Horizontal Class Fragmentation For Advanced Object Models in a Distributed Object Based System*

C.I. Ezeife and Ken Barker

Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

{christie,barker}@cs.umanitoba.ca

Abstract

Optimal application performance on a Distributed Object Based System (DOBS) requires class fragmentation and the development of allocation schemes to place fragments at distributed sites so data transfer is minimized. Fragmentation enhances application performance by reducing the amount of irrelevant data accessed and the amount of data transferred unnecessarily between distributed sites. Algorithms for effecting horizontal and vertical fragmentation of *relations* exist, but fragmentation techniques for class objects in a distributed object based system is a relatively untouched field of study. This paper first reviews a taxonomy of the fragmentation problem in a distributed object base. The paper then contributes by presenting an algorithm for horizontally fragmenting the most complex class model on the taxonomy, consisting of complex attributes using complex methods. The fundamental approach is top–down, where the entity of fragmentation is the class object. Our approach consists of first generating primary horizontal fragments of a class based on only applications accessing this class, and secondly generating derived horizontal fragments of the class arising from primary fragments of its subclasses, its complex attributes (contained classes), and its complex methods classes. Finally, we combine the sets of primary and derived fragments of each class to produce optimal fragments. Thus, these algorithms account for inheritance and class composition hierarchies as well as method nesting among objects.

1 Introduction

Many researchers have demonstrated the importance of entity fragmentation in distributed relational database design. Optimal application performance on a Distributed Object Based System (DOBS) requires class fragmentation and the development of allocation schemes to place fragments at distributed sites so data transfer is minimized [3]. A DOBS supports an object oriented data model including features of *encapsulation* and *inheritance*. A distributed object based system is a collection of local object bases distributed at different sites, interconnected by a network. We assume the database management system (DBMS) is distributed and the objects making up the object base are placed around the network.

The data in a DOBS consists of a set of encapsulated objects. The data values (attributes) are bundled with the methods (procedures) for manipulating them to form an *encapsulated* object. Objects with common attributes and methods belong to the same class, and every class has a unique identifier. Inheritance allows reuse and incremental redefinition of new classes in terms of existing ones. Parent classes are called *superclasses* while classes that inherit attributes and methods from them are called

*This research was partially supported by the Natural Science and Engineering Research Council (NSERC) of Canada under an operating grant (OGP-0105566) and a grant from Manitoba Hydro.

subclasses. The database contains a root class called *Root* which is an ancestor of every other class in the database. The overall inheritance hierarchy of the database is captured in a class lattice. The lattice presents the inheritance relationship between classes after the resolution of conflicts associated with multiple inheritance, and thus, a subclass in the lattice is linked to only one superclass. A class is an ordered relation $\mathcal{C} = (\mathbf{K},\mathcal{A},\mathcal{M},\mathcal{I})$ where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[1]. Other features of the model used to succinctly address the problem include the type of fragmentation that can be performed on a class, as well as the type of attributes and methods contained in the class. These features are discussed next. Each horizontal fragment ($\mathcal{C}_h$) of a class contains all attributes and methods of the class but only some instance objects ($\mathcal{I}^{'} \subseteq \mathcal{I}$) of the class. Thus, $\mathcal{C}_h = (\mathrm{K},\mathcal{A},\mathcal{M},\mathcal{I}^{'})$. Two types of attributes in a class are possible (simple and complex). Simple attributes have only primitive attribute types that do not contain other classes as part of them. A complex attribute has a database class for its type. The composition relationship between a class (due to its complex attributes) and other classes in the database is captured using a "part–of" or class composition hierarchy. If class C_i has a complex attribute that uses class C_j for its type, then class C_j is a contained class of C_i while C_i is a containing class of C_j. Two possible method structures in a distributed object based system are simple and complex methods. Simple methods are those that do not invoke other methods of other classes. Complex methods are those that can invoke methods of other classes. The classes making up the DOBS are classified based on the nature of the attributes and methods they contain as discussed earlier in [1, 2]. Although two basic method types exist, a simple method of a contained (part-of) class is referred to as a contained simple method because it is a simple method of a class that is contained in another class. Thus, the variety of class models that could be defined in a DOBS are: Class models consisting of simple attributes and simple methods. Class models consisting of complex attributes and contained simple methods. Class models consisting of simple attributes and complex methods. Class models consisting of complex attributes and complex methods.

The problem of distributed database design comprises first, the fragmentation of database entities and secondly, the allocation of these fragments to distributed sites. Two approaches are possible in a distributed database design – top-down and bottom-up. This paper adopts the top-down approach, and thus, input to the design process is the global conceptual schema (GCS) and the access pattern information, while the output from the design process is a set of local conceptual schemas (LCSs) [6]. The input to the design process is obtained from an *a priori* static and system requirements analysis which defines the system environment and collects an approximation of both the data and processing needs of all potential database users [8]. Statistical information collected from these design activities include the frequency of user applications and reference patterns of applications. The output of this static and system analysis are the GCS and the access pattern information which constitute inputs to the distribution design process described here. This work generates LCSs by defining class fragments to be distributed later.

Many distributed and client/server object based systems exist which will benefit from fragmentation [3]. A partial list of benefits include: Different applications access or update only portions of classes so fragmentation will reduce the amount of irrelevant data accessed by applications. Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base. Fragmentation allows parallel execution of a single query by dividing it into a set of subqueries that operate on fragments of a class. Fragmentation reduces the amount of data transferred when migration is required. Fragment replication is more efficient than replicating the entire class because it reduces the update problem and saves storage.

Three fragment types are defined on a database entity: Horizontal Fragmentation: Breaking up of a relation/class into a set of horizontal fragments with only subsets of its tuples/instance objects. Vertical Fragmentation: Breaking up of a class into a set of vertical fragments with only subsets of its attributes and methods. Hybrid Fragmentation: Breaking up of a class into a set of hybrid fragments with both subsets of its tuples/instance objects as well as subsets of their attributes and methods.

The overhead and difficulty involved in implementing distributed design techniques include the generation of inputs from static analysis. Earlier work has argued that since 20% of user queries account for

[1]We adopt the notation of using calligraphic letters to represent sets and roman fonts for non-set values.

80% of the total data accesses, so we argue that this analysis is feasible [7]. Secondly, these distribution techniques work best for domains without frequent drastic changes in requirements. Accommodating major changes in a domain would entail a re-analysis of the system and re-running of the distributed design algorithms. Future research will investigate how these techniques can be incorporated into a dynamic system.

This paper reviews possible DOBS models and contributes by presenting algorithms for horizontally fragmenting the advanced class model consisting of complex attributes and complex methods. The balance of the paper is organized as follows. We complete this section by briefly reviewing previous work on distributed database design. In Section 2 a summary of our earlier work reporting fragmentation algorithms for an object model with simple attributes and simple methods is first presented [1, 2]. This paper contributes by extending the earlier work to capture the most complex models, a model consisting of complex attributes and complex methods (Section 2). Section 3 concludes the paper and suggests future research directions.

1.1 Related Work

This section reviews previous work on horizontal class fragmentation in distributed database systems.

Horizontal Fragmentation (objects): Karlapalem, Navathe and Morsi [3] identify some of the fragmentation issues in object bases including: How are subclasses of a fragment of a class handled? Which objects and attributes of the objects are being accessed by the methods? What type of methods are considered: simple methods that access a set of attribute values of an object or complex methods that access a set of objects and instance variables?[2] Further, they argue that a precise definition of the processing semantics of the applications is necessary. They do not present solutions for horizontally fragmenting class objects but argue that techniques used by Navathe *et al.* [5] for fragmenting relations could be applied.

2 Horizontal Fragmentation

This section first summarizes horizontal fragmentation algorithms for classes consisting of objects that have simple attributes using simple methods as discussed in [1, 2]. It then presents the horizontal fragmentation algorithm for classes consisting of objects that have complex attributes using nested method invocations (complex methods).

2.1 Simple Attributes and Methods

The explicit assumptions made are: Firstly, objects of a subclass physically contain only pointers to objects of its superclasses that are logically part of them. In other words, an object of a class is made from the aggregation of all those objects of its superclasses that are logically part of this object. Secondly, there are no cycles in the dependency graphs discussed later. This means that two classes cannot both depend on each other at the same time so if class C_i is part-of class C_j, class C_j cannot be part–of class C_i.

Primary Horizontal Fragmentation is the partitioning of a class based on only applications accessing the class. For example, an application running on a class **Student**, with an attribute **dept**, could have predicates $\{P_1$:dept="Math",P_2:dept="Computer Sc",P_3:dept="Stats"$\}$. *Derived horizontal fragmentation* occurs in two ways. (1) Partitioning of a class arising from the fragmentation of its subclasses, and (2) fragmentation of a class arising from the fragmentation of some complex attribute (part–of) of the class. *The cardinality of a class* is the number of instance objects in the class (denoted **card**(C_i)).

Definition 2.1 *A user query* accessing database objects is a sequence of method invocations on an object or set of objects of classes. The invocation of method j on class i is denoted by M_j^i and a user query Q_k is represented by $\{M_{j1}^{i1}, M_{j2}^{i2}, \ldots, M_{jn}^{in}\}$ where each M in a user query refers to an invocation of a method of a class object. ■

[2] They take complex methods as being synonymous with an application.

Definition 2.2 *Access frequency of a query* is the number of accesses a user application makes to data. Data in this context can be a class, a fragment of a class, an instance object of a class, an attribute or method of a class. If Q = $\{q_1,q_2,\ldots,q_q\}$ is a set of user queries, acc(q_i,d_j) indicates the access frequency of query q_i on data item j. We represent the access frequency of a minterm m_i as acc(m_i,d_j) since minterms formalize queries. ■

Definition 2.3 *Object Pointer join* ($\uparrow$) between a fragment F_o^p of an owner class C_o and a member class C_m returns the set of objects in the member class which are pointed to by objects in the fragment of the owner class F_o^p. ■

Definition 2.4 *Instance Object join* ($\odot$) between a pointer to an instance object of a superclass and an instance object (I_j) of a class C_i returns the aggregate of instance objects of the two classes that represent the actual instance object of the class. ■

To illustrate the aggregate returned by the object join function, we consider the following example. In a database with *Student* a superclass of *Grad*, an instance object I_3 of *Grad* is represented as (Student pointer5) $\odot$ {537msc11,Mary Smith}. This means that the actual I_3 of *Grad* is the quantity representing the instance object I_5 of the superclass *Student* added to the quantity {537msc11,Mary Smith}. *A method invocation* M_j^i on the objects of a class C_i is a set of simple predicates that describe which objects to access. These simple predicates are represented as $\{Pr_{j1}^i,Pr_{j2}^i,\ldots,Pr_{jn}^i\}$ for class C_i. *Minterm selectivity* is the number of instance objects of the class accessed by a user query specified according to a given minterm predicate.

The algorithms are guided by the intuition that an optimal fragmentation keeps those instance objects accessed frequently together while preserving the inheritance and class composition hierarchies. Secondly, the fragments defined are guaranteed correct by ensuring they satisfy the following correctness rules: Completeness: every instance object belongs to a class fragment. Disjointness: every instance object belongs to only one class fragment. Reconstructibility: the union of all class fragments should reproduce the original class.

The Algorithm (HorizontalFrag)

Input to the fragmentation process consists of the set of applications or user queries, the set of database classes, and the inheritance hierarchy information. The output expected from this fragmentation process is a set of horizontal fragments for all classes in the database. The four-step algorithm required for horizontally fragmenting this first class model consisting of simple attributes and simple methods is called *HorizontalFrag*. Summary of the four steps follow and also appear in [1, 2].

Step One – Define the Link Graph For all Classes in the Database

Generally, the object base information needed by the fragmentation procedure are of two types: **The class lattice** showing the superclass–subclass relationship between all classes. The second is **the dependency graph**[3] which captures the method link or attribute link between any two classes in the database. We capture both the class lattice information and dependence information of the database schema using a link graph. Since only simple attributes and simple methods are used in this first model, the link graph is restricted to class lattice information and is generated as follows. Every subclass C_j depends on its superclass C_i in the link graph and has an arc ($C_j \rightarrow C_i$) inserted in the link graph. Starting from the leaf classes of the class lattice, we define a link from each class to its superclasses.

Step Two - Define Primary Horizontal Fragments of Classes

A primary horizontal fragmentation is defined by the effects of user queries on objects of the owner classes of links. A primary horizontal fragment of a class is a set of objects of this class accessed together by only applications running on this class. Therefore, to get the set of simple predicates on any class C_i, we form the union of all predicates of all methods in all the user queries accessing this class. We generate the set of complete and minimal predicates as done in [6]. From the predicates, we generate the minterm predicates[4] We now use the semantics of the class to eliminate meaningless minterm predicates.

Step Three – Derived Horizontal Fragmentation on Member Classes

A derived horizontal fragmentation is defined on the member classes of links according to its owner. We partition a member class according to the fragmentation of its owner class and define the resulting

[3] An example of a graph depicting dependency information is the class composition hierarchy shown in Kim [4].

[4] Note that, each simple predicate can occur in minterm predicates either in its natural or negated form.

fragment on the attributes and methods of the member class only. Therefore, given a link L where owner(L) = S and member(L) = R, the derived horizontal fragments of R are defined as: $R_i = R \uparrow S_i$, where $1 \leq i \leq w$ and w is the number of fragments defined on R based on the owner. The link between owner and member classes is a pointer reference ($\uparrow$) which generates those instance objects of a member class that are pointed to by instance objects in an owner fragment. For example, an instance object of a subclass points to an instance object of its superclass which is logically part of this subclass instance object. A derived fragment of a member class based on a primary fragment of an owner class is a set of member class objects accessed together by applications running on the owner class.

Step Four – Combining Primary and Derived Fragments

The final horizontal fragments of a class is composed of objects accessed together by both applications running only on this class and those running on its owner classes. Therefore, we must determine the most appropriate primary fragment to merge with each derived fragment of every member class. A precise approach is described that captures the environment and attempts to use it in merging derived fragments with primary fragments. The approach is to select a primary fragment of a member class that is most appropriate to combine with a derived fragment of the same class. The primary fragment of choice is the one that has highest affinity with the current derived fragment by applying affinity Rule 2.1. If the class has no primary fragments, the derived fragments are made the final horizontal fragments and instance objects not yet contained in a derived fragment are placed in one of the fragments.

Capturing the quantitative information requires a few additional definitions. First, we define the access frequencies of an object as:

Definition 2.5 *Access frequency acco(I)* of an object (I) is the sum of the access frequencies of all the applications $\mathcal{Q}_i$ accessing the object. Thus, acco(I) = $\sum_i^q acc(q_i, I)$ for all q applications accessing the object. ■

This definition can be used to define the number of relevant and irrelevant accesses to an object with respect to another fragment.

Definition 2.6 *Relevant accesses(F^d,F^p)* are those made to local objects of both fragments F^d and F^p and are defined as all of the access frequencies of objects belonging to both the derived and primary fragments. Thus, Relevant accesses(F^d,F^p) = $\sum_{I_i}$ acco(I_i), for all $I_i \in (F^d \cap F^p)$. ■

Definition 2.7 *Irrelevant access(F^d,F^p)* are those made to instance objects which are in the primary fragment, F^p, or the derived fragment, F^d, but not in both. Irrelevant access(F^d,F^p) = $\sum_{I_j}$ acco(I_j), for all $I_j \in ((F^p \cup F^d)$ - $(F^p \cap F^d))$. ■

Definition 2.8 *Affinity between a derived Fragments F^d and a primary fragment F^p of the same class, aff(F^d,F^p)* is a measure of how frequently objects of these two fragments are needed together by applications.
aff(F^d,F^p) = Relevant access(F^d,F^p) − Irrelevant access(F^d,F^p). ■

Definition 2.9 *The Object Affinity $affo(I_i, I_j)$*, the affinity between two objects I_i and I_j is the sum of each object's accesses (affinity) for all queries that access both objects.

$$affo(I_i, I_j) = \sum_{\{q_k \| q_k \in \mathcal{Q} \wedge q_k \,\text{access} I_i \wedge I_j\}} (acc(q_k, I_i) + acc(q_k, I_j))$$

■

Definition 2.10 *Affinity between an object I_i and a fragment F, faff(I_i,F)* is the sum of object affinities between this object I_i and all objects of the fragment F. Thus, faff(I_i,F) = $\sum_{I_j \in F}$ affo(I_i,I_j), $i \neq j$. ■

To determine the primary horizontal fragment in the member class that is most suitable to merge with a particular derived fragment in the member, we define the following rule.

Affinity Rule 2.1 Select the primary fragment that maximizes the affinity measure aff(F^d, F_i^p) where F^d is the derived fragment and F_i^p is a primary fragment in the class ranging over all candidate fragments. This is the primary fragment that has the highest affinity with this derived fragment. ■

This rule selects the most suitable primary fragment to merge with the derived fragment. The merging process results in some overlap of objects in a set of primary fragments. Disjointness requires that an instance object appearing in more than one fragment is not permitted so a technique is required to determine the object's best location. Our algorithm uses an object's affinity to its fragments to determine the best final placement for the object. This is accomplished with the following rule.

Affinity Rule 2.2 The primary fragment F_j^p that maximizes the function faff(I_i, F_j^p) is where I_i is placed. ■

The final step of this process converts derived fragments to primary for a class that has no primary fragments due to application access pattern.

2.2 Complex Attributes and Complex Methods

Horizontal fragmentation of classes consisting of objects with complex attributes using complex methods requires that we know *à priori* those objects accessed by a method invocation and that encapsulation is not violated. The following additional definitions are needed for this model.

Definition 2.11 *Object reference set of method M_j^i* denoted oref(M_j^i) contains all objects referenced by method M_j^i of some object in class C_i. ■

We can enumerate the objects in oref(M_j^i) by $\{I_1^{k1}, I_2^{k2}, \ldots, I_m^{kn}\}$ where I_m^{kn} is the m^{th} object in class fragment C_{kn}.

Definition 2.12 *Class fragment object reference set coref($F_k^{c_i}$,C_m)* are objects of class C_m referenced by methods of class fragment F_k^i of class C_i. This is derived from oref(M_j^i) as :
coref$(F_k^i, C_m) = \bigcup_{I_k^m \in oref(M_{jn}^i)} I_k^m$ for all M_{jn}^i, $1 \leq n \leq$ card(M_{jn}^i). ■

Definition 2.13 *Number of references* of method M_j^i to an object I_n^k of class C_k is denoted numref(M_j^i, I_n^k) which is the number of times method M_i^i accesses object I_n^k. ■

Definition 2.14 *Class number of reference cnumref(F_k^i,I_n^k)* of class fragment F_k^i on object I_n of class C_k, is the sum of all accesses made to this object by all methods of this class fragment. Thus, cnumref(F_k^i, I_n^k) $= \sum_{M_j^i \in C_i}$ numref(M_j^i, I_n^k). ■

Definition 2.15 *The Method Object Affinity maffo(I_i,I_j)* is the sum of accesses made by any methods of a class fragment or applications to both objects I_i and I_j.

$$\text{maffo}(I_i, I_j) = \text{(application accesses to both objects)} + \text{(method accesses to both objects)}$$
$$\text{maffo}(I_i, I_j) = \sum_{\{q_k \| q_k \in \mathcal{Q} \wedge q_k \text{ access} I_i \wedge I_j\}} (acc(q_k, I_i) + acc(q_k, I_j)) +$$
$$\sum_{\{M_j^{c_k} \| M_j^{c_k} \in C_k \wedge M_j^{c_k} \text{ access} I_i \wedge I_j\}} (numref(M_j^{c_k}, I_i) + numref(M_j^{c_k}, I_j))$$

■

Definition 2.16 *Method Access frequency macco(I)* of an object (I) is the sum of the access frequencies of all the applications $\mathcal{Q}_i$ accessing the object plus the sum of all method references to the object. Thus, macco(I) $= \sum_i^q acc(q_i, I) + \sum_{\forall M_j^i | I \in oref(M_j^i)} numref(M_j^i, I)$. ■

Definition 2.17 *Method relevant access MRA(F^d,F^p)* measures the number of times objects of both fragments are accessed by methods of other objects. Thus, MRA$(F^d, F^p) = \sum_{I_k \in \{F^d \cap F^p\}}$ numref(M_j^i, I_k) for any M_j^i. ■

Definition 2.18 *Method Irrelevant access MIA(F^d,F^p)* measures the number of times objects members of either of the fragments but not both are accessed by methods of other objects.
MIA$(F^d, F^p) = \sum_{I_k \in \{((F^p \cup F^d) - (F^d \cap F^p))\}}$ numref(M_j^i, I_k) for all M_j^i. ■

Definition 2.19 *Object Fragment Affinity* between an object I_i and a fragment F, denoted ofaff(I_i,F), is the amount of accesses made to a fragment *vis a vis* an object.
ofaff(I_i,F) = $\sum_{I_j \in F}$ maffo(I_i,I_j), i $\neq$ j. ■

Definition 2.20 *Method Affinity between Fragments maff(F^d,F^p)* is the difference between relevant and irrelevant access: maff(F^d,F^p)= MRA(F^d,F^p) - MIA(F^d,F^p). ■

Affinity Rule 2.3 Select the primary fragment that maximizes the method affinity measure maff(F^d, F_i^p) where F^d is the derived fragment generated from method dependencies and F_i^p is a primary fragment in the class ranging over all candidate fragments. ■

Affinity Rule 2.4 The primary fragment F_j^p that maximizes the function ofaff(I_i, F_j^p) is where I_i is placed. ■

Achieving optimal fragmentation requires that objects which invoke methods on a set of other related objects should be contained within the same fragment. Obviously, if this can be achieved it will be possible to allocate highly related objects together. This goal is reflected in the above definitions because we can develop an algorithm that maximizes local relevant access and minimizes local irrelevant access.

Thus three dependence types between classes are captured in a complete design with the most complex object model, formally defined as algorithm Hor_CA_CM, of Figure 1. The inheritance relationship between classes is captured in the fragmentation scheme in the first iteration by propagating the fragmentation of a subclass to a superclass so that derived fragments of a superclass, based on the fragments of its subclasses, is obtained (Line 1). The second type of dependence is from the complex hierarchy. If a class C_i contains another class C_j as part of it, the fragmentation of the contained class C_j is used to obtain derived fragments of the containing class C_i. This dependence is captured in the second iteration of the algorithm (Line 2). The third iteration captures the method dependence information. If objects of a fragment of a class C_i invoke methods in objects of another class C_j, we want to group all objects of the class C_j invoked by all objects of the fragment of C_i into a derived fragment of C_j, so that these two fragments are allocated together later. The first and second iterations of this algorithm involve generating a set of horizontal fragments that capture the inheritance hierarchy and attribute link information using the same technique presented in Section 2. During the third iteration, the method link information is captured by generating a set of derived fragments as described in the following narrative. The first step is to take every class and create a set of objects referenced by each class's fragments, called the *derived fragments*. Having generated derived fragments arising from complex methods, we proceed to determine which primary fragment is appropriate to merge with each derived fragment using the same merge technique described in Section 2.1 with the simple class model. Finally, since it is possible to place an object in more than one primary fragment, Affinity Rule 2.2 is used to find the primary fragment with which it has the greatest affinity and it is removed from all other.

3 Conclusions

This paper defines issues involved in the class fragmentation in a distributed object based system. The issues considered include the inheritance hierarchy, the nature of attributes of a class, and the nature of methods in the classes. The paper proposes algorithms for horizontal fragmentation of advanced class object model: consisting of classes with complex attributes and complex methods. The algorithm necessary to support the first simple model is summarized. A description and formal algorithm showing the iterative applications and extensions of this algorithm for fragmentation of the more complex models are further provided. An interesting feature of this design is reusability of design for a simpler class model in a more complex class model. The algorithm necessary to support the first simple model is summarized. A description and formal algorithm showing the iterative applications and extensions of this algorithm for fragmentation of the most complex model consisting of complex attributes and methods, are presented.

Current research efforts include developing performance measurements to demonstrate the utility of our algorithms. Such performance analysis is difficult because very few of these systems (or more

Algorithm 2.1 *(Complex Attribute and Complex Method)*

Algorithm Hor_CA_CM
input: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$: set of database classes
$\mathbf{L(C)}$: the class lattice
$\mathbf{A(C)}$: the class composition hierarchy with attribute link between classes
$oref(M_i^j)$: set of object references for all $M_i^j \in C_i$.
output: $\mathcal{F}_{c_i}$: set of horizontal fragments of the classes
var
$\mathcal{P}_i$: set of simple predicates for class C_i
$\mathcal{F}_i^p$: set of primary horizontal fragments for class C_i
$\mathcal{F}_i^d$: set of derived horizontal fragments for class C_i
$\mathcal{M}_c^L$: set of member classes for the link graph L(C)
begin
$\mathcal{F}_{c_i}$ = **HorizontalFrag**($\mathcal{Q}_i$, $\mathcal{C}_d$, $\mathbf{L(C)}$) (1)
$\mathcal{F}_{c_i}$ = **HorizontalFrag**($\mathcal{Q}_i$, ($\mathcal{F}_{c_i}$, $\mathbf{A(C)}$) (2)
for each class $C_i \in C_d$ **do** (3)
Get Derived Fragments from Complex Methods and Merge
using $oref(M_i^j)$ **and current fragments of the class** (4)
end; {**for** C_i}
end; {**Hor_CA_CM**}

Figure 1: Horizontal Fragmentation For Complex Attribute and Complex Methods

specifically, applications on these systems) have been developed. Ideally these techniques can be modified so they can be used in a dynamic environment where data is added and removed. The initial step in this research requires that we determine a performance threshold below which dynamic redesign is required so the system will continue to meet its performance goals. The current research is attempting to determine if these techniques can be modified so that each iteration of the design process can be accomplished by only analyzing new data added to the system while updating those fragments that had been previously allocated. Furthermore, we are working on vertical and hybrid fragmentation schemes for these complex class models.

References

[1] C.I. Ezeife and K.E. Barker. Horizontal class fragmentation in a distributed object based system. Technical Report TR 93-04, Univ. of Manitoba Dept of Computer Science, October 1993.

[2] C.I. Ezeife and K.E. Barker. Horizontal class fragmentation in distributed object based systems. In *Proceedings of the Second Biennial European Joint Conference on Engineering Systems Design and Analysis*. ASME Publications, 1994.

[3] K. Karlapalem, S.B.Navathe, and M.M.A.Morsi. Issues in distribution design of object-oriented databases. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 148–164. Morgan Kaufmann Publishers, 1994.

[4] W. Kim. Object-oriented databases: Definition and research directions. *IEEE Transactions on knowledge and Data Engineering*, 2(3), Sept. 1990.

[5] S.B. Navathe, K. Karlapalem, and M. Ra. A mixed fragmentation methodology for initial distributed database design. In *Technical Report*. CIS Dept, Univ. of Florida, Gainesville, FL, 1990. TR 90-17.

[6] M.T. Ozsu and P.Valduriez. *Principles of Distributed Database Systems*. Prentice Hall, 1991.

[7] G. Wiederhold. *Database Design.* McGraw-Hill, New York, 1982.

[8] S.B. Yao, S.B. Navathe, and J.L. Weldon. An integrated approach to database design. New York, 1982. Lecture Notes in Computer Science 132.