HEWLETT PACKARD

Object SQL - A Language for the Design and Implementation of Object Databases

Jurgen Annevelink, Rafiul Ahad*, Amelia Carlson*,
Dan Fishman, Mike Heytens, William Kent
Software Technology Laboratory
HPL-94-02
March, 1994

object databases, query language, information services, distributed environment, relational databases

Object SQL (OSQL) is a language for the design and implementation of object databases. The OSQL language is computationally complete and provides a rich set of constructs that allow definition, implementation and integration of information services in a distributed environment. It also provides a declarative query capability, similar to that provided by SQL for relational databases. This chapter includes examples of OSQL types and functions used in actual distributed applications, based on Hewlett-Packard's OpenODB implementation of OSQL.

Internal Accession Date Only
To be published in the *ACM Press Books (Association for Computing Machinery)*, "Database Challenges for the 90's" edited by Won Kim,
*Cooperative Computing Systems Division, Cupertino, CA

1 Introduction

Object SQL (OSQL) is a database (programming) language that combines an expression-oriented procedural language with a high-level, declarative and optimizable query language. The OSQL language combines the object-oriented features found in such languages as C++ [Ellis and Stroustrup 1990] and Smalltalk [Goldberg and Robson 1983] with a query capability that is a superset of the familiar SQL relational query language. Consequently, OSQL provides many of the advantages of object orientation, including a more intuitive model, improved productivity, code reuse and extensibility, together with all the features of current database technology, such as query optimization, integrity constraints, multi-user access, authorization and security. The OSQL language was developed as part of the Iris project at Hewlett-Packard Laboratories [Fishman et al. 1989; Lyngbaek 1991; Wilkinson, Lyngbaek and Hasan 1990]. It has evolved to include general computational primitives [Annevelink 1991] and is now a computationally complete, extensible database language. The design of OSQL was influenced by pioneering work on semantic and functional database models, notably the functional language Daplex [Shipman 1981] and the language Taxis [Mylopoulos, Bernstein and Wong 1980].

The design goals for OSQL can be summarized as follows:

- based on a simple, orthogonal object-oriented model and type system
- computationally complete and independent of specific application programming languages
- provides constructs for specifying declarative queries and allows such queries to be compiled and optimized, similar to the capabilities offered by relational query languages
- extensible, that is allows the user to (dynamically) define new types and operations
- no artificial distinctions between meta-data objects and user-defined objects
- allows separate definition of the interface of an object (type) and the corresponding implementation(s).

OSQL is object-oriented in that it provides object identity, a type system with multiple inheritance, polymorphic functions and built-in aggregate object types such as sets and lists. It differs from other object-oriented languages, in particular C++, in that it does not mix the definition of the interface of an object type with a particular representation of the instances of the type. OSQL allows the interface of an object type to be defined independent of a specific choice for implementing the interface and allows the implementation to change over time[1]. In OSQL, the state of an object is not an intrinsic part of the object itself; rather, it is defined by functions which model attributes of the object, interobject relationships, and arbitrary computations. By disassociating the interface of an object type from any specific representation of the instances of the type, one can allow objects to dynamically acquire and lose types, thus enabling one to model the evolution of objects over their lifetime in a natural way.The OSQL type system allows the OSQL compiler to do compile-time type checking. However, since OSQL allows objects, including types and functions, to be created dynamically, compile-time type checking has to be supplemented by run-time type checks.

Functions are a major modelling construct of OSQL and are used to model attributes of the object, interobject relationships, and arbitrary computations. Functions can be implemented as stored functions (e.g. by storing a direct representation of the relationship in the form of a table in the database) or they

1. Future versions of the language may include additional constructs to allow the specification of multiple implementations of a given interface.

can be computed. The implementation of a computed function is specified by an expression, the *body* of the function; the free identifiers in this expression are the formal parameters of the function and the value computed by the expression is the value returned by the function. The body of a computed function may include query expressions. In addition to stored and computed functions, OSQL also supports external functions. External functions provide a crucial measure of extensibility, because they allow a function to be implemented by a routine written in an external programming language (e.g. C, C++ or COBOL).

The OSQL authorization mechanism is also designed around functions [Ahad et al. 1992]. Users are members of UserGroup's that are assigned call and/or update privileges to functions. The OSQL authorization mechanism is not further discussed in this chapter.

The OSQL language is independent of specific application programming languages (e.g. C, C++, Smalltalk, COBOL) and specific implementations. The examples used in this chapter are slightly stylized versions of actual OSQL functions used by applications running on top of OpenODB, Hewlett-Packard's object-oriented database management system and implementation of OSQL [Ahad and Dedo 1992; Ahad and Cheng 1993; Hewlett-Packard 1992]. The programmatic interface provided by OpenODB allows client applications to call any OSQL function and map the results returned by OSQL functions to the data-types provided by the programming interface.

In this chapter we will give an outline of the OSQL model (section 2), followed by a discussion of the major constructs found in the OSQL language (section 3), with special attention to the *select* construct (section 4). In section 5 we will give a number of examples highlighting some of the more advanced applications made possible by OSQL.

2 *OSQL Object Model*

The OSQL language is centered around three basic concepts: objects, types and functions. Objects, types and functions are related as shown in Figure 1.

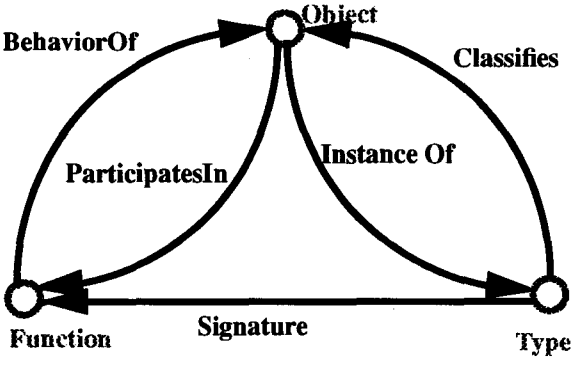

FIGURE 1. OSQL - basic data model elements and relationships

Objects

Objects represent the real-world entities and concepts from the application domain that the database is storing information about. For example, in a clinical database, objects may represent clinics, physicians, nurses, patients, problem lists, and so on. In OSQL, objects can be classified in one of three categories:

- literals, for example, integers, character strings and binary objects
- aggregates, for example, a problem list or a tuple containing demographic data for a patient, such as name, age and social security number
- surrogates, for example, patients and clinics

Surrogate objects are characterized by a system-generated, unique object identifier (oid). Surrogate objects also represent entities used to implement OSQL, e.g. system types and functions. Surrogate objects are explicitly created and deleted.

Types

The second major concept in OSQL is that of a type. Types are used to classify objects on the basis of shared properties and/or behavior. For example, it is natural to group together all patient objects and similarly group all physician objects, all nurse objects and all clinic objects. Types are also used to define the signature of functions (i.e. their argument and result type). The extension of a type is the set of objects that are instances of the type. Some types, for example Integer, have pre-defined extensions. Surrogate types have dynamic extensions that change depending on the type(s) and order in which objects are created and deleted. Aggregate types and aggregate objects can be constructed from other types and objects respectively, using system defined aggregate type and object constructors[2]. For example, an instance of the type `SetType(SmallInteger)` is denoted by the expression: `Set(1,2,3)`. Note that this does not construct a new object, but rather returns a specific object from the extent of the type `SetType(SmallInteger)`, in the same way that the expression `1` does not return a new object but returns an instance of the type `SmallInteger`.

Types are related in a subtype/supertype hierarchy that supports multiple inheritance. The type hierarchy enforces type containment, that is, if an object is an instance of a given type T, it must also be an instance of all supertypes of the type T. An overview of the (pre-defined) system type hierarchy is shown in Figure 2. User-defined types can be added as subtypes of the type UserSurrogate[3]. OSQL surrogate objects can be instances of any number of types, even if the types are not related by a subtype/supertype relationship. This is obviously required in the real world, where say a person may belong to many different groups, and assume different roles depending on the context. For example, the group of cancer patients can be distinguished from the group of diabetics; different types of properties are applicable and relevant to each of the members of the two. Moreover, people can change their membership in a group and thus change what roles and what properties and behavior are applicable. For example, a person can be cured and thus no longer be a cancer patient or he can get sick and be diagnosed with diabetes.

Functions

The third concept, functions, is used to model attributes of the object, interobject relationships, and arbitrary computations. One of the key distinctions of OSQL as compared to other models (e.g. those inspired by object-oriented programming languages) is this unifying notion of a function to model

---

2. OpenODB supports four aggregate type and object constructors, BagType, SetType, ListType and TupleType to construct aggregate types and Bag, Set, List and Tuple to construct the corresponding objects.

3. The version of OSQL implemented by OpenODB currently does not allow the creation of user-defined subtypes of pre-defined system types.

stored and derived attributes, stored and derived relationships and arbitrary computations (behavior). In OSQL the distinction between these is relegated to the implementation domain, thus making the actual modeling more independent of implementation trade-offs and allowing a greater freedom in choosing an implementation, including the possibility to evolve an implementation (e.g. choosing to re-implement something that was a stored attribute as a derived or computed attribute).

An OSQL function takes an object as an argument and may return an object as a result[4]. OSQL functions can be overloaded, that is, there can be multiple functions with the same name but different argument types. OSQL does not currently allow overloaded functions to have different result types. OSQL refers to an overloaded function as a generic function. The resolvents of a generic function are called specific functions. For a given function f, the argument object must be an instance of the argument type specified for one of the specific functions that resolve the function f. A function can only return an object that is, an instance of the result type of the function. The result type of functions that never return a value is `Void`. Functions may change the state of the database as a side effect of their application, by updating other functions. Functions that perform updates are said to have side effects and can not be called as part of a query. Similar to types, functions have extensions. The extension of a function is the mapping from its arguments to its results. Function extensions can be explicitly stored, or they can be computed. Functions whose extent is computed can be implemented either as an OSQL expression, or as a program (subroutine, procedure) written in a general-purpose programming language. These latter are called external functions and give OSQL a unique form of extensibility by allowing the encapsulation of (entry points in) external libraries.

An important property of functions is their updatability. A function f that is, updatable has a companion function, say `set_f`, that will set the value to be returned by f for a given argument, when called. If a function f returns an aggregate type, then it will have three such companion functions: one to set the value as before, the other two to add or remove a value in the aggregate. The set, add and remove functions can be specified explicitly, or they can be generated automatically by the system (e.g. when the extent of the function f is explicitly stored). For example, a stored function `translate` can be defined as follows:

```
create function translate(Char english) -> Char /*french */;
```

Since this is an atomic valued stored function, the system will automatically create a second function to allow it to be updated. This function has no name and can be found by evaluating the (system-defined) function `FunAssign`. The latter function will be invoked when the function `translate` is to be updated; for example,

```
translate('one') := 'un';
```

4. OSQL functions can take aggregate objects as argument and/or return them as results. Functions with multiple arguments are implemented by combining the arguments into a single tuple value and applying the function to the tuple. The argument type of function with multiple arguments is the tupletype corresponding to the types of the arguments.

OSQL System Types

To make an OSQL system work, a great many built-in objects, types and functions need to be defined. The basic type hierarchy is shown in Figure 2. The root object type is called Object and is the supertype of all other object types. Subtypes of object are:

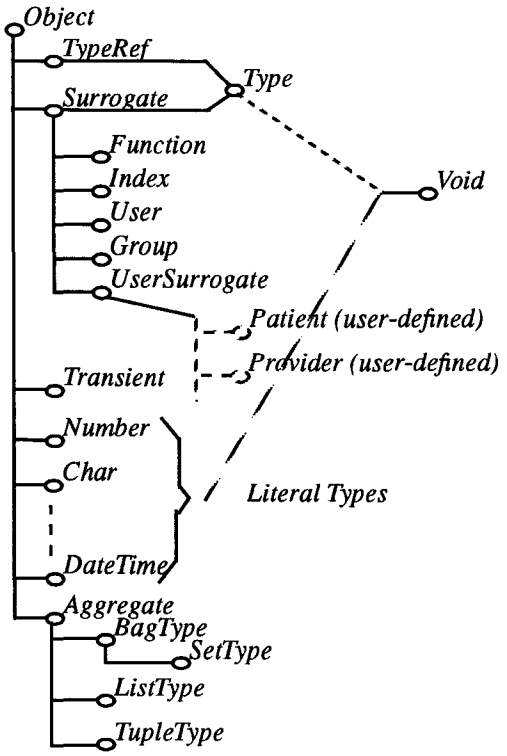

FIGURE 2. OSQL System Type Hierarchy (simplified)

- TypeRef - the type whose extension is the set of all type objects, including aggregate types; for example, SetType(Integer) and surrogate types (all the instances of type Type)
- Surrogate - the supertype of all types whose instances have oid's, including Function, Type, Index, User, UserGroup and UserSurrogate
- UserSurrogate - supertype of all user-defined types; for example, Patient, Provider, and so on.
- Type - the type whose extension is the set of all surrogate types(i.e. all types other then aggregate types).
- Transient - the supertype of all transient object types (i.e. the types whose instances are transient, for example Transaction, Savepoint, Session and Cursor).
- Aggregate - the supertype of all aggregate types. Aggregate types supported include BagType, SetType, ListType and TupleType. Instances of Bag, Set and List types can have any number of components that must all be instances of the component type of the Bag, Set or List. A tuple object on the other hand has a fixed number of components, each with its own type.
- Literal Types - The literal types supported by OSQL include Number (Integer, SmallInteger, Double, Real), Char, Binary, Date, Time, DateTime and Interval.
- Void - Void is a subtype of all types except aggregate types; its extension is the empty set.

OSQL System Functions

There are also a large number of system-defined functions that provide the functionality necessary to implement a full-fledged object-oriented data manager. Some of the more important functions provided are the following:

- CreateType - create a new (user-defined) type and (optionally) create one or more functions that have the new type as their argument type
- CreateFunction - create a new (user-defined) function
- ImplStored - implement a function, or set of functions, as a stored function (i.e. a function whose extent is explicitly stored in the database).
- ImplOSQL - implement a function as an OSQL expression, either as a procedural language expression (section 3) or as a query expression (section 4).
- ImplExternal - implement a function as an arbitrary program, written in an external programming language
- CreateObj - create a new object, an instance of a user-defined type, optionally initializing one or more functions for the new object

The OSQL language implemented by OpenODB provides convenient syntactic sugaring to invoke the above and other functions. For example, to create a new type and a set of associated functions, one can submit an OSQL statement, as follows:

```
create type Patient subtype of Person functions (
      patId Char (var 32)
);
```

A statement such as the one above will be parsed into a call to the function CreateType as follows:

```
CreateType('Patient',Set(type Person),
             List(Tuple('patId',type Char (var 32))))
```

It is also important to note that users can define their own functions that involve system objects such as types and functions. For example, to allow type objects to be annotated, one can define and use a stored function, help, as follows:

```
create function help(Type t) -> Char as stored;
help(Type Patient):= 'Patient object type help descriptor';
```

3 *Expression Language (DML)*

In the previous section we described the computational model of OSQL as being one of expression evaluation. In this section we will discuss the various types of expressions allowed and the means provided to compose expressions.

In addition to the function application expression as described above, there are a number of special forms, including if-then-else, quote, and a number of iterative constructs. An abstract syntax[5] that defines the types of expressions is shown below:

```
Expr        :=  Constant                                        (1)
            |   Identifier                                      (2)
            |   FuncAppl                                        (3)
            |   Assign                                          (4)
            |   Conditional                                     (5)
            |   BlockExpr                                       (6)
            |   ForLoop                                         (7)
            |   WhileLoop                                       (8)
            |   Quote                                           (9)
            |   Select                                         (10)
            ;                                                  (11)
FuncAppl    :=  func_ref: Expr arg: Expr;                      (12)
Conditional:=pred: Expr then: Expr else: Expr;                 (13)
Assign      :=  id: Identifierval: Expr                        (14)
BlockExpr   :=  declare: Declaration+ exprs: Expr+;            (15)
ForLoop     :=  id: Identifier domain: Expr loop: Expr;        (16)
WhileLoop   :=  pred: Expr loop: Expr;                         (17)
Quote       :=  Expr;                                          (18)
Declaration:= t: TypeRefid: Identifier;                        (19)
```

FIGURE 3. OSQL abstract syntax

The first rule defines an expression to be either a constant, an identifier, a function application, an assignment, a conditional, a block expression, a for loop, a while loop, a quoted expression or a select expression (discussed separately in section 4).

Constants and identifiers are defined as usual, except that OSQL also supports aggregate constants, that can be specified using the tuple, set, bag (multi-set) and list constructors. For example, a set constant can be specified as `Set(1, 2, 3 + 4)`. The elements of the set can be specified as expressions themselves. An aggregate object will in general have many types that can be inferred from the types of their elements. The OSQL language as implemented by OpenODB also supports a special notation to denote type and function constants. For example, the generic function `name` is denoted by the constant expression: `function name`. Similarly, the specific `name` function whose argument type is `Person`, is denoted by: `function name.Person`, and the type T is denoted by the expression: `type T`.

The most frequently used kind of expression is the function application. Abstractly, a function application expression consists of two expressions; a function reference (labelled func_ref in Figure 3 line 2), and an argument (labelled arg). The func_ref expression evaluates to a (generic or specific) function identifier, which may be the same as the function that the expression is a part of, thus allowing recursive function invocations. The expression labelled arg evaluates to an arbitrary object or aggregate object. The semantics of evaluating function applications was discussed in detail in section 2. For example, to set the name of a person, we evaluate the following expression:

5. The notation used to define the abstract syntax uses three constructs, respectively choice, denoted by |, aggregation, denoted by a tuplelike notation that labels the components of the syntactic construct, and repetition, denoted by $^+$, to mean one or more, or *, to mean zero or more. Note that the labels used to identify the components of an aggregate construct resemble, but are not the same as the keywords used in specifying a concrete syntax.

```
FunAssign(function name.person) (p1,'John')
```

In this example, the first expression is itself a function call, applying the function `FunAssign` to the function `name.person` (an example of a specific function reference). This returns the oid of the function that sets a person's name, which is subsequently applied to a tuple of two elements, the oid of the person and the new name (a string object), and sets the name of the person accordingly. The parentheses and ',' are used here to denote an operator that creates a tuple. The OSQL language as implemented by OpenODB provides a convenient syntactic shorthand for the above expression, as shown below:

```
name.person(p1) := 'John'
```

OSQL provides an imperative model of variables and assignment similar to C. Variables can be declared and have scope equal to the block expression in which they are declared. Within such scope, one can assign a value to the variable using an assignment, similar to what one would do in a language like C. The introduction of assignment in OSQL is not strictly necessary, but is needed to support such imperative constructs as while loops. Alternatively, one can use recursion and recursive functions to avoid the need for assignment. The OSQL language as implemented by OpenODB provides the following concrete syntax for variable assignment:

```
i := 2
```

The conditional or if-then-else expression consists of three sub-expressions: a predicate expression, a then expression, and an else expression. A specific example using the concrete syntax implemented by OpenODB is shown below:

```
ticketPrice := if age(p1) >= 60 then 27 else 40
```

The semantics of its evaluation is to first evaluate the predicate expression; for example, `age(p1) >= 60`. If it returns true, we evaluate the then expression; if it returns false, we evaluate the else expression. The if-then-else expression returns the value of the then or else expression.

The next expression is the block expression or begin-end expression. It consists of a list of (local) variable declarations and a list of expressions. Its semantics is to evaluate the expressions in the list in order, in the environment created by extending the environment of the block expression with bindings for the local variable identifiers. The value returned by the block expression is the value returned by the last evaluated expression in the list. For example, a simple block returning 5 is defined as follows:

```
begin declare Integer i, j; i := 1; j := 4; i + j; end
```

Next there are two types of iterative expression; a for loop expression and a while expression. The for loop provides iteration over the elements of an aggregate. It returns no value (i.e. its return type is Void), and is thus evaluated only for its side effects. For example, to set the primary provider for a set of patients to a given provider, `prov`, one could evaluate the following expression:

```
for p in Set(p1, p2, p3, p4) do primProvOf(p) := prov
```

The while expression allows one to iteratively evaluate an expression (the loop expression), until another expression (the pred expression) returns true. The while loop also returns no value, similar to the for loop.

The next expression type is the so-called quote special form. A quoted expression returns its arguments unevaluated. A variation of the quote is the so-called backquote expression. This also returns its

argument expression unevaluated, except for those parts that are wrapped in a call to the unquote function. The quote function is required to allow functions to evaluate their own arguments. This is often useful, for example, when defining functions that create and implement other functions.

The last construct shown in the abstract syntax is a type declaration. This is not an expression, but rather is required to be able to define variables. A type declaration defines a binding between an identifier and a variable (i.e. a storage location used for storing the value). The identifier serves as the name of the variable. Declarations can only be specified as part of a block expression, which is the scope of the declaration.

In addition to the special forms discussed above, there are a number of system functions that provide a degree of non-local control that is often convenient when defining a function. The first such function is the `return` function. The effect of calling the `return` function is that the function containing it immediately returns the value of the only argument of the `return` function. Another function is the `raise-error` function. This function raises an exception that will transfer control back to the client application, undoing any changes the function containing the call made.

An example highlighting a number of these constructs is shown in Figure 4. Given a tuple consisting of a function `f` and a set of objects `arg`, the `filter` function will return the set of objects `e` in `arg` for which `f(e)` returns `true`. Note that the parameter `f` is function- valued and that the function application `f(e)` in the filter function above will not be resolved until run-time.

```
create function filter(Function f, SetType(Object) arg) ->
              SetType(Object) as osql
begin
   declare SetType(Object) r;
   declare Object e;

   for e in arg do
         if(f(e)) then r := r + e; endif;
   return r;
end;
```

FIGURE 4. Example OSQL function - filter

4 *Query Language*

OSQL supports a query language whose semantics is based on domain calculus, with support for aggregate domains, functions and multisets (bags). The OSQL SELECT function provides the basic query facilities of OSQL and closely resembles the Select statement of SQL. The abstract syntax of a select expression shows that it consists of six parts:

```
Select:=  resStruct: ResStruct
             resList: Expr+
             forEach: Declaration*
             where: Expr
             groupBy: Group*
             having: Expr
             orderBy: orderSpec*;
```

All of the parts of the select expression shown above, except for the resList, are optional. For example, using the concrete syntax used by OpenODB, to retrieve the name of a person we evaluate the following expression[6]:

```
select name(p)
```

The query compiler will infer that the type of this expression is BagType(TupleType(Char)). We can use the resStruct options of the select expression to change the result type. For example, to have the select expression return a bag of strings, we add the keyword atomic. Similarly, to have the query return just a single name, we add the keyword single. A query that returns the name of a person (or null) can be defined as follows:

```
select single atomic name(p)
```

Select expressions are evaluated by parsing them into a call to the (system defined) select function, and passing the select function its arguments, that is, the parts of the select expression as indicated by the abstract syntax above, without evaluating them. The select function compiles the query by creating an unnamed function with no arguments, whose body is the compiled and optimized query expression. The query is evaluated by calling this function without any arguments. In the case of a stand-alone query expression, the unnamed function is transient and deleted after the function is evaluated. If the query expression is part of the body of a function, the compiled and optimized expression becomes part of the body of that function.

More complicated query expressions than the ones shown above can be easily expressed as well. For example, to return the set of names of all persons living in San Jose, CA, we evaluate the following expression:

```
select distinct atomic name(p) for each Person p where
     City(p) = 'San Jose' and State(p) = 'CA'
```

The type of this expression is SetType(Char). The reason this query returns a set instead of a bag is that we included the keyword distinct as part of the resStruct clause. Similarly, a query that includes an orderBy clause will return a list object instead of a bag. For example, to return a list of the names and

6. Note: we assume that the identifier p is bound to the oid of a person object

oids of the next n persons whose names are lexically greater than a given prefix string fr, we can define a function PersonsByName as follows:

```
create function PersonsByName(Char fr, Integer n) ->
        ListType(TupleType(Char, Person)) as osql
   begin
        declare Cursor c;
        declare ListType(TupleType(Char, Person)) r;
        if( n > 0) then
          open cursor c for
             select name(p), p for each Person p where
                name(p) >= fr order by name(p);
        else begin
          open cursor c for
             select name(p), p for each Person p where
                name(p) <= fr order by name(p) desc;
          n := n * -1;
        end; endif;
        r := fetch(c,n); /* fetch first n elements of cursor c */
        close(c);
        return r;
   end
```

The function above can be called directly by an application or via another function. When called by an application, it can for example be used to fill in the elements of a menu used to select a person from among all the persons in the database. The second argument of the function PersonsByName is used to specify the number of persons to return, as well as to control the direction of the scan, e.g. allowing a client application to use this function to scroll up and/or down a list of persons. Note also that the efficient evaluation of the function PersonsByName is dependent on both the presence of an index on the result of the name function and the query optimizer choosing this index to access the corresponding storage structure. OSQL provides the necessary system functions to define such an index and the query optimizer will select it when compiling a query such as the one above.

The forEach clause[7] defines the search domain of the query by declaring a list of identifiers and their types. The domain of the query is formed by taking the cartesian product of the domain of each of the identifiers in the forEach clause. Doing so will generate a set of bindings for the identifiers; one binding per element of the domain. From the examples shown so far, it can be inferred that the where clause of the select is a functional expression that returns a boolean value. In fact, in OSQL, the where clause can return any natural number. In case the expression actually returns boolean TRUE, the value returned is 1; similarly, in case the expression actually returns boolean FALSE, the value returned is 0. The counter value returned by the where clause determines the number of times the result clause of the query should be evaluated for a given binding in the search domain of the query. A full treatment of the semantics of OSQL queries is beyond the scope of this chapter, but can be found in [Kent 1993].

The groupBy and having clauses of a query are similar to those found in SQL and will not be further explained here.

---

7. Note: The concrete syntax for the forEach clause as implemented by OpenODB allows the keyword `from` to be used instead of `for each` for reasons of compatibility with SQL

OSQL does not restrict the kinds of functions allowed in the expressions associated with the resList and where clause of a query expression, other than that functions can not have side effects. A function has a side effect[8] when it updates another function, either directly or indirectly. To be able to include external functions in query expressions, the query compiler has to ensure that their arguments are bound before they are called.

OSQL query expressions support late binding semantics of functions, that is, function resolution is postponed until the query is actually evaluated, unless the query compiler can determine a unique resolvent at query compile time. For example, suppose that we have a function named area defined on Circle, Rectangle and Polygon, all subtypes of type Shape, the following query will return the area of all shape objects found in the database, using the appropriate specific function for each:

```
select area(s) for each Shape s
```

5 *Annotated Examples*

In this section we give examples of OSQL functions that are intended to highlight the various capabilities of OSQL. These examples are all derived from actual OpenODB applications and reflect the capabilities of OSQL as it stands today. We hope to evolve these capabilities by increasing our understanding of applications and extending the language to provide more and higher level constructs, aimed at simplifying the development of applications such as these, to increase programmer productivity and to simplify maintenance of the resulting systems.

Queries

The first set of examples shows OSQL's query capabilities and illustrates the modeling of objects with nested structure, the use of function overloading and the use of recursion in queries.

```
create type part functions (
      name Char (var 128)
);
create type complexPart subtype of part;
create function subparts(complexPart) -> bagtype(part);

create function price(part) -> Real;
create function price(complexPart p) -> Real as osql
 sum(select atomic price(q) for each part q
              where q occurs in subparts(p));
```

The first two statements above define type part and a subtype complexPart. The definition of type part is combined with the definition of a function, name, that returns the name of a part. A complex part is distinguished from a part in that it has subparts. The function subparts, defined in the third statement above, returns a bag of parts, thus allowing a complex part to include a given part multiple times. Function price illustrates how easily a transitive closure operation can be expressed as an OSQL query, mostly because the price function can be overloaded on the types part and complexPart and will be dynamically resolved as part of the query to compute the price of a complex part p. Note that the occurs in clause in the where clause of the query serves as a multiplier, that is, the price of a given part will be

8. The OpenODB OSQL function compiler determines whether a function has a side effect automatically and marks the function as such in the system dictionary.

duplicated in the result of the query as many times as the part is included as a subpart of a complex part, so that the summation of the elements of the bag returned by the query returns the intended result. The function to compute the price of a complex part could also have been implemented procedurally, as follows:

```
create function price(complexPart p) -> Real as osql
   begin
         declare Real pPrice;
         declare part sp;
         pPrice := 0;
         for sp in subparts(p) do pPrice := pPrice + price(sp);
         return pPrice;
   end;
```

The implementation as a procedure is semantically equivalent to the implementation as a (declarative) query, but the query form is more amenable to optimization.

Another example, showing transitive closure for a parts explosion, is the following,

```
create type material functions (name char(64));
create type simplePart subtype of part functions (
   mat material
);
create function partExplode(part p) -> bagtype(simplePart)
   as osql bag(p);
create function partExplode(complexPart p) ->bagtype(simplePart)
   as osql select atomic sp
   for each simplePart sp, part q where q occurs in subparts(p)
         and sp occurs in partExplode(q);
```

The function `partExplode` defined above can be used in queries, as follows,

```
select name(sp) for each part p, simplePart sp
where sp occurs in partExplode(p) and name(p) = 'mercedes' and
            name(mat(sp)) = 'asbestos';
```

This query will find the names of all components of parts whose name is 'mercedes' that are made of 'asbestos'.

Distributed Application Integration example

In a distributed environment, the database system, or more precisely, the database server processes, must be able both to receive messages from other applications, as well as to send messages to other applications, including other database servers. In general, it is easy to provide a customized interface (client front-end) to a database that enables it to receive messages from other applications, and to respond to these messages. The other way around is more difficult and will require that the database system provide the ability for users (developers) to invoke their own code, both inside the database server process or a specialized external server process, and as part of the client application. Such a capability can then be used to notify other applications of events in the database, such as a change in state in the database as a result of an update, thus implementing a basic 'active database' capability, or to request from other applications (services) information that is needed inside the database.

For example, a slightly stylized function that can be used to send 'messages' to objects residing in other applications can be defined as follows:

```
create function genEvent(Object sender, Object rec, Object msg,
                         Object msgArgs) -> Object as
        simpleextfun 'gen_event';
```

The function `genEvent` sends an event or message `msg` with arguments `msgArgs` to an Object `rec` from an Object `sender`. Note that the sender and receiver objects may be instances of subtypes that have properties that will enable the actual message sending code to retrieve their external identifiers, for example, some kind of globally unique object identifier that enables identification of sender and receiver objects. The `genEvent` function is implemented as a 'simple external' function, meaning that when it is called the C routine registered as gen_event is called. OpenODB allows such C routines to be called in the context of the client application, thus allowing the database server to have access to the state of the client application and providing a degree of isolation between application code and database code[9]. The gen_event routine will format the actual message and forward it to the receiver object. The reason we call this capability active database is because it allows the database to affect the state of an object outside the database itself. The actual capabilities are to a large extent determined by the external environment. For example, in a CORBA [Object Management Group 1991] environment, the routine `genEvent` could format and send a message to a remote object, according to the actual arguments supplied as part of the `genEvent` function invocation.

Functions like genEvent can be used as a building block in other functions, to be called whenever an action is required outside of the database. An example of an actual invocation is as follows:

```
genEvent(LabTechNamed('John'),ProviderNamed('Dr.Heartdoc'),
         'send_email',
         'call 415-123-4567, regarding your patient:
             John - critical high K+ 14.5');
```

This call shows how, in a hospital environment, a lab tech object can notify a provider object about a critically high lab value. In this case, notification involves sending a relevant message, via e-mail, to the receiver.

Another example, along the same lines, shows how we can create functions that spawn and control other processes.

```
create function sendMail(Char to, Char subject, Char msg) -> Void
as osql
begin
 declare TupleType(Integer,Integer) pty;

 pty := PtySpawn('/usr/bin/mailx',List('-s',subject,to));
 PtySend(pty,StrAppend(msg,'\n.\n'));
end;
```

This function shows the use of some system-defined process control functions (`PtySpawn`, `PtySend`) that allow programs to be executed in a terminal emulator (pseudo terminal device) and control the program by sending it data or reading back data generated by the program. In this case we create an external process, using the function `PtySpawn`, and then send that process some data (i.e. the

9. Future releases will also allow such routines to be made part of so-called 'external server processes' that are independent of any specific applications.

contents of the mail message) using the function PtySend. The mailx process sends the mail message when it receives a terminating '.' on a line by itself and then terminates, as if it was invoked interactively from a terminal and the mail message had been typed in directly.

```
create function dhcp_eval(Char cmd) -> ListType(Char) as osql
begin
   declare TupleType(Integer,Integer) pty;
   declare TupleType(Integer,Char) ans;

   /* lookup or establish connection with msm interpreter */
   pty := LookupMsmPty();
   if(NotExists(pty)) then begin
         pty := PtySpawn('/mumps/msm',List());
         LookupMsmPty() := pty;
   end;
   else if (not(PtyAlive(pty))) then begin
         pty := PtySpawn('/mumps/msm',List());
         LookupMsmPty() := pty;
   end;  endif; endif;

   /* send command to interpreter and receive echo back */
   PtySend(pty,cmd);
   PtyReceive(pty,List(Tuple(0,MkRegExprToMatch(cmd))));

   /* receive, parse (split in lines) and return answer */
   ans := PtyReceive(pty,List(Tuple(0,'\r> ')));
   return SplitLines(ans[1]);
end;
```

The function dhcp_eval above is similar to the sendMail function but shows an example of the type of processing required to establish a connection with another process and then repeatedly send commands and receive replies. Note the use of LookupMsmPty(), a stored function used to store the values of the external process id and file descriptor. The function PtyAlive is called to check that the process whose process identifier is in the variable pty is still present and responding; if not, a new process will be spawned. The function PtyReceive provides capability to do pattern matching on the output stream sent back by the external process, allowing the function dhcp_eval to recognize the end of a reply. The function MkRegExprToMatch takes a string as an argument and returns a string which is a regular expression that matches the argument. The newline characters in the result returned by the second call to PtyReceive are used to split the answer into a list of strings, which is returned as the result of the function dhcp_eval.

The program started by the function dhcp_eval is a stand-alone interactive interpreter for the MUMPS language. This language is used extensively in hospitals to implement information systems that manage all kinds of data associated with patients, for example, laboratory test results, prescription records, demographic information and so on [Dept. of Veterans Affairs 1990]. The details are outside the scope of this chapter but the important thing to note is that the msm program provides us access to all of this information by evaluating specific commands. For example, using MUMPS interface routines that we developed [Annevelink, Young and Tang 1991], we can request a list of the next n names and identifiers for patients, starting from a given prefix. Given the identifier of a patient, we can then request values of attributes for these patients and so on, thus enabling us to access the external database.

Constraints, rules and triggers

The next example utilizes genEvent and shows how OSQL procedures can be used to encode rules, constraints and triggers. In this case, we want to create a function to update a patient's lab values that generates a message when the lab value is abnormal.

```
create function newLabResult(LabTech lt, Patient p, TestOrder t,
            Char value) -> Boolean as osql
begin
   declare isAbnormal;
   LabtestResults(LabTestOf(t),p,CurrentDate()) := Tuple(value,lt);
   isAbnormal := CheckCriticalRange(LabTestOf(t),value);
   if(isAbnormal) then
    genEvent(lt,PrimaryPhys(p),'send-email',StringAppend('call ',
          ToChar(work_phone(lt)),
          'regarding your patient: '
          name(p),
          ' critical value of lab test: ',
          name(LabTestOf(t)),
          'value: ',
          value);
   endif;
   return isAbnormal;
end;
```

The function shown above will update the LabTestResults function to reflect that a labtest result has come in. It will then compare the value of the labtest with the critical range defined for it and determine whether it is abnormal. If so, the function will format a message using information stored in the database and send that to the primary physician of the patient.

This example is not intended to show the utility of OSQL as a language for defining rules or constraints. Rather, it intends to show how one can enforce arbitrarily complex rules or constraints by 'programming' them by hand. Future versions of OSQL may include more specialized rule and constraint subsystems that can be invoked directly, or used to compile functions similar to this one automatically, from a more declarative specification.

Integration of legacy applications (external data sources)

In many situations, there is a need to integrate existing, so-called *legacy* systems and applications to simplify access to these systems and to provide a capability to formulate queries that range over data included in several such systems or applications.

The types and functions below are prototypical for the situation in which one wants to integrate an external system with an OSQL database. The type ExtObject provides the capability to define an external key for an object. The external key must provide the ability to uniquely identify the object in the external system, in the same way that its object id allows that inside the OSQL database.

The function CrExtObject provides the capability to create instances of local objects that represent external objects. It provides an example of the capability to call system functions such as CreateObj with arbitrary parameters. CreateObj is a system function that provides the capability to create objects and initialize one or more functions that have the newly created objects as their argument. CrExtObj also shows the capability of OSQL functions to contain arbitrary OSQL queries. The object created by the function CrExtObject is an instance of the type t, which must be a subtype of the type ExtObject.

```
create type ExtObject functions (xkey Char);

create function CrExtObj(Type t, Char key) -> ExtObject as osql
begin
   declare ExtObject eObj;
   declare ListType(ExtObject) eObjs;
   eObj := select single atomic o for each ExtObject o
             where xkey(o) = key;
   if(isNull(eObj)) then
        eObj := CreateObj(t,List(FUNCTION xkey),
List(Tuple(key));
    endif;
   return eObj;
```

The latter constraint is enforced by a run-time typecheck on the value returned by the function CreateObj, added by the OSQL compiler.

Using the functions described so far, it is now relatively straightforward to define the function ReadExtSurrAttr which returns the value of an attribute of an object stored in an external MUMPS database. The external database is accessed through the dhcp_eval function defined before. The argument of the function dhcp_eval is an example of a MUMPS expression used to retrieve the value of the attribute.

```
create function ReadExtSurrAttr(Char id, Type objType,
    Char attrName) -> Object as osql
begin
   declare ListType(Char) ans;
   ans := dhcp_eval(strAppend(
             'S DR=',attrName,',OID=',id,' D EN^HPRD\r'));
   return CrExtObj(objType,ans[0]);
end;
```

The function above returns the value of a surrogate-valued attribute of an external object. For example, the following call will return the oid of the spouse of the patient whose id is '^DPT(^2301^2'[10]:

```
ReadExtSurrAttr('^DPT(^2301^2',type patient,'SPOUSE')
```

The arguments are the external key (id) of the external object, the type of the surrogate object, and the name of the external attribute. The function ReadExtSurrAttr invokes the function dhcp_eval to retrieve the value of the external key of the surrogate attribute from the MUMPS database and calls the function CrExtObj to convert the external key into a corresponding oid, which is then returned.

The next two functions show how OSQL system functions can be used to dynamically create types and functions. In this example, the function ImpExtType creates a type whose extension maps to a set of objects residing in an external system. It also creates, by calling the function CrAttrF, a set of attribute functions that can be used to access the attributes of the objects that reside in the external database. The functions ImpExtType and CrAttrF access the data dictionary of the external system to determine different properties of the attributes, for example what type of value they return and what indexes, if any, are defined to access the data in the external system. The functions ImpExtType and CrAttrF in

10. This string is an example of an external id. It contains the information necessary to uniquely identify an object in an external database, in this case a record that contains a.o. a reference to the spouse.

```
create function ImpExtType(Char tName, Char extTypeName,
   ListType(TupleType(Char /* fname */, Char /* attrName */,
                Integer /* attrMod */)) attrs)
   -> TupleType(Type, ListType(Function)) as osql
begin
 declare Function f;
 declare ListType(Function) attrFncs;
 declare Type extType;
 declare TupleType(Char, Char, Integer) attr;

 extType := MkExtType(tName, extTypeName);
 attrFncs := List();
 for attr in attrs do begin
   f := CrAttrF(attr[0],extType,attr[1],attr[2]);
   attrFuncs := attrFuncs + f;
 end;
 return Tuple(extType,attrFncs);
end;
```

effect map the schema specified by the external data dictionary into an OSQL schema. Similar functions can be defined to import other data sources and/or applications.

```
create function CrAttrF(Char fname, Type argtype, Char attrName,
   Integer attrMod) -> Function as osql
begin
   declare ExtAttr attr;
   declare Type resType;
   declare Function f;

   attr := FindExtAttrByName(argtype,attrName);
   if(isNull(attr)) then
         raise error StrAppend('Attribute does not exist: ',
                 attrName);
   endif;
   resType := TypeOfExtAttr(attr);
   f := CreateFun(fname,argtype,resType,List());
   if(SurCategory(resType) = 0) then
         ImplOsql(f,List('ARG'), BACKQUOTE(ReadExtCharAttr(
                  xkey(ARG), UNQUOTE(attrName))));
   else
         ImplOsql(f,List('ARG'),BACKQUOTE(ReadExtSurrAttr(
              xkey(ARG), UNQUOTE(resType), UNQUOTE(attrName))));
    endif;
    return f;
end;
```

The function ImpExtType creates a new type by calling the function MkExtType. This function is not shown here, but it will ultimately create the new type by calling the system function CreateType. After creating the new type, the function CrAttrF is called repeatedly to create the functions that will allow us to access the attributes of the objects residing in the external system.

The function CrAttrF creates a surrogate or character valued attribute function for an external attribute. CrAttrF is implemented as a procedure that first checks the existence of the attribute. If the attribute does not exist, an error will be raised. Next the function TypeOfExtAttr is called to determine the type

of the attribute. This information can be found by accessing the data dictionary of the external database. Given the type, we can create the attribute function by calling the system function CreateFun. This function takes the name of the function to be created, its argument and result type and a list of constraints and returns the oid of the newly created function. The next step is to implement the just created function. Essentially the body of the function will be a call to either the function ReadExtCharAttr or ReadExtSurAttr, depending on whether the attribute's type is a string or a surrogate type. The body of the function is specified as a backquoted expression, to allow us to substitute the attribute name and possibly the attrType into the function body. The function ReadExtSurAttr is the one defined earlier. The function ReadExtCharAttr is similar to ReadExtSurAttr, except that it returns a string object instead of a surrogate object.

The examples shown here are in many ways simplifications of the ones we have actually implemented. Due to the fact that OSQL is computationally complete and all of the system functionality is accessible by calling the appropriate system functions, there are few limits on what can be done. Another example of extensions that we have implemented are functions to control caching of data residing in external systems. This is often necessary to overcome performance limitations inherent in accessing the external system, for example when the external system does not provide the indexes needed to efficiently evaluate queries posed by the applications running on top of our system.

6 *Alternative Approaches*

In this section we want to compare the approach we have taken with OSQL to that of extending the C++ programming language with persistence mechanisms. The first observation that we make is that, due to the fact that extension constructs provided by C++ are limited, approaches that provide a seamless integration between programming language constructs and query constructs and that allow for the compilation and optimization of queries (e.g. those described in [Agrawal and Gehani 1989; Blakeley 1994]) require the use of a language preprocessor. In effect, one defines a new and extended language. Secondly, the C++ object model is at the same time more complex and less flexible as the OSQL object model. As alluded to before, the C++ object model mixes the representation of an object with its interface definition. Also, the C++ model does not provide support for adding or removing types from objects, a requirement if one wants to realistically model persistent objects. Thirdly, the C++ language provides only very limited mechanisms for function polymorphism and the type system requires that the types of the identifiers in an expression be known at compile time. These restrictions, while useful at times, also prevent the definition of functions like those discussed in section 5, where we pass types and functions as arguments.

In our view, C++ is not a suitable basis for defining a database programming language, because it falls short off meeting some of our essential design goals as summarized in the introduction. That notwithstanding, C++ can be an excellent choice as an applications programming language. By carefully designing and implementing the C++ application programming interface, taking advantage of the data abstraction capabilities offered by C++, one can easily provide a set of classes to provide a seamless interface to the OSQL data model.

7 *Conclusions*

We have presented OSQL, a language developed to facilitate the design and implementation of object databases. OSQL is a computationally complete language that combines the power of relational query languages with the expressiveness and extensibility characteristic of a functional programming language. OSQL provides a type system that allows objects to evolve over time, acquiring new types and losing old ones as their roles change. OSQL provides support for aggregate types and avoids the impedance mismatch problem between programming language and query language by tightly integrating the two. OSQL is very well suited to developing applications in a distributed world. By providing the capability to evaluate (user-defined) functions in the database server, the application developer can implement functions and then share these across a large number of applications. In addition, as shown by the examples in this chapter, database functions can interact with other application services, providing the capability to integrate data residing in external applications and/or databases and building integrated information management systems that preserve the integrity and autonomy of legacy applications. Moreover, by providing this capability as an orthogonal extension to the base language, one can use OSQL's declarative query processor to transparently query data residing either in the object database or in one or more external databases.

The OSQL language is characterized by the fact that it can evolve by extending the set of (system) types and functions provided. As a result, a lot of our development effort is directed towards building libraries of functions and types that facilitate the rapid development of applications in specific application domains and that provide the constructs needed to integrate information available in existing databases and other types of legacy applications. This is a first step towards an approach that would allow application developers to quickly construct information services by picking from an existing set of type and function definitions, refining them as needed and interface them with their (application specific) user interfaces using the communications facilities and application development tools provided in a distributed computing environment.

Acknowledgments

We would like to acknowledge the many contributions made by our colleagues in the database department of Hewlett-Packard Laboratories, where most of the early work on OSQL was done and where the Iris prototype was developed, as well as the Commercial Systems Division of Hewlett-Packard, where OpenODB was designed and implemented.

References

Agrawal, R., and Gehani, N.H. 1989. ODE (Object Database and Environment): The Language and Data Model. ACM SIGMOD 1989, May 1989, Portland, 36-45

Ahad, R. and Dedo, D. 1992. OpenODB from Hewlett-Packard: A Commercial OODBMS. Journal of Object Oriented Programming, Vol. 4, Number 9, Feb. 1992

Ahad, R. et.al. 1992. Supporting Access Control in an Object-Oriented Database Language. Proc. Intl. Conf. on Extending Database Technology, Vienna, Austria, March 1992

Ahad, R. and Cheng, T. 1993. OpenODB - An Object-Oriented Database Management System for Commercial Applications. HP Journal, Vol 44, No 3, June 1993, 20-30

Annevelink, Jurgen 1991. Database Programming Languages: A Functional Approach. ACM SIGMOD 91, May 1991, Denver, 318-327

Annevelink, J., Young, C.Y., Tang, P.C. 1991. Heterogeneous Database Integration in a Physician's Workstation. Proc. 15th Annual Symposium on Computer Applications in Medical Care, McGraw-Hill, New York, 1991, 368-372

Blakeley, J. A. 1994. ZQL[C++]: Extending the C++ Language with an Object Query Capability. this book

Ellis, M.A. and Stroustrup, B. 1990. The Annotated C++ Reference Manual. Addison-Wesley, 1990

Fishman, D. H. et.al. 1989. Overview of the Iris DBMS" in: Object Oriented Concepts, Databases and Applications. W. Kim and F.H. Lochovsky, Eds. New York, ACM, 1989

Goldberg, A. and Robson D. 1983. SMALLTALK-80 The Language and its Implementation. Addison Wesley, 1983

Hewlett-Packard 1992. OpenODB Reference Manual B3185A 1st ed 9/1992

Kent, W. 1993. A Model-Independent Query Paradigm Founded on Arithmetic. in preparation

Lyngbaek, Peter 1991. OSQL: A Language for Object Databases. Hewlett-Packard Laboratories technical report, HPL-DTD-91-4

Mylopoulos, J. Bernstein, P.A. Wong, H. K. T. 1980. A Language Facility for Designing Database-Intensive Applications. ACM TODS, 5(2), Jun., 1980.

Object Management Group 1991. The Common Object Request Broker: Architecture and Specification. Document Number 91.12.1

Shipman, D. 1981. The Functional Data Model and the Data Language DAPLEX. ACM TODS, 6(1), Mar, 1981

Department of Veterans Affairs 1990. VA Fileman User's Manual. Version 18, September 1990,

Wilkinson, K. Lyngbaek, P. and Hasan, W. 1990. The Iris Architecture and Implementation. IEEE Transactions on Knowledge and Data Engineering 2(1), Mar. 1990