# Clustering Strategies in $O_2$: an Overview

Véronique Benzaken
Claude Delobel
Gilbert Harrus
Altaïr
BP105
78153 Le Chesnay Cedex

April 26, 1995

**Abstract**

This paper addresses the problem of clustering complex data on disk to minimize the number of I/O in data intensive applications. It describes the clustering strategies adopted in the $O_2$ system. As clustering depends on both structural aspects (composition hierarchy of the classes) and dynamic aspects (the methods associated with the classes) the paper details a cost model in order to evaluate the benefits of the clustering strategies. This model will permit to automatically derive new clustering strategies. To this end, a derivation algorithm which builds an optimal strategy in linear time is presented. Last, we give a short summary of past and future performance measurements activities.

## 1 Introduction

In the past five years, object-oriented database systems have evolved to what is to become the major commercial database technology of the 1990's, [Ba88], [Ki88]. Whereas the relational model provided the user only with sets of tuples as data modelling primitives, object-oriented database systems provide the database programmer with a wide variety of data modelling primitives (type constructors) including sets, list, tuples, and references between objects. In addition, one can compose these type constructors almost arbitrarily [AB87], [LRV88], [LR89a] [AK89]. Furthermore, object-oriented

database systems almost always provide an inheritance hierarchy for defining types.

The differences between relational database systems and object-oriented database systems is not, however, limited to their corresponding type definition systems. Database accesses in an object-oriented database system are much more complex due to the rich variety of type constructors provided. In particular, in addition to providing relational-like scans of sets or collections of objects, object-oriented database systems foster a navigation-like access among related objects. While such inter-object references would not present a performance problem if the entire database fitted in main memory, with a disk resident database, however, following each such inter-object references may result in random disk accesses.

Without effective clustering, following a reference from one object to another, may involve a disk I/O. Since a typical disk drive can only perform 30 such random accesses per second, the performance of the system may be severely limited if objects that reference one another are not clustered together. *A clustering strategy* is an attempt to reduce the I/O overhead by storing the components of a complex object in the same unit of storage.

Most of the prototypes already developed [ZH87], [KBC87], [At85], [MSOP86], [HK89],
[SPSW90], attempt to perform clustering. In all such systems the segment or the page is the clustering unit and clustering is done when the object is created. The principal goal is to store a complex record as a linearized consecutive string preserving the hierarchical structure. If the string is larger than a page, it will be mapped to a minimum number of contiguous disk pages. The approach adopted in Orion [KBC87] defines the unit of clustering as the composite (exclusive/dependent) object. This is done at the schema level. We think instead that the user should not need to worry about physical issues: the clustering information has to be transparent from an user point of view. Furthermore, if the concept of cluster (via the notion of composite object) is part of the schema, when we want to modify the placement in secondary storage we have to perform a schema update! Data independence is not insured anymore.

In [CK89], it is shown that clustering exploits structural information such as composition hierarchy and inheritance properties. A simulation model has been developed which gives a comparative evaluation of different clustering strategies. The clustering algorithm studied in [CK89] chooses an initial placement for newly created objects which depends on the most frequently used composition links between instances.

The grouping strategies presented in the Encore system [ZH87] seem flexible and powerful (in particular the clustering-by-value feature). In Encore, the following clustering rules may be chosen: (i) ability to store one object per segment if it is large, (ii) ability to store components of a complex object together, (iii) ability to group all the instances of a given class in the same segment, (iv) grouping by value properties (e.g, all blue cars will be stored together). The power of these strategies relies on the fact that, in Encore, objects are shared by means of copies. However, in an environment where update operations are frequently performed, managing such copies seems penalizing with respect to performance.

The approach adopted in the PRIMA project [HMMS87] uses the concept of *type-molecule* as a clustering mechanism for complex objects [SS89]. This concept allows the description of flexible clustering strategies. The main drawback of this approach is that some objects have to be duplicated between several complex objects and that clustering is update dependent.

These strategies only rely on the structure of objects and do not take into account the operations really performed on the database. Design decisions about the clustering are under the control of the Database Administrator. Nevertheless, to help the Database Administrator in taking his decision we have established a link between the access patterns of the methods of a class and the "optimal" placement tree for this class.

Our technique is similar of the one proposed by [Sc77] for the hierarchical data model. The idea of Schkolnick is to partition a hierarchical tree structure into subtrees and to find the best partition according to some access patterns inside the hierarchical structure. Our strategy follows the same idea, but differs in the following: (i) the data model is not the same (objects can be shared, inheritance relationship), (ii) clustering is done at commit time as well as persistency, (iii) clustering is transparent to the programmer, thus placement trees can be modified at any time without affecting programs.

We propose a set of flexible clustering strategies which can evolve when the need arises. We will not try to cluster all the components of an object together; instead, we try to define grouping strategies which take into account both the structural part of the objects (i.e, the composition hierarchy) and the operations performed on the database (i.e, the methods). To this end, we propose a cost model which evaluates the profits of a given strategy, then an algorithm is presented which builds the optimal strategy with respect to the model. Last, we give a short summary of past and future performance measurements activities. Most of the results presented in this paper are part

of three earlier papers ([BD90], [Be90b], [BDH90]). Each of them dealing respectively with: the clustering strategies, the cost model and the design of a clustering benchmark.

## 2 The $O_2$ Clustering Strategies

In the $O_2$ system [LRV88], [LR89a], two main concepts coexist: *values* and *objects*. Values are instances of *types* which specify their structure and are manipulated through predefined primitives. Objects have an identity (internal identifier) and encapsulate values and *methods*. Objects belong to *classes*. Each class has a name and is associated with a type as well as with a set of objects. This set of objects may be referred, at the schema level, as the *extension* of the class. However, some classes may be declared to have no extension. Types are recursively constructed from basic types such as integer and string using set, list and tuple constructors. The following example illustrates class and type construction (notice that the elements of a Menu are identified by a number):

**add class** Restaurant
**type tuple**(`name`: string
`menu`: Menu
`chef`: Chef
`customer`: **set**(Person))

**add class** Person
**type tuple**(`name`: string
`address`:string)

**add class** Menu
**type tuple**(`price`: integer
`courses`: Courses)

**add class** Chef inherits Person
**type tuple**(`specialty`: Courses)

**add class** Courses
**type set**(integer)

Classes are partially ordered according to an inheritance hierarchy. Types and classes may be represented by a directed labeled graph which we call the *Type Inheritance Graph (TIG)*. Such graphs are similar to those described in [KBG89]. Each node denotes a class or a type and is associated with the constructor used (tuple [ ], set { } or list <>). When a node denotes a class it is labeled by the class name. Each edge represents either a composition link, and is labeled by an attribute name in case of tuples or an inheritance link (the dotted edges in the graph). Basic types may be represented in the $TIG$ but as we assumed that atomic values are always stored within the corresponding object we limit our description to the elements which are essential for clustering purposes. The $TIG$ for the Restaurant example is given in Figure 1.

In the $O_2$ system, objects are uniquely identified and are represented by a pair $(i, v)$ where $i$ is an object identifier and $v$ a value. The value of an object may be composed of several other objects and/or values. Values are recursively constructed from atoms (such as integer, string, object identi-

Figure 1: Type Inheritance Graph

fiers, etc) using set, list and tuple constructors. A *named* object or value is an object or value with a user-defined name.

Again, given a set of objects, the links between these objects may be represented by a labeled directed graph. We call this graph a *Composition Object Graph (COG)*. The nodes denote either objects or complex values. These concepts are illustrated in the following example which represents the database state:

($i_1$, [ `name`: "Taillevent"
`menu`: $i_3$
`chef`: $i_4$
`customers`: $\{i_5, i_6\}$ ])
($i_2$, [ `name`: "La tour d'argent"
`menu`: $i_7$
`chef`: $i_8$
`customers`: $\{i_6, i_9\}$ ])
($i_3$, [ `price`: 500
`courses`: $\{i_{10}\}$ ])
($i_4$, [ `name`: "P.Dupond"
`address`: "Rue des dames"
`specialty`: $i_{12}$ ])
($i_5$, [ `name`: "F.Mitterand"
`address`: "Elysee" ])
($i_6$, [ `name`: "J.Chirac"
`address`: "Hotel de ville" ])
($i_7$, [ `price`: 600
`courses`: $i_{11}$ ])
($i_8$, [ `name`: "P.Dupont"
`address`: "Rue des abbesses"
`specialty`: $i_{11}$ ])
($i_9$, [ `name`: "G.Bush"
`address`: "The White House" ])
($i_{10}$, {1, 2, 3, 4, 5, 6 } )
($i_{11}$, {1, 2, 3, 4, 5, 6, 7 )}
($i_{12}$, {1, 2, 3, 4})

The `customers` attribute of object $i_1$ is a set-structured complex value and the attribute `menu` of object $i_1$ is an object. The $COG$ associated with these data is described in Figure 2.

Figure 2: An example of COG

The $COG$ is an abstraction which allows us to represent a complete database as well as any subset of objects of a database. The clustering strategies, which we present next, rely on the structural part of data and on

inheritance. Then the concept of method will be taken into account in order to evaluate the benefits of clustering and to derive, if the need arises, new grouping strategies adapted as well as possible to the processing actually performed on the database.

In the $O_2$ system, pure object identity is maintained. There is no sharing by copy. The clustering strategies are used when new objects are created or when links between objects are updated. At this time, we have to decide whether to create a new cluster for the object or to place it in an already existing one. This is the major reason for which we bind the clustering process to the concept of transaction. Indeed, when an object is brought into main memory during a transaction, its location has been determined by the transaction which created it. On the other hand, newly created objects have an existence in main memory only and it is impossible to state, before the end of the transaction, if they will be persistent. Thus the need to delay as long as possible the physical placement on secondary storage. The clustering algorithm will be invoked at the end of the transaction, i.e, at commit time.

### 2.1 Placement Trees

Intuitively, the type structure of a class in the $O_2$ system may be represented as an infinite tree (in case of recursive types) [AK89]. A placement tree is any finite subtree extracted from this infinite tree. A *clustering strategy* for the system classes is a set of placement trees. Given a clustering strategy, each class in the system is the root of at most one placement tree. We have chosen a finite tree for operational reasons. A placement tree for class $c$ expresses the way in which components of an instance of $c$ are to be clustered. Notice that the definition of a placement tree allows us to handle type recursiveness at the physical level. Figure 3 shows examples of clustering strategies.

In the first alternative for the Restaurant class, the placement tree states that `menu` (which is an instance of class Menu), the type `customers` and each of the "customers" elements will be grouped with the corresponding instance of Restaurant. This means that, in a single cluster, we shall store the "menu" object, its atomic components, the value `customers`, the elements of the corresponding set and the atomic values which compose them. An instance of Courses may be shared by a Chef or a Menu. Therefore, a Course object may either be clustered with a Chef or with a Menu. Unlike the Orion system, such a decision is only taken at run time and is not part of the database schema. The placement tree for the class Person for the second

Figure 3: Placement Trees

alternative reduces to a single node. Indeed, instances of this class are only composed of atoms and we assume that atomic values are stored with the instance to which they are related. Placement trees are the inputs of the clustering algorithm. They only indicate how objects may be stored, they are not part of the schema (there is neither a class placement tree nor instances of placement trees).

### 2.1.1 Placement Trees and Complex Values

The concept of placement tree allows the database administrator (DBA) to describe, for a given class, any sub-tree extracted from the type graph associated with it. Such a tree may contain types (which denote complex values). Assume that the class Person is modified by the following declarations (cf. Figure 4):

**add class** Person
**type tuple**(`parents`: **tuple**(`father`: Man
`mother`: Woman)
`children`: **set**(Person))
**add class** Man **inherits** Person
**add class** Woman **inherits** Person

We can choose the following placement trees for Person: the first one groups

Figure 4: Placement Trees and Complex Values

all the types which compose the class Person, while the second one only groups the value `parents` and `father`. This means that in a same cluster we shall have the instance of Person, the tuple complex value `parents` and the `father` attribute of this value which is an instance of class Man. The `mother` attribute of the tuple value will not be grouped with the instance of Person. Nevertheless, the reference to the value `mother` is stored within the object Person. If we have instead an atomic type (such as string) rather than Man, the attribute `father` will not appear in the placement tree, the atomic value `father` will be stored with the tuple complex value `parents`. A placement tree is thus a tool which can be used in order to describe how complex values may be mapped onto the physical memory.

### 2.1.2 Placement Trees and Recursiveness

Since types may be recursive, there might exist cycles in the *TIG* (such cycles are fully described in [KBG89]). Though a placement tree breaks such cycles, it allows us to manage the depth of recursivity. For the class

Person of Figure 4, we have a composition cycle with the value `children`. We may choose either of the placement trees of Figure 5 for the class Person, where

Figure 5: Placement trees for Person

the cycle is broken at the second level. Notice that in the rightmost placement tree of Figure 5 the children of a given person are grouped in a different way (the value `parents` is grouped with them).

### 2.1.3 Placement Trees and Inheritance

Given two classes, one being a subclass of the other, they may have completely different placement trees. If this is the case, this means that the behavior of their instances (the methods which are actually performed) is different and so there is a real need to adopt different grouping schemes for them. However, if they have the same behavior, we want to force their placement trees to be similar. We thus introduce the concept of *consistent placement tree*. Two placement trees are consistent if the edges present in the first one (i.e the superclass tree) are either replicated or specialized in the second one (i.e, the subclass tree). The concept of consistency is closely related with the subtyping relationship and will have consequences on the relationships between placement trees and file allocation. In Figure 6 the placement tree for the class Chef is consistent with the one of Person in the same figure. On the other hand, if the tree for Chef had only the `specialty` edge, they would no longer be consistent.

In the $TIG$, cycles other than composition cycles may be present. Both composition links and inheritance links compose them, as is the case for

Figure 6: Consistent Placement Trees

the class Person: (Person, `parents`, `father`, Man, Person) is a mixed cycle. Assume then that the placement tree in Figure 6 is defined for the class Person. Again the cycle is broken, however, the placement trees defined for Man and Woman are not forced to be consistent, since this is unnecessary. The definition of placement trees is orthogonal to inheritance. Nevertheless, when a class has no placement tree, we shall use for it the placement tree of its (first selected) direct superclass.

#### 2.1.4 Placement Trees and File Allocation

The clustering algorithm (see Section 2.2) generates logical clusters. This means that some space limitations introduced by the file system features have not been taken into account. Indeed, the size of a cluster is considered unlimited even though it depends on the characteristics of the file system. In the actual implementation of the $O_2$ system, we have to take into account the WiSS (Wisconsin Storage System) [WiSS85] features. Wiss uses files and records. We thus have to map logical clusters to a set of records of the same file.

We shall associate a file $F$ with each placement tree $PT(c)$. A default file $D$ will be reserved for those instances corresponding to classes that do not belong to any placement tree. Thus the mapping between objects of a given class and files is not unique and depends on the following rules:

**$R_1$** If the class $c$ is the root of a placement tree and if $c$ does not belong to any other placement trees then all instances of $c$ will be stored in the file, say $F$, associated with $PT(c)$.

**R**$_2$ If $c$ is the root of $PT(c)$ but also belongs to many other placement trees, then instances of $c$ might be stored either in the file associated with $PT(c)$ or in the files associated with the other placement trees containing $c$.

**R**$_3$ If there is no placement tree for $c$, but $c$ belongs to other placement trees then some instances of $c$ will be stored in the corresponding files. However, it may be the case that some instances of $c$ do not belong to any other objects. We consider, then, the poset of the super classes of $c$. Then the three following rules apply:

- **R**$_{3.1}$ The poset has a unique least element - say $c''$ - then instances of $c$ will be stored in the file associated with $PT(c'')$ (if such a file exists).
- **R**$_{3.2}$ The poset has more than one least element, each being the root of a placement trees. This may happen in case of multiple inheritance. Instances of $c$ will be stored in the file corresponding to the class selected first.
- **R**$_{3.3}$ The poset is empty; in this case, instances of $c$ will be stored in the default file $D$.

**R**$_4$ If class $c$ does not belong to any placement tree, the three subcases of **R**$_3$ apply.

For the first alternative of Figure 3) in the Restaurant example, the file allocation is given by Table 1. All instances of the class Restaurant will be

| *Classes* | F1 | F2 | F3 | F4 | D |
|---|---|---|---|---|---|
| Restaurant | • | | | | |
| Chef | | • | | | |
| Menu | • | | • | | |
| Courses | | • | • | • | |
| Person | • | | | | • |

Table 1: **File Allocation Table**

stored in $F_1$. On the other hand, instances of the class Person which are not components of a Restaurant instance will be stored in $D$. These rules

are particularily useful in the following case. When a class $c$ is declared to have an extension, the system automatically generates a value $\mathbf{set}(c)$ where each element is an object identifier. It is thus possible to retrieve all the instances of class $c$ scanning this set but it would be also possible to scan the corresponding files. Such a decision is left to the optimizer, which has to choose the best strategy. For the classes which are not declared to possess an extension, it is impossible to know where their instances are stored. Thus, with the tables generated by the file allocation rules, it is possible to scan the files in which those instances may be stored.

As we state above, we associate a file with each placement tree. However, this mapping is not one to one. Indeed, if placement trees corresponding to subclasses of a given class are consistent they are assigned to the same file. If one of the subclasses has a placement tree which is not consistent, it is assigned its own file.

Now, we present our clustering algorithm.

## 2.2 A Clustering Algorithm

The clustering algorithm operates on a set of objects which consists of the objects newly created by the transaction (which have a temporary identifier) and those objects which have been accessed by the transaction (which have a persistent identifier). This set is described by a $COG$. This $COG$ is a subgraph of the $COG$ which describes the whole database and represents the objects which must persist at the end of the transaction. Objects are placed in logical clusters using placement trees associated with the class of which they are instances and according to their identifiers (disk or memory). A logical cluster is composed of a root object and a set of objects which are grouped with it. We assume that the size of a logical cluster is unlimited. The algorithm presented here is a greedy tree pattern-matching algorithm. There exist different alternatives for this algorithm based on either depth first or breadth first traversals of placement trees.

**ALGORITHM**

INPUT : The $COG$ corresponding to the transaction being executed and the list of the objects manipulated by the transaction.

1) While (not end of list): Select an object o in the list if it is not marked.

**a)** If there is a placement tree associated with the class $c$ of this object or with a superclass of $c$ (if there is no PT for $c$) then SET = succ(o, PT($c$)); /* succ(o, PT) is the set of objects o' such that there exist a path from o to o' in PT($c$) and either o' is a memory identifier or o' is a disk identifier and is in the cluster of o (each object of the path is in succ(o, PT($c$)). */

**a.1** If o has never been stored on disk create a logical cluster; Group each element of SET;

**a.2** If o is a root object open the old logical cluster; group all elements of SET;

**b)** mark o; mark each element of SET;

End While

2) When the end of list is reached, all the objects which are not marked are grouped in a default cluster.

OUTPUT : A set of logical clusters.

We give an illustration of the clustering algorithm for the classes Restaurant, Menu and Chef. We consider the first alternative of Figure 3 for PT's and assume that all objects are new. The set of all generated clusters is

Figure 7: Example

a partition of the $COG$ nodes, and each logical cluster is a connected subgraph of the $COG$. From the point of view of localization, this algorithm is non deterministic. Localization depends on the order in which objects were selected from the list. For this example, we have chosen instances of Restaurant first. Thus the algorithm generates five clusters (cf Figure 7). If we

had selected Courses instances first, and then Menu, Chef and Restaurant, we would have had nine clusters: three for the "courses" and two for each of "menu", "chef" and "restaurant". Our objective is to minimize the I/O overhead; as such, given a grouping strategy, it is desirable to minimize the number of clusters produced by the clustering algorithm. Indeed, we assume that the exchange granularity between disk and main memory is the cluster. Under this assumption, it is possible to order the classes in such a way that we obtain as few clusters as possible. Such an order is described in [Be90a], and relies upon the existence of placement trees and on the presence of the given class in other placement trees. The objects are selected according to this order. To group as many objects as possible, we have chosen to select those objects which were already stored on disk and to group some of their newly created components with them. Assume that object $i_1$ has already been placed on disk and that object $i_5$ has been created during this transaction, we shall cluster $i_5$ with $i_1$. Objects already existing on disk are not re-clustered except in the case mentioned in [BD90].

The clustering algorithm is actually implemented in the $O_2$ system. However, we need to define what constitutes good placement trees. Thus the need arises to define a cost model in order to evaluate the benefits of our clustering strategies. This model must take into account the treatments performed on the database. We present in the next section our cost model.

# 3 Self-Adaptative Clustering Strategies

## 3.1 Problem Statement

The clustering problem consists in finding a partition of the set of the objects in the database. Our approach is similar of the one adopted in [Sc77] for the hierachical model. When objects are grouped together, it is desirable that these objects be accessed together in the future. We have an operational approach, our model relies on the following information: the type graph of the system, a clustering strategy (a set of placement trees) and the set of methods associated with the classes. Given this information, we will evaluate the benefits of our clustering strategy and show how to dynamically adapt it. First, we have to define what is a good placement tree.

### 3.2 Definition of Good Placement Trees

We rely on the following assumptions. Given a class $c$ and its associated methods (inherited or not), the following two criteria will be taken into account in order to define good placement trees:

- When a method is applied to an object (instance of $c$) and the object has been brought into main memory, in one disk access, all the objects components accessed by the method must also be brought in the same disk access. There is no object fault. This means that in a logical cluster (created according to $PT(c)$) all the objects the method needs to access are present.

- All objects that have been brought into main memory, in one disk access, should be used. Indeed, we do not want to waste main memory space.

Notice that the two criteria might be contradictory. Assume, for example, that a class is built from ten other component classes, and that there are two methods for this class; the first one accessing two components and the second one accessing the ten components. In order to fulfill the first criterion, we would be tempted to cluster the ten components together. But, if we now assume that in 90% of the cases it is the first method that is applied, then the second criterion is not fulfilled. We thus have to take into consideration the activation frequencies of methods. In the next subsection, we characterize the operations which can be performed in a method. We shall use them to derive an evaluation model for our clustering strategies. Finally, we shall use this evaluation to derive optimal strategies.

#### 3.2.1 Syntactical Characterization of Operations

The problem we face is to get some information for a given class and its associated methods which describes the paths the methods follow in the graph which represent the type structure of this class. Given a class $c$, we shall characterize any of its methods $m$ by means of expressions representing which edges in the graph describing class $c$ it needs to follow.

Any $O_2$ method is defined by its signature and its body. The signature describes the parameters of the method, while the body is written in $CO_2$ source code. We are only interested in the structural part of class $c$ which method $m$ needs to access. Thus we shall transform any method body into

a regular expression. We call this transformation a *characterization* and we denote it by $\mathcal{C}$. The alphabet is composed of the labels of each edge in the graph denoting class $c$. The connectors are "." which stands for edge concatenation, and "+ " which stands for conjunction. We assume that each edge in the type graph is uniquely labeled either by an attribute name (in case of tuples) or by a unique arbitrary label — say $\ell_{x,y}$ — (in case of sets and lists, where $x$ denotes the set/list structured type $x = set(y)$ or $x = list(y)$). The following example illustrates such a characterization. Suppose we have for the class Restaurant the following method (written in $CO_2$ source code) which returns for a given Restaurant its Chef's specialties:

```
add method chef–specialty in class Restaurant
      body Courses
            in class Restaurant co2
{
      o2 Courses result;
      result = self → chef → specialty;
      return(result)
}
```

The characterization of chef-specialty is: $\mathcal{C}$(chef–specialty) ≡ `chef.specialty`. In this example the characterization expresses the path followed by the method. Not only is it possible to describe simple paths but also paths where set-iterations may be described. Given such characterizations, we have an access-pattern of the class.

A placement tree which is a good candidate with respect to the first criterion of Section 3.2, is a tree which provides a minimal covering of the set of edges covered by the methods. We call it a *Minimal Covering Tree* and denote it $PT^0()$.

### 3.3 Simplifying Assumptions

- We assume that there is no space problem. In other words, given a placement tree $PT(c)$, for the class $c$, all the instances of $c$ will be stored on the disk according to this placement tree. Thus all the components which should be stored in the cluster are actually stored in it.

- We suppose that when a method has to access a component $o'$ of an object $o$, it must access each of the components of the path $o, o_1, ..., o_i, ..., o'$

(i.e., we do not take into consideration index).

- When we characterize a method, we do not take into account the fact that it might have parameters. We thus assume that the set of methods associated with a class reflects the current usage-pattern of the class.

These simplifying assumptions are not too restrictive and they will permit us to focus on only one class at a time and its associated methods. We next give details about the cost function associated with a placement tree.

### 3.4 Cost Function

This section defines the cost function of a placement tree PT($c$) for $c$. Let us consider the following notation:

- $m_1, m_2, \ldots, m_l$ are the methods associated with $c$.
- $p_1, \ldots, p_l$ are their associated frequencies (the number of times they are called).
- $t_1, \ldots, t_n, \ldots, t_p$ are the types which compose $c$; $c, t_1, \ldots, t_n$ are only accessed by the methods $m_i$ associated with $c$.
- $cost(i,j)$ is the cost associated with method $m_i$ and type $t_j$.
  - $cost(i,j) = \alpha$ if, for the method $m_i$, component $o(t_j)$ has been brought into main memory and then used. $\alpha$ is a profit and thus $\alpha > 0$.
  - $cost(i,j) = \beta$ if, for the method $m_i$, component $o(t_j)$ has been brought into main memory and has not been used. In this case there is waste of main memory space. $\beta$ is a loss and thus $\beta < 0$.
  - $cost(i,j) = \gamma$ if, for the method $m_i$, component $o(t_j)$ has not been brought into main memory and was not needed. $\gamma$ is a profit and thus $\gamma \geq 0$.
  - $cost(i,j) = \delta$ if, for method $m_i$, component $o(t_j)$ has not been brought and was needed. There is, in this case, an object fault. $\delta$ is a loss and thus $\delta < 0$.

Let $PT(c) = (V, E)$, be the placement tree associated with $c$. Let us consider the matrix

$$M_{l,1} = (p_1, p_2, \ldots, p_l)\,(cost(i,j))$$

- If $t_j \in V$ and if $m_i$ uses $t_j$ then $cost(i,j) = \alpha$.
- If $t_j \in V$ and if $m_i$ does not use $t_j$ then $cost(i,j) = \beta$.
- If $\neg(t_j \in V)$ and if $m_i$ does not use $t_j$ then $cost(i,j) = \gamma$.
- If $\neg(t_j \in V)$ and if $m_i$ uses $t_j$ then $cost(i,j) = \delta$.

The cost function $F(PT(c))$ associated with $PT(c)$ will be obtained by adding the coefficients of matrix $M_{l,1}$. It will reflect the object faults and the main memory overloading.

$$F(PT(c)) = \sum_{i=1}^{i=l} \sum_{j=1}^{j=n} cost(i,j) p_i$$

$$F(PT(c)) = \alpha \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \alpha) + \beta \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \beta) +$$

$$\gamma \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \gamma) + \delta \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \delta)$$

where $Nbr(j; cost(i,j) = \aleph)$ denotes the number of times $cost(i,j)$ is equal to $\aleph$. Our purpose is to find the placement tree which maximize the cost function (i.e, which gives the greater profit). We posit:

$$USED = \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \alpha)$$

$USED$ corresponds to the number of objects used for all the methods of $c$.

$$WASTE = \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \beta)$$

$WASTE$ corresponds to the number of objects uneeded and brought into main memory.

$$UN_USED = \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \gamma)$$

$UN_USED$ corresponds to the number of objects which have neither been brought into main memory nor used.

$$DEFAULT = \sum_{i=1}^{i=l} p_i Nbr\,(j; cost(i,j) = \delta)$$

$DEFAULT$ corresponds to the number of object faults. Assume that with this placement tree, $k$ methods among $l$ lead to an object fault. Let $p_{i1}, \ldots, p_{ik}$ be their associated frequencies. If $k$ methods among $l$ lead to an object fault, this means that the associated tree was not a minimal covering. It is interesting to know whether the minimal covering tree $PT^0(c)$ (which eliminates object faults) should be better (has a greater cost function) than a non covering one. We thus now compare the costs of $PT(c)$ and $PT^0(c)$. Objects which were *UN-USED* are loaded and thus associated the loss of $WASTE$ objects. Objects which were $DEFAULT$ are brought and are associated the profit of *USED* objects. Thus the cost function associated with $PT^0(c)$ is:

$$F(PT^0(c)) = \alpha\,(USED + DEFAULT) +$$

$$\beta\,(WASTE + UN_USED)$$

In order to determine which of $PT(c)$ and $PT^0(c)$ is the better candidate, we compute the difference of their associated cost functions.

$$F(PT(c)) - F(PT^0(c)) = (\delta - \alpha)DEFAULT + (\gamma - \beta)WASTE$$

$PT^0(c)$ is the better candidate if and only if:

$$(\delta - \alpha)DEFAULT + (\gamma - \beta)WASTE \leq 0$$

$$\frac{DEFAULT}{WASTE} \geq \frac{(\gamma - \beta)}{(\alpha - \delta)}$$

$DEFAULT$ and $WASTE$ depend on the call frequencies of the methods associated with $c$. This inequation indicates that, if object faults correspond to methods with a low frequency, it is better to use a placement tree which is not covering and which will permit a gain of space in main memory. The coefficients $\alpha, \beta, \gamma, \delta$ depend on the characteristics of the system and will be fixed in time. However, if the penalization ($\delta$) associated with object fault is the only one to vary, we obtain the curve in Figure 8:

Figure 8: Cost Function

The cost function described in this section does not take sets into account. Indeed, when a method accesses a set structured type, we don't know how many elements of the set actually will be used by the method. In our first approach, we considered that either all the elements were brought in one disk access (if the placement tree allowed it) or none of them. In all cases, only one object was taken into account in the computation of the cost function. This does not correctly reflect reality. On the other hand, it might not be correct to consider that a method will always access all the components of a requested set. A refined version of the cost function which takes sets into account is described in [Be90b]. Nevertheless, we are able to introduce the automatic derivation of placement trees without any lack of generality.

### 3.5 Automatic Derivation of Placement Trees

The objective of this section is to use the cost analysis in order to derive automatically an optimal placement tree for a class — say $c$. The naive method, which consists in building all the possible trees and computing their cost functions and then selecting the best, is not tractable. Indeed, such a construction is exponential with respect to the number of edges covered by the methods. We instead propose a linear algorithm with respect to the number of edges covered by the set of methods associated with $c$. This algorithm derives an optimal tree based on the following assumptions: the set of methods associated with class $c$ is fixed, and their frequencies are also fixed. We assume that there exists $PT(c)$.

Let $F_\tau(c)$ be the cost function associated with $PT(c)$ at time $\tau$. Assume that at time $\tau + d\tau$ we add the edge $a$. Let $\mathcal{M}_a$ be the set of methods which use edge $a$. We have:

$$F_{\tau+d\tau}(c) = F_\tau(c) + \Delta$$

with

$$\Delta = \alpha\Delta_\alpha + \beta\Delta_\beta + \gamma\Delta_\gamma + \delta\Delta_\delta$$

$$\Delta_\alpha = -\Delta_\delta = \sum_{i \in \mathcal{M}_a} p_i$$

Methods which before adding $a$ caused an object fault for the edge $a$ will not cause this fault again.

$$\Delta_\beta = -\Delta_\gamma = \sum_{i \neg\in \mathcal{M}_a} p_i$$

Methods which were not using edge $a$, now cause a main memory overload. It is crucial to notice that the fact of adding an edge has no retroactive effect on the initial value of the cost function $F_\tau(c)$. This guarantees that there are no local maxima or minima.

### 3.5.1 A Derivation Algorithm

The algorithm we now present builds an optimal placement tree from the initial placement tree which reduces to a single node, by adding an edge, step by step, if this improves the cost function.

**INPUT**: The set of edges covered by the set of methods. The frequencies associated with each edge.

DERIVATION

**while** (not empty set) **do**

extract an edge $a$;

compute $\Delta$;

**if** $(\Delta > 0)$ **then**

$PT(c) = PT(c) + a$;

**end while**

END-DERIVATION

**OUTPUT**: A placement tree.

**Property 1** Algorithm DERIVATION builds an optimal placement tree in linear time.

The proof of Property 1 above is given in [Be90a].

# 4 Performance Measurements

As seen in Section 3.3, the cost function described in Section 3 and the algorithm for deriving an optimal placement tree are obtained under the simplifying assumption that there is no space problem, i.e., if a cluster fits in a physical page. Moreover, this model does not take into account sharing of components between two or more objects. In practice, if these two conditions are not met, the actual benefits of the clustering strategies might differ from the expected. So it was important to make a study on the effect of the size of objects on the number of disk accesses in the actual system (see Section 4.1 and [BD90]).

Another argument for the studies outlined in this section is the effect of the choice of a placement tree, optimizing a traversal of a composition graph, following particular edges, on other kinds of queries. The experiments we designed in order to study this problem led us to the very first version of a clustering benchmark, CluB-0 (see Section 4.2).

## 4.1 Early Measurements

The complete description of the measurements outlined in this section can be found in [Be90a]. The main goals of these measurements were:

(i) a first partial validation of the cost model: if the assumptions made are met, the best theoretical placement tree is expected to be also the best in the actual implementation of the $O_2$ system!

(ii) the study of the effect of parameters neglected in the model, mainly the size of objects or the relationship between *logical* cluster and *physical* page.

(iii) a first study of the behaviour of the clustering algorithms in the actual implementation, in order to get information for a second, more precise theoretical model.

We decided to make these first experiments simple in the sense that the number of parameters has to be small enough for the results to be easily understable. Here is a sketch of the experiments and an abstract of the results and conclusions.

The class we studied is in fact a simplification of the class `Restaurant` described in previous sections (see Figure 9).

Figure 9: Type Inheritance Graph for the basic measurements.

A number of parameters can be studied:

- the placement tree: none (no clustering) or one of the eight non empty placement trees,
- the number of objects in each class,
- the size of objects of each class,
- the sharing degree of objects (only objects of class C4 can be shared)
- the calling frequencies for the methods.

For the last parameter, we chose the worst case, i.e., equal frequencies for all methods. For a number of values for these parameters, the number of disk accesses made during the execution of the set of methods was recorded. The main conclusions of the study are:

- if the assumptions are satisfied, mainly if the size of a logical cluster is less than the size of a physical page, the theoretically optimal placement tree is actually optimal. It saves from 30% to 50% on the number of disk accesses (except maybe for one sub-optimal tree).
- as soon as the cluster derived from the optimal placement tree is larger than the size of a physical page, clusters smaller than this size give the best results (up to 40%).

Even on the study of this simple class, we can see that the cost model has to be modified to take into account the size of objects.

### 4.2 From Hypermodel to CluB-0

After these basic measurements, we felt unsatisfied as we did not catch in the results all the power of the clustering strategies in $O_2$. For example the ability to cluster along two or more different relationships has not been completly explored. Moreover, a placement tree can be optimal for a certain kind of query on complex composition graph, but very bad for other kinds of query. Last, we think that comparing clustering strategies and algorithms require the definition of a common platform. We describe here what we think is a good basis for such a benchmark.

The Hypermodel Benchmark ([ABMPS90]), also called the Tektronix Benchmark, was a good candidate for our goals. This benchmark presents a fairly complex composition graph, with a wide range of operations. However, we did not want to reuse exactly the same composition graph, as we wanted to have a total control on a small number of parameters. Moreover, a number of operations of the Hypermodel Benchmark does not directly concern clustering in $O_2$. For example operations such as *rangeLookups*, i.e., selections, are not part of the first version of our benchmark.

The structure of the composition graph we choose for the CluB-0 benchmark is as follows. Only one class `N` is needed. The *TIG* for the class `N` is shown on Figure 10. The database consists of objects of class `N`, called nodes, forming a tree (relationship $T$) with a constant fanout $f$, overlapping with a graph (relationship $G$) where each internal node of $T$, say a node at depth $i$, is connected to exactly $f$ distinct nodes randomly chosen from the set of nodes at depth $i+1$ (each branch in the tree has the same depth). Of course, at level 0, the two kinds of links are the same. At any other level, objects can be shared inside the graph induced by $G$.

On Figure 11 is shown an instance of a database (solid lines represent the $T$ relationship, dotted lines the $G$ relationship). The value of $f$ is 5 in the benchmark description, 3 in Figure 11.

Figure 10: The Type Inheritance Graph for the class `N`.

Figure 11: A four levels composition graph.

Attached to each node are some more informations, mainly an unique

identifier, the lists of sons for each one of the two relationships $T$ and $G$, and a variable length string. The latter allows to make measurements on various configurations without changing physical parameters such as the page size of the internal buffers, i.e. the underlying parameter is the mean number of objects per physical page.

For this study, we are interested only in queries whose performance can be modified by clustering decisions. So not all of the requests described in [ABMPS90] were retained. Queries that will benefit from good clustering decisions are for example *closure traversals*, e.g., choose a node at a given level of the tree and construct the list of the subtree or subgraph that can be attained from this node (depth-first traversal). Some other queries are of interest, such as scans, e.g. read the unique identifier of all nodes, traversing the tree in a breadth-first manner.

The complete study consists in generating a number of databases obtained for various values of the parameters, e.g., mean number of objects per physical page, and for various placement trees (spanning over up to two levels of the hierarchies). For each of these databases, for the operations described above, the complete trace of accessed objects is used to derive some indices of performance such as the total number of pages read during the operation, the *switch rate*, i.e. the proportion of accesses to objects that are made in the same page than the previous access.

The main phenomena observed are:

- For a given operation (e.g., *closure traversals*), the best placement tree leads to 75% less object faults than without any clustering. However, with the same tree and another operation, we loose 50% of the benefits we previously had with the best tree (i.e., if we had 10 disk operations we now have 15 disk I/O's). Nevertheless, clustering has a real benefic impact on the performance when it corresponds to the operations actually performed against the database.

- Object sharing has a crucial importance with respect to clustering performance. Such performance may be severely altered for the same tree even if few objects are shared (for the same tree, when objects are shared, we observed that the number of disk operations increases in the proportion of 30%, the probability for an object to be shared is, in this case, of 0.26). However, the tree which is expected to be the best candidate for the corresponding operation still remains the best.

The complete benchmark and results are described in [BDH90].

# 5 Conclusion

The concept of placement trees which we have proposed in this paper is simple and powerful enough to define clustering strategies. These strategies can evolve over time when the need arises. Our work differs from the Orion or PRIMA approach in the following:

- We rely on the specificity of the $O_2$ data model which incorporates the notions of objects and values.

- Our strategies are not part of the schema.

The clustering algorithm allows us to group objects according to a given strategy. If this strategy changes, the algorithm will continue to run according to the new strategy without penalizing the global performance of the system. The algorithm is actually implemented in the $O_2$ system.

The measurements performed in the system, allow us to select an optimal placement tree under certain hypotheses. This tree leads to significatively fewer object faults. However, when the hypotheses are violated, the results are quite different. Nevertheless, we are able to provide the DBA with a tool in order to help him to choose adequate placement trees. In future work, we shall design another cost model which will take into account the sizes of objects.

In this paper, we do not take into account the relationships between clustering and concurrency control. As it is desirable to group objects (to have a means to do it) it would be desirable too to have the ability to ungroup objects (in case of highly transactional objects). This point will be studied in future work.

# 6 Acknowledgments

We are thankful to: G.Arango for his helpful comments about the characterization of methods, P.Biriotti for the discussions we had about the cost model and J.Ioannidis for his helpful suggestions on the derivation algorithm. We also would like to thank P.Richard, C.Lécluse, C.Bauzer-Medeiros, J.M.Larchevêque and F.Bancilhon for helpful advice and careful reading of the paper.

## References


[AB87] M.Atkinson, P.Buneman, "Types and Persistence in Database Programming Languages." *ACM Computing Surveys*, June 1987.

[AK89] S.Abiteboul, P.Kannelakis, "Object Identity as a Query Language Primitive" *In Proc Int Conf ACM Sigmod* Portland 1989.

[ABMPS90] T.Anderson, A.Berre, M.Mallison, H.Porter, B.Schneider, "The Hypermodel Benchmark"" *Proceedings of the EDBT'90 Conference*, Venis, Italy, March 1990.

[At85] T. Atwood, "An Object Oriented DBMS for Design Support Applications" *in proc IEEE COMPINT* Montreal 85.

[Ba88] F.Bancilhon, "Object-Oriented Database Systems" *Proceedings of the ACM PODS conference*, 152-162 1988.

[BD90] V.Benzaken, C.Delobel, "Enhancing Performance in a Persistent Object Store: Clustering Strategies in $O_2$", *to appear in proceedings of the fourth international workshop on Persistent Object Systems Design, Implementation and Use*, Martha-Vineyard, Massachussetts, USA, 23-27 September 1990.

[BDH90] V.Benzaken, C.Delobel, G.Harrus, "Measuring Performance of Clustering Strategies: the CluB-0 Benchmark" *Altaïr internal report* 90.

[Be90a] V.Benzaken, "Regroupement d'Objets sur Disque dans un Système de Bases de Données Orienté-Objet" *Phd Dissertation* Université d'Orsay, Jan 1990.

[Be90b] V.Benzaken, "An Evaluation Model of Clustering Strategies in the $O_2$ Object-Oriented Database System", *in proceedings of the 3rd ICDT conference*, Paris Dec 1990.

[CK89] E. E. Chang, R. H. Katz, "Exploiting Inheritance and Structure Semantics for Effective Clustering and Buffering in an Object Oriented DBMS" *in proc ACM SIGMOD conference* Portland 1989.

[HK89] S. Hudson, R. King, "Cactis: A Self-Adaptative, Concurrent Implementation of an Object-Oriented Database Management System." *ACM TODS, vol 14 n° 3* September 89.

[HMMS87] T. Harder, K. Meyer-Wegner, K. Mitschang, A. Sikeler, "PRIMA: A DBMS prototype supporting engineering applications" *Proceedings of the VLDB'87 conference*, Brighton 87.

[KBC87] W.Kim, J.Banerjee, H.T.Chou. "Composite Object Support in an Object-Oriented Database System", *OOPSLA '87 Proceedings.*

[KBG89] W.Kim, E.Bertino, J.Garza, "Composite Objects Revisited" *in Proc ACM Sigmod conf* Portland 1989.

[KCF87] S.Khoshafian, M.J. Carey, P.Franklin, "Storage Management for Persistent Complex Objects" *Technical Report MCC 87.*

[Ki88] W.Kim, "A Foundation for Object-Oriented Databases", *Technical Report MCC 88.*

[LR89a] C.Lécluse, P.Richard "Modeling Complex Structures in Object-Oriented Databases." *Proceedings of the ACM PODS conference*, Philadelphie 89.

[LR89b] C.Lécluse, P.Richard "The $O_2$ Database Programming Language" *in Proc., of the VLDB'89 conf.*, Amsterdam, August 89.

[LRV88] C. Lécluse, P. Richard and F. Velez, "$O_2$ an Object Oriented Data Model", *Proceeding of the ACM SIGMOD conference*, Chicago, 88.

[MSOP86] D.Maier, J.Stein, A.Otis, A.Purdy, "Development of an Object Oriented DBMS" *Technical Report CS/E-86-005* 1986.

[Sc77] M. Schkolnick, "A Clustering Algorithm for Hierarchical Structures", *ACM Transactions on Database Systems*, Vol 2, n°. 1, March 77.

[SPO89] P.Scheuermann, Y.C.Park, E.Omiecinski, "Heuristic reoganization of clustered files" *Proceedings of the 3rd FODO conference* Paris 89.