CSE781 Database Management Systems

Final Report

December 11, 1997

Ozgur Balsoy

Introduction

This final report accompanies an object-oriented database management system developed as a term project. It has three main sections dedicated to database schema maintenance, object data model, and query processing.

Data Structures for a DB Schema

By their nature, object-oriented databases should support class hierarchy. In this project, tree structures are used to implement class hierarchy.

Schema Object

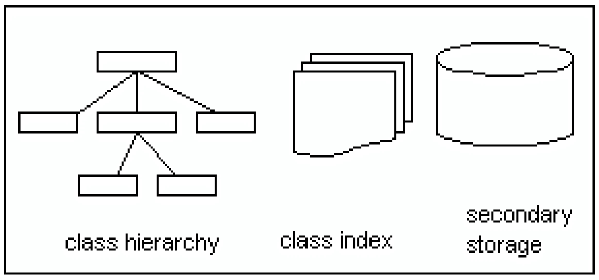


Figure 1

The schema has three major components as illustrated in Fig.1. The first one is a tree of class types which holds all user defined classes and definitions. The second one is an index to ease the class retrieval, and the last one is a reference to a disk object by which the disk storage is performed.

Class Hierarchy

Initially, the schema has a root node, *baseClass*, which serves as the base class in the hierarchy. The schema also provides necessary methods to add and delete classes from the hierarchy, to traverse the tree, to save and reload the schema, and to search for individual classes.

New classes are constructed by allocating a new *SchemaNode* object through the *newClass* method of the schema. If the class has no parents, then the *baseClass* is assigned as the parent class. Otherwise, it is placed in its parent's sub tree, and its links, such as *parent*, *children* and *siblings*, are set to appropriate values. A close look to the hierarchy and its structure is given in Figure 2.

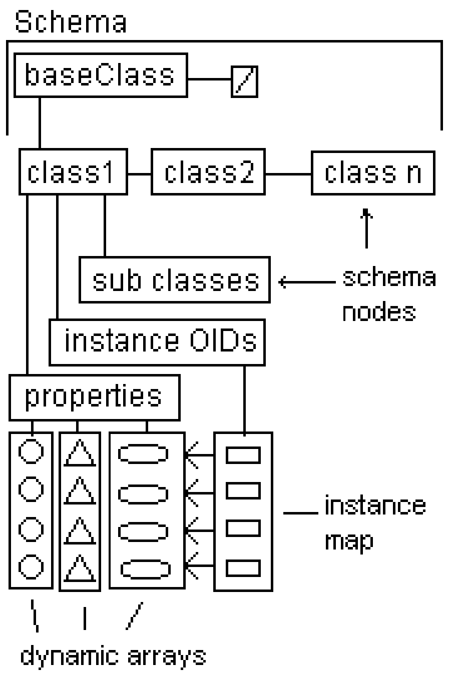


Figure 2

In addition to the class addition, users can remove a class from the schema along with its entire sub tree, and the instances of these classes. Otherwise, all schema evolutions are static in a sense that users have to make changes in the data definition language.

Inheritance among user defined classes arise whenever a user extends a new class from a parent class which is already defined in the system. In this process, first, the parent is search. If it exists, all of its properties are copied to the child class and their inherited flags are set on. Later, the properties of the new class are added. In this way, database objects are collected in leaf nodes. Using this method, upper level classes are prevented from hosting large databases, and database manipulations from evaluating unnecessary objects in queries involving one or two particular classes rather than a class hierarchy.

Class Definitions

Properties (Attributes & Relationships)

Schema nodes have data fields to hold class definitions as well. These are a property list which holds attributes and relations defined by the user, and an instance index, which is mainly an associative array keeping database objects in the memory keyed by their object identifiers [see Objects in Memory Format].

An abstract class, *Property*, is developed to hide differences among attributes (ie. Long, Short, String) and relationships, and to use their similarities. Common fields grouped in this class definition are the property name (i.e. “name”, “address”), the property type (i.e. ptLONG, ptSHORT, ptLIST), the size of the type in terms of memory units, and a flag to hold inheritance information.

Primitive Classes

More specialized, but still abstract versions of Property class are Attribute and Relation classes. All attribute objects corresponding to atomic literals such as long, short, double, etc., are intances of classes derived from Attribute class. Relation class as well is the parental class of relationship classes. Each of these property objects keeps an array of instances of primitive classes. For example, if an attribute “age” in domain “short” is defined, an attribute object is constructed from attr<short>, inserted into the property list in its schema node, and a dynamic array is created to hold all instances of class “short” in the primitive object set, “age.”

Struct, Enum, and Collections

Structures are considered as inner classes, and treated as such. Structure definitions are converted into new class definitions by giving a name made out of its owner class and its type name connected by a period. Later, these new classes are inserted into the class hierarchy at the top level which means that they do not inherit any parent class. The structure attribute, then, is converted into a one-to-one relationship. Whenever an instance of the class type which aggregates a structure is constructed, the system constructs a corresponding structure object, and establishes a relationship to it.
For example, if the following interface is given,

```
interface Students {
      attribute String name;
      attribute Struct Address {
            String street,
            String city,
            String state,
            String zip
      } address;
};
```

the system converts the definitions to the following:

```
interface Students {
      attribute String name;
      relationship      Student.Address
address;
};
interface Students.Address {
      attribute   String street,
      attribute   String city,
      attribute   String state,
      attribute   String zip
};
```

All of these processes are hidden from users, and the system response to structure evaluations as it is supposed to. Using this method, the system saves a lot of bookkeeping memory and time specialized to structures. It eases the queries which structures are involved in such that an access to a structure attribute does not need any different implementation other than class attributes do. The street property can be accessed as a_student.address.street.

Enumerated type sets are considered as static definitions, and stored with the enumerator attribute definitions. Any instance of these types is an unsigned char which is an index to the element in the type set. For example, if an attribute of Professor class is defined Enum Rank {full,associate,assistant}, then for an associate professor, this attribute will have an index value of 1 which costs one byte in the main and secondary storage. However, in case of user interaction, these indexes have to be converted back and forth.

Collections are implemented through the collection module. The interface of the module provides required methods specified by the collection class specifications. Internal implementations of all the collection types are extensive associative arrays which support indexed accesses as well as insertion, deletion, sort, and element distinction for sets and bags.

Relationships

There are three types of relations: lists, sets, and one-to-one. Lists allow indexed accesses and duplicate values. Sets, however, can only have unique values, and does not allow indexed accesses.

In relational properties are stored only object identifiers. Since the identifiers are composed of the object type name and a number, and also the collections can only store the same type objects, the system has an optimization in the implementation of relationships. A relational property keeps only one copy of the target class name, and an array of long typed numbers. Using this method, the numbers can be used as indexes to the instances of the aggregate classes, which makes access times constant. This also simplifies data storage since the identifiers require fixed length memory.

Class Indexing

Object type information, class information stored in the schema nodes, is required in many different cases. For example, when a user defines a sub class of a parent class, all the system knows about the parent class is its name. Although the class hierarchy is stored in tree structure, a search for some class information on this tree is not very efficient because user-defined classes are placed based on parent-children relations rather than alphabetical order of their names.

To reduce the search time of class information retrieval, a map object (an associative array implementation defined in Standard Template Libraries) is used. Each element of the map is a pair of the class name and a pointer to the related schema node sorted according to the class name. Internal implementation of this map is a red-black tree with a time complexity O(log N). This indexing method is illustrated in Figure 3.

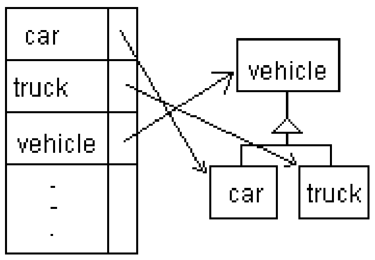


Figure 3

Schema Storage

Users can construct the database schema by editing a DDL text file, and using a more user-friendly language. However, each time this file is needed by the system, first, it has to be parsed; second, it has to be error-freed, and finally the schema can be constructed.

Instead, a schema file is created whenever the DDL file is processed for the first time. On the later references to the database schema, this file is used. It is guaranteed to be error-free, and faster to retrieve into the main memory. The format of the schema file is given in Figure 4.

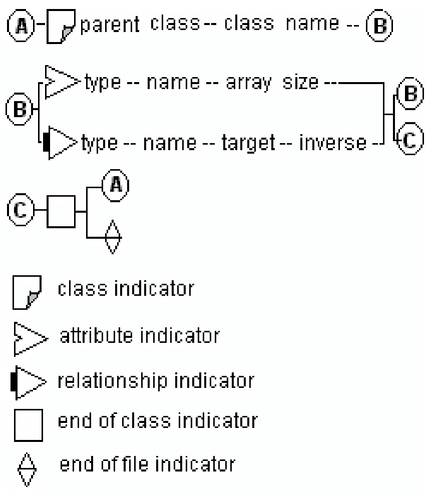


Figure 4

Objects

In-memory format

Each object in the system is defined by an object identifier, and by a pointer to its domain class stored in an object descriptor. This type of representation illustrated in Figure 5 is very similar to Resident Object Descriptors (RODs) in [1]. The attribute instances of an object, then, can be retrieved through its identifier and the pointer.

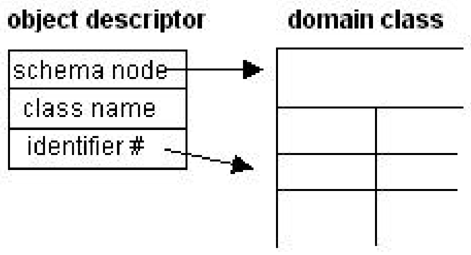


Figure 5

In domain classes (schema nodes), a tuple class C with n attributes is stored in n vectors, one vector for each attribute. In addition, an associative array (i.e. a map) is used to store object identifiers for user-defined objects.

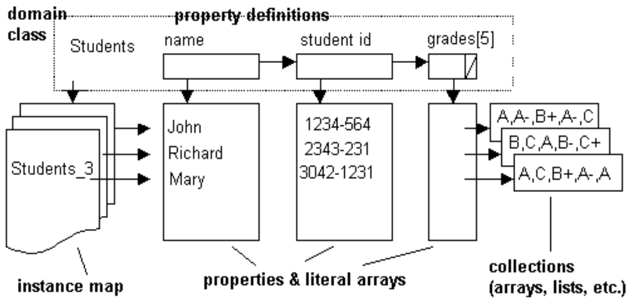


Figure 6

Figure 6 illustrates the memory representation of a domain class along with its instances. As an example, Students class is given. It has three properties, namely "name", "student_id", and "grades" which is defined as an array.

Object Access Mechanism

An object is referenced using its object descriptor. For example, the object descriptor for Mary has Students, 3, and a pointer to the schema node for the class Students in its data fields. Whenever Mary's attribute instances are requested, her OID is verified through the instance map, her record number is obtained, and individual attributes are retrieved using the record number as an vector index.

Disk format

All the database information and database objects are stored in one directory hierarchy. Each domain class is given a directory under the main database directory, and a file called *class.inf* is created in it holding valid and user-defined object identifiers. The object identifier numbers are used as indexes to the other property files, just like they are used in the main memory model. If an object, *class_n* is not created, the nth record stores the identifier number as zero meaning that the records is empty, and that object is not a valid object.

In individual class directories, a file is created for each property defined by the user. If a property is not a collection, then its file is one binary file (file type *pro*) with a fixed record length having attribute instances in its each record. For the example given above, the directory structure would be as in Figure 7. For each collection and string attributes, an accompanying file (file type *dat*) is also created. This file stores variable length data, and the pro file is used as an index file.

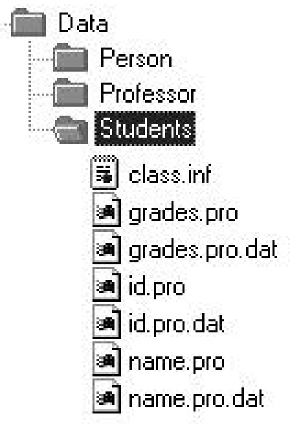
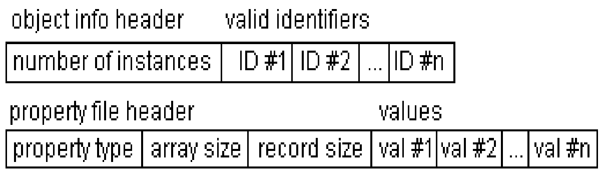

```
Data
  Person
  Professor
  Students
    class.inf
    grades.pro
    grades.pro.dat
    id.pro
    id.pro.dat
    name.pro
    name.pro.dat
```

Figure 7


object info header	valid identifiers			
number of instances	ID #1	ID #2	...	ID #n

property file header			values			
property type	array size	record size	val #1	val #2	...	val #n

Figure 8

Advantages and Disadvantages of the Object Model

Both in the main memory and in the physical storage, objects and attribute instances are stored in vectors. This type of transposed storage model has some advantages and disadvantages [2].

First, it avoids the reorganization of the physical storage. In case of an attribute deletion, only those files or physical pages are removed from the physical storage. Otherwise, each record has to be retrieved, modified, and rewritten onto the disk.

Second, it reduces the page faults in case of a query evaluation. Each object set in the query expression corresponds to a vector of attribute instances. In each iteration, the next attribute instance to test in predicates, or to retrieve in projections is going to be the very next instance in the memory. In traditional systems in which objects are represented record based, the number of attribute instances a memory page can hold is the same as the number of objects a page can hold. This is approximately P/S where P is the page size, and S is the size of a single object. It is obvious that bigger objects do not help query processes. In this system, however, a page can hold as many attribute instances as its size. A

Identifier	Binding	Property	Object set	Iterator	Last element in iteration
S	nil	nil	Students	Students_1	Students_n
T	nil	nil	TA	TA_1	TA_n

Table 1

character object or a enumerator object takes just one byte, and in one page P objects can be evaluated.
Transposed storage can be disadvantageous when retrieving all the attribute values of some instances of a class C. In this case, it is quite possible that each attribute value is going to be in different memory pages or physical pages.

Query Processing

Internal representation

A query expression is converted into internal memory representation after it is parsed, and error-freed. A query is evaluated by initializing a query object which has four components. These are the function, the projection, the range, and the predicates. The related parts of an expression are used to construct each component of the query.

The function component is used to keep the function name, and methods to execute the appropriate function over the query projections. In this project, no function is implemented.

The range component aggregates a list of bindings as well as identifiers. Each identifier is mapped to a binding object, and its scope lives as long as the query evaluation lasts. For example, for a range clause like

(1) "select … from Students as s, TA as t where …",

the following table, Table 1, is constructed.

This table is used to evaluate the natural join defined in the range clause. The evaluation process and the implicit joins are explained later.

The projection component is composed of a list of projection attributes, each attributes corresponding to a user-defined projection attribute. These attributes aggregate binding objects which are explained later.

The predicate part is a tree of expressions. Each node can be either another predicate, a single predicate, a simple comparison expression, boolean expression, a quantification expression, or a cardinality expression. Each expression can be composed of either arithmetic operands and operators, membership operators, etc. Each dotted operand is represented by a DotOperand object which aggregates a binding object.

Bindings

For each dotted path expression in the query statement is defined a binding object. Each binding object has a pointer to the domain class, a list of implicit join objects each composed of a property, an object set, an iterator, and another iterator for the last element in the iteration. For example, consider a query which returns all the titles of the books students have to buy for their courses. The range clause could be defined as,

(2) "select s.name, b.title from Students as s, s.takes as c, c.textbook as b",

then the binding table in the range component is constructed like Table 2.

If the property is not a collection then the iterator and the last element iterator are set to the same instance.
Query Evaluation Cycle

The evaluation cycle is simply the interpretation of an algorithm. This algorithm is summarized as follows:

For all permutations (iterations) in the range clause object sets, if the predicates evaluates true, pick those instances currently available in the projection attributes' bindings.

The following piece of code is mainly the heart of the query evaluation, and exact implementation of the above algorithm.

```
bool ok=range->init();
if(ok)
  do{
    // predicate evaluations
    if(predicates->value())
      projections->pickData();

    // next iteration is generated
    // starting from the right most
```

Identifier	Binding	Property	Object set	iterator	Last element in iteration
S	nil	nil	Students	Students_1	Students_n
C	S (1st row)	takes	Courses	1st course Students_1 takes, say Course_s1	Last course Students_1 takes, say Course_sn
B	C(2nd row)	textbook	Books	1st book taught for Course_s1	Last book taught for Course_s1

Table 2

```
// object set in the range clause
    os = object_sets.rbegin();
    for(; os != object_sets.rend(); os++)
      if ((*os)->iterate()) break;

    // re-initialization of bindings
    // completing the last iteration
    if(os!=object_sets.rend())
      for(init_os=os, init_os--;
       init_os != object_sets.rbegin()-1;
       init_os--)
       (*init_os)->initialize();

    // loop continues until the left most
    // object set in the range clause
    // completes its iteration
  } while( os != object_sets.rend() );
```

The predicate tree is traversed using the postorder method to evaluate the current value. The top node in the tree may have erither no children, only one left child node, or two child nodes. If there is no child at all, the predicate evaluation returns true to assure the semantics of no predicates in the query expression. If there is only the left predicate which means only one predicate is defined, the value of this predicate is returned. In case of two predicates defined meaning that there are at least two or more predicates, and one or more nodes in the predicate tree, both of the predicates are evaluated and result is return after the boolean operation of the results.

Path expressions: Hidden Implicit Joins

In the query samples above in (1) and (2), the natural join operation and implicit join operations are explicitly defined. However, the evaluation of a path expression is an implicit join by itself. For example, the range clause in (2) can be defined as a path expression as follows:

(3) select s.name, s.takes.textbook.title from Students as s;

The evaluation of path expressions is very similar to the methods explained above. However, this time, the binding table has a slight change. Instead of constructing row based, it is constructed column based. The rows for the identifiers C and B in Table 2 become the right most columns of the first row. The evaluation of the join is performed exactly the same way.

Conclusion

This project was quite a challenge for me at this semester. I had not completed any other project as big as this one in the course of my education life before. I had chances to develop other database management systems, but these were usually team works in which I took a part. However, I believe that this was a good experience, and will be a chance to show off.

The project has some weaknesses in terms of implementations of the requirements. I have tried to complete all the requirements, but this was not an easy task. So, there are some requirements missed. These are listed in the README file included in the package.

---

[1] W. Kim, J.F. Garza, N. Ballou, and D. Woelk "Architecture of the ORION Next-Generation Database System", *IEEE Transactions on Knowledge and Data Engineering* vol.2 No.1 March 1990.

[2] L. Al-Jadir, T. Estier, G. Falquet, and M. Leonard, "Evolution Features of the F2 OODBMS", *Advanced Database Research and Development Series* vol.5 April 1995 pp. 284-291.