Object Database and Enviroment: The Language and the Data Model

Presented by
Wes Luo

1. INTRO

ODE is a database system and enviroment based on the object paradigm. The database is defined, queried and manipulated using the database programming language O++.

database object vs. conventional PL object:

- database objects persist beyond the lifetime of the program creating them
- capability to create and access multiple versions
- support for active database -- triggered action on condition
- the ability to associate constraints with objects

O++ provides one integrated model for both database and general purpose manipulation

. extended from C++: persistence, versions, associating constraints & triggers with object
. provide iterators on sets/clusters, add “join” query capability
. set iteration also facilitate “recursive query”
. ways to specify iteration subset & order -- useful for optimization

2. DATA STRUCTURE CONSTRUCT

A database is a collection of persistent objects, each identified by a unique identifier
database object id -- a pointer to a persistent object.

2.1 Object Definitions: C++ Class Spcification and Body

2.2 Inheritance, Multiple Inheritance

2.3 Persistent Objects

Principles:

. Persistence should be orthogonal to type
. no run-time penalty for not dealing with persistent object
. Allocation/manipulation of persistent objects should be similar to volatile objects
. Inadvertent fabrication of object ids should be prevented
. Minimum language change

2.4 Dual Poointers

. a pointer that can refer to a volatile or persistent object.
. runtime determine what it points to
. dual pointer <= persistent object pointer / volatile object pointer
  persitent object pointer <= dual pointer which refer to persistent object
  volatile object pointer <= dual pointer which refer to volatile object

2.5 Clusters of Persistent Objects

. All persistent objects of the same type are grouped together into a *cluster*
. cluster name = type name
. cluster must exist before creating corresponding persistent object
. create and destroy macro

2.5.1 Subclusters

. for logical organization or efficiency reasons
. subcluster name = a string qualified by the cluster name
. better than using inheritance mechanism (which need to use many subtypes)

2.6 Sets

. Multisets (elements do not have to be unique)
. set operations: assignment, union, difference, insertion, and deletion
. also iteration facility

3. QUERY PROCESSING CONSTRUCTS

3.1 Iterating Over Sets and Clusters

. for loop with suchthat and by clauses
. can be simulated by if statment within the loop and by sorting set in volatile store
. but it facilitate optimization

3.1.1 Iterating over Cluster Hierarchies ------ forall loop
3.1.2 Arbitray Joins ------ multiple loop variables
3.1.3 Type Test ------ **is** operator

3.2 Fixpoint Queries (???)
just recursive queries?

4. (VERSIONING)
. many database system (CAD, sw mgmt) require capability to create/access multiple versions of an object
. also important for historical databases

. all persistent objects can have versions
. no predefined limit on number of versions
. current version updatable, old versions may be read only
. object property, not class preperty
. an object and all its versions are treated as one logical object with one id

. logical id vs. version pointer
. macros to coercing version pointers to logical id and vice versa (how?)
. macro for creating new version **newversion**
. macro to access different versions **previous**
. **pdelete** delete the object and all its versions

5. CONSTRAINTS

. constraints is the notion to maintain consistency beyond type system
. constraints can be associated with classes
. objects must satisfy all the constraints of the class
. can be used in query optimization

in ODE:
. constraint section in class definition
. constraint is Boolean expression of components (member data)
. constraints are checked at the end of constructor and member(friend) function
. if member data is public, then programmer responsible to enforce constraint

5.1 Constraint Inheritance
. inherit parent class constraints
. added constraints used to specialize classes

6. Triggers
. *once-only* vs. *perpetual*

Design Alternatives, Rationale for Choices Made

Dual Pointers:

Alternatives:
i. Have only one type of pointer that can point to both volatile and persistent object

Pro:
. two less pointer type
. all code will work with both volatile and persistent object
. strictly orthogonal to persistence
. only syntatic addition would be pnew and pdelete

Con:
1. performance over head
   . a run-time check must be made when dereference a pointer
   . avoiding run-time overhead: trap references to persistent object
   . trapping mechanism and resultant actions differ
   . erroneously generated pointer may refers to persistent object
   . no machine-independent way of generating illegal addresses
2. limits the size of pointers to persistent objects (same as ordinary ones)

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

But 3 pointer type is better since:
. better program readability
. compiler has better error checking

Clusters of Persistent Objects

Bloom and Zdonik: using type extents to partition the database.

Disadvantage:
1. Unrelated objects of the same type will be grouped together
2. impossible to optimize queries involving subset of objects in cluster

Solution: Subclustering with inheritance mechanism

Disadvantage:
1. static mechanism
2. inheritance mechanism is being overloaded to create subcluster (??)
3. need large number of types for large number of subclusters

Solution II: subclestering with distinguished qualified name (as in O++)

Query Processing Construct

Alternative Approach:
Define two additional functions in each class definition to iterate over persistent objects:
1. initializing
2. next (step through)

Disadvantage:

1. put the responsibility on programmer to define those functions
2. hard to provide funcionality of by and suchthat clause, let alone optimization
3. lower level iteration interface
4. may not work properly in case of nested iteration on same cluster
5. may not provide functionality of forall
6. O++ then need to provide low-level mechanish for accessing objects in cluster

Related Work:

. E
. Avalon/C++
. Vbase
. O2

A Transformation-Based Approach to Optimizing Loops in Database Programming Languages

I. INTRO

. DB programming language compilers must include database-style optimizations
. optimize set iterations correspond to joins

Source of set iteration:
. value-based join using nested iterators (this paper concentrate on this)
. following pointers within a set iteration
. calling a function within a set iteration

Definitions:
. set loop
. group-by loop
. simple group-by loop
. iterator variables

. associativity and commutativity
. removing ordering constraints and reorder
. transformations: add extra set scan, use temporary set, sort set, and rewrite statements embeded in loop
. source-to-source transformations, tree rewrites

II. RELATED WORK

III. SELF-COMMUTATIVITY

(5) for (X_1 of Set_1) suchthat ($Pred_1(X_1)$)
. . .
for (X_m of Set_m)
suchthat ($Pred_m(X_1, . . ., X_m)$)
S;

. order of iteration in sets are not specified
. Meaning - set of program states can be reached after execution
. actual state may vary from run to run, nondeterministic

The statement S in (5) is **self-commutative** relative to X_1,...X_m if the meaning of
(6) for (X_1 of Set_1; . . . ; X_m of Set_m)
suchthat ($Pred_1(X_1)$ && . . . && $Pred_m(X_1, . . ., X_m)$)
S;
in any starting program state is identical to the meaning (5), and the meaning is a singleton set.

Types of operation that are self-commutative:
. computaton of a sequence of aggregates
. reductions
. insertion/deletion other than those being iterated over
. for more, refer to LIEU91a

IV. REPRESENTATION

AST doesn't fit for set loop optimization:
. no fixed arity
. have to design search strategy
. have to regularly search statement list

Tree representation of set loops:

(9) for (X_1 of Set_1) suchthat ($Pred_{1,1}(X_1)$) {
    $S_{1,1}$; . . .
    for (X_i of Set_i) suchthat
     ($Pred_{i,1}(X_i)$ && $Pred_{i,2}(X_1, . . ., X_i)$) {
        $S_{i,1}$; . . .
        for (X_n of Set_n) suchthat
         ($Pred_{n,1}(X_n)$ && $Pred_{n,2}(X_1, . . ., X_n)$) {
            $S_{n,1}$;
        } . . .
        $S_{i,2}$;
    } . . .
    $S_{1,2}$
}

Tree representation of (9):

. each leaf node represents a set iteration
. contents of the Xi set node

Var = X_i
First = $S_{i,1}$
Second = $S_{i,2}$
StmtsDesc = {*self-comm*, *empty*, *not-self-comm*}
Flow = {*noflow*, *flow-in*, *aggregate*, *innermost*, *io-flow-in*,
flow-out-from-inner}
SetName = Set_i
SelectPred = $Pred_{i,1}(X_i)$

2.1

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



item::item(Name xname, double xwt)
{
  nm = xname;
  wt = xwt;
}
Name item::name()
{
  return nm;
}
double item::weight_kg()
{
  return wt;
}
double item::weight_lbs()
{
  return (wt * 2.205);
}
```

2.2

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

```
class supplier {
  Name nm;
  Addr addr;
public:
  supplier(Name xname, Addr xaddr);
  Name name();
  Addr address();
}

class stockitem: public item, public supplier {
  int consumption;
  int leadtime;
public:
  int qty;
  double price;
  stockitem(Name iname, double iwt, int xqty,
          int xconsumption, double xprice,
          int xleadtime, Name sname, Addr saddr);
  int eoq();
  int reorder_level();
  Name itemname();
  Name suppliername();
  Name name();
};
```

2.3

Persistent storage opertors pnew and pdelete, type qualifier persistent:

```
persistent stock item *psip;
...
psip = pnew stockitem(initial values);
...
pdelete psip;
```

copying from volatile to persistent object and vice versa:

```
*sip = *psip; ... *psip = sip;

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

2.4

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

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

2.5
int create (*type-name*);
int destoy(*type-name*);

create(student::“Cascadilla Hall”);
sp = pnew student(“Jack”)::“Cascadilla Hall”;

2.6
persistent part *partset<MAX>;
real s<100>

userid users<>; /* maximum need not be specified */

3.1
for *i* in *set-or-cluster-or-subcluster*
  [*suchthat-claus*] [*by-clause*] *statement*;
suchthat-claus = suchthat(*e*)
by-clause = by(*e* [, *cmp*])
e must contain loop variable *i*

Example:
```
for p in person
  for c in p->children
    suchthat (c->Profession == Name(“CS”))
    printF(“%s %s\n”, p->name,  c->name);
```

3.1.1
```
for ip in item
  tot_wt += ip->weight_kg;

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

3.1.2

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

3.1.3

```
np = ns = nf = incomep = incomef = 0;
forall p ini person {
   incomep += p->income(); np++;
   if (p is persistent student *)
      { incomes += p->income(); ns++;}
   else if (p is persistent faculty *)
      { incomef += p->income(); nf++;}
```

3.2

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

4.

```
newversion(p)

persistent Employee *e, *ve;
...
for (i=0; i<5; I++)
   if ((ve = previous(e,i)) == NULL)
      error("%s has worked < 5 yrs\n", e->name);
   total += ve->salary;
}
```

Task 1. Describe the database

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

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

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

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

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

```
persistent basepart *bp;

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

Task 3: Compute the total mass and total cost of a part named "mast".

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

forall p in part suchthat (p->Name == Name("mast"))
{
  if p is persistent basepart * { /* part has no component */
    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;
      }
  }
}
```