A New Approach in Fragmentation of Distributed Object Oriented Databases Using Clustering Techniques

Adrian Sergiu Darabant

December 13, 2005

Abstract

Horizontal fragmentation plays an important role in the design phase of Distributed Databases. Complex class relationships: associations, aggregations and complex methods, require fragmentation algorithms to take into account the new problem dimensions induced by these features of the object oriented models. We propose in this paper a new method for horizontal partitioning of classes with complex attributes and methods, using AI clustering techniques. We provide quality and performance evaluations using a partition evaluator function and we prove that fragmentation methods handling complex interclass links produce better results than those ignoring these aspects.

1 Introduction

As opposed to centralized databases where the design phase handles only logical and physical data modeling, the design process in Distributed Object Oriented Databases involves as well data partitioning and allocation to the nodes of the system. Horizontal fragmentation, in Object Oriented Database Systems, distributes class instances into fragments. Each object has the same structure and a different state or content. Thus, a horizontal fragment of a class contains a subset of the whole class extension. Horizontal fragmentation is usually subdivided in primary and derived fragmentation.

Many of the existing Object Oriented (OO) fragmentation approaches are usually inspired from the relational fragmentation techniques. While this proves to be a good starting point for approaching the fragmentation problem, there is definitely a limit in applying these techniques to data models featuring all the complex characteristics of a real OO model. The OO model is inherently more complex than the relational model. Inheritance, polymorphism, class aggregation and association all induce complex relations between classes in an object oriented database. In order to cope with the increased complexity of the OO model, one can divide class features as follows: *simple attributes* – attributes with scalar types; *complex attributes* – attributes with complex types (other classes), sets, bags, etc. as their domain; *simple methods* – methods accessing only local class simple attributes; *complex methods* - methods that return or refer instances of other classes.

In this paper we approach the horizontal fragmentation problem of classes with complex attributes and methods. We rely on AI clustering as an alternative to the current state of the art fragmentation techniques derived from the relational approaches.

1.1 Related Work

Fragmentation methods for OODB environments, or flat data models have been generally considered in Karlapalem [1], [4], [5], Ezeife [2]. Ravat [6] uses the Bond Energy Algorithm (BEA) for vertical and horizontal fragmentation. Ezeife [7] presents a set of algorithms for horizontally fragmenting models with simple attributes/methods and complex attributes/methods. She is using the algorithm developed for horizontal fragmentation in relational data models. Bellatreche et al. [9] propose a method that emphasizes the role of queries in the horizontal fragmentation.

We have already discussed an alternative AI clustering fragmentation method for OO models with simple attributes and simple methods in [12].

1.2 Contributions

We propose a new technique for horizontal fragmentation in object-oriented databases with complex attributes and methods. Fragmentation in complex OO hierarchies is usually performed in two steps: primary fragmentation and derived fragmentation. Primary fragmentation groups class instances according to a set of class conditions [12] imposed on their simple attributes.

Derived fragmentation takes into account the class relationships (aggregation, association, complex methods). It groups instances of a class in fragments according to the fragmentation of the related classes. There are generally two approaches in derived fragmentation: *left order derived fragmentation* (*parent first*) and *right order derived fragmentation* (*child first*). They differ in the order in which two related classes are fragmented. In the left order derived fragmentation, the referring class is fragmented first and determines a partitioning of the instance set of the referred class. In the right order derived fragmentation, the referred class is fragmented first and determines the partitioning of the instances of the referring class.

We propose an algorithm that unifies the two fragmentation steps: *primary* and *derived* into a single step. Both class conditions and class relationships are modeled together in a vector space. Each object is represented as a vector and we use the k-means clustering algorithm for separating clusters (fragments) of objects.

The paper is organized as follows. The next section of this work presents the object data model and the constructs used in defining the object database and expressing queries. Section 3 introduces the vector space model we use to compare objects, methods for constructing the object characteristic vectors and similarity metrics over this vector space. Section 4 presents our fragmentation algorithm. In section 5 we present a complete fragmentation example over a class hierarchy and we evaluate the quality of our fragmentation scheme by using a variant of the Partition Evaluator [12].

2 Data Model

We use an object-oriented model with the basic features described in the literature [8], [11]. Object-oriented databases (OODB) represent data entities as objects supporting features like inheritance, encapsulation, polymorphism, etc. Objects with common attributes and methods are grouped into classes. A class is an ordered tuple $C{=}(K, A, M, I)$, where A is the set of object attributes, M is the set of methods, K is the class identifier and I is the set of instances of class C. Every object in the database is uniquely identified by an object identifier (OID). Each class can be seen in turn as a class object. Class objects are grouped together in metaclasses. This allows us to consider classes as being instances of higher-level classes that describe the database schema. This way the database schema is self-describing.

Classes are organized in an inheritance hierarchy, in which a subclass is a specialization of its superclass. Although we deal here, for simplicity, only with simple inheritance, moving to multiple inheritance would not affect the fragmentation algorithm in any way, as long as the inheritance conflicts are dealt with into the data model. An OODB is a set of classes from an inheritance hierarchy, with all their instances. There is a special class Root that is the ancestor of all classes in the database. Thus, in our model, the inheritance graph is a tree.

An *entry point* into a database is a metaclass instance bound to a known variable in the system. An entry point allows navigation from it to all classes and class instances of its sub-tree (including itself). There are usually more entry points in an OODB.

Given a *complex* hierarchy H, a *path expression* P is defined as $C_1.A_1.\ldots A_n$, n≥1 where: C_1 is an entry point in H, A_1 is an attribute of class C_1, A_i is an attribute of class C_i in H such that C_i is the domain of attribute A_{i-1} of class C_{i1} (1≤ i ≤ n). In the general case, A_i can be a method call. If i<n, then A_i must return a single complex type value (an object).

As presented in [12], a *query* is a tuple with the following structure q=(Target class, Range source, Qualification clause), where:

- *Target class* – (query operand) specifies the root of the class hierarchy over which the query returns its object instances;

- *Range source* – a path expression starting from an entry point and specifying the source class hierarchy;

- *Qualification clause* – logical expression over the class attributes and/or class methods, in conjunctive normal form. The logical expression is constructed using atomic predicates: *path_expression* θ *value* where $\theta \in \{<, >, \leq, \geq, =, \neq, \in, \supset, \supseteq\}$.

3 Vector Space Modeling

3.1 Primary Fragmentation Modeling

We denote by $Q = \{q_1, \ldots, q_t\}$ the set of all queries in respect to which we want to perform the fragmentation. Let *Pred*=$\{p_1, \ldots, p_q\}$ be the set of all atomic predicates Q is defined on. Let *Pred(C)*=$\{p \in$*Pred*$|$ p imposes a condition to an attribute of class C or to an attribute of its parent$\}$. Given the predicate $p \equiv C_1.A_1. \ \ldots A_n \ \theta \ value$, $p \in Pred(C_n)$, if class C_i is the complex domain of A_{i-1}, $i = 2..n$, and A_n has a complex type or simple type.

Given *two* classes C and C', where C' is subclass of C, $Pred(C') \supseteq Pred(C)$. Thus the set of predicates for class C' comprises all the predicates directly imposed on attributes of C' and the predicates defined on attributes of its parent class C and inherited from it [12].

We construct the object-condition matrix for class C, $OCM(C) = \{a_{ij}, 1 \leq i \leq |Inst(C)|, 1 \leq j \leq |Pred(C)|\}$, where $Inst(C) = \{O_1, \ldots O_m\}$ is the set of all instances of class C, $Pred(C) = \{p_1, \ldots, p_n\}$:

$$a_{ij} = \begin{cases} 0, \ if \ p_j(O_i) \ = \ false \\ 1, \ if \ p_j(O_i) \ = \ true \end{cases} \quad w_{ij} = \frac{\sum\limits_{l=\overline{1..m}, a_{lj}=a_{ij}} [(a_{lj}|a_{lj}=1) + (1-a_{lj}|a_{lj}=0)]}{m} \tag{1}$$

Each line i in $OCM(C)$ is the object-condition vector of O_i, $O_i \in Inst(C)$. We obtain from $OCM(C)$ the characteristic vectors for all instances of C. The characteristic vector for object O_i is $w_i = (w_{i1}, w_{i2}, \ldots, w_{in})$, where each w_{ij} is the ratio between the number of objects in C respecting the predicate $p_j \in Pred(C)$ in the same way as O_i, and the number of objects in C. We denote the characteristic vector matrix as $CVM(C)$ [12].

3.2 Attribute Induced Derived Fragmentation Modeling

We have captured so far all characteristics of simple attributes and methods. We need to express the class relationships in our vector space model. We first model the aggregation and association relations.

Given two classes C_O (owner) and C_M (member), where C_M is the domain of an attribute of C_O, a path expression traversing this link navigates from instances of C_O to one or more instances of C_M. In the case of left derived fragmentation C_O will be fragmented first, followed by C_M. In the right derived fragmentation variant the order in which the two classes are fragmented is reversed. Each of the two strategies is suitable for different query evaluation strategies. For example, in reverse traversal query evaluation strategy, the right derived fragmentation variant gives the best results.

We assume here, for space reasons, that right derived fragmentation method is used. However, both: the algorithm and the vector space model remain the same when considering left derived fragmentation order.

Thus, in right derived fragmentation method, when fragmenting C_O we should take in account the fragmentation of C_M [13]. We want to place in the same fragment of C_O objects aggregating instances from a fragment of C_M. Objects of a fragment of C_O should aggregate as much as possible objects from the same fragment of C_M.

Let $\{\mathrm{F}_1, \ldots \mathrm{F}_n\}$ be the fragments of C_M. We denote by $Agg(O_i, F_j)$=$\{O^m \mid O^m \in F_j, O_i \textit{ references } O^m\}$. Given the set of fragments for C_M, we define the *attribute-link induced object-condition vectors for derived fragmentation* as ad_i = $(ad_{i1}$, ad_{i2}, ..., $ad_{in})$, where each vector component is expressed by the following formula:

$$ad_{ij} = sgn(|Agg(O_i, F_j)|), \quad j = \overline{1, n} \tag{2}$$

For an object $O_i \in Inst(C_O)$ and a fragment F_j of C_M, ad_{ij} is 1 if O_i is linked to at least one object of F_j and is 0 otherwise.

Given the set of fragments for C_M, we define the *attribute-link induced characteristic vectors for derived fragmentation* as $wad_i = (wad_{i1}, wad_{i2}, \ldots, wad_{in})$, where each vector component is expressed by the following formula:

$$wad_{ij} = \frac{|\{O_l \in Inst(C_O) | sgn(|Agg(O_l, F_j)|) = sgn(|Agg(O_i, F_j)|)\}|}{|Inst(C_O)|}, j = \overline{1, n} \tag{3}$$

Each wad_{ij} component gives the percentage of objects in C_O that aggregate/refer in the same way as O_i objects from F_j. Two objects O_i and O_l are said to aggregate F_j *in the same way* if they are both either linked or not linked with objects from F_j. According to this criterion, two objects are candidates to be placed in the same fragment of C_O in respect to F_j if they are both related in the same way to F_j.

3.3 Method Induced Derived Fragmentation Modeling

In the following paragraphs we model the class relationships induced by the presence of complex methods. Given a class with complex methods C(owner) that has to be fragmented, we need to take in account, when fragmenting it, the fragmentation of classes referred by its complex methods. In order to model the method reference dependencies in the fragmentation process we need to express this type of relationships in our vector space.

We denote by $MetComplex(C)$=$\{m_i|$ m_i *complex method of* $C\}$ – the set of all complex methods of class C.
Let $SetCRef(m,C) = \{C_R|C \neq C_R, C_R$ is referred by method $m \in MetComplex(C)\}$ be the set of classes referred by the complex method m of class C. For a given instance of a class C with complex methods we denote as:
$SetORef(m,O_i,C_R)$=$\{O_r^{'}{\in}Inst(C_R)$ | $C_R{\in}SetCRef(m,C), m \in MetComplex(C), O_r^{'}$ *is referred by method* $m\}$ – the set of instances of class $C_{R,}$referred by the complex method m of class C.

For each pair $(m_k, C_R) \in \{m_k \in MetComplex(C)\} \times SetCRef(m_k, C)$ we quantify the way each instance of C refers - through complex methods - instances from fragments of C_R. Given a class C_R referred by a complex method m_k of class C, and the fragments of class $C_R \rightarrow \{F_1, \ldots F_n\}$, we define the *method-link induced object-condition vectors for derived fragmentation.* For each instance O_i of C let md_i= $(md_{i1}$, md_{i2}, ..., $md_{in})$ be the *method-link induced object-condition vector.* Each vector component is defined by the following formula:

$$md_{ij} = sgn(|\{O_l \in Inst(C_R)|O_l \in F_j \cap SetORef(m_k, O_i, C_R)\}|), j = \overline{1,n} \quad (4)$$

Each md_{ij} evaluates to 1 when object $O_i{\in}Inst(C)$ refers objects from fragment F_j of class C_R and 0 otherwise. For each object O_i we obtain, for each pair (m_k, C_R), one *method-link induced object-condition vector.* We derive from it the *method-link induced characteristic vector for derived fragmentation,* wmd_j =(wm_{i1}, wmd_{i2},...,wmd_{in}), where:

$$wmd_{ij} = \frac{|\{O_l \in Inst(C)|md_{lj} = md_{ij}\}|}{Inst(C)}, j = \overline{1,n}\,, l = \overline{1, |Inst(C)|} \qquad (5)$$

Each wmd_{ij} quantifies the way objects of class C refer objects from fragments of C_j through complex methods.

When modeling relationships induced by the presence of complex methods, we obtain as many referring condition vectors (object-condition and characteristic), for each instance O_i of C, as the number of elements of the Cartesian product $\{m_k \in MetComplex(C)\} \times SetCRef(m_k, C)$.

3.4 Derived Fragmentation Modeling

As the number of elements in $\{m_k \in MetComplex(C)\} \times SetCRef(m_k, C)$ is usually large we need to use some heuristics in order to retain only the pairs with significant impact in the fragmentation. In order for a pair (m_k, C_R) to be kept it should satisfy the following combined restrictions:

- The number of calls to the method m_k should be significant compared to the contribution brought by all method calls made by applications running on the database;
- The number of instances of C_R referred by the method m_k should be significant compared to the number of instances of all classes generally referred by the applications.

The above conditions are expressed in the following formula (*significance factor*):

$$Sig(m_k, C_R) = \frac{NrCalls(m_k)}{\sum\limits_{m_l \in MetComplex(C)} NrCalls(m_l)} \times \frac{\sum\limits_{O_i \in Inst(C)} |SetORef(m_k, O_i, C_R)|}{\sum\limits_{C_p \in SetCRef(m_k)} \sum\limits_{O_r \in Inst(C)} SetORef(m_k, O_r, C_p)} \tag{6}$$

In (6) the first factor gives the ratio between the number of calls to method m_k and the number of calls of all complex methods of class C. The second factor gives the ratio between the number of C_R instances referred by m_k and the number of all objects referred by m_k. In reality the actual method parameters would normally influence the set of objects referred by the method. Even more, the set of referred objects could be as well influenced by the internal state of the object. However, tracking all the possible combinations is computationally intractable – even in simple situations. The statistical heuristic proposed in (6) is still manageable and helps reducing the problem space dimensions.

Usually, the fragmentation of a class C is performed in two steps: primary fragmentation, according to query conditions, and derived fragmentation, according to the fragments of the member or owner classes. We merge the two phases into one single step capturing the semantic of both primary and derived fragmentations. For this we unify the characteristic vector, the attribute-link and method-link induced characteristic vectors for each object O_i of the class C, and we obtain the *extended characteristic vector*. Each extended characteristic vector quantifies all the information needed for fragmentation: the conditions imposed on the object and the relationships of the object with instances of related classes, induced either by complex attributes or by complex methods.

If the class C is related with classes $C_{A1}, C_{A2}, \ldots, C_{Ap}$ by means of complex attributes, and with classes $C_{M1}, C_{M2}, \ldots, C_{Mr}$ by means of complex methods, the *extended characteristic vector* we_i for object $O_i \in Inst(C)$ is obtained by appending the p attribute-link induced characteristic vectors and the number of $mc = |\{m_k \in MetComplex(C)\} \times SetCRef(m_k, C)|$ method-link characteristic vectors to the characteristic vector of O_i. However, as we have already mentioned above, we are using the *significance factor* to filter out non-relevant pairs (m_k, C_R) and vectors derived from them. As observed experimentally, a significance factor around 0.27 will filter out most of the inappropriate (m_k, C_R) pairs.

The *extended object-condition vector* ae_i for an object O_i is obtained in the same way by appending its attribute-link and method-link induced object-condition vectors to its object-condition vector. We denote by $EOCM(C)$ and $ECVM(C)$ the extended object-condition and characteristic matrices for class C.

3.5 Similarity between Objects

The aim of our method is to group into a cluster those objects that are similar to one another. Similarity between objects is computed using the Euclidian and Manhattan metrics:

$$d_E(we_i, we_j) = \sqrt{\sum_{k=1}^{n} (we_{ik} - we_{jk})^2}\,,\ d_M(we_i, we_j) = \sum_{k=1}^{n} |we_{ik} - we_{jk}| \quad (7)$$

Given two objects $O_i$and O_j, we define two similarity measures between them in (8):

$$sim_E(O_i, O_j) = 1 - \frac{d_E(we_i, we_j)}{|Inst(C)|}\,,\ sim_M(O_i, O_j) = 1 - \frac{d_M(we_i, we_j)}{|Inst(C)|} \quad (8)$$

We use sim_E and sim_M in (8) to measure how similar two objects are. Both measures take values in [0,1] and are expressed using one of the two distances from (7). The distance functions and the similarity measures are inversely proportional in [0,1]. As the distance between two objects increases, their similarity decreases. We should note that all characteristic vectors have positive coordinates by definition.

4 K-means Centroid-based Fragmentation

We apply an algorithm we have used to fragment classes with simple attributes and methods: the k-means centroid based clustering algorithm [12]. The classical k-means algorithm takes the input parameter k and partitions a set of m objects into k clusters so that the resulting intra-cluster similarity is high but the inter-cluster similarity is low. Cluster similarity is measured in regard to the *mean* value of the objects in a cluster, which can be viewed as the cluster's *center of gravity* (*centroid*). First, the k-means algorithm randomly selects k of the objects, each of which initially represents a cluster mean or center. For each of the remaining objects, an object is assigned to the cluster to which is the most similar, based on the distance between the object and the cluster centroid. It then computes the new centroid for each cluster and redistributes all objects according to the new centroids. This process iterates until a criterion function converges. The criterion tries to make the resulting k clusters as compact and separate as possible.

Our version of the algorithm improves several aspects of the original algorithm with regard to the semantic of object fragmentation. First of all, we implement a variant where we choose as initial centroids the most representative objects in respect with fragmentation predicates, rather than choosing them arbitrarily. At each iteration, if an object should be placed in any of several clusters (same similarity with the centroid), we choose the cluster to which the object has maximum similarity with. We also choose as criterion function the degree of compactness/homogeneity H of all clusters. For a given cluster F, this value is the difference between the maximum and minimum similarity of all pairs of objects in F:

$$H(F) = \max\{sim(a,b) \in F \times F, a \neq b\} - \min\{sim(a,b) \in F \times F, a \neq b\} \quad (9)$$

```
Algorithm k-meansFrag is:
Input:  Class C, Inst(C) to be fragmented, the similarity function
$sim : Inst(C) \times Inst(C) \rightarrow [0,1], m = |Inst(C), 1 < k \leq m$ desired
```

```
number of fragments, $OCM(C), CVM(C), threshold\_value$.
Output:  the set of clusters $F = \{F_1, \ldots, F_f\}$, where $f \leq k$.
Begin
  Centr=$\{c_1, \ldots, c_k\}$=InitCentr(Inst(C), OCM(C), CVM(C), k);
  $F = \{F_i | F_i = \{c_i\}, c_i \in Centr, i = 1..k\}; F' = \emptyset$;
  // initial object allocation to clusters;
  For all objects $O_i$ do
    $F_{candidates} = \{argmax_{centr}(sim(O_i, c_l),\ l = 1..k)\}$;
    $F_{u*} = argmax_{sim}(sim(O_i, f_c), f_c \in F_{candidates}); F_{u*} = F_{u*} \cup \{O_i\}$;
  End For;
  While $F' \neq F$ and H(F)<threshold_value do
    For all $F_i \in F$ recalculate centroid $c_i$;
    $F' = F$;
    For all objects $O_i$ do
      $F_{candidates} = \{argmax_{centr}(sim(O_i, c_l), l = 1..k)\}$; (i)
      $F_{u*} = argmax_{sim}(sim(O_i, F_c), F_c \in F_{candidates})$; (ii)
      $F'_v = F'_v - \{O_i\},\ where\ O_i \in F'_v$;
      $F'_{u*} = F'_{u*} \cup \{O_i\}$;
      $F' = F' - \{F'_l | F'_l = \emptyset\}$; // eliminate empty clusters
    End For;
  End While;
End.

Function InitCentr(Inst(C),OCM(C),CVM(C),k) is
Begin
  Centr=$\emptyset; n = |Pred(C)|$;
  For i=1 to k do
    $c_i = argmin[d_M(OCM(O_j), u_i)], Oj \notin Centr, i \leq n$; (iii)
    $c_i = argmin(sim(O_j, Centr)), O_j \notin Centr, i > n$; (iv)
    $Centr = Centr \cup \{c_i\}$;
  End for;
  Return Centr;
End Function;
```

Function *InitCentr* chooses the initial centroids as described above. In line (iii) u_i is the identity vector of degree i, which has 1 only on the i^{th} position and 0 on the other positions. Each u_i represents the corresponding predicate from $Pred(C)$. Line (iii) chooses as centroid the closest object to u_i, i.e. the most representative object for that predicate. We note that we can choose this way as many centroids as the number of predicates in $Pred(C)$. If we need more clusters than $|Pred(C)|$, we choose as their initial centroids the objects most dissimilar to the already chosen centroids (line (iv)). We try this way to minimize the impact of "losing" clusters in the following iterations. This occurs when all objects in a cluster relocate to other clusters because the initial centroid is not semantically representative to our set of predicates.

We use in lines (i) and (ii) the similarity of an object O_i with a cluster F_c, defined as (the average similarity with all objects of the cluster):

$$sim(O_i, F_c) = \frac{\sum_{a \in F_c} sim(O_i, a)}{|F_c|} \tag{10}$$

5 Results and Evaluation

In this section we illustrate the experimental results obtained by applying our fragmentation scheme on a test object database. Given a set of queries, we first obtain the horizontal fragments for the classes in the database; afterwards we evaluate the quality and performance of the fragmentation results. The problem with the evaluation method is that it is difficult to quantify a fragmentation result without allocating the fragments to the nodes of a distributed system. On the other side, the allocation problem must be solved in order to be able to evaluate the fragmentation. As resolving the allocation problem in the general case is not a trivial task, we need a simplified allocation model, yet a valid one. Our solution is to consider a distributed system running database applications (queries). Some of the nodes are part of the distributed object oriented DBMS as well. They hold fragments of the database and a database engine. All applications run with different frequencies on different nodes of the system. We chose to allocate each fragment on the node where is most needed (accessed).

Another issue that might affect the results is the fact that the order in which classes are fragmented is significant as it captures the semantic of query path expressions into the fragmentation process [13]. It might be possible to obtain better results by using a different order for fragmenting classes. We do not handle here the ordering problem, but we address it in [13].

The sample object database represents a reduced university database.

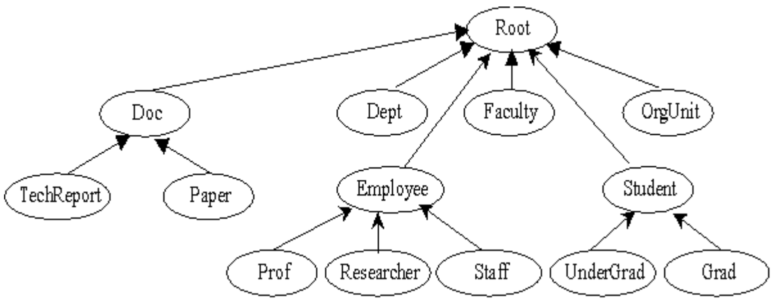

Figure 1: The database class hierarchy

The inheritance hierarchy is given in Figure 1 and a trimmed down version of the aggregation/association graph is shown in Figure 2.

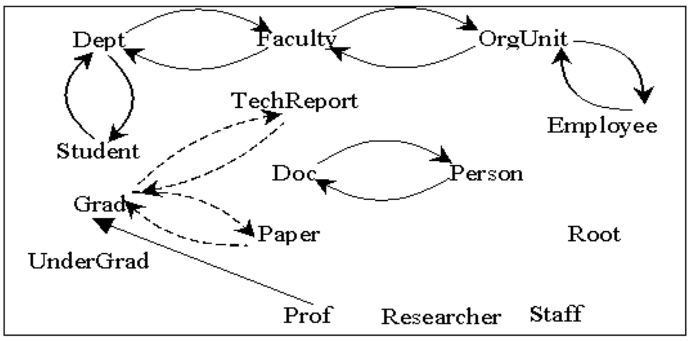

Figure 2: The database aggregation/association graph

The queries running on the classes of the database are given bellow:

q_1 =(Grad, Faculty.Dept.Student, Grad.Supervisor.OrgUnit.Name in (“ProgMeth”, “InfSyst”))

q_2 =(UnderGrad, Faculty.Dept.Student, UnderGrad.Dept.Name like “CS%” and UnderGrad.Grade between 7 and 10)

q_3 =(UnderGrad, Faculty.Dept.Student, (UnderGrad.Dept.Name like “Math%” or UnderGrad.Dept.Name like “CS%”) and UnderGrad.Age()≥24)

q_4 =(Researcher, Doc.Person, Researcher.count(Reasercher.doc) ≥2)

q_5 =(Prof, Faculty.OrgUnit.Employee, Prof.OrgUnit.Name in (“ProgMeth”, “InfSyst”) and Prof.salary≥40000)

q_6 =(Prof, Doc.Person., Prof.Paper.Publisher in (“IEEE”, “ACM”) and Prof.Position=”prof”)

q_7 =(TechReport, Doc, TechReport.year>1999)

q_8 =(Set(Student.Dept), Person, Student.Grade<5)

q_9 =(Employee, Person, Employee.salary>35000)
q_{10} =(Grad, Person, Grad.count(Grad.Paper)≥1)
q_{11} =(Student, Person,Student.Dept.Name like “CS%”)
q_{12} =(Student, Person,Student.Dept.Name like “Math%”)
q_{13} =(Staff, Person, Staff.salary>12000)
q_{14} =(Person, Person, Person.Age()>30)

In Figure 2 the links between Doc and Person should be inherited by all subclasses of Person and Doc. This is graphically represented in the figure by the dotted arrows. Similar inherited links are present for other classes in this graph (links not represented here). The motivation for aggregation/association inheritance is presented in [13].

For measuring the fragmentation quality we determine the cost of remote accesses combined with the cost of local irrelevant accesses to each fragment. Remote accesses are made by applications running on a given node and accessing objects that are not stored on that node. Local irrelevant accesses are given by local processing incurred when a query accesses a fragment. Each access to a fragment implies a scan to determine objects that satisfy a condition. Irrelevant local access measure the number of local accesses to objects that will not be returned by the query. Intuitively, we want that each fragment be as compact as possible and contain as much as possible only objects accessed by queries running on the fragment's node. We use the following measure for calculating the fragmentation quality:

$$PE(C) = EM^2 + ER^2 \tag{11}$$

$$EM^2(C) = \sum_{i=1}^{M}\sum_{t=1}^{T} freq_{ts}^2 * |Acc_{it}| * \left(1 - \frac{|Acc_{it}|}{|F_i|}\right) \tag{12}$$

$$ER^2(C) = \sum_{t=1}^{T} \min\left\{\sum_{s=1}^{S}\sum_{i=1}^{M} freq_{ts}^2 * |Acc_{it}| * \frac{|Acc_{it}|}{|F_i|}\right\} \tag{13}$$

The EM term calculates the local irrelevant access cost for all fragments of a class. ER calculates the remote relevant access cost for all fragments of a class. Acc_{it} represents the set of objects accessed by query t from fragment F_i. The value $freq_{ts}$ is the frequency of query t running on site s. In (12) s is the site where F_i is located, while in (13) s is any site not containing F_i. M is the number of clusters for class C, T is the number of queries and S is the number of sites [12]. The fragmentation is better when the local irrelevant costs and the remote relevant access costs are smaller. Each term of PE calculates in fact the average square error of these factors. Globally, *PE* measures how well fragments fit the object sets requested by queries.

Using the given query access frequency, the fragments above are allocated to 4 distributed sites. Query frequency at sites is presented in Table 1.

Table 1: Access frequencies of queries at distributed sites

Freq(q,s)	S1	S2	S3	S4
q1	0	10	5	20
q2	0	10	5	25
q3	20	0	15	10
q4	15	10	5	0
q5	25	20	0	20
q6	30	0	20	10
q7	30	25	0	10
q8	10	0	0	10
q9	20	20	10	0
q10	15	25	0	0
q11	5	10	5	0
q12	0	0	0	10
q13	15	0	0	5
q14	20	5	0	0

We qualitatively compare the results of our fragmentation method with a centralized and a full replicated database in Figure 3. The centralized version of the database is allocated to node S1, while in the replicated case each node holds a copy of the entire database. It can be seen that our fragmented database obtains smaller *PE* costs, with both measures, than the centralized and full replicated database. The full replicated case obtains the worst costs as the irrelevant access cost explodes in this case. Even though remote accesses tend to zero in the replicated case, the local irrelevant accesses increase considerably as each node holds an entire copy of the database, thus many irrelevant objects for each query. In reality, the full replicated case performs well only when there are no updates to the database. The Manhattan similarity measure applied on *OCM* obtains the best results, followed by Manhattan similarity applied on characteristic vectors and by the Euclid measure at last.

In Figure 4 we present the *PE* costs induced on each fragmented class with each method. Here it can be seen that the Manhattan and Euclid measures behave approximately the same on all classes except *Undergrad*. In our example, classes

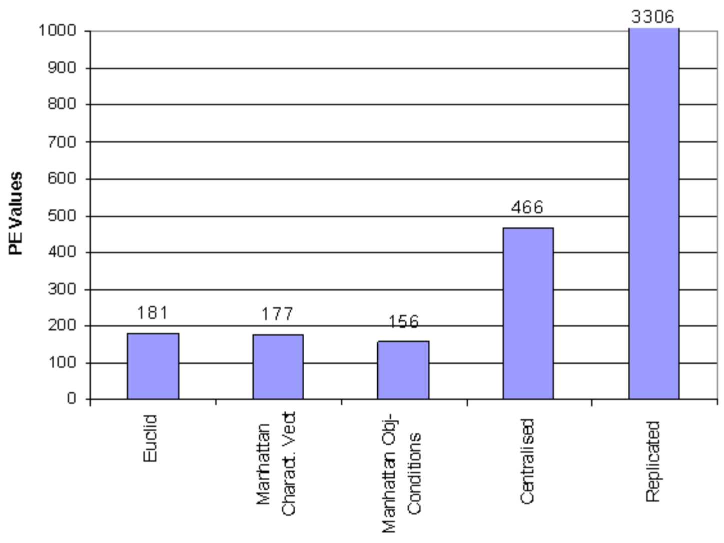

Figure 3: Comparative *PE* values for our fragmentation method, centralised and replicated databases

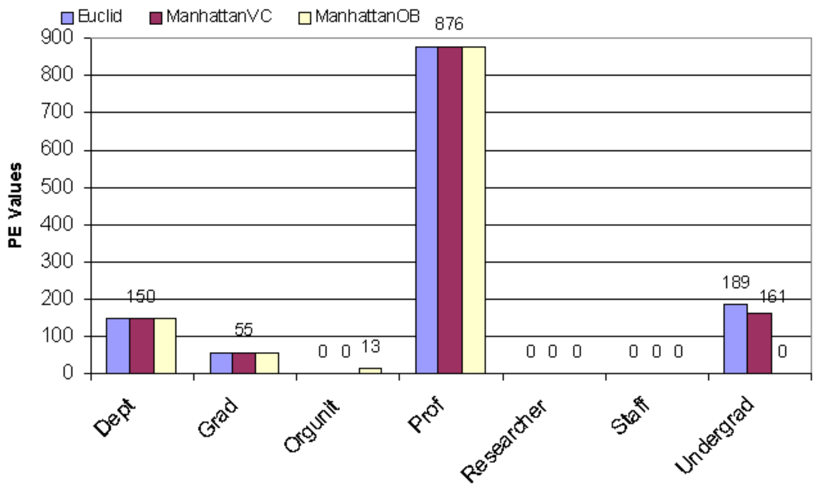

Figure 4: Comparative class PE values for each similarity measure

have been fragmented in the order they appear in Figure 4, from left to right, Undergrad being the last fragmented class. Even if *PE* scores for other classes are approximately the same – the resulting fragments are not identical for different similarity measures.

This leads to the fact that when fragmenting Undergrad, the resulting fragments are influenced by the fragmentation of the related classes. Manhattan applied on *OCM* does the best fragmentation on the intermediate (related) classes, which leads to a better score when the last class (Undergrad) is fragmented.

Finally in Figure 5 we compare the results of the same fragmentation algorithm in two cases: when complex class relationships are considered and when complex class relationships are ignored, i.e primary only fragmentation. The *P-Euclid, P-Manhattan Charact. Vect.* and *P-Manhattan Obj. Conditions* denote the primary only versions of the fragmentation algorithm.

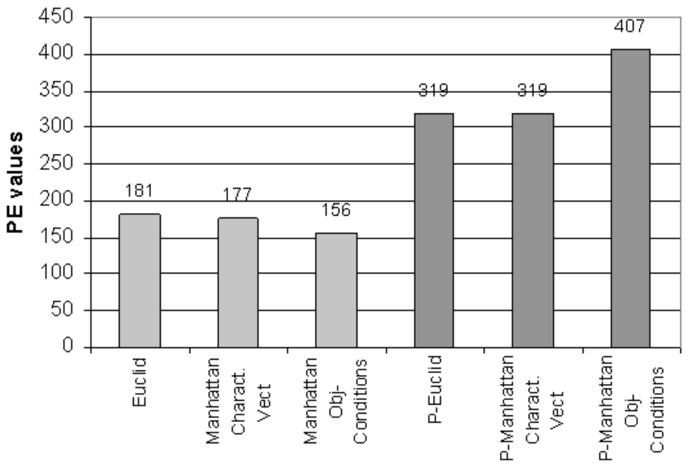

Figure 5: Comparative *PE* values for *primary only fragmentation* and our complex fragmentation method (*primary + derived fragmentation*)

It can be seen that the fact of considering the complex class relationships improves quality. All similarity measures in this case behave better than the best case of the fragmentation without derived fragmentation.

6 Conclusions and Future Work

We proposed in this paper a new horizontal fragmentation method for object oriented distributed databases. Our method takes into account all complex relationships between classes: aggregations, associations, and links induced by complex methods. Primary and derived fragmentations are modeled together and are performed in a single step. This reduces the complexity of our technique compared to traditional approaches that perform primary and derived fragmentation in two distinct steps, processing twice the entire database.

We have shown that taking complex relationships into account significantly improves fragmentation quality as opposed to methods considering only primary fragmentation. The order in which classes are fragmented is important as class relationships may induce mutual transitive class dependencies. There is always a fragmentation order that produces better results than the average of all other orders. We proposed an algorithm for determining the fragmentation order in [13].

We aim to find new ways of expressing inter-class relationships and compare their results and impact in the fragmentation process.

References

1. Karlapalem, K., Navathe, S.B., Morsi, M.M.A.: Issues in distribution design of object-oriented databases. In: Tamer Ozsu, M., Dayal, U., Valduriez, P. (eds.): Distributed Object Management, Morgan Kaufmann Publishers (1994) 148-164
2. Ezeife, C.I., Barker, K.: A Comprehensive Approach to Horizontal Class Fragmentation in a Distributed Object Based System, International Journal of Distributed and Parallel Databases, 3(3) (1995) 247-272
3. Han, J., Kamber, M., Data Mining: Concepts and Techniques, The Morgan Kaufmann Series in Data Management Systems (2000)
4. Karlapalem, K., Li, Q.: Partitioning Schemes for Object-Oriented Databases, In Proceedings of the Fifth International Workshop on Research Issues in Data Engineering-Distributed Object Management, Taiwan (1995) 42–49
5. Karlapalem, K., Li, Q., Vieweg, S.: Method Induced Partitioning Schemes in Object-Oriented Databases, In Proceedings of the 16th Int. Conf. on Distributed Computing System (ICDCS'96), Hong Kong (1996) 377–384
6. Ravat, S.: La fragmentation d'un schema conceptuel oriente objet, In Ingenierie des systemes d'informaton (ISI), 4(2) (1996) 161–193
7. Ezeife, C.I., Barker, K.: Horizontal Class Fragmentation for Advanced-Object Modes in a Distributed Object-Based System, In the Proceedings of the 9th International Symposium on Computer and Information Sciences, Antalya, Turkey (1994) 25-32
8. Bertino, E., Martino, L.: Object-Oriented Database Systems; Concepts and Architectures, Addison-Wesley (1993)
9. Bellatreche, L., Karlapalem, K., Simonet, A.: Horizontal Class Partitioning in Object-Oriented Databases, In Lecture Notes in Computer Science, volume 1308,

Toulouse, France (1997) 58–67
10.Savonnet, M. et. al.: Using Structural Schema Information as Heuristics for Horizontal Fragmentation of Object Classes in Distributed OODB, In Proc IX Int. Conf. on Parallel and Distributed Computing Systems, France (1996) 732-737
11.Baiao, F., Mattoso, M.: A Mixed Fragmentation Algorithm for Distributed Object Oriented Databases, In Proc. Of the 9th Int. Conf. on Computing Information, Canada (1998) 141-148
12.Darabant, A. S., Campan, A.: Semi-supervised learning techniques: k-means clustering in OODB Fragmentation, IEEE International Conference on Computational Cybernetics ICCC 2004, Vienna University of Technology, Austria, August 30 - September 1 (2004) 333-338
13.Darabant, A.S, Campan, A.: Optimal Class Fragmentation Ordering in Object Oriented Databases, In Studia Universitatis Babes Bolyai Informatica, Volume XLIX, Number 1 (2004) 45-54