Supporting Physical Independence in an Object Database Server

Nicola ALOIA[2], Svetlana BARNEVA [1], Fausto RABITTI [1]

[1] IEI-CNR,
Via S. Maria 46, 56126 Pisa, Italy

[2] CNUCE-CNR,
Via S. Maria 36, 56126 Pisa, Italy

Abstract. An approach for supporting physical independence in Object Database Servers is proposed in this paper. While in current implementations the strategy for storing data objects reflects the logical object definitions, a certain degree of independence in the physical database organization would be essential to meet specific performance requirements. In this paper, a canonical object data model and a storage object data model are presented. In the first, the objects are organized in classes; in the second, physical objects with similar structures are grouped in collections. Mechanisms for mapping data structures and operations from the logical level to the physical level are discussed, and a comprehensive example is given.

This work was funded in part by the ESPRIT Basic Research Action Project No. 3070, FIDE (Formally Integrated Data Environment) and in part by Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo of C.N.R.

1. Introduction

The object-oriented technology is rapidly gaining popularity also in the area of database systems. Object Database Systems, ODBSs, are based on a set of common ideas, such as objects with identifiers, classes and class inheritance, etc. These ideas have led to the development of a number of systems, such as IRIS [6], GemStone [4], Orion [7], and O2 [5].

One of the most important features of today's database management systems is the ability to separate the logical view of a database from its physical storage organization. This makes it easier to redefine and restructure the database and is, in fact, the key feature when dealing with performance problems of database systems. This feature is known as *physical data independence*. It means that storage structures and access algorithms can be changed in response to changes in processing requirements without altering existing applications.

Physical data independence is not supported in current ODBSs. In today implementations, the physical database organization is hard-coded in the system, i.e., the strategy for storing data objects is fixed and cannot be changed by the database administrator. The physical object organization usually reflects only the logical object definitions (i.e. the class definitions in the database schema).

In this paper, we will propose an approach for supporting physical independence in an Object Database Server. We will investigate the mechanisms needed to provide the database designer with the capabilities which permit him to choose the most suitable physical organization for an object database in order to meet the performance requirements of particular applications.

Our goal is to support, in an Object Database Server architecture, a physical description of the object database which can be different from the logical description of the same object database, thus introducing tools for performance optimization which act on the physical data organization. It will thus be necessary to give rules for translating data structures and operations from the logical data model level (here called *canonical level*, since the Object Database Server is supposed to interface several object-oriented database programming languages) to structures and operations at the storage/physical data model level (here called *storage level*). Operations for grouping and partitioning canonical objects into physical objects are provided. As result, a collection may correspond to one or more classes (or parts of classes) of the canonical database schema.

The paper is organized as follows. In Section 2, we introduce the object structures in the canonical object data model and the storage object data model. In Section 3, we introduce the operations at canonical level and the operations at storage level. In Section 4, we present the process of mapping object structures from the canonical level to the storage level, giving also a comprehensive example. In Section 5, we discuss the mapping of operations defined in the canonical object data model to corresponding operations defined in the storage object data model. Conclusions are presented in Section 6.

2. Object Data Structures

In this section, we define the object data structures at canonical level and at storage level.

2.1 Canonical level object structures

The canonical data model constitutes the common interface to the Object Database Server. It is supposed to support more than one high level languages (i.e. object-oriented database languages, like Orion [7] and O2 [5], and conceptual programming languages, like Galileo [1]). The language for the description of the data structures in the canonical model is given in Appendix 1 (more details in [2]).

The canonical data model supports flat class definitions, i.e. it is not allowed to use nested definitions of classes. Each class definition consists of two lists: one is the list of its superclasses (i.e. the classes from which it inherits attributes), and the other is the list of the class attributes (each of them can be a basic attribute or a reference to another class). For each class, it is also specified whether its objects are *dependent* or *independent* and *shared* or *not shared*. An object is *independent* if its existence does not depend on the existence of any other object. An object is *dependent* if it can exist in the database only when referenced, also indirectly, by an *independent* object. This is the reason sometimes the deletion of objects to have a cascade effect. The *dependent/independent* specification may be useful in translating *composite* objects, as in [7]. An object is defined as *not shared* if it can be referenced by no more than one object, otherwise it is *shared*. This specification is necessary when determining groups of classes (see Subsection 4.3, in the following). If the object is of a *shared* class, for each reference entering to it, we should specify whether this reference is *shared* or *not shared*, i.e. whether more than one object of the same class may refer to it. The last specification is not important for the translation mechanisms discussed in the present paper, and therefore will not be considered in the examples. All multi-valued attributes of the high level model are represented in the canonical model as *multiple* attributes without distinction between sets, lists, arrays, etc.

In the remainder of the paper, we will always refer to the example database which is shown in Figure 1.

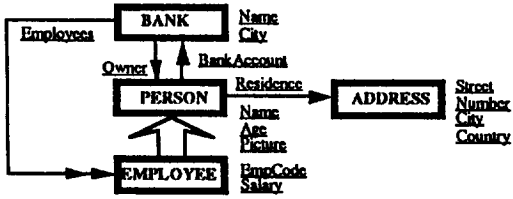

Figure 1. Example database

In this figure, classes are represented by boxes. Note that class Bank is *independent* and *shared*, classes Person and Employee are *independent* and *not-shared*, class Address is *dependent* and *not-shared*. Simple attributes (i.e. attributes whose domain is a basic data type) are listed on the right of each class box. Complex attributes (i.e. attributes whose domain is another class, and therefore describe references between objects in the database) are associated to a single arrow, from a class (i.e. codomain) to another (domain). A two-headed single arrow is associated to multiple references. A double arrow indicates a generalization/specialization link: it goes from the subclass to the superclass.

2.2 Storage data model

Physical objects are grouped in collections and stored in the physical store. There is not an exact correspondence between canonical classes and physical collections. Collections are constructed by grouping and/or partitioning classes in order to accelerate the access to the physical store. This means that a physical object may consist of the attributes of several logical objects and a logical object may be split into two or more physical objects. The description of the storage data model can be found in Appendix 2. A storage schema is obtained from the canonical schema applying, in a defined order, the mapping operations described in Section 4.

3. Object Operations

In this section, we define the operations on objects at canonical level and at storage level.

3.1 Canonical Level Operations

At the canonical level, we have defined three groups of operations. The first group consists of operations for creating and modifying objects of the canonical classes. There are operations for:

- creating a new object in a given canonical class;
- updating an object with a given *oid* or a set of objects which satisfy a given clause;
- deleting an object with a given *oid* or a set of objects which satisfy a given clause.

Depending on the way of giving the values for the attributes, we propose two variants for the operations for creation and modification of objects. The first is to give the values of the new object (or of the object to be updated) directly in the operation. The second allows to refer the area of the memory in which the values are stored, and in this way the execution of the operation is more efficient. For example, let us consider the syntax of the operation for creating new objects:

Make instance of <class-name> : <specifications>;

The item <specifications> may be either a reference to the area of the store in which *the values are stored in the appropriate format, or an expression like*

values(a_1=v_1,...,a_n=v_n). Here, a_i (i=1,...,n) are the names of the attributes, and v_i may be basic values or class names or nested *Make instance* operations. When an attribute a_i is multi-valued, the corresponding v_i may be a sequence of basic values, or class names, or nested *Make instance* operations. Let us consider the following *Make instance* operation (it refers to the example database in Figure 1):

```
oid1 =Make instance of Person
    values (Name='Nicola Aloia',Age=21, Picture=photo1,Residence=
        Make instance of Address values
            (Street='Roma',Number=23,City='Pisa',Country='Italy'));
```

With this operation, an object of class Person will be created. Some of its attributes (Name, Age, Picture) have primitive values. The attribute Residence is more complex: creating its value means creating a new object in the class Address with the given values.

For the operations *Update instance* and *Delete instance* we have to indicate the *oid* of the object that we want to modify or remove. There are also the operations *Update set* and *Delete set* that allow to modification or deletion of all objects that satisfy a given clause (for query clause definition, see the *Find* operation). A deletion of an object may cause cascade deletions of referenced objects of *dependent* classes. For example, if we execute the operation:

```
Delete instance oid1 from Person;
```

the object of class *Address* referred by *oid1* will be also removed from its class because it is *dependent* and *not shared*.

The second group of operations are intended to support the migration of objects from a class to its specialization classes (using the *Extend instance* operation) or generalization classes (using the *Reduce instance* operation). In case of *Extend instance* operation, the values for the new attributes must be supplied. The *Reduce instance* operation, like the *Delete* operation, may cause the recursive deletion of some referenced *dependent* objects.

The last group consists of operations for object access and retrieval. We may either directly indicate the *oids* of the objects that we want to access, or specify a query clause, which determines the set of objects needed. The first operation is called *Retrieve*, while the second is called *Find*.

We discuss briefly here how set-oriented access is defined at the canonical level. The *Find* operation returns identifiers of objects belonging to the given class and its subclasses satisfying the specifications. The specifications (i.e. the query clause) are a combination of basic access operations to the objects in the database. The basic access operations are: Filter operations, Map operations and Set operations. (for more details, see [2].) Consider the following query:

> Find all the Banks with at least one BankAccount owned by a Person with Address having City = "Pisa"

At canonical level, it corresponds to the following *Find* operations.

```
Find  Bank where
(T1 = F  [City = 'Pisa'] Address
T2 = MB  [Residence] T1
R = MF  [BankAccount] T2)
```

In this example, the *F* (i.e. filter) operation identifies a subset of the objects Address satisfying the predicate [City='Pisa']. The *MB* (i.e. map-backward) operation is a navigation operation that identifies objects of other classes following the backward references. Here it identifies the set of objects Person whose Residence is an object (of class Address) of the previous set T1. The *MF* (i.e. map-forward) operation is a

navigation operation that identifies objects of other classes following the forward references. Here it identifies the set of objects Bank corresponding to the BankAccount attribute of an object (of class Person) of the previous set T2.

3.2 Storage Level Operations

The operations at the storage level (see [2] for more details) are similar to the operations on the canonical level but they operate on physical objects and cannot be nested. There are operations for:

- creating and deleting collections;
- creating and modifying physical objects;
- creating and deleting references between physical objects;
- retrieval;
- creating and modifying indexes.

The first group of operations is used for the creation and modification of the storage database schema which is obtained as a result of the mapping from the schema at the canonical level.

The second and third groups include operations for the management of physical objects. Each canonical operation for creation, deletion and modification of canonical objects should be translated in a sequence of storage operations. A low-level transaction mechanism is supported [2]. Thus, each canonical operation is a low-level transaction which consists of one or more storage operations. For example, because of the fragmentation of a canonical class, a *Make instance* operation may be translated in a sequence of operations for creating physical objects and references.

The fourth group consists of two operations. The first (*Get object*) returns a given physical object. A canonical operation *Retrieve* may correspond to a sequence of *Get object* operations. The second operation (*Search*) is set-oriented. This operation returns a list of identifiers of physical objects in the specified collection which satisfy the given specification. The specification, at this level, is a combination of Filter, Map, and Set basic access operations on collections or on temporary sets of physical objects, obtained during this *Search* operation. However, Filter and Map basic access operations are different in the *Search* operation (i.e. at storage level) with respect to the *Find* operation (i.e. at canonical level), since they must contain also the indication of the *physical path* to be used in their execution on the physical store. In fact, there are several possibilities (i.e. *physical access methods*) in performing one of these basic access operations at the storage level. Each possibility differs on the amount of time required in performing the basic access operations and the choice is a matter of optimization of the *Search* operation (taking into account physical storage parameters and statistics associated to the collections and access structures, like indexes).

The definition of Filter and Map basic access operations in the *Search* operation is the following (Set basic operations are the same as in the *Find* operation):

$S_2 = F$ [<expression>] S_1 <F-ph-acc-meth>

where <expression> is defined as in the Filter basic operation at the canonical level, and <F-ph-acc-meth> can be:

scan
: It means that the file containing the collection must be scanned sequentially, and <expression> is evaluated on all the objects of the collection.

index <index-name>
: It means that the index <index-name> is accessed to select a subset of the objects in the collection which satisfy <expression>.

$S_2 = MF$ [<f-path>] S_1 <Map-ph-acc-meth>

$S_2 = MB$ [<b-path>] S_1 <Map-ph-acc-meth>

where <f-path> and <b-path> are defined as in the Map basic operations at the canonical level, and <Map-ph-acc-meth> can be:

join

It means that the navigation is performed computing a *join*, like in relational databases, between the identifiers contained in the referencing objects (i.e. reference attributes) and the identifiers of the referenced objects.

link

It means that the navigation is performed using the physical pointers between the objects. This is always possible for Map forward operations, but for Map backward operations is possible only if backward pointers are supported at physical level.

index <index-name>

It means that the navigation is performed using a special index, identified as <index-name>. whose function is to speed up the navigation between objects in the object store, without physically access the objects in the collections. Several types of these indexes have been proposed in literature, like join indexes [9], path indexes and nested indexes [3], and, more recently, navigation indexes [11].

The last group consists of operations that deal with the management of the index structures.

4. Mapping of Data Structures

The problem to be addressed here is the translation between the two levels of the system architecture. In this section, we present the rules for mapping data structures from the canonical level to the storage level. In the canonical data model, objects are grouped in classes which have a direct correspondence in the conceptual data model. In the storage data model, objects are grouped in collections. A collection may correspond to one or more classes (or parts of classes) of the canonical database schema.

The problem which will be discussed here is how the various generalization hierarchies or the fragmented or grouped canonical classes can be represented at the storage level. In the following, we present the language for the mapping of canonical data structures and then discuss the algorithms for the mapping of canonical operations (in Section 5).

The first step to obtain the storage level is an initial *one-to-one* translation of the class descriptions (which are written in canonical level language) to storage level language. As a result we obtain collections which are in exact correspondence with the classes. The attributes of a class become attributes or attribute-references of the corresponding collection depending on their types: if their types are primitive they become attributes, if their types are classes, they become attribute-references. The list of generalization references of the collection is formed from the list of superclasses of the class by adding the name of the corresponding collection to each of the superclasses.

Let us now see how the description of the storage level schema is obtained from the canonical level schema of the example database in Fig.1.

Step 1: One-to-one transformation

Bank(INDEPENDENT,SHARED)*

(Bank.Name,Bank.City)
(Bank.Owner:Person,Bank.Employees:MULTIPLE Employee*)*
0 0

Person(INDEPENDENT,NOT-SHARED)*
(Person.Name,Person.Age,Person.Picture)
(Person.Residence:Address,Person.BankAccount:Bank*)*
0 0

Employee(INDEPENDENT,NOT-SHARED)*
(Employee.EmpCode,Employee.Salary)
()
(Person:Person) ()*

Address(DEPENDENT,NOT-SHARED)*
(Address.Street,Address.Number,Address.City,Address.Country)
0 0 0

The above descriptions are of the collections corresponding to the canonical class. The names of the collections are obtained by adding the symbol * to the end of the names of the corresponding classes. Note that the Employee* collection contains a generalization reference towards the Person* collection.

4.1 Generalization hierarchy mapping

We support two alternative techniques for the resolution of the generalization hierarchies: the *Schema Division Solution* (SDS) and the *Objects Distribution Solution* (ODS).

In the first solution (*Schema Division Solution* - SDS), there is an exact correspondence between the canonical and the storage level, i.e. there are two collections: one for the objects of the generalization class and one for the objects of the specialization class. This means that the second collection contains only those values which specialize objects of the first collection. In addition, a reference will be generated from each object of the specialization class to the corresponding object in the generalization class (this solution is adopted in the implementation of Galileo). References entering or exiting the various classes will remain unchanged.

In Figure 2, a graphical example of this solution is shown. The rectangles correspond to classes of the canonical database schema and are indicated by A and B while ovals correspond to collections of the storage schema and are indicated by A' and B'. Arrows correspond to references and double arrows to inherited properties.

In the second solution (ODS), we again have two collections: in the first all the properties (both inherited and not) of the objects of the specialization class are stored, whereas in the second, only objects belonging to the superclass are stored (this storage solution is adopted in the implementation of Orion). Each reference entering the generalization class corresponds to two references entering the two collections. A reference entering or exiting the specialization class will enter or exit the corresponding collection. A reference exiting the generalization class corresponds to two references exiting the both collections. No references between the two collections will be created. An example of this situation in given in Figure 3.

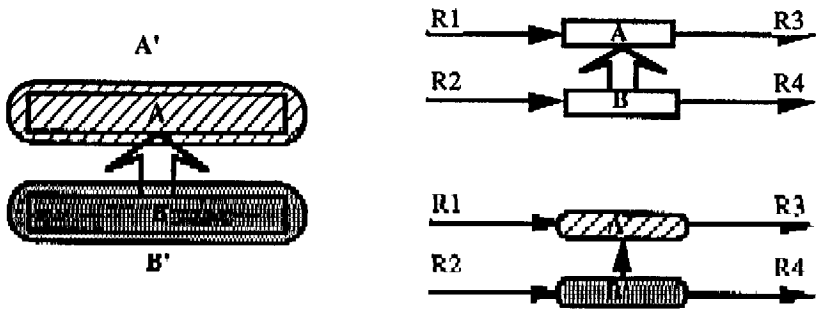

Figure 2 Schema Division Solution

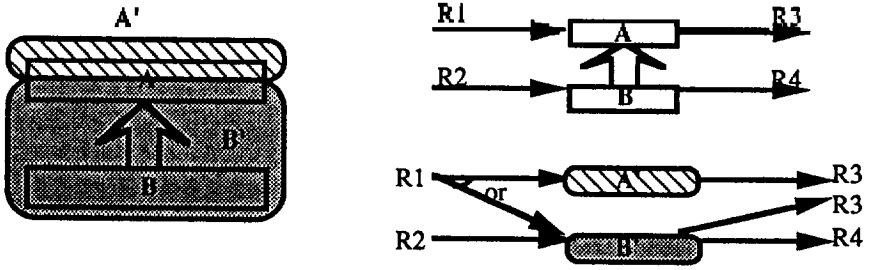

Figure 3 Objects Distribution Solution

4.1.1 Generalization Solution Algorithm

After the initial translation, we obtain a set of collections in which all generalization hierarchies are solved according to the Schema Division Solution (SDS) way. If desired, some of them can be changed, i.e. be stored using the Object Distribution Solution (ODS). To do this, an ODS operation should be performed for each generalization reference. The syntax of the ODS operation is the following:

ODS <gen-coll>, <spec-coll>

where <gen-coll> is included in the list of generalization references of the collection specified by <spec-coll>.

The result of the ODS operation will be that the second collection will also contain all attributes and references, inherited from the first collection, and the name of the first collection will no more be present at the list of generalization references of the second collection. The objects of the specialization collection will be *not shared* if both collections have been *not shared,* and *shared* otherwise. Moreover, all collection definitions will be scanned and each time <gen-coll> is found as an ending of a reference, it will be decomposed into the expression:

<gen-coll> <pred$_1$> OR <spec-coll><pred$_2$>

and the whole ending will be simplified, if necessary. Here, <pred$_1$> and <pred$_2$> are the following predicates:

<pred$_1$> = is not of <spec-coll>,

<pred$_2$> = is of <spec-coll>.

This means that the reference may point to either of the two collections according to whether the object belongs only to the generalization collection or also to the given specialization collection. It is not always possible to evaluate the predicates, especially when a search operation is to be executed. In this case, the object must be searched in both collections. But, when a new object is to be created, these predicates

can be evaluated and the collection into which the object is to be stored can always be determined.

The ODS operation can be applied optionally for each pair formed by a generalization and a specification collection. The order in which this operation is applied in the case of complicated structures of generalization references is important. For example, if we have three collections A, B and C, and B is a specialization collection of A and C is a specialization collection of B, then the results of the following two sequences of operations are different.

ODS A,B;	ODS B,C;
ODS B,C;	ODS A,B;

In both cases, we obtain three collections A, B and C. In the first case, A will have the same definition as before but will contain only those objects that do not belong to B and C. In the definition of B the attributes of A will be added, therefore B will contain the same objects but also with their attributes inherited from A. Finally, in the definition of C the attributes of A and B will added, and it will contain its objects with their attributes inherited from A and B. B will no longer contain a generalization reference to A and C will no longer contain a generalization reference to B. Therefore, in this case A remains unchanged, while B and C contain also all of their inherited attributes. In the second case, the definition of A will be unchanged, the definition of B will contain also the attributes inherited from A, and the definition of C will contain also the attributes inherited from B. The values contained in A will be relative to objects of A and C while the values contained in B will be relative to objects of A and B. C will contain values for its own attributes and for the attributes inherited from B. It will also contain a generalization reference to A. Therefore, different solutions can be obtained according to which one is the most convenient.

Suppose the following transformation is done on the example database:

Step 2 : ODS Person*,Employee*

Only the collection definitions of Bank* and Employee* have been changed in this step.

Bank(INDEPENDENT,SHARED)*
(Bank.Name,Bank.City)
(Bank.Owner:Person(is not an Employee) OR Employee*(is an Employee),*
Bank.Employees:MULTIPLE Employee)*
0 0

Employee(INDEPENDENT,NOT-SHARED)*
(Employee.EmpCode, Employee.Salary,
Person.Name, Person.Age, Person.Picture)
(Person.Residence:Address,Person.BankAccount:Bank*)*
0 0

Note that the reference Bank.Owner of the collection Bank* is addressed to the collection Person* if the pointed object is not an Employee or towards the collection Employee* if the pointed object is an Employee. Note also that the collection Employee* now contains also the attributes (primitives and references) of the collection Person* and it has no more any generalization references.

4.2 Collection partitioning

As stated in the introduction, the object store proposed will support functions to fragment the classes of the canonical level into two or more collections of the storage level. Fragmentation problems have been investigated in dept in distributed database research area. The fragmentation capability is not necessarily restricted to data distribution in computer networks, but can also be useful in other database architectures for efficiency purposes and to supply the capabilities which implement specific access mechanisms for particular portions of data (e.g. image data stored on optical disks) or security devoted mechanisms. The object store proposed supports operations for two kinds of collection fragmentation: *vertical partitioning* and *horizontal partitioning*. In the first case, a collection is split into two or more collections, each of which contains a subset of the properties of the starting collection. In the second case, a collection is split into two or more collections with the same structures each of which contains a subset of the physical objects of the source collection according to a fragmentation predicate. The two operations may be applied repeatedly. Collections which are generated as results of the fragmentation operation form partitions of the starting collection.

4.2.1 Vertical partitioning

When an operation for vertical partitioning is applied, some parts (i.e. some subsets of attributes or attribute-references) of the objects of the starting collection are stored as objects of the newly generated collections and a reference is created from the rest of the object to each of its parts, i.e. to each new object. All newly generated collections are *dependent* on the starting collection and are *not shared*. All references entering the starting collection remain unchanged. An exiting reference will come out of the starting collection or of one of its parts depending on whether it remains in the starting collection or not. Figure 4 shows an example of vertical partitioning.

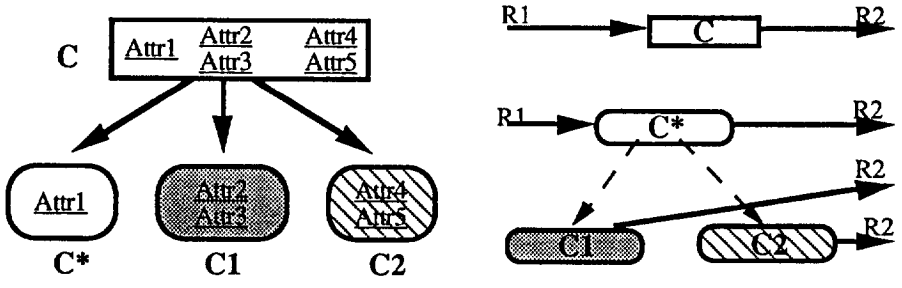

Figure 4 Vertical partitioning

The syntax of the operation for vertical partitioning is the following:

VERTICAL PART<coll-name>:
<coll-name$_1$> (<list-of-attributes$_1$>,<list-of-attr-references$_1$>),
<coll-name$_2$> (<list-of-attributes$_2$>,<list-of-attr-references$_2$>),
. . .
<coll-name$_n$> (<list-of-attributes$_n$>,<list-of-attr-references$_n$>)

The result of this operation is that *n* new collections are created which contain the attributes and attribute-references indicated and are all *dependent* and *not shared*. The definition of the i-th collection will include <attributes$_i$> and<attr-references$_i$>.

The source collection will also be changed. It does not contain any of the attributes and attribute-references cited in the VERTICAL PART operation but these are added to the list of vertical references as references to the corresponding new collections.

Let us apply the following transformation to the storage schema of the example database obtained in the previous step:

Step 3 :VERTICAL PART Person*: Photo(Person,Picture)

The resulting schema (limited to the collections changed) follows:

Person(INDEPENDENT,NOT-SHARED)*
(Person.Name,Person.Age)
(Person.Residence:Address,Person.BankAccount:Bank*)*
()
(Person.Picture:Photo)

Photo(DEPENDENT,NOT-SHARED)
(Person.Picture)
()
0 0

In this step, the definition of the Person* collection has been changed adding a vertical reference and the definition of the new collection Photo has been added.

4.2.2 Horizontal partitioning

When an operation for horizontal partitioning is applied, the collection is divided into a set of new collections with the same structure, such that the objects belonging to each collection satisfy certain conditions, given by predicates. The syntax of this operation is the following:

HORIZONTAL PART <coll-name>:
<coll-name$_1$> <predicate$_1$>
<coll-name$_2$> <predicate$_2$>
. . .
<coll-name$_n$> <predicate$_n$>

The result of this operation is that n new collections are created which have the same definitions as the source collection. Then all collection definitions present in the database are scanned and each time the <coll-name> is found at the ending of a reference, it is decomposed into the expression:

<coll-name$_1$><predicate$_1$>OR ...OR <coll-name$_n$><predicate$_n$>
OR<coll-name><predicate>

where <predicate>=NOT<predicate$_1$> AND ... AND NOT<predicate$_n$> and then the whole ending is then simplified, if necessary. In Figure 5, an example of this operation is given.

Let us consider an horizontal fragmentation on the example database:

Step 4 : HORIZONTAL PART Address*: Pisa (Residence.City='Pisa')

In this step, the definitions of the Person* and Employee* collections have been changed, and the definition has been added of the new collection named Pisa. Note that the definition of the Address* collection has not been changed but some of its objects will no more belong to it since they will be stored in the Pisa collection.

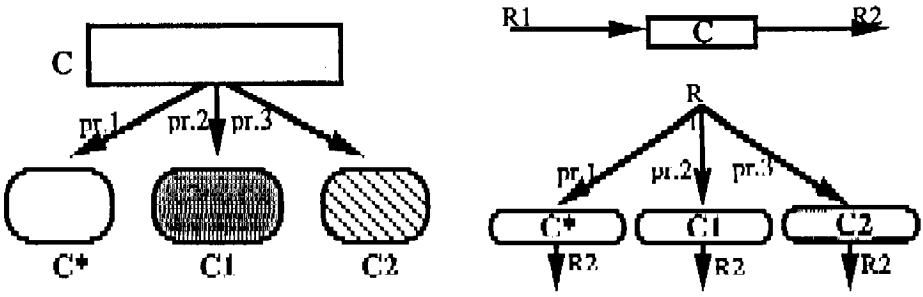

Figure 6 Horizontal partitioning

Person(INDEPENDENT,NOT SHARED)*
(Person.Name,Person.Age)
(Person.Residence:Address(City!='Pisa') OR Pisa(City='Pisa'),*
Person.BankAccount:Bank)*
()
(Person.Picture:Photo)

Employee(INDEPENDENT,NOT-SHARED)*
(Employee.EmpCode, Employee.Salary,
Person.Name,Person.Age,Person.Picture)
(Person.Residence:Address(City!='Pisa') OR Pisa(City='Pisa'),*
Person.BankAccount:Bank)*
0 0

Pisa(DEPENDENT, -SHARED)
(Address.Street,Address.Number,Address.City,Address.Country)
0 0 0

4.3 Collection grouping

The *clustering* mechanism has been long investigated in traditional DBMSs. This mechanism consists in storing, in possibly adjacent positions, data items that are supposed to be frequently accessed together. This mechanism is also proposed for Object Database Systems. What we call *grouping* is a kind of static clustering (i.e. determined at schema definition time). Conceptually, a collection which is obtained by grouping two or more different collections contains objects which have values for all the properties of all the collections that have been grouped. A collection generated by a grouping operation can have objects with different structures. For instance, let us consider the following definition:

$$C = A \oplus B$$

in which C, A and B are collections and $\oplus$ is the grouping operation. Let A and B correspond to the classes A_C and B_C of the canonical model. A physical object of collection C may correspond to one or two different objects, each belonging to one of the canonical classes A_C and B_C. A problem arises here concerning how to understand whether a given object of C represents one or two objects of the canonical schema at a given moment. The problem has already been solved in the clustering techniques of CODASYL DBMS [8] and consists in assigning to each object of the collection a binary code of n bits, where n is the number of the source collections. The i-th bit indicates whether the object contains the values of an object of the i-th collection or not.

In order to make the grouping operation feasible, and to simplify the Storage Manipulation language, we stipulate some restriction rules:

Rule 1. *Two collections can be grouped if and only if there is a reference from one to the other.*

Rule 2. *When defining a group, all collections except the root collection must be dependent and not shared.*

The first rule avoids groups that have no sense from semantic point of view, while the second avoids replications of values and the problem of managing dynamic re-grouping. In this way we simplify the Storage Manipulation language.

After the GROUP operation, all references entering the root collection will enter the grouped collection and all references exiting the collections will exit from the grouped collection (see Figure 6).

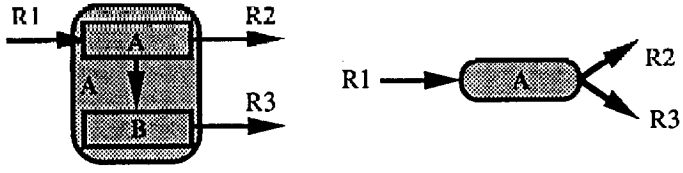

Figure 6 Grouping

Grouping can be performed in two different ways depending on the type of reference which connects the two collections. If it is an attribute-reference, we have to deal with *grouping by attribute-reference*. The syntax of the operation in this case is the following:

GROUP <coll-name$_1$>,<ref-name>,<coll-name$_2$>

The reference <ref-name> must be present in the list of attribute-references of the first collection. It may be a simple reference or part of a reference with a complex ending (e.g. an OR-list). The result of the GROUP operation is that for each of its objects the first collection will also contain the attributes and the references of the referred objects of the second collection. All attributes and references of the second collection will be added to the definition of the first collection, each of them followed by the predicate which specifies the <coll-name$_2$> in the ending and then the whole lists should be simplified. The definition of the second collection remains the same and all of its objects which are not referred by objects of the first collection will be stored in it.

The case when the reference is vertical is called *grouping by vertical reference*. The syntax of the GROUP operation in this case is the following:

GROUP <coll-name$_1$>,<coll-name$_2$>

As in the first case, all attributes and all references of the second collection are added to the first one. The vertical reference between <coll-name$_1$> and <coll-name$_2$> is then removed from the list of vertical references of <coll-name$_1$> in the way described in the previous case. In addition to this, in both cases, if the second collection is not referred by other collections, its definition is cancelled, i.e. it does not exist any more.

In the following, we consider a group transformation on the example database schema obtained in step 4.

Step 5 : GROUP Person*,Person.Residence:Pisa

Person(INDEPENDENT,NOT SHARED)*
(Person.Name,Person.Age,Address.Street(City='Pisa'),
Address.Number(City='Pisa'),Address.City(City='Pisa'),
Address.Country(City='Pisa'))
(Person.Residence:Address(City!='Pisa'),Person.BankAccount:Bank*)*
()
(Person.Picture:Photo)

In this step, only the definition of Person* has been changed. Even though the Person* collection and the Pisa collection have been grouped, the collection Pisa will continue to exist because those of its objects which are referenced by objects of the Employee* collection must be stored in the collection Pisa. The final storage level schema is shown in Figure 7.

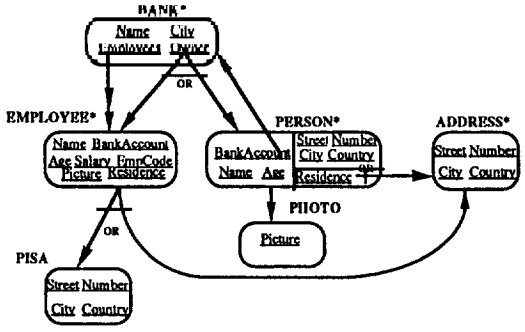

Figure 7 Example of data structure translation

After the structures have been mapped, a *transformation table* must be created to give the correspondence between the canonical classes and the physical collections, i.e. in which collections the attributes of the objects of a given class are stored (including inherited attributes if this class is a specialization class) and which attributes are substituted by the referred objects as a result of GROUP operations. The transformation table can be obtained by scanning all collection descriptions in the storage data model or by analyzing the ODS, VERTICAL PART, HORIZONTAL PART and GROUP operations. It is used in the mapping of the operations on the canonical level.

5. Mapping of Operations

This section addresses the problem of translating the operations from the canonical level to the storage level (a full description of the algorithms is given in [2]).

The mapping of a *Make instance* operation consists of three steps:

- construction, in a temporary buffer, of the sets of basic values of the physical objects corresponding to the canonical object to be created;
- determination of the physical collections of these objects;
- creation of the physical objects and transfer of their values from the temporary buffer into the collections.

For example, the translation of the example given in Subsection 3.1 will be:

pid1=Create object (Person.Name='Nicola Aloia',Person.Age=21,
Address.Street='Roma',Address.Number=23,Address.City=Pisa,
Address.Country=Italy) *of* Person*;
pid11=Create object (Person.Picture) *of* Photo;
Add reference Person.Picture *from pid1 to pid11*;

oid1=pid1;

Besides changes in the basic values, updating a canonical object may cause changes in its structure, i.e. after performing the *Update instance* operation we may obtain a set of physical objects that belong to collections which are completely different from the initial ones. For example, if a horizontal fragmentation has been declared, changing a simple value of an attribute which is involved in the fragmentation predicate may cause a migration of a physical object from one collection to another. This means creating of a new object and deleting of the old object.

Moreover, since some of the values v_i in the list of values of the operation may be nested *Make instance* operations, we have to deal with problems similar to those of the mapping *Make instance*. (e.g. using temporary buffers, etc.). The algorithm for the mapping of the *Update instance* operation consists in performing a *Make instance* operation whose list of values is constructed by merging the values (here value may also mean a nested *Make instance* operation) given in the *Update instance* operation and those of the starting object. The kind of algorithm is used also for the mapping of the *Extend instance* and *Reduce instance* operations.

The mapping of the *Delete instance* operation consists in removing those physical objects which correspond to the given canonical object and all their references. In turn, the deletion of each physical object may cause a cascade deletion of all *dependent* objects referred by it which are not referred by other *independent* objects.

The mapping of the *Retrieve* operation is trivial: for each of the given *oid*s the corresponding physical objects that contain the attributes in the <target-list> should be read from the collections. If there is no <target-list> given in the operation, the entire canonical object (i.e. all corresponding physical objects) will be read. If the <target-list> is present, a considerable amount of data transfer can be avoid when a vertical fragmentation involves the <list-of-oid>.

The translation of the *Find* operation involves the resolution of the clause specified in the <query-spec> part of the command. This implies the use of query optimization techniques in order to reduce the amount of data to be read from the secondary storage. The optimization process can be a very complex problem since is usually influenced by the use of accelerators, i.e. the use of value-based indexes (i.e., secondary key access methods [8]) for Filter operations and navigation indexes (as in [3], [9] and [11]) for Map operations.

In the following, we only show how a *Find* operation, at canonical level, is translated into a *Search* operation, at storage level, referring to the example in Subsection 3.1.

Search Bank* *where*
(T_1 = *F* [City ≠ Null] Person* *index* I-Bank-City
T_2 = *MF* [BankAccount] T_1 *link*
T_3 = *MB* [Residence] Pisa *index* JI-Empl-Residence
T_4 = *MF* [BankAccount] T_3 *link*
R = $T2 \cup T4$)

where *link* means the use of physical pointers stored in the objects, I-Bank-City is a B+-tree index on the values of the attribute City in collection Person*, and JI-Empl-Residence is a join index [9] implementing the backward references from the objects in collection Pisa to the objects in collection Employee*, through reference Residence.

6. Conclusions

We have investigated the problem of *physical data independence* in Object Database Systems, i.e. the possibility of separating the logical schema of a database from its physical storage organization. In traditional database systems, this has been shown to be essential in order to overcome many performance problems, since it means that the physical database organization can be restructured without changing the logical database schema. The introduction of the principle of physical data independence is new in current Object Database Systems, which usually adopt the simplistic approach by which the physical database organization mirrors the logical database organization, it is hard-coded in the system and cannot be changed to increase the performance of the applications.

In this paper, we have defined a canonical object data model and a storage object data model. In the first, which supports several conceptual models, the objects are organized in classes and basic operations can be performed on classes and logical objects. In the second, physical objects with similar structures are grouped in collections, and basic operations can be performed on collection and physical objects. We have focused on the algorithms which map data structures and operations from one level to the other.

We believe that the solutions proposed can improve the effectiveness of Object Database Systems, leading to the design of more flexible and efficient Object Database Servers [10]. In order to evaluate the trade-offs between the increased complexity of the system and the resulting performance gains, a prototype implementations is in progress [2], within the FIDE Project (and its continution $FIDE_2$) which is part of the ESPRIT Basic Research Action.

References

1. Albano A., Cardelli R. and Orsini R. "Galileo: a Strongly Typed, Interactive Conceptual Language." *ACM Transactions on Database Systems*, Vol. 10, No. 2, 1985, pp. 230-260.
2. Aloia N., Barneva S., and Rabitti F. "Supporting Physical Independence in Object Databases." *Technical Report FIDE/90/33, ESPRIT BRA Project No. 3070*, FIDE (Formally Integrated Data Environment), Nov. 1991.
3. Bertino E. and Kim W. "Indexing Techniques for Queries on Nested Objects." IEEE Transactions on Knowledge and Data Engineering, Vol. 1, No. 2, June 1989, pp. 196-214.
4. Bretl R., Maier D., Otis A., Penney J., Schuchard B., Stein J., Williams E. H., and Williams M. "The Gem-Stone Data Management System." *In Object-Oriented Concepts, Databases, and Applications*, edited by Won Kim and Frederick H. Lochovsky, ACM Press Frontier Series, 1989, pp. 283-308.
5. Deux O. et al. "The Story of O2". IEEE Transactions on Knowledge and Data Engineering, Vol. 2, No. 1, March 1990.
6. Fishman D. H. et al. "IRIS: an Object-Oriented DBMS." ACM *Transactions on Office Information Systems*, Vol. 5, No. 1, Jan. 1987.
7. Kim W., Ballou N., Chou H. T., Garza J. F., and Woelk D. "Features of the ORION Object-Oriented Database." In *Object-Oriented Concepts, Databases, and Applications*, edited by Won Kim and Frederick H. Lochovsky, ACM Press Frontier Series, 1989, pp. 251-282.
8. Teorey, T. e Fry J. P. "Design of Database Structures" Prentice-Hall Inc., Englewood Cliffs, New Jersey, 1982
9. Valduriez P. "Join Indices." *ACM Transactions on Database Systems*, Vol. 12, No. 2, June 1987, pp. 218-246.

10. Zezula P. and Rabitti F. "High Performance Object Store." *Technical Report FIDE/90/4, ESPRIT BRA Project No. 3070,* FIDE (Formally Integrated Data Environment), Dec. 1990.
11. Zezula P. and Rabitti F. "Navigation Index for Object Stores." *Technical Report FIDE/90/25, ESPRIT BRA Project No. 3070,* FIDE (Formally Integrated Data Environment), Aug. 1991.

Appendix 1: Language for the Description of the Canonical Data Model Structures

<class> := <class-name>(<class-type>,<shared>)
SUPERCLASSES <list-of-superclasses>;
ATTRIBUTES <list-of-attributes>;
<class-name> := <name>
<class-type> := DEPENDENT | INDEPENDENT
<shared> := SHARED | NOT-SHARED
<list-of-superclasses> := <class-name> | <list-of-superclasses>,<class-name>
<list-of-attributes> := <attribute-def> | <list-of-attributes>,<attribute-def>
<attribute-def> := <attr-name>:<attr-type> | <attr-name>: MULTIPLE <attr-type>
<attr-name> := <name>
<attr-type> = <primitive-type> | <shared> <class-name>
<primitive-type> = integer | real | string | image

Appendix 2: Language for the Description of the Storage Data Model Structures

<collection> := <coll-name>(<coll-type>, <shared>)
<attributes>
<attr-references>
<gen-references>
<vert-references>
<coll-name> := <name>
<coll-type> := DEPENDENT | INDEPENDENT
<shared> := SHARED | NOT-SHARED
<attributes> := () | (<list-of-attributes>)
<list-of-attributes> := <attribute> | <list-of-attributes>,<attribute>
<attribute> := <field> | MULTIPLE <field>
<field> := <class-name>.<attr-name> | <class-name>.<attr-name><predicate>
<attr-references> := () | (<list-of-attr-references>)
<list-of-attr-references> := <ref> | <list-of-attr-references>, <ref>
<ref> := <reference> | MULTIPLE <reference>
<reference> := <class-name>.<attr-name> : <ref-ending>
<ref-ending> := <coll-name> | <OR-list>
<OR-list> := <coll-name> <predicate> | <OR-list> OR <coll-name> <predicate>
<gen-references> := () | (<list-of-gen-references>)
<list-of-gen-references> := <gen-ref> | <list-of-gen-references>,<gen-ref>
<gen-ref> := <list-of-endings>
<list-of-endings> := <ref-ending> | <list-of-endings>,<ref-ending>
<vert-references> := () | (<list-of-vert-references>)
<list-of-vert-references> := <vert-reference> |
<list-of-vert-references>,<vert-reference>
<vert-reference> := (<list-of-ref-names>):<ref-ending>
<list-of-ref-names> := <class-name>.<attr-name> |
<list-of-ref-names>,<class-name>.<attr-name>