e.g., an object type can have a structured literal type as one of its attributes; the structured literal type can have an object type as one of its components; and so on). The OQL query language defined in the ODMG-93 specifications is capable of querying any of these structures. IDL is more limited, in that it allows only user-defined mutable types. It is also more limited in the structures that are supported. These limitations could be addressed by Object Services.

[2] [MM94] was prepared as an activity of the ANSI X3H7 committee, in support of its goal of harmonizing the object standards being developed in different standardization activities. Further details of these models are also contained in X3H7's Object Model Features Matrix [Man94], which in addition describes numerous other object models according to a specified set of object model features. This information is accessible electronically via URL http://info.gte.com/ftp/doc/activities/x3h7.html.

SQL3 to a large extent provides two ways to model application entities: rows in tables (and structures created using them) and instances of Abstract Data Types (ADTs). OBJECT ADTs implement unique identities with OIDs, and are similar to ODMG-93 objects; VALUE ADTs are similar to ODMG-93 literals in having a state-based identity. Tables involve the concept of a row type (a list of column-name/type pairs), and may include the explicit definition of *row types* and *row identifier types*. The row type facility allows complete rows or collections of rows to be stored as values of columns or ADT attributes. With the row identifier facility, which allows rows to be assigned system-defined unique identifiers and allows columns in other tables to reference these row identifiers, the SQL3 rows begin to resemble (structured) objects.

In SQL3 it is also possible to specify whether a column or attribute having an OBJECT ADT as its type is to contain the OID of an ADT instance that is located elsewhere, or an actual ADT instance. This allows for both the usual *reference* semantics found in many object models (including the IDL and ODMG-93 models) as well as for *embedding* or *containment* semantics. This distinction is peculiar, and can create a number of anomalous situations. This has been recognized by X3H2, which is busy working on this aspect of the model.

As currently defined, SQL3 effectively requires a table at the top level of a database schema; i.e., tables (and routines) are the things that have independent existence in the database. The current SQL3 specifications contain considerable overlap between the facilities associated with tables and those associated with ADTs. For example, there are two separate inheritance hierarchies (one for tables and one for ADTs), and both object and row identifiers. Some of this reflects the need for upward compatibility between SQL3 and previous SQL standards—an important consideration for the SQL3 developers. To some extent, these different ways of modeling application entities also parallel the facilities for defining both mutable and immutable objects provided by ODMG-93. However, the integration of ADT and table facilities in SQL3 is not as tight as the integration of mutable and immutable objects in ODMG-93, which results in some apparently redundant concepts. There also currently appear to be some limitations on the flexibility with which ADT and table concepts can be intermixed in SQL3 (although recent additions, such as the row type facility, have improved SQL3 in this regard). For example, instances of ADTs must be stored in tables to be persistent, and the row type and table declaration facilities have not yet been fully integrated. In addition, the reference vs. embedding options for object ADTs are not very clean. At the same time, it is not clear how much ODMG has really thought about the integration of user-defined atomic literal types into the model (aside from saying they can be defined).

It appears possible to define ODMG-93 structures that fully subsume the structures that can be defined in SQL3. For example, a named object of type multiset (or other collection) of tuples could be defined in ODMG-93 to represent each SQL3 table. If ODMG-93 extents can be named (as suggested by the OQL specification) a more direct mapping would be to define an SQL3 table as the extent of an object type. The models would be more parallel if, in SQL3, things besides tables could have names and be persistent, and if the table-oriented and ADT-oriented facilities in SQL3 were more thoroughly integrated into the same type system (recent work on SQL3 illustrates progress in this direction). Also, it seems as if the parallels between ODMG-93 and SQL3 could easily be the basis for defining a “relational subset” of OQL that could, if necessary, have specialized syntax that would be identical to SQL.

2.2 Persistence

ODMG-93 and SQL3 differ in the way lifetimes are associated with entities in the system and in their specification of what kinds of entities can be persistent. In ODMG-93, lifetime is specified at object creation and, once specified, cannot be changed. The ODMG-93 object model indicates that objects have an explicit “delete” operation; however, the language bindings specify that objects are deleted in accordance with the semantics of the particular language (thus, explicit deletion is supported in the C++ binding, and reachability persistence is supported in the Smalltalk binding). ODMG-93 currently specifies three lifetimes that are tied to the storage location of objects (persistent objects are “coterminous with database”). Objects can reference other objects regardless of lifetime, although references to objects are only valid during their lifetime.

Any object of a "persistence capable" type can be persistent; persistence capability is part of the ODL type specification and is implemented as an inherited behavior in the C++ binding. Lifetime is not applicable to literals (immutable objects) since they always exist implicitly. Names can be assigned to individual objects, regardless of their lifetime, and can be used as "handles" for identifying the object.

In SQL3, tables are persistent unless declared otherwise. SQL3 currently requires that ADT instances, to be persistent, must either be stored as column values of (persistent) tables, or as attribute values of other ADT instances that, directly or indirectly, are stored as such column values. These instances can be of any defined type. ADT instances are removed from the database by invoking the instance's destructor function (this is done implicitly when a row containing an instance is deleted from a table). There is no facility for assigning a name to an individual persistent object, except via the table mechanism (directly or indirectly, as described above).

IDL does not support object creation using an operation applied to a type, but rather uses a "factory object" approach that is defined by the Lifecycle Service. The different lifetimes specified by ODMG-93 could presumably be implemented using OMG Object Services by one or more factory objects offering these options. Persistence could similarly be provided using the Persistence Service (one option of which uses ODMG facilities).

2.3 Types and Inheritance

In IDL and ODMG-93, a type is a specification; it can have one or more implementations. A *class* is the combination of a type specification and a specific implementation. ODMG-93, unlike IDL, allows a type interface to specify that an *extent* (a set of all instances of the type) is to be automatically maintained for instances of the type. ODMG-93 does not specify how declaration of an extent interacts with object lifetimes (this will depend on the implementation of the create operations). No implicit conversions are given for either object or structured literal types. Some explicit conversion expressions are defined in the ODMG OQL specification (e.g., element(e) returns the element in the singleton collection designated by e).

In SQL3, an ADT is both a specification and an implementation. DISTINCT types with the same implementation can be specified, but it is not clear how one would specify multiple implementations for the same type. Since ADT values cannot be persistent outside of tables, the extent of an ADT would have to be explicitly created (and populated) by the user. SQL3 allows the definition of CAST functions for converting between types. In addition, a number of built-in type conversions are defined.

The IDL and ODMG-93 models define type inheritance only (i.e., subtyping). If S is a subtype of T, then S inherits all operations and properties of T, and S may define new operations and properties applicable to its instances. A subtype can specialize the properties and operations it inherits, but there are no rules given to indicate what kinds of refinement are correct. In ODMG-93, a type can inherit from multiple supertypes, but must rename same-named inherited operations or properties. "Subtyping" of structured literals requires that both have the same structure at every level (thus, a subtype may not have *more* structure than its supertype), and that the type of each 'subobject' of the supertype is the same or a subtype of the corresponding subobject of the subtype.

Subtyping for ADTs in SQL3 is similar to ODMG-93 subtyping: two objects have the same type if and only if they are instances of the same named type; an object of a given type can be assigned to an object of any of its supertypes; a type can inherit from multiple supertypes, but must rename same-named inherited operations or properties. ADT subtyping in SQL3 also defines implementation inheritance (type and implementation inheritance are bundled together).

SQL3 also supports table inheritance. The row type of a subtable is an extension of the row type of its supertables (i.e., a subtable adds columns) and the row identifier type of a subtable is a subtype of the row identifier type of its supertables. Rules are defined to keep the row membership in sub- and super-tables consistent, so that they reflect the containment semantics expected of the extents of sub- and super-types.

Both ODMG-93 and IDL support the polymorphism implicit in subtyping and operator overloading. SQL3

supports similar polymorphism for both ADTs and for tables in subtable hierarchies. ODMG-93 also supports parameterized collection types. SQL3 supports parameterized collection types and, in addition, supports *template types* to allow the specification of user-defined parameterized types (which may be either structured or atomic). There appears to be nothing corresponding to user-defined template types in ODMG-93.

2.4 Methods and State

In ODMG-93 and IDL, operations are defined on individual types. The interface of a type includes operation signatures: argument names and types, possible exceptions, and result types. Both IDL and ODMG-93 define *classical* object models; each operation has a distinguished argument (the first argument) considered for dispatching. Operation names may be overloaded, in which case dispatching is based on the most specific type of the first argument of the operation call. Operations are implemented by methods defined in type implementations; an implementation might also define additional methods (not visible at type interfaces as operations).

Since SQL3 defines both the interface and the implementation of ADTs, it includes PUBLIC, PRIVATE, and PROTECTED encapsulation levels for both ADT attributes and routines, and mechanisms for actually defining (or referring to) aspects of the implementation. ADT operations are implemented by *routines*, which may be defined either entirely in SQL3 (using SQL3's new programming constructs) or in arbitrary programming languages. SQL3 also supports operations that are independent of ADTs or other specific data structures, and defines a module facility for managing such routines. Such operations can be defined to take table rows (from tables with row identifiers) as arguments, in which case the operations resemble operations on objects. SQL3 defines a *generalized* object model; operations are dispatched based on all argument types, using a "most specific routine" concept.

In ODMG-93, object state is modeled by the properties of an object. An ODMG-93 property is defined as part of the type interface and can be an attribute or a relationship. Attributes take literals as their values; relationships can only be defined between two nonliteral object types. This definition for attributes means that ODMG-93 is not exactly a superset of IDL, since IDL attributes may be of either literal or object types, and IDL does not explicitly define a separate relationship concept.

State, in both ODMG-93 and IDL, is *abstract* state; there is no implication that attributes or properties are necessarily implemented directly as stored data. ODMG-93's ODL, like CORBA IDL, defines only interfaces, hence all object characteristics are, in SQL3 terms, PUBLIC. Mechanisms for specifying object implementations (either methods or internal state) are not explicitly provided in ODL; however, object implementations could be specified using the ODMG-93 language bindings.

Both ODMG-93 and SQL3 support non-object state, in the form of literals associated with schema-defined names (e.g., rows in tables in SQL3). In SQL3 (one-way) relationships are modeled by ADT attributes containing OBJECT ADT or row identifiers; n-ary relationships can be modeled using tables.

ODMG-93 supports standard object type inheritance and operation overloading. SQL3 supports this for ADTs, and supports a separate table inheritance concept with the ability to define overloaded operators.

2.5 Language Bindings

IDL has a C binding defined, and a C++ binding is being voted on; a Smalltalk binding is under discussion. ODMG-93 has bindings for C++ and (the beginnings of) one for Smalltalk, but none for other languages. The ODMG-93 bindings define the "tight" sort of programming language interface generally found in ODBMSs; i.e., a direct mapping of ODMG-93 object model concepts to programming language object model concepts, support for both persistent and transient instances of programming language types, more-or-less transparent movement of objects between the database and application program, etc.

The current SQL standard (SQL92) defines language bindings for a number of standard languages. These must be extended in SQL3 to support the additional facilities. These bindings assume a "looser" type of pro-

gramming language/DBMS interface than ODMG-93 (the assumption being that the type systems are in two different "spaces"). A key aspect of the individual language bindings is the definitions of correspondences between SQL data types and host language data types. In SQL92, these type correspondences are defined only at the level of elementary scalar data types. There are no type correspondences defined for structured types, e.g., between a row of an SQL table and a flat record or structure in a programming language (although some such correspondences would be relatively straightforward to define). There are currently no bindings defined between the SQL3 ADT extensions (or rows with row identifier values) and object classes or types in object-oriented programming languages such as C++ or Smalltalk, although some preliminary papers have been submitted to X3H2 on this subject.

2.6 Model-Defined Languages

OMG's IDL is intended only for defining object interfaces. Implementation and manipulation of these objects is done via programming language bindings. IDL-based query languages are included in some of the proposals for an Object Query Service.

ODMG-93 specifies an object definition language (ODL) that supports the ODMG object model[3] and is compatible with IDL. ODL is programming language independent. ODMG-93 also specifies OQL, an SQL-like object query language that provides declarative access to objects (the OQL syntax and semantics differ from those of SQL in some cases; some examples are given below). SQL3 also provides both data definition language and query language facilities. In addition, new facilities in SQL3 are intended to support complete programming language capabilities.

ODMG-93 mentions that multiple concrete syntaxes for OQL are possible. Also, as noted earlier, the ODMG-93 data structures appear to be a superset of those that can be defined in SQL3. Hence, it should be possible to define a binding of "pure" SQL3 syntax to the OQL semantics, together with a corresponding mapping of SQL3 data structures to a subset of the ODMG-93 data structures. Further work should be done to merge the capabilities of SQL3 and ODMG-93 based on this idea.

OQL provides a number of facilities for querying and/or returning complex data structures. For returning data structures, OQL allows building instances of any (combination of) existing types as a query result. The type of the result must first be defined so that an appropriate create operation exists; a query cannot return an entirely new object type (constructed on the fly) as a query result (general facilities of this type are, however, in the area of research rather than practice). SQL3 appears somewhat more limited in this respect, given the current lack of integration of the ADT-oriented and table-oriented facilities. It is still only possible to return tables from SQL3 queries, even though those tables can contain ADT instances (including structured ADT types) and some forms of nested literal structures.

Both OQL and SQL3 provide facilities for operating on user defined classes of objects, and allow queries to range over, e.g., multiset and list collections, as well as sets. Both languages allow these collections to be results produced by nested query expressions as well as schema-defined collections. Both allow the use of procedurally-specified operations in query predicates (in SQL3, ADT operations must be invoked on ADTs accessed directly or indirectly via columns). Both also restrict queries to object (ADT) characteristics defined at object interfaces (queries do not break encapsulation). SQL3 provides encapsulation for tables only to the extent that views are considered as providing encapsulation. Both OQL and SQL3 also provide facilities for querying nested structures (e.g., *path expressions* for following references between objects, or for querying embedded values). In the case of SQL3, these nested structures may be defined using the row type and row identifier facilities.

One specific way in which OQL and SQL3 differ is the GROUP and SORT operations in OQL, which are somewhat different from what appear to be corresponding GROUP BY and SORT BY clauses in SQL (a point also raised in [Kim94]). The differences between these facilities is not just syntactic. The OQL operations

[3]There appear to be inconsistencies in [Cat94] that still need to be addressed between the description of the model, the ODL definition and the programming language bindings. For example, parameterized collection types in the model are not fully supported in the ODL.

are not additional clauses for a SELECT query, but are stand-alone operations that could be nested within a SELECT query (or vice versa). For example, GROUP BY WITH is a stand-alone query in OQL; the WITH clause is actually used to define a computation over groups (it is different from GROUP BY HAVING in SQL in which the HAVING clause is like a WHERE for grouping). For example, the query (from [Cat94])

```
group e in Employees
by (department: e.deptno)
with (avg_salary: avg(select x.salary from x in partition))
```

returns a set of structures matching each department with the average salary of all employees in the department ("partition" is the default name for a grouped set). Although queries can be written that result in grouping or sorting as defined in SQL (e.g., GROUP x in Select(etc.) BY etc.), having Group and Sort as separate operations allow them to be combined in interesting ways with other operations. These kinds of operation are possible in OQL because the resulting nested structures are definable in the ODMG-93 object model, whereas, since they are not relations, they cannot be treated as separate structures in SQL.

Another difference between the languages is that in OQL there are no statements defined as part of the query language syntax corresponding to SQL's INSERT, DELETE, and UPDATE statements for creating new objects, updating a set of objects, etc., based on search conditions [Kim94]. OQL explicitly notes that it relies on type-defined operations for doing updates. This is possible because such operations can be defined (and in fact *must* be defined) for individual object types, rather than defining general UPDATE operations for tables, as in SQL. For example, in OQL, creation operations are defined for each defined type, and operations for inserting and removing members of sets are defined for set types. The use of these operations in performing general types of updates should, however, be more thoroughly explored in ODMG specifications (or related explanatory material). Discussion within the SQL community has noted that combining ADT-specific update operations with SQL's current update syntax for tables is an issue that also must be resolved in SQL3's object facilities.

2.7 Formal foundations

SQL92 has the formal foundation provided by the relational algebra, and the associated concept of "relational completeness". However, this foundation does not apply to many of the extensions provided in SQL3. In particular, a formal foundation for the object extensions remains to be defined.

ODMG-93 currently has no explicit algebraic foundation (although one could be developed) and thus no "completeness" property corresponding to "relational completeness" (although one could presumably be defined for its relational subset). However, OQL is largely based on the O_2 query facility, and a considerable amount of work has been done on the formal foundations of this facility (see [O291]). Such work, possibly extended with other work on object query algebras, could be the basis for such a foundation.

3 Conclusions

Both ODMG-93 and SQL3 provide object models that include both atomic and structured literals and objects. Both also provide object query facilities that are similar in many respects. Strictly as an *object-oriented query* facility, ODMG-93 is currently more complete and better developed, since it starts from an object-oriented base, and thus is not restricted by a relational "legacy". It is also tightly coupled with IDL and OMG facilities in general. ODMG-93 really attempts (without saying so) to provide a superset of the relational model (using structured literals) in addition to its object capabilities. At the same time, ODMG-93 currently lacks object variants of the view, access control, and other facilities provided in SQL.

Strictly as an *object-oriented query* facility, SQL3 is currently somewhat less complete, and requires additional work. This is not unreasonable, since the SQL3 work on objects is at more of an interim stage than

ODMG- 93. The SQL3 object facilities are still very much under development, and changes are regularly taking place. Hence it is likely that a far more polished facility will finally be produced than exists today. Also, the SQL3 work involves many extensions to SQL other than the ones related to object facilities that are the main emphasis in this paper. Recent changes such as the row type facility, and recent proposals addressing the object facilities, reflect movement in the direction of trying to address the outstanding difficulties. Hopefully, further work of this sort will result in a more complete specification, with better integration of table and object facilities. It should be possible to define a mapping from substantial parts of SQL3 to a subset of ODMG-93 facilities.

ODMG-93 and SQL3 each have something to offer the other. Specific aspects of ODMG-93 that should be considered in SQL3 include:

- Objects have separate persistent existence in ODMG-93, and structures such as tuples of objects reference those objects. SQL3 should consider this approach instead of the current mixture of embedding and reference semantics. This would simplify the integration of objects into SQL3, and allow them to more closely resemble objects in other object-oriented systems. (SQL3 appears to be evolving in this general direction already.)
- ODMG-93 uses a classical object model. SQL3 should consider replacing its generalized model with a classical model. It is not clear how helpful the generalized model of SQL3 will actually be in defining object operations. Also, relatively few of the object-oriented programming languages that might be used with SQL3 use generalized models.
- ODMG-93 fully integrates structured literals and objects. Maintaining the separation of table and ADT facilities as currently defined in SQL3 would be unfortunate.
- ODMG-93 defines tightly-coupled bindings between its specifications and object-oriented programming languages. Similar bindings should be considered in SQL3, in order to reduce the "impedance mismatch" between such programming languages and SQL3 data structures. Also, the same sorts of client/server architectures that ODBMSs use now for supporting their object-oriented programming language interfaces might be useful in future Object/Relational DBMSs supporting SQL3.

ODMG-93 has already clearly borrowed some things from SQL, including aspects of OQL's SQL-like syntax (and, to some extent, semantics) and many of its built-in data types. Specific additional aspects of SQL3 specifications that should be considered by ODMG include:

- SQL3's template types allow a user to define new parameterized types. This facility should be considered in ODMG-93. Also, SQL3 has a more complete (or, at least, more completely specified) VALUE ADT facility that could be viewed by ODMG as a model for specification of the semantics of user-defined literals.
- SQL3 provides facilities for views, access control, triggers, dynamic schema changes, etc., that are missing from ODMG-93. Such facilities are required in many applications, and will need to be specified by ODMG. (View facilities specifically for objects/ADTs need to be addressed further by both groups.)
- ODMG should consider SQL3's procedure and module facilities as a model for incorporating support for "free-standing" operations (noted in [Cat94] as a topic for a future revision of ODMG-93).
- SQL3 defines bindings to non-object-oriented programming languages. ODMG-93 defines only (two) bindings to object- oriented languages. SQL3 bindings could be used as a basis for ODMG bindings to non-object-oriented languages.
- SQL3 syntax could be used for a "relational subset" of ODMG-93 structures, since this would support portability of the "relational parts" of object database systems.

- SQL3 specifies user-defined CAST operations. ODMG-93 has no implicit conversions, and provides some explicit conversions in OQL. ODMG should consider a more complete specification of conversions, including guidelines for the specification of user-defined conversions.

Joint work should be encouraged between OMG, ODMG, and X3H2 to address harmonization of their object models (and syntax, where that is necessary). This work should address the issue of defining cross mappings, and pinning down fuzzy areas that exist in both sets of specifications (and, in fact, such joint work between ODMG and X3H2 is in progress).

Acknowledgments

The authors would like to thank Len Gallagher (NIST), Phil Shaw (Oracle), Bill Kelley (UniSQL), and Amelia Carlson (Sybase) for their help with SQL3 concepts, and Drew Wade (Objectivity) for clarifying a particular ODMG-93 concept.

References

[Cat94] R. G. G. Cattell (ed.), *The Object Database Standard: ODMG-93, Release 1.1*, Morgan Kaufmann, 1994.

[Kim94] Won Kim, "Observations on the ODMG-93 Proposal for an Object-Oriented Database Language", SIGMOD Record 23(1), March 1994, 4-9.

[Man94] Frank Manola (ed.), "X3H7 Object Model Features Matrix", Document No. X3H7-93-007v8, May 2, 1994.

[Mel94] Jim Melton (ed.), *ISO-ANSI Working Draft, Database Language SQL/Foundation (SQL3)*, March 1994.

[MM94] Frank Manola and Gail Mitchell, "A Comparison of Candidate Object Models for Object Query Services", X3H7-94-32v1, August 28, 1994.

[O291] F. Bancilhon, C. Delobel, and P. Kanellakis, editors. *Building an Object Oriented Database System: The Story of* O_2. Morgan Kaufmann, San Mateo, CA, 1991.

[OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Rev. 1.1, 1991.

[OMG92] Object Management Group, *Object Management Architecture Guide, Revision 2.0*, 2nd. Ed., OMG TC Document 92.11.1, Sept. 1, 1992.

[OMG94] Object Management Group, *Common Object Services Specification, Volume I, Revision 1.0*, OMG Document Number 94-1-1, March 1, 1994.

[TI93] Texas Instruments, Inc., *Open OODB Technical Overview Release 0.2 (Alpha)*, 1993.

David L. Wells and Craig W. Thompson
Texas Instruments, Incorporated
P.O. Box 655474, MS 238
Dallas, TX 75265
{wells, thompson}@csc.ti.com

Abstract

This article reviews the four specifications for the Object Query Service (OQS) that were submitted to the Object Management Group (OMG) in response to OMG Object Services Task Force RFP #4. The article begins by summarizing and commenting on each OQS submission in turn, then identifies the major technical issues raised by the competing submissions. Finally, we suggest a course of action for resolving the issues. The goal of a consensus submission, acceptable to all parties and beneficial to the community as a whole, is well worth achieving but will require considerable work. It should be noted that the OMG process is dynamic and fast moving; this article is a snapshot circa late November 1994, and the situation is expected to change quickly in the coming months.

1 Background

The *Object Management Architecture Guide*[4] [OMG92a] defines the overall OMG software architecture, which consists of an object-oriented Interface Definition Language (IDL) and message-passing bus (together constituting CORBA) [OMG91], plus an extensible collection of object services. The OMG *Object Services Architecture (OSA)* [OMG92b] describes a collection of basic object services that many other services will depend on. The OMG *Common Facilities Architecture (CFA)* [OMG94a] (in progress) describes another collection of widely useful services.

OMG populates its architecture by issuing "Requests for Proposals" (RFPs) to industry for interface specifications for each service described in the OSA and CFA (also requiring that there be commercially available implementations). OMG OSTF RFP #1 covered the life cycle, events, naming, and persistence services (specifications now adopted, see [OMG94b]); OMG OSTF RFP #2 covered the externalization, concurrency, transactions, and relationships services (adopted); OMG OSTF RFP #3 covers the security service (outstanding); OMG OSTF RFP #4 covers the properties, query, and licensing services (outstanding); and future OMG OSTF RFPs will cover change management, data interchange, and other services. The database community should note that, taken together, these basic object services provide much of the functionality of a componentized DBMS, including both Object DBMS and Relational DBMS functionality.

As mentioned, OMG *Object Services RFP #4* [OMG94c] solicits specifications for the OMG Object Query Service. This OMG object query service is especially important since it represents the confluence of two multi-billion dollar software infrastructure industries—application integration frameworks represented by OMG with its 400-plus member companies, and relational and object DBMSs. Both communities support information sharing in distributed, increasingly object-oriented, client-server and peer-peer environments. A carefully designed unification of these two areas will save a generation of programmers from having to solve for themselves interoperation problems that otherwise will inevitably occur if separate, incompatible framework and DBMS specifications become industry standards.

By the due date of September 26, 1994, OMG received the following four submissions for the Object Query Service (OQS):

[4]Hard copies of OMG documents can be obtained by sending E-mail to `documents@omg.org`. Electronic versions of numbered OMG documents are available through OMG's E-mail server: send E-mail to `server@omg.org` with the line `help` in the body of the message for instructions, or use the `get` command to retrieve a specific document (e.g., `get docs/94-9-29.ps`).

- OMG document 94-9-29: Object Query Service (IBM, Dan Chang; Taligent, Roger Harvey). Paper, PostScript.
- OMG document 94-9-37: Object Query Service (Oracle, Kenneth Ng). Paper, PostScript.
- OMG document 94-9-38: Joint Object Query Service (Itasca; Objectivity, Drew Wade; Ontos; O2; Servio; SunSoft). Paper, PostScript, RTF.
- OMG document 94-9-44: Object Query Service (Craig Thompson, Texas Instruments). Paper, PostScript. See also appendix, document 94-9-45.

Also relevant to the OQS submissions are the two submissions for the property service:

- OMG document 94-9-39: Properties Service (SunSoft, Geoff Lewis). Paper, PostScript.
- OMG document 94-9-43: Object Property Service (George Copeland, IBM; Taligent). Paper, PostScript.

Unless an extension is requested and granted, revised proposals are due to OMG by January 4, 1995. Competing proposals may be merged into a single or small number of proposals.

2 Review of the Object Query Service Submissions

We begin by summarizing the main points of RFP #4 related to the Object Query Service, and then summarize and comment on the individual submissions. We apologize in advance for any over-simplifications of the actual submissions; the interested reader is referred to the original submissions for complete details. Some liberties have been taken with terminology to make it consistent between proposals.

2.1 Requirements for an Object Query Service

The Request for Proposals calls for proposals for the interface specifications for an Object Query Service (OQS) to become part of the OMG Object Services Architecture. The OQS must be based on the OMG object model IDL, and should provide high-performance (i.e., optimized) access to fine grained objects, standard operations for manipulating user-defined collections of complex objects (i.e., more than just sets), make use of attributes, inheritance, and procedurally specified operations in predicates, allow the use of the OMG Relationship and Property Services, and enable federation of heterogeneous query servers.

2.2 The Submissions

IBM/Taligent. The IBM/Taligent submission is a minimalist proposal stressing the structural characteristics needed to federate existing query engines. It does not propose a particular query language, and in fact, argues that no particular query language be standardized at this time. Responsibility for processing queries is split between the Query Processor, which partitions queries into subqueries and collects results, and various Collection classes, each providing an interface allowing predicate based selection of the collection's contents. Most work is done by the collection classes. This approach has the advantage that new collection types can be added without requiring changes to the Query Processor itself. A disadvantage is that, because all collections are required to present a query interface, it is impossible to naively iterate over or index collections designed for other purposes that do not themselves define a query interface. Separating the class hierarchy into Collections and Queriable-Collections that inherit from Collections would fix this problem. This would allow for collection types such as Graphs that may not have a query capability defined for them, but that still manage collections of objects.

Non-collection objects are also queriable, as long as they present the appropriate interface to the Query Processor, giving a sort of seamlessness in that one needs to know less about an object to ask it a question, allowing the OQS to handle a wider variety of requests.

The boundary between collections and non-collections is slippery. For example, is an index with a lot of internal structure considered to be a collection (it has more structure than any of the sample collection types listed in Section 4.2 of the submission) or is it a queriable single object? Does the distinction even matter?

Queries are passed between federated query processors as any kind of object including strings, graphs, etc. This lack of standard query language for internal communication is troublesome for the goal of federation, because it requires each query processor to know what other query processors understand and makes adding new kinds of query processors difficult.

The Query Framework does not provide an interface to specify a new algebra for objects. How is information about optimizability to be made available? Since collections are specifiable separately, where is the knowledge of processing algorithms held? Can one project a result? Can one create a result of a collection type other than that of the underlying collection types from which objects came? Is there a notion of a `JOIN` to create new object types? If so, where does this fit in? Since `JOIN` is not an operation over any given collection, it does not seem to fit in a model where most work is done by the collection classes. If base objects come from two different collection types, say `list` and `set`, what is the result type? In general, it appears that this proposal is biased toward algebraic selection from collections, with no other operations. Limiting operations to filtering collections seems to be a limited view of what queries do. Perhaps this a consequence of not providing a query language.

The strengths of this proposal are that it provides a federation framework, separates collections out from the query processor, and has a mechanism to support additional collection types. Weaknesses are that there is not a strong query model, there is no language for exchanging subqueries within the federation, all collections must support a query interface, and indexing is bundled with collections.

ODMG. The ODMG submission proposes a query language called OQL based on a merger between SQL syntax and an extension to the OMG/IDL called the Object Definition Language (ODL). ODL is upward compatible with IDL, adding visible object identity, explicit relationships, cardinality constraints, a specific class library, which includes collection classes, and other features described in *The Object Database Standard: ODMG-93 (Revision 1.1)* [ODMG]. The definition by the OQS of relationships when there is an OMG Relationships Service being specified, while historical (ODL existed before the Relationship Service), breaks service orthogonality but is not an important part of the query submission.

The ODMG proposal addresses federation. Objects either have the predicate applied to them by brute force, or know how to internally evaluate a predicate passed to them. The latter are called queriable collection types. It appears that queriable collections are the way that existing execution engines are encapsulated, since both manage collections and know how to evaluate queries over the managed collections.

Queries are passed internally as strings using OQL. Name resolution and scoping rules are not specified.

ODMG OQL is not quite SQL compatible, although subsequent to the submission, ODMG has declared its intention to make OQL totally compatible with SQL92 Entry Level and to try to influence X3H2's SQL3 so that ODMG can adhere to that emerging ANSI/ISO standard. Issues related to covering OQL and SQL92 are discussed in the next section. In this section, we address OQL as presented in the OQS submission.

OQL is largely a `SELECT`-only language. `DELETE` and `UPDATE` are not supported, the intent being that these operations will be performed through the normal interface of selected objects. This is in keeping with object encapsulation, but precludes optimization. Also, it is not clear how `PROJECT` and `JOIN` are supported, particularly with regards to the handing of object identity in the joined or projected results.

The strengths of this proposal are the support of a substantial part of the OODB vendor community, an IDL-compatible object model used by the OQS, and a federation story with a common internal interchange format

(OQL). The weaknesses are that not all ODMG vendors participated in the submission, lack of compatibility with SQL, unspecified details about how object identity is preserved through projection and join, and not identifying collections, relationships, properties, and indexing as being provided by other Object Services.

Oracle. The dominant aspect of the Oracle submission is the use of SQL3 ADTs as the data model to enable the reuse of existing (modified) SQL engines. The use of SQL3 is both a strength and a weakness. SQL3 ADTs provide an object model different from OMG IDL. SQL3 has multiple inheritance; multi-methods instead of IDL's classic object model; public, private, and protected state; friend functions; a method implementation language; a distinction between objects and object references; enumerated types; and unions. For a more complete description of the differences, see the article by Manola in this issue as well as [Man94] and [MM94].

There is a claim that SQL3 maps to IDL via intermediate bindings of SQL3 to C++ and IDL to C++. However, this does not appear to be the case. There is, in fact, no C++ to IDL mapping; rather there is an IDL to C++ mapping which is an injection that does not cover full C++. In particular, there are C++ constructs that do not map back to IDL. Since SQL3 has many features not present in IDL, it is likely that the SQL3 to C++ mapping uses features of C++ that do not map to IDL, making a straightforward SQL3 to IDL mapping unlikely. This will have interesting consequences beyond the obvious "impedance mismatch." Specifically, the difference in dispatch model leaves the possibility that executing an operation `FOO(A, B, C)` using the IDL dispatch model based on classic objects, and executing `FOO(A, B, C)` within the query engine using multi-methods could cause different functions to be executed, with the result that user-defined scans of collections and queries over the same collections produce different results.

SQL3 itself defines collection types, but differently from ODMG (which is different again from the proposed C++ Standard Template Library). This is a problem for two reasons. First, SQL3 appears set-centric, with other kinds of collections (e.g., arrays and trees) being ignored. Secondly, collections are not always manipulated via the OQS. Application programs in languages other than SQL3 are written to iterate through collections and insert, delete, and reorder elements without queries. To force the OQS to be present simply to provide a collection class interface seems unnecessarily heavy-weight. Similarly, indices are apparently managed by the OQS, even though indices may also be separately useful.

Query specifications are passed as SQL3 strings. This has the advantage of being a common language for federation (assuming all query engines speak SQL3), but some issues are outstanding. The principal problem is where names get bound, and the scope of allowable names. The assumption seems to be that all names can be resolved by the OQS. However, objects will have internal state, predicates may use program variables, etc. These can be resolved using "`%`" notation as in SQL, but if queries are passed around, what binding environment is used: the enclosing environment of the query, the object's scope, or the local environment of the OQS that actually does the processing?

The major strengths of this proposal are that Oracle is an established DBMS vendor and they have proposed a full query language. The major weakness is that the query language is not based on IDL and is not obviously compatible with it. Other weaknesses are in how separate query engines compose in a federation and the tight coupling of collections and indices with the OQS.

Texas Instruments. TI proposes a query language, OQL[IDL], using SQLi = (SQL89, ...) syntax with IDL replacing relations and ADTs as the data model. All queriable objects must have definitions in IDL. A binding of OQL[IDL] to OQL[C++] is provided using the OMG approved IDL to C++ binding. In the proposed language, class definitions and user-defined predicates may appear in the `SELECT-FROM-WHERE` clauses in the obvious places.

Collections are expected to be defined externally to the OQS because they are useful to other services. Similarly, it is proposed that indices be managed by a newly identified Object Indexing Service that can maintain indices over collections that do not know about the existence of the object query service.

Federation is supported by minimal commitment on the part of query engines as to their internal operation. Subqueries are transported in OQL[IDL]. Name resolution occurs based on calls to the Object Name Server for globally known objects and by the local OQS for program variables unknown outside the environment in which the query originated. The submission details how the various query servers would interconnect in a federated mode. The submission is careful to ensure that the OQS does not define any capabilities potentially used by other services that should be partitioned out into their own service; hence the call for separate templates, collections, indexing, and relationships specifications.

Interfaces are sketched but not fully specified for connecting to query servers (based in part on ODBC[5]) and for extending a query optimizer with algebraic transforms known to be applicable to methods of application-defined objects. The need to inform the OQS about which methods are side-effecting is pointed out as essential to the ability to optimize queries safely.

The strengths of this submission are that it partitions the OQS design into a number of orthogonal decisions, and that it replaces SQL ADTs with OMG IDL. Weaknesses are that not all interfaces are fully specified, and that TI is not a mainline DBMS vendor.

3 Issues that must be Resolved

As can be seen even from the brief descriptions above, there are substantial differences between the OQS submissions. Some of these are areas where the submissions emphasize different aspects of the OQS specification (e.g., query language definition vs. support for federation) and, in these cases, a simple unioning may be possible. In other cases, functionality is bundled in different ways (e.g., should collections be defined as part of the OQS or separately), but even here there may be no great philosophical differences. In the more vexing cases, there are fundamental differences (e.g., SQL3 ADTs vs. an IDL-based data model) that can be resolved only by long discussion. The issues are orthogonal corresponding to separable decisions that the submitters must eventually agree on. The issues fall into the following broad categories:

- object model (ADTs vs. IDL)
- query language
- SQL3 PSM
- legacy database compatibility
- model of federation
- query server (engine)
- collections
- templates
- standard class libraries
- relationships and properties
- indexing service
- query optimization

[5] ODBC is Open Database Connect, Microsoft's implementation of Remote Data Access (RDA), a standard for SQL gateways.

Object Model (ADTs vs. IDL): OMG OSTF RFP #4 requires that the query service use the OMG object model IDL but mentions “bindings,” allowing the underlying object model defined by IDL to be represented in other languages. The IBM, ODMG, and TI submissions are all based on IDL. The IBM submission defines collections using IDL. The TI submission adopts IDL as the object model but notes opportunities for growing IDL to meet SQL3 or ODMG needs, perhaps as an OMG Object Model Profile. The OMG Object Model Task Force has defined provisions for this. The ODMG submission adopts ODL, which is an upwards compatible of OMG IDL plus ODMG relationships (not OMG relationships), extent and key keywords, a specific (idiosyncratic?) class library that includes collections and templates. The Oracle submission takes a different tack entirely, providing the separate world of SQL3 with its ADT-based object model, assuming a “binding” to IDL will suffice. The binding is not really provided; that both IDL and ADTs map to C is too weak since both are injections.

Both ADTs and IDL are C++ derivatives, but as sketched previously are substantially different and not easily reconciled. Both are also supported by huge communities, the multi-billion dollar relational database community, and the 400-plus member OMG. If the OQS supports a different object model than the rest of OMG Object Services, an enormous internal interoperability problem will be created within the OMG world. This is a **huge** issue, that dwarfs all others by comparison. If we can work out a merger of the X3H2 ADT object model and OMG IDL, *any* amount of convergence will benefit millions of programmers, who face the prospect of writing $B of glue software to map between dissimilar ADT and IDL. Note that ANSI X3H7’s charter is object model harmonization, and they are involved in the process of helping to understand the differences between SQL3 ADTs and IDL.

Query Language: To a large extent, the query language is independent of the data model. For instance, the Oracle, ODMG, and TI submissions all use a variant of SQL, despite having different data models as discussed above. The IBM submission wants to remain Query Language neutral. This separation of query language from data model and query service is architecturally desirable, since it allows flexibility in the way queries are embedded into various host languages. However, to support federation, there is a need for some internal query language syntax for exchange of subqueries. The three submissions that recommend SQL-like languages all use that same language for federation.

While several query languages are clearly possible, the consensus seems to be at least to provide a default one as close to SQL as possible, both to make use of existing query engines and for programmer convenience. This does not preclude other query languages, for instance, a lightweight query language for searching Property Lists.

We have to be careful about “near subsets” of SQLi. The ODMG submission references ODMG-93, which is less and more than SQL, but at the recent ODMG meeting there was a decision to move to SQL92 Entry Level for the non-object (relation-like) portion of OQL. The TI submission states SQL89 but means SQLi where i = 89, 92 level x, or SQL3 or the new NIST SQL for files. Both ODMG and TI would replace SQL Relations and ADTs with OMG IDL, possibly with ODMG ODL extensions. Reaching detailed agreement here should not be too difficult, since most differences appear gratuitous.

It is desirable to retain as much as possible the heritage of SQL3 syntax (e.g., `CREATE` statements) and also the IDL syntax (e.g., `INTERFACE`) DDL command. In other words, the issue is, can the languages be made semantically the same while allowing syntactic variants?

What are the semantics of “joins” involving, say, sets and ordered lists? Does precedence determine whether the result is a set or list? No one’s submission specifies this except possibly by reference.

ODMG allows no bulk update operators corresponding to `UPDATE-WHERE` or `DELETE-WHERE`. It seems useful to add these for SQL compatibility and to permit additional opportunities for optimization.

Does the fact that IDL is inherently static with a clumsy dynamic interface cause problems? SQL3 appears much more dynamic. Is increased dynamism essential for query processing?

Navigation via IDL with relationships and properties is a form of query. What OQSs, at least those based on

SQL, provide is the associative query or filtering capability.

SQL3 PSM: No submissions explicitly defines a computationally complete procedural specification language for writing methods, but Oracle's submission references SQL3's PSM, which is invented for this purpose. In so doing, far from being programming language neutral, SQL3 has invented yet-another-programming-language (YAPL), PSM, for defining methods. They also provide an interface description language for externally defined operations, defined in programming languages. IDL does the latter and provides interface definition, so the interface description parts of IDL and SQL3 ADTs compete. OMG has steered clear of a PSM for IDL.

Legacy DB Compatibility: ODMG-93 talks about mappings but does not provide them. Of course there is now an OMG IDL → C++ Mapping and IDL → Smalltalk is coming. Subsequently, ODMG agreed to reach SQL92 compatibility and is cooperating with the X3H2 committee. TI also provides a mapping for how to wrap legacy relations in RDBMSs and have IDL sets of IDL instances corresponding one to one to the tuples. This is just one mapping, but is straightforward, useful, and consistent with the rest of a full IDL in place of ADTs.

The SQL3 community has a Call Level Interface (CLI) standard progressing. We still need to look at on how this combines SQL statements and host programming environments.

Model of Federation: Given that we have more than one instance of query engines on federated CORBAs, there is a need for a object query service federation architecture. It must account for the homogeneous and heterogeneous cases and for gateways. All submissions cover this in one way or another. However, we think that the specification for federation should be separated from the OQS engine discussion, the requirements for it should be made clearer, cross-OQS optimization should be accounted for, and an intermediate language (e.g., OQL[IDL]) for exchanging subqueries must be defined. There are three flavors of federation proposed: the IBM submission seemed to have a recursive descent model, ODMG has a nested pass through model, and Phil Shaw of Oracle proposes to support federation "under the covers" by subclassing off the defined OQS (i.e., federated servers are a subclass). Another issue is whether subqueries are sent to collections or to OQS engines for processing. We prefer the latter because it better supports multi-collection operators such as `JOIN`.

Query Server (engine): The ANSI/ISO CLI (Call Level Interface) defines the facilities for connecting to query servers and for submitting query requests. There was little discussion of this aspect of the OQS interface (only TI even mentioned it) but there appears to be little reason for defining the same capabilities in another way.

There may be several server subclasses for various flavors of SQL (e.g., SQL89, SQL92, level X, ...) or lightweight ones such as what the Object Property Service seems to require. All submissions cover query servers and these descriptions need detailed comparison.

A key question is exactly what a query server is. If *any* kind of query engine is allowed, how do we distinguish query engines from general function servers? What is special about them? Other operators also operate on collections. Is it just a convenience for traders to group query engines under one class, or do general query optimizers know something special about the class of all query optimizers? Phil Shaw suggests that query engines are distinguished by providing cursors to iterate through large collections. All submissions either support or do not preclude cursors.

Another question about query servers has to do with the processing of collections and join operations. Some of the proposals made collection classes responsible for predicate-based selection from themselves, while others took a more traditional approach. This is somewhat more than an implementation issue, since requiring collections to support predicate-based selection means that generalized collections become heavier weight than they would need to be to support the OQS. It also begs the question of whether indexing is provided by the collection, by the OQS, or by a separate Object Index Service so that indices could be used by services other than the

OQS. This level of detail will not be apparent to users of the OQS, but will affect the developers of other object services.

Related to the question of selection being performed by collections is the handling of joins. If, as IBM suggests, most processing is done by the collection objects, what performs operations (such as `JOIN`) on multiple collections? If it is still the responsibility of the collection, then collections must know something about other collections, which seems to violate the cleanness of the model. This needs more thought.

Collections: OQS could only specify a single minimalist default collection library, like IBM's, or we could push for a single common collection library, or we could assume that there will be more than one common collection class library and that queries should obey a protocol to work on one or more of these.

There seem to be many Collection Class Libraries to choose from. Obviously, there is need of collections for other purposes than queries. IBM, ODMG, and Oracle all recommend different collection class libraries. The TI submission additionally identifies the C++ Standard Template Library (STL) as an additional candidate. Finally, the OMG Common Facilities Task Force has separately identified a Collection Facility (service) as a common horizontal service. We need to alert CFTF and TC of the opportunity for synergy in cooperating to select a Common Class Library. The right approach to selecting a common class library will probably be controversial. Whatever the case, collection classes will need to be specified in IDL (or ADTs if you grew up in the SQL3 camp). The decision on Class Libraries may be deferrable if we find that the Object Query Service, query language, and object model are independent of most of the details of the collections supported in the collection class library.

The IDL-based submissions all agree that collections are orthogonal to persistence, security, and other services just as IDL is. It is not clear whether SQL3 ever expects anyone to program transiently in SQL3. The IDL-based submissions apparently also agree that instances can exist without being part of some interface's explicitly maintained extent. SQL3 appears to require membership in a collection as a prerequisite for existence. The use of an `EXTENTS` key word that tells an interface to maintain a collection of all instances it ever created might be a way to support both views.

Finally, is it really necessary for queriable collections to inherit from IdentifiableObject as the IBM and ODMG submissions suggest?

Templates: As far as we are aware, IDL only supports two built in templates, `string` and `sequence`, and has no general purpose template facility analogous to C++ templates. The TI submission notes this defect in IDL (if it is not being addressed in the ORB Task Force) and suggests IDL templates based on C++ templates. Templates for IDL are implicit in the ODMG submission since their ODL collections are defined with templates, which do not appear to be defined in ODMG-93. SQL3 defines Templates, but does not propose them in the OQS submission since the goal is to have a working implementation quickly.

Other Standard Class Libraries: A logical consequence of ADTs and PSMs is that someone will suggest a work item for standard class libraries for GIS, Text, and Multimedia all written in this ADT-PSM language to allow sharing and queriability. But then we will have parallel ones in C++ and other languages, which is not a good idea. Note that the STEP Express community is already in the business of defining common vertical class libraries, and the OMG Common Facilities and various OMG SIGs are beginning to explore this same terrain. We need to keep these communities coordinated.

Relationships and Properties: The ODMG proposal defines its own "relationships" that are not the same as those that are defined by the OMG Relationship Service. Relationships are definitely needed independently of an object query service. Some resolution of ODMG relationships and OMG Relationships is needed. A main

issue of dependency between Relationships and the Query Service involves the opportunity to optimize across relationships (and path expressions).

Indexing Service: Only TI identified the need for a separate Object Indexing Service and did not specify an interface. This interface will need to be specified eventually, since optimizers will need to understand the properties of indices. This can be left to later since most of OMG never needs to see this interface, which affects the optimizer and possibly federation.

A related issue is how indices are to be maintained if an updated object appears in two or more indexed collections. Does the Event Service notify the collections and the Index Service? The question is where the locus of control resides.

Query Optimization: Like indexing, optimization can be hidden for now, but federation of OQSs will eventually require defining some Query Optimization Service interface. Some submissions seem to make optimization the job of the collection being queried. There is a very general Lisp based optimization primitive for wrapping any function with an optimizer. This area needs more work.

4 Toward An Action Plan

As the last section described, there are many issues that will need consensus before a merger of OMG query submissions can take place. Of these, the major issue is going to be to resolve the competing object models while retaining some form of compatibility with SQL92, SQL3 ADTs, OMG IDL, and ODMG ODL. A start was made at this objective at an ad hoc meeting of X3H2 and ODMG in Palo Alto, California, on November 17, 1994. The meeting focused on SQL92 and ODMG OQL compatibility. Continued collaboration is expected over the next several months. In future meetings, it will be important that points of view from OMG also be represented if the merger of the object query service submissions is to reflect true consensus of not just the SQL3 and ODMG communities but also the OMG community.

As a final note, it appears essential that the OQS submitters request an extension from OMG of four to six months to achieve the object model convergence necessary for even the first cut at an OQS. It will be much better to lengthen the OMG process to arrive at consensus than to adopt any one minority position prematurely.

At the next meeting of OMG in New Jersey, the submitters will evaluate the submissions and discuss an action plan for arriving at consensus for a merged OQS specification.

References

[ODMG] *The Object Database Standard: ODMG-93*, Release 1.1, R. G. G. Cattell (Ed.), 1994, Morgan Kaufmann.

[Man94] Frank Manola (ed.), "X3H7 Object Model Features Matrix," Document No. X3H7-93-007v8, May 2, 1994. See also: `http://info.gte.com/ftp/doc/activities/x3h7.html`.

[MM94] Frank Manola and Gail Mitchell, "A Comparison of Candidate Object Models for Object Query Services", X3H7-94-32v1, August 28, 1994.

[OMG91] Object Management Group, *The Common Object Request Broker: Architecture and Specification*, OMG Document Number 91.12.1, Rev. 1.1, 1991.

[OMG92a] Object Management Group, *Object Management Architecture Guide, Revision 2.0*, 2nd. Ed., OMG TC Document 92.11.1, Sept. 1, 1992.

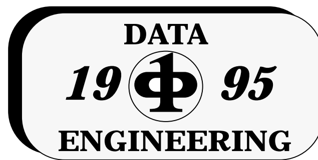
[OMG92b] Object Management Group, *Object Services Architecture*, Issue 6.0, OMG TC Document 92-8-4, August 4, 1992.

[OMG94a] Object Management Group, *Common Facilities Architecture*, Issue 3.0, OMG TC Document 94-11-9, November 9, 1994.

[OMG94b] Object Management Group, *Common Object Services Specification, Volume I, Revision 1.0*, OMG Document Number 94.1.1, March 1, 1994.

[OMG94c] Object Management Group, *Object Services RFP #4*, OMG Document Number 94-4-18, April 18, 1994.

CALL FOR PARTICIPATION ICDE 11



11th International Conference on

Data Engineering

March 6-10, 1995

The Grand Hotel, Taipei, Taiwan, R.O.C.

Sponsored by the IEEE Computer Society-TC on Data Engineering, National Tsing Hua University, and Providence University

SCOPE OF THE CONFERENCE

Data Engineering deals with the modeling and structuring of data in the development and use of information systems, as well as with relevant aspects of computer systems and architecture. The 11th Data Engineering Conference will provide a forum for the sharing of original research results and engineering experiences among researchers and practitioners interested in automated data and knowledge management. The purpose of the conference is to examine problems facing the developers of future information systems, the applicability of existing research solutions and the direc ions for new research.

TECHNICAL PROGRAM HIGHLIGHTS

Research Papers Sessions on:

• Data Mining and Information Discovery • Performance Evaluation • Query Processing in Object-Oriented DBMS • Active Databases • High Performance and Parallel Systems • Change Management • Query Optimization • Temporal and Sequence Databases • Integration of Heterogeneous Systems • Application Systems • Transaction Management • Access Methods and Caching • Fuzzy and Proximity Query • Database Modelling • Multimedia Servers • Logic and Artificial Intelligence • Interoperable Systems • Sampling and Compression • Object-Oriented Systems.

Keynote Speakers:

- Hector Garcia-Molina (Stanfor University): "Challenges and Pitfalls on the Way to the Digital Library of the Future"
- Won Kim (UniSQL): "On Object-Relational Database Technology"

Panel Discussions:

• Document Repositories • Information Resource Discovery
• The Futures of Active Databases • Workflow Automation.

Technology and Application Track:

- Practice-oriented presentations of applications of database technologies.

TUTORIAL PROGRAM HIGHLIGHTS

1. **Multimedia Database Systems:** Arif Ghafoor (Purdue University), March 6, 9:00 a.m. - 12:00 noon.
2. **High Performance Transaction Processing:** C. Mohan (IBM Almaden), March 6, 9:00 a.m. - 12:00 noon.
3. **Indexing Multimedia Databases:** Christos Faloutsos (University of Maryland),, March 6, 1:30 p.m. - 4:30 p.m.
4. **QueryProcessinginParallel Database Systems:** Ming-Syan Chen, Hui-I Hsiao (IBM Watson), March 6, 1:30 p.m. - 4:30 p.m.
5. **Implementation of Document Retrieval Sysems:** Dik L. Lee (Ohio State University and Hong Kong Uniersity of Science and Technology), March 7, 9:00 a.m. - 12:00 noon.
6. **Multidatabase Interoperation: Perspectives of Researchers and Practitioners**: Amit Sheth (University of Georgia), March 7, 9:00 a.m. - 12:00 noon.
7. **Spatial Databases:** Hanan Samet (University of Maryland), March 7, 1:30 p.m. - 4:30 p.m.
8. **From Database Systems to Knowledge-Base Systems: An Evolutionary Approach:** JiaWei Han (Simon Fraser University, Canada), March 7, 1:30 p.m. - 4:30

LOCAL ARRANGEMENTS

The 11th ICDE will be held at the Grand Hotel in Taipei, Taiwan, R.O.C. Taipei is the principal cultural, economic, and political center of Taiwan. This metropolis is one of the most fascinating in all of Asia. The Grand Hotel, the official conference site, is a major landmark in Taipei, commanding a panoramic view of the city. Designed after the Forbidden City of Peiking, the Grand is symbolic of ancient architectural grandeur.

VISA INFORMA TION

Please contact Allen Wu, **chunghaw@cs.nthu.edu.tw**

THE ORGANIZING COMMITTEE

General Chairs: R.C.T. Lee, Providence University and C. V. Ramamoorthy, University of California - Berkeley
Program Chairs: Philip S. Yu, IBM T. J. Watson Research Center and Arbee L.P. Chen, National Tsing Hua University
Steering Committee Chair: Benjamin W. Wah, University of Illinois at Urbana-Champaign
Tutorial Program Chair: Yao-Nan Lien, ITRI/CCL
Panel Program Chair: Maria Zemankova, MITRE/NSF
European Coordinators: Elisa Bertino, Univ. of Genoa
Far East Coordinators: M. Takizawa, and M. Papazoglou
Publication Chair: Kung-Lung Wu, IBM Watson research
Registration: Chuan-Yi Tang, N. Tsing Hua U**.**
Industrial Program and Chairs: Ahmed Elmagarmid and Omran Bukhres, Purdue University
Publicity Chairs: Abdelsalam Helal, U. of Texas at Arlington and Chiang Lee, National Cheng-Kung Univ.
Financial Chair: Steve Y.L. Lin, National Tsing Hua University, and Jeffrey Tsai, Univ. of Illinois at Chicago
Local Arrangements: Allen Wu, National Tsing Hua University and Chen-Pang Lin, III.

IEEE TRANSACTIONS ON KNOWLEDGE AND DATA ENGINEERING

CALL FOR PAPERS

Research Surveys and Correspondences on Recent Developments

We are interested to publish in the *IEEE Transactions on Knowledge and Data Engineering* research surveys and correspondences on recent developments. These two types of articles are found to have greater influence in the work of the majority of our readers.

Research surveys are articles that present new taxonomies, research issues, and current directions on a specific topic in the knowledge and data engineering areas. Each article should have an extensive bibliography that is useful for experts working in the area and should not be tutorial in nature. Correspondences on recent developments are articles that describe recent results, prototypes, and new developments.

Submissions will be reviewed using the same standard as other regular submissions. Since these articles have greater appeal to our readers, *we will publish these articles in the next available issue once they are accepted.*

Address to send articles:	Benjamin W. Wah, Editor-in-Chief Coordinated Science Laboratory University of Illinois, Urbana-Champaign 1308 West Main Street Urbana, IL 61801, USA Phone: (217) 333-3516 (office), 244-7175 (sec./fax) E-mail: b-wah@uiuc.edu
Submission Deadline:	None
Reviewing Delay:	One month for correspondences, three months for surveys
Publication Delay:	None; articles are published as soon as they are accepted
Submission Guidelines:	See the inside back cover of any issue of *TKDE* or by anonymous ftp from manip.crhc.uiuc.edu (128.174.197.211) in file /pub/tkde/submission.guide.ascii
Length Requirements:	40 double-spaced pages for surveys, 6 double-spaced pages for correspondences
Areas of Interest:	See the editorial in the February'94 issue of *TKDE* or by anonymous ftp from manip.crhc.uiuc.edu in file /pub/tkde/areas.of.interest

TENTATIVE PROGRAM

RIDE-DOM'95

Fifth International Workshop on Research Issues on Data Engineering:
DISTRIBUTED OBJECT MANAGEMENT
Taipei, Taiwan, March 6-7, 1995

Sponsored by the IEEE Computer Society

RIDE-DOM'95 is the fifth of a series of annual workshops on Research Issues in Data Engineering (RIDE). RIDE workshops are held in conjunction with the IEEE CS International Conferences on Data Engineering. As in the past, this year's RIDE Workshop concentrates on a particular topic: *distributed object management*. The objective of the workshop is to provide a forum for the discussion and dissemination of original and fundamental advances in all aspects of distributed object management. The following technical program will be complemented by discussants at the end of each paper session to enable further in-depth discussions.

The Workshop will be held at the **Science Technology Center** (No. 106, Sec 2, Ho-Ping East Road) and the Workshop Hotel is **Howard Plaza Hotel** (160, Jen Ai Road, Section 3, Tel: +886-2-700-2333 ext. 2417; Fax: +886-2-708-2376). The program includes an invited talk, 15 regular and 4 short paper presentations organized in single stream format. In addition, there are two panel sessions being organized: *Distributed Object Management Platforms* which will bring together representatives from OMG, IBM, Microsoft, Sun, HP and other vendors, and *Research Issues in Distributed Object Management* which will address the technical issues that require solution for this technology to be used widely.

Detailed information on the Workshop can be found on the World Wide Web at the address: **http://web.cs.ualberta.ca/~database/ride95**. To register for the Workshop, please return the form below to the indicated address with your registration check.

WORKSHOP REGISTRATION

	IEEE Member	Non-Member
Early registration (by Jan. 31,95)	US$ 180	US$ 230
Late/On-site registration	US$ 200	US$ 250
Student	US$ 80	US$ 100

Registration fee includes refreshment breaks and proceedings.

Please complete this form (TYPE or PRINT), and return with your payment (**please make checks payable to RIDE-DOM'95**) to:

Prof. Randal J. Peters, Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada R3T 2N2, Tel: +1-204-474-8685, Email: randal@cs.umanitoba.ca

Dr./Mr./Mrs./Ms./Prof.____ (For Name Badge)

Affiliation:____ (For Name Badge)

Address:____

____

Country/Postal code: ________________________________ Phone:_______________________________

Fax:__________________________ Email:____

IEEE Computer Society
1730 Massachusetts Ave, NW
Washington, D.C. 20036-1903

Non-profit Org.
U.S. Postage
PAID
Silver Spring, MD
Permit 1398