Rationale for the Design of Persistence and Query Processing Facilities in the Database Programming Language O++

R. Agrawal
N. H. Gehani

AT&T Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

ODE is a database system and environment based on the object paradigm. It offers one integrated data model for both database and general purpose manipulation. The database is defined, queried, and manipulated in the database programming language O++, an extension of C++. O++ uses the C++ object definition facility, called the class, to provide data encapsulation and multiple inheritance. O++ extends C++ by providing facilities for creating persistent and versioned objects, defining and manipulating sets, organizing persistent objects into clusters, iterating clusters of persistent objects, and associating constraints and triggers with objects. In this paper, we present the O++ facilities for persistence and query processing, the alternatives considered, and the rationale behind the design choices.

1. INTRODUCTION

ODE is a database system and environment based on the object paradigm [2]. The database is defined, queried, and manipulated using the database programming language O++. O++ is an upward compatible extension of the object-oriented programming language C++ [Stroustrup 1986] that offers one integrated data model for both database and general purpose manipulation. The C++ object definition facility is called the *class*, which supports data encapsulation and multiple inheritance. O++ extends C++ by providing facilities for creating persistent and versioned objects, defining and manipulating sets, organizing persistent objects into clusters, iterating over clusters of persistent objects, and associating constraints and triggers with objects.

In this paper, we present the O++ facilities for persistence and query processing, the alternatives that we considered, and the rationale behind our design choices. We assume that the reader is familiar with the basic issues in the design of database programming languages (see, for example, [6, 10, 11, 26, 33, 38] for introduction) and concentrate on our design. Our discussion is oriented around O++, but we think that lessons we learned have wider applicability.

The organization of the rest of the paper is as follows. In Section 2, we give a brief overview of the C++ object definition facility. In Section 3, we consider the persistence model of O++, which was inspired by our desire to provide facilities so that persistence becomes a property of object instances and not object types, and persistent objects are accessed and manipulated like volatile heap objects, without imposing run-time penalty for code that does not deal with persistent objects. We present design goals, alternatives considered, and the rationale for the choices made. In Section 4, we consider the query processing constructs provided in O++. These constructs were motivated by our desire to provide facilities in an object-oriented framework for set-oriented query processing and optimizations similar to those found in relational database systems. We again present design goals, alternatives considered, and the rationale for the choices made. We discuss related work in Section 5

2. OBJECT DEFINITION: A BRIEF OVERVIEW

The C++ object facility is called the class. Class declarations consist of two parts: a specification (type) and a body. The class specification can have a private part holding information that can only be used by its implementer, and a public part which is the type’s user interface. The *body* consists of the bodies of functions declared in the class specification but whose bodies were not given there. For example, here is a specification of the class `item`:

```
class item {
    double wt; /* in kg */
public:
    Name name;
    item(Name xname, double xwt);
    double weight_lbs();
    double weight_kg();
};
```

The private part of the specification consists of the declaration of the variable `wt`. The public part consists of one variable `name`, and the functions `item`, `weight_lbs` and `weight_kg`.

C++ supports inheritance, including multiple inheritance [35], which is used for object specialization. The specialized object types inherit the properties of the base object type, i.e., the data and functions, of the ‘‘base’’ object type. As an example, consider the following class `stockitem` that is derived from class `item`:

```
class stockitem: public item {
   int consumption;
   int leadtime;
public:
   int qty;
   double price;
   stockitem(Name iname, double iwt, int xqty, int xconsumption,
             double xprice, int xleadtime);
   int eoq();        /*economic order quantity*/
};
```

`stockitem` is the same as `item` except that it contains other information such as the quantity in stock, its consumption per year, its price and the lead time necessary to restock the item.

We have only given the details of class definitions that are necessary for the discussion in this paper. Please see [36] for further details.

3. PERSISTENCE

We visualize memory as consisting of two parts: volatile and persistent. *Volatile* objects are allocated in volatile memory and are the same as those created in ordinary programs. *Persistent* objects are allocated in persistent store and they continue to exist after the program creating them has terminated. A database is a collection of persistent objects, each identified by a *unique* identifier, called the object identity [19]. We shall also refer to this object identity as a *pointer to a persistent object*.

3.1 Design Goals

When incorporating persistence in O++, we kept the following design goals in perspective.

- Persistence should be orthogonal to type [6]. Persistence should be a property of object instances and not types. It should be possible to allocate objects of any type in either volatile or persistent store.
- There should be no run-time penalty for code that does not deal with persistent objects.
- Allocation and manipulation of persistent objects should be similar to the manipulation of volatile objects. For example, it should be possible to move objects from persistent store to volatile store and vice versa in much the same way as it is possible to move objects from the stack to the heap and vice versa.
- Inadvertent fabrication of object identities should be prevented.
- Language changes should be kept to a minimum.

3.2 Persistence in O++

We first give an overview of the persistence model of O++, and then discuss the rationale behind our decisions.

3.2.1 Allocating and Manipulating Persistent Objects Persistent objects are referenced using pointers to persistent objects (that is, their identities). Persistent objects are allocated and deallocated in a manner similar to heap objects. We choose to view persistent store as being similar to heap because most programmers are already familiar with manipulating heap objects. Persistent storage operators `pnew` and `pdelete` are used instead of the heap operators `new` and `delete`. Here is an example:

```
persistent stockitem *psip;
...
psip = pnew stockitem(initial-values);
```

`psip` is allocated on stack but `pnew` allocates the `stockitem` object in persistent store and its id (returned by `pnew`) is saved in `psip`. Note that `psip` is a pointer to a persistent `stockitem` object, and *not* a persistent pointer to a `stockitem` object.

Persistent objects can be copied to volatile objects and vice versa using simple assignments:

```
*sip = *psip; /*copy the object pointed to by psip*/
              /*to the object pointed to sip      */
*psip = *sip; /*and vice versa */
```

Components of persistent objects are referenced like the components of volatile objects, e.g.,

```
w = psip->weight_kg();
```

A persistent object can be deleted with the `pdelete` operator.

3.2.2 Dual Pointers Having only ordinary pointers and pointers to persistent objects has the following ramifications:

i. If a class is used to build a linked data structure, then the same class cannot be used to create the data structure both in volatile memory and in persistent store. For example, consider a class `node` that contains a component `next` that points to the next node in a list. Now if this pointer refers to a volatile `node` object, then we cannot use `node` to build a list in persistent store. The reverse happens if `next` points to a persistent object.
ii. We cannot write a function that can take as arguments pointers to either volatile objects or to persistent objects.

We avoid these problems by providing `dual` pointers that can reference either a volatile object or a persistent object. Whether the pointer refers to a volatile object or to a persistent object is determined at run time. Here is an example illustrating the use of dual pointers:

```
class node {
    ...
    dual node *next;
public:
    ...
    dual node *add(dual node *n);
};

persistent node *p, *proot; /*proot refers to a list in persistent store*/
node *v, *vroot;            /*vroot refers to a list in volatile store*/
...
proot = proot->add(p);
vroot = vroot->add(v);
```

3.3 Some Alternatives

O++ has three kinds of pointers: pointers to volatile objects (ordinary C or C++ pointers), pointers to persistent objects, and dual pointers. We were reluctant to add two new pointer types and gave much thought to the following two alternatives:

i. Have only one type of pointer that can point to both volatile and persistent objects.
ii. Have two types of pointers: an ordinary pointer that can only point to volatile objects and another that can point either to a persistent object or a volatile object, i.e., a dual pointer.

3.3.1 One-Pointer Alternative Using one pointer to refer to either a volatile object or a persistent object has several advantages: the language has two less pointer types and all (including existing) code will work with both volatile objects and persistent objects. This code compatibility could be at the source level or even at the object level depending upon the implementation strategy. Having one pointer type also makes types strictly orthogonal to persistence. Finally, the only syntactic additions required to C++ would be the operators `pnew` and `pdelete`.

We rejected the one-pointer alternative mainly on performance grounds: we wanted to have facilities that were easy to port and efficient to implement on general purpose machines and on standard operating systems (particularly on the UNIX® system) without requiring special hardware assists.

With the one-pointer alternative, a run-time check must be made to determine whether a pointer refers to a volatile object or a persistent object. Performing this check for every pointer dereference imposes a run-time penalty for references to volatile objects. This overhead is unacceptable in languages such as C and C++ (and therefore O++) which are used for writing efficient programs. Note that in these languages pointers are used heavily, e.g., string and array manipulation are all done using pointers.

One way of avoiding this run-time overhead is to trap references to persistent objects, for example, by using illegal values such as negative values for pointers to persistent objects (assuming that the underlying hardware or operating system provides the appropriate facilities). References to volatile objects then would not incur any penalty. This scheme has several problems:[1]

i. Trapping illegal addresses is like trapping illegal memory references. The trapping mechanism and the resultant actions differ from one machine architecture and operating system to another. On the UNIX system, memory traps can be implemented by catching the segment violation (`SIGSEGV`) signal. However, interception of this signal is under user control, and the user (or some library routines) may (inadvertently) turn off this signal making persistent object inaccessible.
ii. A program may erroneously generate a pointer that refers to a persistent object which can result in the erroneous update of a persistent object. If the database programming language cannot prevent such errors, then it will be hard to write robust applications. Note that illegal pointer references are a common source of errors in C programs.
iii. There is no machine-independent way of generating illegal addresses. For example, negative addresses are illegal in the UNIX implementations on VAXen but not so on AT&T 3B computers.

Another problem with the one-pointer approach is that it limits the size of pointers to persistent objects to the size of ordinary pointers. This could become a problem for large databases, and it reduces flexibility of the implementation in deciding what information can be stored in a pointer to a persistent object.

These problems do not arise when type information can be used to determine, at compile time, references that involve persistent objects. There is no need to use a memory trap mechanism to differentiate between pointers to volatile and persistent objects and appropriate compile-time checks will ensure that illegal references to persistent objects are not generated inadvertently. Also, the implementation will then not have to limit the size of pointers to persistent objects to the size of ordinary pointers.

1. Some of these problems were pointed out by S. Buroff.

As a brief aside, let us remark that a technique called ‘‘pointer swizzling’’ has been used in some systems (for example, PS-Algol [15]) to optimize accesses to persistent objects.[2] The first reference to a persistent object results in its being cached in volatile memory and the original pointer is replaced by the pointer to the cache location. When an object is written back to the persistent store from the cache, any swizzled pointers (in this or in other objects) that contain the cache address of this object must be ‘‘deswizzled’’. Pointer swizzling is orthogonal to the issues of portability, robustness, and implementation flexibility; the problems with the one-pointer approach do not go away with pointer swizzling.

3.3.2 Two-Pointer Alternative Instead of using three pointers, we seriously considered using only two pointers: ordinary pointers and dual pointers, but finally decided in favor of the three pointer approach. The main argument for not having pointers that point only to persistent objects is that the gain in run-time efficiency by avoiding the check needed in dual pointers would not be significant compared to the cost of accessing persistent store. On the other hand, the use of separate pointers for persistent objects leads to better program readability and allows the compiler to provide better error checking, e.g., flagging arithmetic on pointers to persistent objects as being inappropriate. Based on our limited experience (which is writing small sample programs in O++), we feel that programmers will use pointers to persistent objects when appropriate.

3.4 Clusters of Persistent Objects

We view persistent store as being partitioned into clusters each of which contains objects of the same type. Initially, we decided that there would be a one-to-one correspondence between cluster names and the corresponding type names. Whenever a persistent object of a type was created, it was automatically put in the corresponding cluster. Thus, our clusters were type extents [12].

This strategy preserved the inheritance relationship between the different objects in the persistent store, and worked nicely with our iteration facilities (discussed in the next section) allowing us to iterate over a cluster, or over a cluster and clusters ‘‘derived’’ from it[3]. Before creating a persistent object, the corresponding cluster must have been created by invoking the `create` macro (in the same program or in a different program[4]). Additional information (such as indexing) may be provided to the object manager to assist in implementing efficient accesses to objects in the cluster. A cluster, together with all the objects in it, can be destroyed by invoking the `destroy` macro.

Bloom and Zdonik [10] discuss issues in using type extents to partition the database. Although this scheme frees the programmer from explicitly specifying the cluster in which a persistent object should reside, it suffers from the following disadvantages:

i. Unrelated objects of the same type will be grouped together (e.g., employees of unrelated companies, or hash tables of unrelated applications).
ii. It will not be possible to optimize queries that involve subsets of the objects in a cluster.

In short, the disadvantages were due to the absence of a subclustering mechanism.

Although we used type extents to partition the database, our scheme did not suffer from the above problems because subclusters could be created with the inheritance mechanism. For example, employee of a companies A and B could be represented by types `A-employee` and `B-employee` derived from the type `employee`. Each of these derived types would have a cluster corresponding to it. All employees could then be referenced using the cluster `employee`, while employees belonging only to company A or company B could be referenced by using clusters `A-employee` or `B-employee`, respectively. Different indices could also be created for the two clusters.

2. See [26] for a lucid discussion of pointer swizzling, problems with it, and an alternative approach to optimizing accesses to persistent objects.

3. To be precise, clusters are not derived; they simply mirror the inheritance (derivation) relationship between the corresponding classes.

4. The ODE environment can be queried about the clusters that exist in persistent store.

Reaction (from our colleagues) to this clustering scheme was unfavorable on the grounds that our clustering scheme did not allow multiple clusters for the same object type. The use of the inheritance mechanism for creating subclusters was not appealing for the following reasons:

i. The type inheritance mechanism is a static mechanism.
ii. The type inheritance mechanism is being overloaded to create subclusters.
iii. The use of type inheritance to form subclusters becomes unwieldy if the number of subclusters is large. Consider, for example, objects of type `order`, which have to be clustered by order date. If one used inheritance to create such clusters, one would end up with an unacceptably large number of types.

Consequently, we allowed creation of multiple clusters to group together objects of the same type, but retained the notion that some ‘‘distinguished’’ clusters represent type extents to preserve the inheritance relationship between objects in the database. An object implicitly always belongs to the distinguished cluster whose name matches the name of its type. However, it may also belong to another cluster which may be thought of as a subcluster of the corresponding distinguished cluster.

Subclusters are also created and destroyed with the `create` and `destroy` macros. A subcluster name is a string qualified by the corresponding cluster (object type) name, e.g.,

```
create(employee::"A-employee");
destroy(employee::name);        /*name is a string variable*/
```

Objects may be specified to belong to a specific subcluster when they are allocated in persistent store, e.g.,

```
ep = pnew employee("Jack")::"A-employee";
```

4. QUERY PROCESSING CONSTRUCTS

4.1 Design Goals

An important contribution of the relational query languages was the introduction of the set processing constructs. This capability allows users to express queries in a declarative form without concern for physical organization of data. A query optimizer (see, for example, [32]) is made responsible for translating queries into a form appropriate for execution that takes into account the available access structures. Object-oriented languages, on the other hand, typically do not provide set-oriented processing capabilities. Indeed, the major criticism of the current object-oriented database systems is that the query processing in these systems ‘‘smells’’ of pointer chasing and that they may take us back to the days of CODASYL database systems in which data is accessed by ‘‘using pointers to navigate through the database’’ [21]. One of the design goals of O++ was to provide set-processing constructs similar to those found in relational query languages.

Object-oriented languages are ‘‘reference’’ oriented, in that the relationship between two objects is established by embedding the identity of one object in another. Frequently, it is not possible to envision all relationships between the objects at the time of designing the database schema (class definitions). Relational systems are ‘‘value’’ oriented and relationships between objects (tuples) are established by comparing the values of some or all attributes of the objects involved. This lack of capability to express arbitrary ‘‘join’’ queries has been cited as another major deficiency of the current object-oriented database systems [21]. A design goal of O++ was to correct this deficiency.

In [6], Atkinson and Buneman proposed four database programming tasks as benchmarks to study the expressiveness of various database programming languages. One of the tasks, the computation of the bill of materials which involves recursive traversal, was found particularly awkward to express in many of the database programming languages discussed. Considerable research has been devoted recently to developing notations for expressing recursive queries in a relational framework and designing algorithms for evaluating them (see, for example, [1, 7]). Providing capability to express recursive queries in a form that can be used to recognize and optimize recursive queries was another design goal of O++.

The distinguished clusters mirror the hierarchy relationship of the corresponding types. If type *x* is derived from type *y*, then the corresponding clusters also have the same relationship. It is sometimes necessary to collectively access objects in a cluster and those in related ‘‘derived’’ clusters. POSTQUEL [29] allows a * to be specified after the relation name to retrieve tuples from the named relation and all relations that inherit attributes from it. Orion [9] also provides similar functionality. Providing capability for such accesses was another design goal.

4.2 Set-Oriented Constructs

Recall that the persistent objects of a type implicitly belong to the corresponding distinguished cluster which has the same name as the type. Objects in a cluster can also be partitioned into named subclusters. O++ provides a `for` loop for accessing the values of the elements of a set, a cluster or a subcluster[5]:

`for` *i* `in` *set-or-cluster-or-subcluster* **[** *suchthat-clause* **]** **[** *by-clause* **]** *statement*

The loop body, i.e., *statement*, is executed once for each element of the specified set, the cluster or the subcluster; the loop variable *i* is assigned the element values in turn. The type of *i* must be the same as the element type.

The *suchthat-clause* has the form

`suchthat(`e_{st}`)`

and the *by-clause* has the form

5. The square brackets **[** and **]** indicate an optional item.

`by(`e_{by} **[** , *cmp* **]**)

If the `suchthat` and the `by` expressions are omitted, then the `for` loop iterates over all the elements of the specified grouping in some implementation-dependent order. The `suchthat` clause ensures that iteration is performed only for objects satisfying expression e_{st}. If the `by` clause is given, then the iteration is performed in the order of non-decreasing values of the expression e_{by}. If the `by` clause has only one parameter, then e_{by} must be an arithmetic expression. If the `by` clause has two parameters, then the second parameter *cmp* must be a pointer to a function that compares two elements of type *t*, where *t* is the type of the `by` expression e_{by}. *cmp* compares its arguments and returns an integer greater than, equal to, or less than 0, depending upon whether its first argument is respectively greater than, equal to, or less than its second argument.

For example, the following statement prints the name of stockitems heavier than 10kg in order of increasing price:

```
for s in stockitem suchthat(s->weight_kg() > 10) by (s->price)
        printf("%s %f\n", s->name, s->price);
```

Here is the famous ‘‘employees who make more than their managers’’ query:[6]

```
class employee {
public:
    Name name;
    float salary;
    persistent employee *subordinates<>;
};
...
for m in employee
    for s in m->subordinates suchthat(s->salary > m->salary)
        printf("%s\n", s->name);
```

Although the `suchthat` and `by` clauses can be simulated by using an *if* statement within the loop body and by sorting the set of values in volatile memory, these clauses facilitate optimization as follows: we expect to pass these clauses to the object manager to select only the desired object ids and deliver them in the right order for the `for` loop. The object manger is responsible for maintaining appropriate indices to speed up accesses.

To allow expression of order-independent join queries, we allow multiple loop variables in the `for` loops:

`for` i_1 `in` *set-or-cluster-or-subcluster*$_1$`,` ..., i_n `in` *set-or-cluster-or-subcluster*$_n$
[`suchthat(`e_{st}`)` **]** **[** `by(`e_{by}`)` **]** *statement*

The loop body, i.e., *statement*, is executed for every combination of values for the loop variables that, if given, satisfy the `suchthat` expression and in the order specified by the `by` clause.

These loops allow the expression of operations with functionality of the arbitrary relational join operation. For example, we can write

```
for e in employee, d in dept suchthat(e->dno==d->dno && e->salary>100)
    printf("%s %s\n", e->name, d->name);
```

to print the names of the employees who make more than 100K and the names of their departments. We again hand over this query to the object manager which decides how best to execute the query. In particular, the object manager can permute the order of loop variables, reorder the expressions within `suchthat` clause, and use any join algorithm that it determines to be optimal.

6. In O++, angle brackets <> denote a set.

The `suchthat` and `by` clauses were inspired by similar clauses in SQL [14] and Concurrent C [18]. Similar `for` loops have been provided, among others, in Pascal/R [31], Rigel [28], Plain [37] and Trellis/Owl [23]. Multiple loop variables in `for` loops are also allowed in Rigel [28].

4.3 Recursive Queries

When iterating over a set or a cluster, we allow iteration to also be performed over the elements that are added during the iteration, which allows the expression of recursive queries [3]. Thus, given a class `person` with a set component named `children`, the following statements finds all the descendents of `abraham`:

```
for p in person suchthat(p->name == Name("abraham"))
    descendants = p->children;    /*basis for recursion*/
for d in descendants
    descendants += d->children;   /*recursion*/
```

In the Appendix, we show how the `for` loop can be used to easily express the computation of the bill of materials. We hope to recognize recursive queries and use efficient algorithms for processing them.

4.4 Accessing Cluster Hierarchies

All objects in hierarchically related clusters can be accessed using a `forall` loop of the form

`forall` *oid* `in` *cluster* **[** *suchthat-clause* **]** **[** *by-clause* **]** *statement*

Except for the inclusion of objects in derived clusters, the semantics of the `forall` loop are the same as those of the `for` loop for iterating over a cluster. Thus, given the class `item` and the derived class `stockitem` as defined earlier, the statement

```
for ip in item
    tot_wt += ip->weight_kg;
```

computes the weight of only objects of type `item`, but the statement

```
forall ip in item
    tot_wt += ip->weight_kg;
```

computes the weight of all items including stockitems.

4.4.1 Type Test When iterating over objects in a cluster and in hierarchically related clusters, it is sometimes necessary to be able to determine the type of the objects. O++ provides the `is` operator for this test, which is of the form

e `is` *type*

This expression evaluates to true if expression *e* is of the specified type, and to false otherwise.

For example, suppose that we want to compute and print the average income of university employees and, separately, that for faculty and students in a university database (employees can be other than faculty and students, e.g., staff). Class `person` has the member function `income`, and classes `student` and `faculty` have been derived from it. Here is the code to perform the above computation:

```
np = ns = nf = incomep = incomes = incomef = 0;
forall p in person {
   incomep += p->income(); np++;
   if (p is persistent student *)
       { incomes += p->income(); ns++;}
   else if (p is persistent faculty *)
       { incomef += p->income(); nf++;}
}
printf("%g %g %g\n", incomep/np, incomed/nd, incomef/nf);
```

One can simulate this task using C++ virtual functions, but this may require modification of existing class definitions to add appropriate functions, which is not appropriate for posing arbitrary queries.

4.5 An Alternative Approach

An alternative to providing special `for` loops for iterating over clusters would be to define two additional functions in each class definition to iterate over persistent objects. The first function would be used for loop initialization and the second function, say `next`, would return the next element of the iteration. These functions could then be used in conjunction with the existing loop statements in C++ for iterating over clusters. However, such an approach suffers from the following disadvantages:

i. It is the class definer's responsibility to provide the iteration functions.
ii. It would be hard, if not impossible, to provide the functionality of the `by` and `suchthat` clauses, and perform the optimizations that may be possible with these clauses.
iii. The iteration functions provide a lower-level iteration interface than that provided by the iteration facilities in O++.
iv. The iteration functions may not work properly in case of nested iterations involving the same cluster.
v. The iteration functions may not provide the functionality of the `forall` loop.
vi. O++ must provide some low-level mechanism for accessing items in the clusters which would be used to implement the loop iteration functions.

It is for these reasons that we decided to add new iteration facilities to O++.

5. RELATED WORK

Many research efforts have attempted to add the notion of persistence and database-oriented constructs in programming languages (see, for example, [4, 5, 8, 13, 16, 22-26, 28, 30, 31, 34, 37]). Some of these database programming languages have been surveyed and compared in [6]. Issues in integrating databases and programming languages have been discussed in [6, 10, 11, 26, 33, 38]. In the remainder of this section, we limit our discussion to some other ongoing work on combining C and C++ with databases.

Closely related to our work is the language E [26, 27], which also started with C++ and added persistence to it. However, persistent objects in E must be of special types called ‘‘db’’ types. Objects of such types can be volatile or persistent. Persistence orthogonality in E can thus be realized by programming exclusively in ‘‘db’’ types, but all references to volatile objects in that case would incur run time check to see if they need to be read into memory [26]. Otherwise, one has to have two class definitions, one ‘‘db’’ type and one ‘‘non-db’’ type, if objects of the same type are to be allocated both in volatile memory and in persistent store.

Persistent objects in E can be allocated statically by using the storage class `persistent` in object definitions or they can be allocated dynamically by using the predefined ‘‘db’’ class `file`. E does not directly provide the notion of type extents. A programmer can simulate the effect of clusters by inserting the desired objects in an instance of the ‘‘db’’ class `file`. However, E does not provide facilities for maintaining inheritance relationships between objects whose types are hierarchically related. E is intended to be a database implementation language rather than a database system. It therefore does not provide constructs for set-oriented query processing. E has been implemented, and an interesting discussion of the implementation issues appear in [26].

Avalon/C++ [17], a programming language designed for writing reliable distributed systems, also provides facilities for persistence. Types whose objects are to be allocated in persistent store must be *derived* from the class `recoverable`. A persistent object in Avalon/C++ is accessed by explicitly bringing it to volatile store from persistent store (called ‘‘pinning’’ the object), accessing the object, and then moving the object back to persistent store (called ‘‘unpinning’’ the object). The pinning and unpinning operations are provided by the class `recoverable`. A syntactic extension, the ‘‘pinning’’ block is provided to support the accessing of persistent objects. From what we know, Avalon/C++ does not provide query facilities nor does it support the notion of organizing objects into clusters/subclusters.

Vbase [5] combines an object model with C. It presents to the user two languages: the type definition language TDL for specifying classes and operations, and the C superset COP for writing methods to implement the operations. A new product Vbase+, more closely tied to C++, is being currently designed that proposes to implement persistence through library routines.

The O2 system [20] also integrates an object model with C. Type definitions are written in one language and methods are written in the C superset CO2. A class in O2 implicitly owns a persistent collection of objects of the class, which is similar to our notion of a cluster.

6. FINAL COMMENTS

An important goal of research in programming languages design is to provide a better fit between the application domain and the programming notation. We started with the object-oriented facilities of C++ and extended them with features to support the needs of databases, putting to good use all the lessons learned in implementing today's database systems. At the same time, we tried to maintain the spirit of C++ (and C) by adding only those facilities that we considered essential for making O++ a database programming language. In the appendix, we give the O++ versions of the benchmark tasks used in [6] to compare database programming languages. We feel that O++ provides a clean fusion of database concepts in an object-oriented programming language.

7. ACKNOWLEDGMENTS

Our design of persistence has been influenced by the discussions we had with Steve Buroff, Mike Carey, Keith Kelleman, Dennis Shasha, and Bill Weber. We appreciate the suggestions of Chandra Kintala.

8. APPENDIX

In [6], Atkinson and Buneman proposed four database programming tasks as benchmarks to study the expressiveness of various database programming languages. In this Appendix, we code these tasks in O++. The reader is encouraged to compare these programs with the solutions given in [6] for the same tasks but written in other database programming languages. We (of course!) feel that the O++ solutions to these tasks are quite elegant and natural.

Atkinson and Buneman considered a manufacturing company’s parts database. The database stores how a composite part is assembled using other base and composite parts and what mass and cost increments occur due to assembly. The four tasks are:

Task 1: *Describe the database*.

The database schema in O++ is just a set of class definitions:

```
    ...
typedef struct { persistent part *partid; int qty; } partqty;

class part {
public:
    Name name;
    ...
};

class basepart: public part {
public:
    float cost, mass;
    ...
};

class compositepart: public part {
public:
    float costincrement, massincrement;
    partqty subparts<>;   /*set of partid, qty pairs for subparts*/
    ...
};
```

Task 2: *Print the names, cost, and mass of all base parts that cost more than $100*.

This query is implemented by simply iterating over the cluster `basepart` with the appropriate selection condition specified in the `suchthat` clause:

```
persistent basepart *bp;
    ...
for bp in basepart suchthat(bp->cost >= 100)
{
   printf("name=%s,cost=%f,mass=%f\n",bp->name, bp->cost, bp->mass);
}
```

Task 3: *Compute the total mass and total cost of a part named ‘‘mast’’*.

We first find the part ‘‘mast’’. If it is a base part, we get the mass and cost directly; otherwise, it is a composite part, and its mass is the incremental mass of the assembly plus the sum of the mass of its components (which may themselves be composite parts). The cost of the part is computed similarly.

```
persistent part *p;
partqty c, components<>;

forall p in part suchthat(p->Name == Name("mast"))
{
    if p is persistent basepart * { /*part has no components*/
        mass = p->mass; cost = p->cost;
    }
    else {  /*part named "mast" is a composite part*/

        mass = p->massincrement; cost = p->costincrement;
        components = p->subparts;  /*subparts of "mast"*/

        for c in components    /*recursion*/
            if c.partid is persistent basepart * {
                mass += c.qty * c.partid->mass;
                cost += c.qty * c.partid->cost;
            }
            else {
                mass += c.partid->massincrement;
                cost += c.partid->costincrement;
                components += c.partid->subparts;
            }
    }
}
```

Task 4: *Record a new manufacturing step in the database, that is, how a new composite part is manufactured from subparts.*

Given a composite part, its subparts and the assemblies that use it, we add this composite part to the database by allocating a composite part object (with id `nid`), then storing the ids of its subparts (children) and the quantity information in object `nid` and, finally, determining the ids of the assemblies (parents) that use this composite part and storing in these objects `nid` and the quantity of `nid` used.

```
typedef struct { Name name; int qty; } info[MAX];

void add(char *pname, float costincr, float massincr,
         int nchild, info children, int nparent, info parents)
{
    persistent compositepart *nid; /*new part*/
    persistent part *cid;          /*child id*/
    persistent compositepart *pid; /*parent id*/
    partqty c;                     /*partid, qty pair*/
    int i;

    nid = pnew compositepart;
    nid->name = Name(pname);
    nid->costincrement = costincr;
    nid->massincrement = massincr;

    /*store oids of children in the new composite part*/
    for (i=0; i < nchild; i++)
        for cid in part suchthat(cid->name == children[i]->name)) {
            c.partid = cid;
            c.qty = children[i].qty;
            nid->subparts += c; /*store info. in new composite part*/
        }

    /*store oids of new composite part in parts that use it*/
    for (i=0; i < nparent; i++)
        for pid in compositepart
                   suchthat(pid->name == parents[i]->name)) {
            c.partid = nid;
            c.qty = parents[i].qty;
            pid->subparts += c;  /*store information in parent*/
        }
}
```

REFERENCES

[1] R. Agrawal, ‘‘Alpha: An Extension of Relational Algebra to Express a Class of Recursive Queries’’, *Proc. IEEE 3rd Int’l Conf. Data Engineering*, Los Angeles, California, Feb. 1987, 580-590.

[2] R. Agrawal and N. H. Gehani, ‘‘Ode (Object Database and Environment): The Language and the Data Model’’, *Proc. ACM-SIGMOD 1989 Int’l Conf. Management of Data*, Portland, Oregon, May-June 1989, 36-45.

[3] A. V. Aho and J. D. Ullman, ‘‘Universality of Data Retrieval Languages’’, *Proc. 6th ACM Symp. Principles of Programming Languages*, San-Antonio, Texas, Jan. 1979, 110-120.

[4] A. Albano, L. Cardelli and R. Orsini, ‘‘Galileo: A Strongly Typed Interactive Conceptual Language’’, *ACM Trans. Database Syst. 10*, 2 (June 1985), 230-260.

[5] T. Andrews and C. Harris, ‘‘Combining Language and Database Advances in an Object-Oriented development Environment’’, *Proc. OOPSLA ’87*, Orlando, Florida, Oct. 1987, 430-440.

[6] M. P. Atkinson and O. P. Buneman, ‘‘Types and Persistence in Database Programming Languages’’, *ACM Computing Surveys 19*, 2 (June 1987), 105-190.

[7] F. Bancilhon and R. Ramakrishnan, ‘‘An Amateur’s Introduction to Recursive Query Processing Strategies’’, *Proc. ACM-SIGMOD 1986 Int’l Conf. on Management of Data*, Washington D.C., May 1986, 16-52.

[8] F. Bancilhon, T. Briggs, S. Khoshafian and P. Valduriez, ‘‘FAD, a Powerful and Simple Database Language’’, *Proc. 13th Int’l Conf. Very Large Data Bases*, Brighton, England, Sept. 1987, 97-105.

[9] J. Banerjee, H. T. Chou, J. F. Garza, W. Kim, D. Woelk and N. Ballou, ‘‘Data Model Issues for Object-Oriented Applications’’, *ACM Trans. Office Information Systems 5*, 1 (Jan. 1987), 3-26.

[10] T. Bloom and S. B. Zdonik, ‘‘Issues in the Design of Object-Oriented Database Programming Languages’’, *Proc. OOPSLA*, Orlando, Florida, Oct. 1987, 441-451.

[11] P. Buneman, ‘‘Can We Reconcile Programming Languages and Databases?’’, in *Databases – Role and Structure: An Advanced Course*, Cambridge Univ. Press, Cambridge, England, 1984, 225-243.

[12] P. Buneman and M. Atkinson, ‘‘Inheritance and Persistence in Database Programming Languages’’, *Proc. ACM-SIGMOD 1986 Int’l Conf. on Management of Data*, Washington D.C., May 1986, 4-15.

[13] L. Cardelli, ‘‘Amber’’, LNCS 242, 1986.

[14] D. D. Chamberlin, M. M. Astrahan, K. P. Eswaran, P. P. Griffiths, R. A. Lorie, J. W. Mehl, P. Reisner and B. W. Wade, ‘‘SEQUEL 2: A Unified Approach To Data Definition, Manipulation, and Control’’, RJ 1978, IBM, June 1976.

[15] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey and R. Morrison, ‘‘Persistent Object Management System’’, *Software Practice and Experience 14*, 1 (1984), 49-71.

[16] G. Copeland and D. Maier, ‘‘Making Smalltalk a Database System’’, *Proceedings of the 1984 ACM SIGMOD Intl. Conf. on Management of Data*, Boston, Massachusetts, June 1984, 316-325.

[17] D. D. Detlefs, M. P. Herlihy and J. M. Wing, ‘‘Inheritance of Synchronization and Recovery Properties in Avalon/C++’’, *IEEE Computer 21*, 12 (Dec. 1988), 57-69.

[18] N. H. Gehani and W. D. Roome, ‘‘Concurrent C’’, *Software—Practice & Experience 16*, 9 (1986), 821-844.

[19] S. N. Khoshafian and G. P. Copeland, ‘‘Object Identity’’, *Proc. OOPSLA ’86*, Portland, Oregon, Sept. 1986, 406-416.

[20] C. Lecluse, P. Richard and F. Velez, ‘‘O_2, an Object-Oriented Data Model’’, *Proc. ACM-SIGMOD 1988 Int’l Conf. on Management of Data*, Chicago, Illinois, June 1988, 424-433.

[21] E. Neuhold and M. Stonebraker, ‘‘Future Directions in DBMS Research’’, Tech. Rep.-88-001, Int’l Computer Science Inst., Berkeley, California, May 1988.

[22] B. Nixon, L. Chung, D. Lauzon, A. Borgida, J. Mylopoulis and M. Stanley, ‘‘Implementation of a Compiler for a Semantic Data Model’’, *Proc. ACM-SIGMOD 1987 Int’l Conf. on Management of Data*, San Fransisco, California, May 1987, 118-131.

[23] P. O’Brien, P. Bullis and C. Schaffert, ‘‘Persistent and Shared Objects in Trellis/Owl’’, *Proc. Int’l Workshop Object-Oriented Database Systems*, Asilomar, California, Sept. 1986, 113-123.

[24] Persistent Programming Research Group, ‘‘The PS-Algol Reference Manual, 2d ed.’’, Tech. Rep. PPR-12-85, Computing Science Dept., Univ. Glasgow, Glasgow, Scotland, 1985.

[25] J. E. Richardson and M. J. Carey, ‘‘Programming Constructs for Database System Implementation in EXODUS’’, *Proc. ACM-SIGMOD 1987 Int’l Conf. on Management of Data*, San Fransisco, California, May 1987, 208-219.

[26] J. E. Richardson and M. J. Carey, ‘‘Persistence in the E Language: Issues and Implementation’’, *Software—Practice & Experience 19*, 12 (Dec. 1989), 1115-1150.

[27] J. E. Richardson, M. J. Carey and D. H. Schuh, ‘‘The Design of the E Programming Language’’, Computer Sciences Tech. Rep. #824, Univ. Wisconsin, Madison, Feb. 1989.

[28] L. Rowe and K. Shoens, ‘‘Data Abstraction, Views and Updates in RIGEL’’, *Proc. ACM-SIGMOD 1979 Int’l Conf. on Management of Data,* Boston, Massachusetts, May-June 1979, 77-81.

[29] L. A. Rowe and M. R. Stonebraker, ‘‘The POSTGRES Data Model’’, *Proc. 13th Int’l Conf. Very Large Data Bases,* Brighton, England, Sept. 1987, 83-96.

[30] G. Schlageter, R. Unland, W. Wilkes, R. Zieschang, G. Maul, M. Nagl and R. Meyer, ‘‘OOPS – An Object Oriented Programming System with Integrated Data Management Facility’’, *Proc. IEEE 4th Int’l Conf. Data Engineering*, Los Angeles, California, Feb. 1988, 118-125.

[31] J. W. Schmidt, ‘‘Some High Level Language Constructs for Data of Type Relation’’, *ACM Trans. Database Syst. 2*, 3 (Sept. 1977), 247-261.

[32] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price, ‘‘Access Path Selection in a Relational Database Management System’’, *Proc. ACM-SIGMOD 1979 Int’l Conf. on Management of Data*, May 1979, 23-34.

[33] M. Shaw and W. A. Wulf, ‘‘Toward Relaxing Assumptions in Languages and their Implementations’’, *SIGPLAN Notices Notices 15*, 3 (March 1980), 45-61.

[34] J. M. Smith, S. Fox and T. Landers, *ADAPLEX: Rationale and Reference Manual, 2nd ed.*, Computer Corp. America, Cambridge, Mass., 1983.

[35] B. Stroustrup, ‘‘Multiple Inheritance for C++’’, *Proc. European UNIX User’s Group*, Helsinki, May 1987, 189-208.

[36] B. Stroustrup, *The C++ Programming Language (2nd Ed.)*, Addison-Wesley, 1991.

[37] A. Wasserman, ‘‘The Data Management Facilities of PLAIN’’, *Proc. ACM-SIGMOD 1979 Int’l Conf. on Management of Data*, Boston, Massachusetts, May-June 1979.

[38] S. N. Zilles, ‘‘Types, Algebras and Modeling’’, *SIGPLAN Notices Notices 16*, 1 (Jan. 1981), 207-209.