Vertical 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

Tel.: (204) 474-8832

FAX: (204) 269-9178

{christie,barker}@cs.umanitoba.ca

Abstract

Many object based systems exist that support some form of distribution. Optimal application performance on distributed object based systems demands accurate class fragmentation and the subsequent allocation of these fragments to distributed sites. Vertical fragmentation must minimize application execution time by splitting a class so that all class attributes and methods frequently accessed together are grouped together into a single fragment. This paper describes a fragmentation algorithm that statically creates a set of fragments for the most complex object model: namely, one that supports inheritance and includes a part–of hierarchy, and whose method invocation structure is based on a nested transaction model. Our approach consists of grouping into a fragment, all attributes and methods of the class frequently accessed together by applications running on either this class, its descendant classes, its containing classes or its complex method classes.

*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.

1 Introduction

Distributed database design is a two step process. First, database entities are fragmented and secondly, the fragments are allocated to distributed sites. Two approaches are used in distributed database design - top-down and bottom-up. The top-down approach entails generating a set of local conceptual schemas from a global conceptual schema (GCS) and the access pattern information. The GCS describes the global database entities and their relationships while the LCS describes the database entities at each local site and their relationships [9]. The input to the design process is obtained from an earlier static and system requirements analysis which defines the environment of the system and collect an approximation of both the data and processing needs of all potential database users. Our entity of distribution is a class fragment and the top-down approach is used.

A distributed object based system (DOBS) is a collection of local object bases distributed among different local sites, interconnected by a communication network. A DOBS supports an object oriented data model including features of encapsulation and inheritance. The data in a DOBS consists of a set of encapsulated objects. The data values (attribute values) are bundled together 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 *subclasses*. The database contains a root class called *Root*, and Root is an ancestor of every other class in the database. The overall inheritance hierarchy of the database is captured in a class lattice. 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]. There is an object identifying attribute *oid* which is a member of the set of attributes $\mathcal{A}$. The *oid* could be either a system defined object identifier or a user-defined key attribute. Vertical fragmentation is the process of breaking up of a class into a set of possibly smaller classes called vertical fragments. Each object in a vertical fragment is a portion of the original object in the original class. In effect, each vertical fragment ($\mathcal{C}^v$) of a class contains its class identifier, and all of its instance objects for only some of its methods ($\mathcal{M}' \subseteq \mathcal{M}$) and some of its attributes ($\mathcal{A}' \subseteq \mathcal{A}$). Thus, $\mathcal{C}^v$ = (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. Complex attributes have domains in another class. This is often referred to as a

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

"part–of" or composition hierarchy. 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 [5]. Although two basic method types exist, a simple method of a contained (part-of) class is referred to as a 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, and class models consisting of complex attributes and complex methods. This classification enables us accommodate all the necessary features of object orientation and provide solutions for object bases that are structured in various ways. Distributed object based design enhances performance by organizing database entities in fragments such that the amount of irrelevant data accessed by applications is reduced while reducing the amount of data that needs to be transferred between sites.

Many distributed and client/server object based systems exist which will benefit from fragmentation [7]. 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.

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 80% of the total data accesses, this analysis is feasible [9]. However, 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 can be incorporated into a dynamic system.

This paper reviews possible DOBS models as initially presented in [5], and contributes by presenting algorithms for vertically fragmenting the most complex class model consisting of complex attributes with complex methods. The balance of the paper is organized as follows. We complete this section by briefly reviewing previous work on distributed database design. Section 2 presents vertical fragmentation algorithm for class model consisting of complex attributes and complex methods. Finally, Section 3 concludes and suggests future research directions.

1.1 Related Work

Algorithms that fragment *relations* horizontally and vertically exist. Previous work on relational vertical fragmentation is reviewed and then previous work on fragmentation in DOBS.

Vertical Fragmentation (relational): Work on vertical fragmentation in the relational data model includes Hoffer and Severance [6], Navathe *et al.* [8], Cornell and Yu [2], Özsu and Valduriez [9] and Chakravarthy *et al.* [1].

Hoffer and Severance [6] define an algorithm that clusters attributes of a database entity based on their affinity. Attributes accessed together by applications have high affinity so the Bond Energy Algorithm [9] is used to form these attribute clusters. Navathe *et al.* [8] extends Hoffer's work by defining algorithms for grouping attributes into overlapping and nonoverlapping fragments. Cornell and Yu [2] optimized this work by developing an algorithm that obtains an optimal binary partitioning for relational databases. Özsu and Valduriez [9] discuss this earlier work on vertical partitioning for distributed databases using the access frequency information and the Bond Energy Algorithm. Chakravarthy *et al.* [1] argue that earlier algorithms for vertical partitioning are *ad hoc*, so they propose an objective function called the Partition Evaluator to determine the "goodness" of the partitions generated by various algorithms.

Vertical Fragmentation (objects): Karlapalem *et al.* [7] define issues involved in distribution design for an object oriented database system. They identify two types of methods – simple and complex methods. They argue that a model consisting of simple methods can be vertically partitioned using techniques described by Navathe *et al.* [8], while complex methods require a method-based view (MBV). The MBV identifies the set of objects accessed by a method and the set of attributes or instance variables accessed by the method. The sets are further grouped into sets of objects and instance variables based on the classes to which they belong. This generates the set pairs of objects and instance variables (O_i,I_i) accessed from a class C_i by a method. This is called method m_j's view of class C_i. They suggest the use of concepts developed by Pernul *et al.* [10] to fragment classes based on views.

2 Vertical Fragmentation of Classes – Complex Attributes and Complex Methods

This section presents an algorithm for vertically fragmenting classes consisting of complex attributes and complex methods. The database information needed is: the inheritance hierarchy, the attribute link to reflect part-of hierarchy, and the method links to reflect the use of methods of objects of class C_i (being fragmented) by objects of other classes. This algorithm is built on the algorithms used by simpler models [3]. With this class model, vertical fragmentation aims at splitting a class such that all attributes and methods of the class most frequently accessed together by user applications are grouped together. User applications that access attributes and methods of the class are of the following types: (1) those running directly on this class, (2) those running on descendants of this class, (3) those running on containing classes which use this class as a type for their attributes, and (4) those running on complex methods of other classes in the database that use methods of this class. Vertical fragmentation aims at splitting a class so all attributes and methods of the class most frequently accessed together by user applications are grouped together. *Encapsulation* means user applications do not directly access an objects' attribute values except through the objects' methods. Since every method in the object accesses a set of attributes of the class, we first group only methods of the class based on application access pattern applying the same technique used in [6, 8]. Secondly, we extend each method group (fragment) to incorporate all attributes accessed by methods in this group. A problem with this second step is some attributes may belong to the reference set of more than one method, so deciding which method group these attributes belong in, so only non-overlapping vertical fragments are generated, is required. Two alternative approaches for handling this conflict are: (1) use a set of affinity rules similar to those used in horizontal fragmentation schemes presented in [5] to decide which fragment to place these overlapping attributes or (2) from the outset, group both attributes and methods using attribute/method affinity. We adopt the first approach because it drastically reduces the size of the matrices used as input to the Bond Energy and Partitioning algorithms of [8]. It also exploits the abstraction power of the object-oriented data model and thus performs better when there is low overlap in the attribute reference sets of the methods of a class. Thus, we want to place in one fragment those methods of a class usually accessed together by applications. The measure of "togetherness" is the affinity of methods which shows how closely related the methods are. We first present some definitions and assumptions before presenting the algorithm. The explicit assumptions are:

1. 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.

2. Application and database information are pre-determined prior to the fragmentation process.

The major data requirements related to applications is their access frequencies. Let Q = $\{q_1, q_2, \ldots, q_q\}$ be the set of user queries (applications) running object methods from the set of all methods denoted $\{M^{i1.j}, M^{i2.k}, \ldots, M^{in.p}\}$. Then, for each query q_k and each method $M^{i.j}$ (the jth method of class C_i), we associate a method usage value denoted as use(q_k, $M^{i.j}$) where use(q_k, $M^{i.j}$) = 1 if method $M^{i.j}$ is referenced by query q_k, 0 otherwise. Thus, for each class, we define a method usage matrix. 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 C_i is denoted by $M^{i.j}$ and a user query q_k is represented by $\{M^{i1.j}, M^{i2.k}, \ldots, M^{in.p}\}$ where each M in a user query refers to an invocation of a method of a class object. ■

Definition 2.2 *Method Attribute Reference MAR($M^{i.j}$)* of a method $M^{i.j}$ of a class C_i is the set of all attributes of C_i referenced by $M^{i.j}$. ■

Definition 2.3 *A null method of a class C_i, denoted NM(C_i)*, with respect to a superclass C_s is a place holder for the superclass's method. A null method is denoted by: (original class.method name) where original class is the name of the superclass and method name is the method in the subclass. ■

Definition 2.4 *An Extended Method of a class, C_i, $(EM)^{i.k}$* is either an original method of the class, $M^{i.j}$ or a null method of the class $NM(C_i)$.
Thus, $(EM)^i = M^{c_i} | NM(C_i)$. ■

In effect, the extended method set of a class is the union of its actual methods and its null methods (null methods are used to refer to inherited methods).

Definition 2.5 *Access frequency of a query* is the number of accesses a user application makes to "data". 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 d_j where data item d_j can be a class, a fragment of a class, an instance object of a class, an attribute or method of a class. ■

Definition 2.6 *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 "complete" instance object consisting 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) $\odot$ {Grad3,Mary Smith}. This means that the actual I_3 of *Grad* is the quantity representing the instance object I_5 of the superclass *Student* combined with the quantity {Grad3,Mary Smith} from *Grad* per se.

The next four definitions may be used for computing the method affinity matrix of the class being fragmented. The method affinity matrix is the matrix to be clustered and gives the affinity values between extended method pairs of the class being fragmented. While subclass affinity measures the access of the extended methods through the descendant classes of this class, the method affinity value accounts for the use of this method pair both through the descendant classes and directly on the class. The containing class affinity is needed for complex hierarchy to incorporate the use of the extended method pairs of the class being fragmented through its containing classes, while complex method affinity incorporates their use through its complex method classes.

Definition 2.7 *Subclass affinity of two extended methods* $(EM)^{i.j}$,$(EM)^{i.k}$ *of a class* C_i*, denoted saff(*$(EM)^{i.j}$,$(EM)^{i.k}$*)* is a measure of how frequently methods/attributes of the subclasses or descendant classes of C_i and methods/attributes of the class are needed together by applications running at any particular site. saff$((EM)^{i.j}, (EM)^{i.k}) =$ $\sum_{o=1}^{w} \sum_{p||use(q_p,(EM)^{i.j})=1 \wedge use(q_p,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_p)acc_l(q_p)$,
where w is the number of subclasses, p is some application and S_l ranges over all sites. ■

Definition 2.8 *Method Affinity between two extended methods of a class* C_i*,* *MA(*$(EM)^{i.j}$,$(EM)^{i.k}$*)* measures the bond between two extended methods of a class according to how they are accessed by applications.
MA$((EM)^{i.j}, (EM)^{i.k}) =$
$(\sum_{p||use(q_p,(EM)^{i.j})=1 \wedge use(q_p,(EM)^{i.k}=1} \sum_{\forall s_l} ref_l(q_p)acc_l(q_p)) + saff((EM)^{i.j}, (EM)^{i.k})$
where $ref_l(q_p)$ is the number of accesses to methods $((EM)^{i.j}$,$(EM)^{i.k})$ for each execution of application q_p at site s_l and $acc_l(q_p)$ is the application access frequency modified to include frequencies at different sites. This generates the method affinity matrix (MA), an n * n matrix. ■

Definition 2.9 *Containing Class affinity between two extended methods* $(EM)^{i.j}$ *and* $(EM)^{i.k}$ *of a class* C_i*,* $ccaff((EM)^{i.j}, (EM)^{i.k})$ is a measure of how frequently methods/attributes of containing classes of C_i and methods/attributes of the class are needed together by applications running at any particular site.
ccaff$((EM)^{i.j}$,$(EM)^{i.k}) = \sum_{o=1}^{w} \sum_{k||use(q_k,(EM)^{i.j})=1 \wedge use(q_k,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_k)acc_l(q_k)$
where w is the number of containing classes, C_i is the class and S_l ranges over all sites. ■

Definition 2.10 *Complex Method affinity between two extended methods,* $(EM)^{i.j}$ *and* $(EM)^{i.k}$ *of a class* C_i*, cmaff(*$(EM)^{i.j}$*,*$(EM)^{i.k}$*)* measures how frequently methods/attributes of other classes in the database and method/attributes of the class C_i are needed together by applications running at any particular site.
cmaff($(EM)^{i.j}$,$(EM)^{i.k}$) $= \sum_{o=1}^{d} \sum_{k||use(q_k,(EM)^{i.j})=1 \wedge use(q_k,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k)$
where d is the number of database classes, C_i is the class and S_l ranges over all sites. ■

When one attribute becomes a member of more than one vertical fragment, we need to decide with which fragment it has the highest affinity. The next three definitions are used to compute these affinities. While attribute/attribute affinity (AAA) computes the binding of this overlapping attribute with other attributes of a fragment, the attribute/method affinity binds this attribute with methods in this fragment. Thus, attribute/fragment affinity now becomes the combination of the affinities between the attributes and methods of the fragments.

Definition 2.11 *Attribute/Attribute Affinity* AAA($A^{i.j}$,$A^{i.m}$) between two attributes of the same class C_i is the sum of the access frequencies of all methods accessing these two attributes together at all sites. AAA($A^{i.j}$,$A^{i.m}$) =
$\sum_{k|A^{i.j} \in MAR(M^{in.k}) \wedge A^{i.m} \in MAR((M^{in.k})} \sum_{l=1}^{m} acc_l(M^{in.k}, A^{i.j}) + acc_l(M^{in.k}, A^{i.k})$
where $acc_l(M^{in.k}, A^{i.j})$ is the number of accesses made to the attribute $A^{i.j}$ by method $M^{in.k}$ at site s_l. ■

Definition 2.12 *Attribute/Method Affinity* AMA($A^{i.j}$,$M^{i.m}$) between an attribute and a method of the same class C_i is the sum of the access frequencies of all methods using this attribute and this method together at all sites. AMA($A^{i.j}, M^{i.m}$) =

$$\sum_{k|A^{i.j} \in MAR(M^{s.k}) \wedge M^{i.m} \in MMR(M^{s.k})} \sum_{l=1}^{m} acc_l(M^{s.k}, A^{i.j}) + acc_l(M^{s.k}, M^{i.m})$$

where $M^{s.k}$ belongs to some class C_s. ■

Definition 2.13 *Attribute Fragment Affinity AFA(*$A^{i.m}$*,*$F^{i.j}$*)* is a measure of the affinity between attribute $A^{i.m}$ and vertical fragment $F^{i.j}$, and is the sum of all the attribute/attribute and attribute/method affinities between $A^{i.m}$ and all attributes and methods of the class fragment $F^{i.j}$. AFA($A^{i.m}, F^{i.j}$) =
$\sum_{k|A^{i.m} \in F^{i.j} \wedge A^{i.k} \in F^{i.j})} AAA(A^{i.m}, A^{i.k}) + \sum_{k|A^{i.m} \in F^{i.j} \wedge M^{i.k} \subset nF^{i.j})} AMA(A^{i.m}, M^{i.k})$. ■

After generating non-overlapping method fragments, it is possible to obtain overlapping fragments when attributes referenced by methods in the fragments are included. Since our objective is to make the final method/attribute fragments non-overlapping, a technique is needed to decide in which fragment it is most beneficial to keep an overlapping attribute.

Affinity Rule 2.1 Place the overlapping attribute $A^{i.j}$ in the fragment $F^{i.k}$ with maximum AFA($A^{i.j}$,$F^{i.k}$) since this is the vertical fragment with which attribute $A^{i.j}$ has highest affinity. ∎

The proposed algorithm is guided by the intuition that an optimal fragmentation keeps those attributes and methods accessed frequently together while preserving the inheritance, aggregation and method nesting hierarchies. Secondly, the fragments defined are guaranteed correct by ensuring they satisfy the correctness rules of completeness, disjointness and reconstructibility. Completeness requires that every attribute or method belongs to a class fragment, while disjointness means every attribute or method belongs to only one class fragment. Finally, reconstructibility requires that the union of all class fragments should reproduce the original class.

The algorithms also require the following data structures and functions.
$\mathcal{M}^{c_i}$: set of all methods of class C_i.
$\mathcal{C}_i^{des}$: set of all descendant classes of C_i.
$\mathcal{C}_i^{cont}$: set of all containing classes of C_i.
$\mathcal{C}_i^{cmeth}$: set of all complex method classes of C_i.
EM-set(C_i^{des}) : set of extended methods of C_i and its descendant classes.
NM-set(C_i^{des}) : set of null methods of C_i and its descendant classes.
EMapplic-set(C_i^{des}) : set of applications accessing extended methods of C_i and its descendant classes.
L(C_i) : a tree rooted at node (class) C_i.
MAR-set(C_i): set of method attribute references of of all methods of the class C_i.
AF-set(C_i^{des}): set of application frequency matrices of the class and its descendants.
MU-set(C_i^{des}): set of method usage matrices for class C_i and its descendant classes.
UsageMtrx(L(C_i^{des}**))** : a function that returns the original method usage matrices for the class C_i and its descendant classes.
children(C_k) : a function that returns the immediate children of the node (class) C_k.

Thus, in vertically fragmenting this class model, we generate the method affinity matrix of the class iteratively in three increments as follows:

1. This initial method affinity matrix is generated using method usage and application frequency matrices of the class and its descendant classes.

2. The method affinity matrix is modified using method usage and application frequency matrices of the class and its containing classes.

3. It is further modified using method usage and application frequency matrices of the class and its complex method classes.

The steps for generating vertical fragments of classes consisting of complex attributes and complex methods are given below.
Steps

R1. Generate initial Method Affinity matrix for the class as:

a. Obtain the method usage and application frequency matrices of the class and its descendants (use algorithm *UsageMtrx* [4]). This incorporates the inheritance link information using the class lattice with the objective of grouping together the set of methods/attributes of class C_i that are used by applications running on its descendants. The algorithm *UsageMtrx* accepts a tree rooted at a class C_i and generates the original method usage matrices for class C_i and other classes (subclasses of C_i in this case) on the tree. To compute the method usage matrix of a class C_i on the tree, it assigns 1 to the matrix element identified by (row q_j, column $(EM)^i_k$) for some application q_j in the object base and some extended method $(EM)^i_k$ of the class, if use(q_j, $(EM)^i_k$) $= 1$, and 0 otherwise.

b. Define the method affinity matrix of the class from step (R1a) using the usage matrices and application frequency matrices of the class as defined in algorithm *UsageMtrx*. By doing this, we are producing an initial method affinity matrix of the class that includes use of its methods by methods of its descendant classes.

R2. Modify the method affinity matrix from step R1 to include use of the methods through its containing classes.

a. We repeat the operations in step R1 above, using a different type of relationship. Obtain the method usage and application frequency matrices of the class and its containing classes using the algorithm *UsageMtrx* with the Linkgraph [4] that returns a tree rooted at the class showing the attribute link between that class and other classes in the database.

b. Modify method affinity matrix of the class from step (R1b) using the usage matrices and application frequency matrices of the class and its containing classes (from R2a).

R3. Modify method affinity matrix from step R2 above to include usage of methods through complex method classes.

a. We repeat the operations in step R2 above, using a different type of relationship – the complex method link. We first produce a link graph which is a tree rooted at the class being fragmented C_i, that links it to all other classes in the object base whose complex methods are represented in the intra class null method set of this class.

b. Modify method affinity matrix of the class from step (R2b) using the usage matrices and application frequency matrices of the class and its complex method classes (from R3a). This requires adding the cmaff of each method pair to their current method affinity value.

R4. Use the Bond Energy Algorithm developed [9] as presented in [8, 9] to generate clustered affinity matrix of the class. This algorithm accepts method affinity matrix as input and permutes its rows and columns to generate a clustered affinity matrix. The clusters are formed so that methods with larger affinity values are together and the ones with smaller values are also together.

R5. Modify the original method usage matrix to include method usages through complex class and complex method classes.

a. Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* [4] which modifies the method usage matrix of a class C_i to include a row for every application q_j that accesses a null method representative of this class at all its descendant classes.

b. Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* which modifies the method usage matrix of a class C_i to include a row for every application q_j that accesses a null method representative of this class at all its containing classes

c. Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* which modifies the method usage matrix of a class C_i to include a row for every application q_j that accesses a null method representative of this class at all its complex method classes.

R6. Use method-attribute reference information of the methods in each method fragment (MAR of definition 2.2) to include in each method fragment all attributes of the class accessed by methods of the fragment.

R7. Since there may be problems of overlapping attributes in more than one fragment, use Attribute Placement Affinity Rule 2.1 to decide which vertical fragment to keep each overlapping attribute.

The formal algorithm for vertically fragmenting a class consisting of complex attributes and complex methods is presented as algorithm *Vert_CA_CM* of Figure 1. This algorithm is the same as the *VerticalFrag* algorithm [3] of the model with simple attribute and simple method except that the method affinity matrix includes in addition to subclass affinity (saff) between the extended methods, complex class affinity (ccaff) as well as complex method affinity (cmaff). Secondly, the modified method usage matrix used for the partitioning includes additional rows to account for method usage of the class being fragmented by methods of containing classes and complex method classes. The *VerticalFrag* algorithm for the simplest model generates the method affinity matrix of a class C_i to be fragmented using the method usage and application frequency matrices of C_i and all its descendant classes. It then modifies the method usage matrix of C_i to account for usage of C_i's methods through its descendant classes. It then generates vertical method fragments of C_i using the binary partition algorithm with modified usage and method affinity matrices of C_i as input.

We simplify the presentation of the algorithm *Vert_CA_CM* for the most complex model by defining the vertical fragmentation algorithm in terms of the two key matrices of the class C_i being fragmented: the method affinity and the modified method usage matrices of the class. These two matrices constitute the major final inputs to the vertical fragmentation scheme before fragments are produced. Thus, the major difference in the various schemes for fragmenting various class models lies in how these two matrices are obtained. In describing the procedures involved in obtaining the matrices needed before running the vertical fragmentation algorithm, we shall attach the sequence of modifications performed on the matrix as its arguments. Thus, VerticalFrag(MA(C_i^{des}, C_i^{cont}), $MU(C_i^{des}, C_i^{cont})$ means the method affinity matrix is produced using method usage and application frequency matrices of the class and its descendant classes first, followed by a modification using method usage and application frequency matrices of the class and its containing classes. Similarly, the method usage matrix includes rows to account for method usage of the class by applications running on descendant classes and then containing classes. Running Verticalfrag on the two matrices MA and MU entails submitting MA to the Bond energy algorithm for clustering of methods and then partitioning the clustered MA using MU and the binary partition algorithm. Finally, attributes are included in method fragments to generate non-overlapping fragments.

2.1 An Example

This example incorporates class models consisting of complex attributes and complex methods. The extended complex class object base is as given in Figure 2 [2]. The database schema information consists of the class hierarchy of the object base and is given in Figure 3, while the class composition hierarchy is as in Figure 4.

[2]For readability, we precede each key attribute of a class by k, each attribute by an a and each method by an m.

Algorithm 2.1 *(Vertical Fragments of Complex attributes and Complex Methods)*

Algorithm Vert_CA_CM

input: $\mathcal{Q}^{C_i^{des}}$: set of user queries accessing C_i and its descendant classes.
C_i : the database class to fragment; $\mathbf{L(C)}$: the class lattice
$\mathcal{C}_i^{des}$: set of descendant classes of C_i
icnm(C_i) : set of intra class null methods of C_i.
$\mathcal{C}_i^{cont}$: set of containing classes of C_i
$\mathcal{C}_i^{cmeth}$: set of complex method classes of C_i
$(EM)^{C_i}$: extended method set of C_i.
MAR-set(C_i) : method attribute reference set of methods of C_i
AF-set(C_i^{des}): application frequency matrices of C_i and its descendants.
AF-set(C_i^{cont}): application frequency matrices of C_i and its containing classes.
AF-set(C_i^{cmeth}): applic. frequency matrices of C_i and complex method classes.

output: $\mathcal{F}^{c_i}$: set of vertical fragment s of C_i.

var

MU-set(C_i^{des}) : method usage matrices for class C_i and its descendants.
MU-set(C_i^{cont}) : method usage matrices for class C_i and its containing classes.
MU-set(C_i^{cmeth}) : method usage matrices for class C_i and its complex method classes.
MA^i : method affinity matrix of C_i.
CA^i : clustered affinity matrix C_i.
MU^i : the modified method usage matrix of C_i.

begin

//Generate a set of attribute/method fragments of the class C_i //
// using algorithm VerticalFrag with appropriate method affinity //
// and modified method usage matrices. //

$$\text{VerticalFrag}(\text{MA}(C_i^{des},, C_i^{cont}, C_i^{cmeth}), MU(C_i^{des}, C_i^{cont}, C_i^{cmeth})) \quad (1)$$

end {Vert_CA_CM}

Figure 1: Vertical Fragmentation - Complex Attributes and Complex Methods

Person = {Person,{a.ssno,a.name,a.age,a.address},
{m.ssno-of,m.whatname,m.age-in-year,m.newaddr},
{ I_1 {Person1,John James,30,Winnipeg}
I_2 {Person2,Ted Man,16,Winnipeg}
I_3 {Person3,Mary Ross,21,Vancouver}
I_4 {Person4,Peter Eye,23,Toronto}
I_5 {Person5,Mary Smith,40,Toronto}
I_6 {Person6,John West,32,Vancouver}
I_7 {Person7,Jacky Brown,35,Winnipeg}
I_8 {Person8,Sean Dam,27,Toronto}
I_9 {Person9,Bill Jeans,43,Vancouver}
I_{10} {Person10,Mandu Nom,30,Winnipeg} } }
Prof = Person pointer $\odot$ {Prof,{a.empno,a.status,a.dept,a.salary,a.student},
{m.empno-of,m.status-of,m.students-of,m.whatsalary,m.dept-of},
{ I_1 (person pointer5) $\odot$
{Prof1,asst prof,Computer Sc.,45000,students pointers}
I_2 (person pointer6) $\odot$ {Prof2,assoc prof,Math,60000,students pointers }
I_3 (person pointer9) $\odot$ {Prof3,full prof,Math,80000,students pointers}
I_4 (person pointer10) $\odot$ {Prof4,full prof,Math,82000,students pointers} } }
Student = Person pointer $\odot$ {Student,{a.stuno,a.dept,a.feespd,a.coursetaken},
{m.stuno-of,m.dept-of,m.owing,m.course-taken}
{ I_1 (person pointer1) $\odot$ {Student1,Math,Y,[]}
I_2 (person pointer4) $\odot$ {Student2,Computer Sc.,N,[521,632]}
I_3 (person pointer2) $\odot$ {Student3,Stats,Y,[211]}
I_4 (person pointer3) $\odot$ {Student4,Computer Sc.,N,[111,211]} } } }
Grad = Student pointer $\odot$ {Grad,{a.gradstuno,a.supervisor},
{m.gradno-of,m.whatprog}
{ I_1 (Student pointer1) $\odot$ {Grad1,John West}
I_2 (Student Pointer2) $\odot$ {Grad2,Mary Smith}
I_3 (Student Pointer4) $\odot$ {Grad3,Mary Smith} } }
Dept = {Dept,{a.code,a.name,a.profs,a.students},
{m.code-of,m.name-of,m.number-of-profs,m.students-of},
{ I_1 {Dept1,{Computer Science,prof pointers,student pointers}
I_2 {Dept2,{Math,prof pointers,student pointers}
I_3 {Dept3,{Stats,prof pointers,student pointers} } }

Figure 2: The Complex Sample Object Database Schema

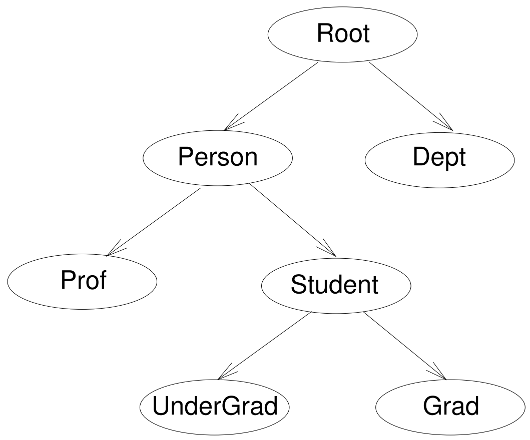


Figure 3: Complex Class Lattice

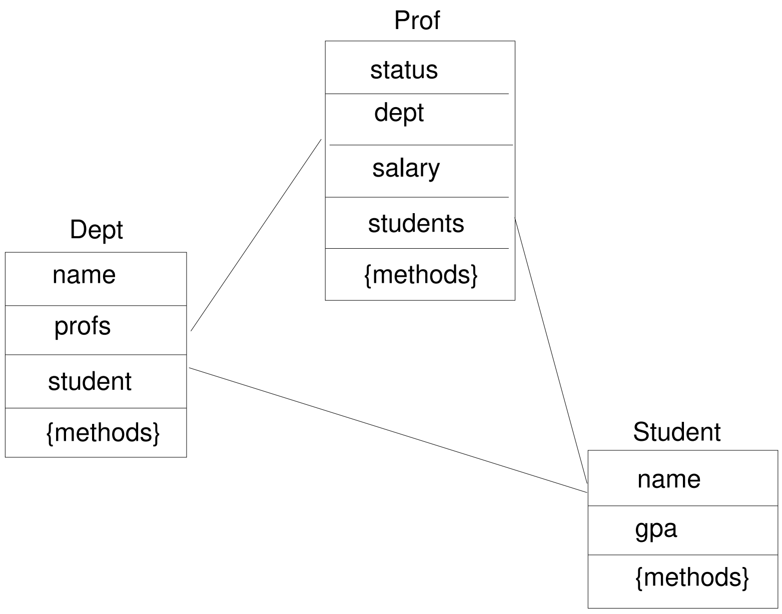


Figure 4: Class Composition Hierarchy

Suppose we want to vertically fragment the class *Student*, a network of method usage and application frequency matrices (obtained from an initial system analysis) needed to compute the method affinity matrix of this class that incorporates *saff*, *ccaff* and *cmaff* is given in Figure 5. The method affinity value of a method pair in the matrix is computed as follows:

MA($M^{student.1}$,$M^{student.2}$) $= \sum_{k=1}^{1} \sum_{l=1}^{3} acc_l(q_1) + saff_{grad}(M^{student.1}, M^{student.2})$.
= (40+0+20) + (0+60+10) = 130 (initial MA).
ccaff($M^{student.1}$,$M^{student.2}$)= $ccaff_{dept}(M^{student.1}, M^{student.2})$ +
$ccaff_{prof}(M^{student.1}, M^{student.2})$.
= [(50 + 15 + 0) + (25 + 40 + 5)] + 0 = 135
MA($M^{student.1}$,$M^{student.2}$) = 130 + 135 = 265 (after first modifi cation)

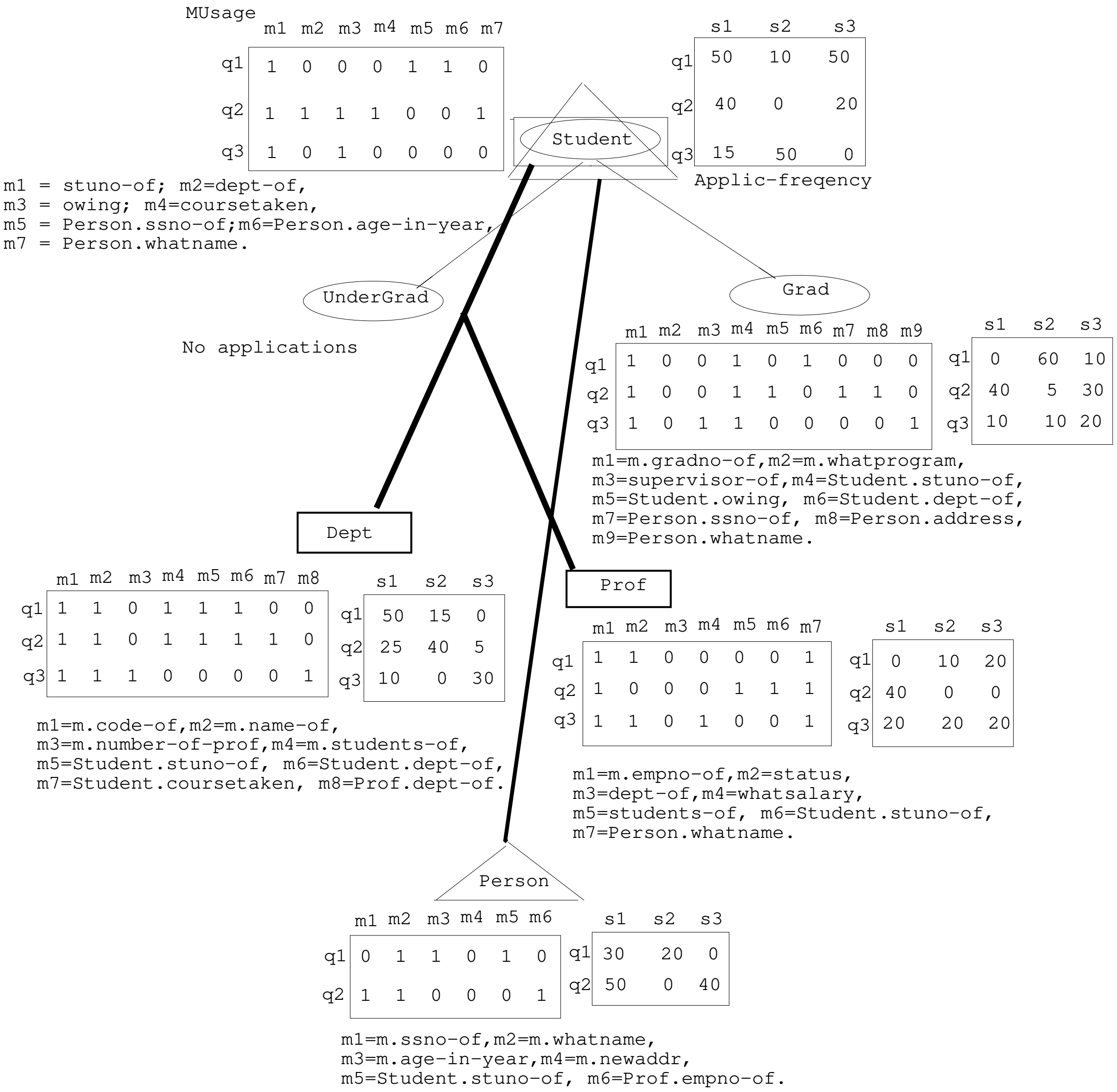
The method affinity, clustered affinity and the modified method usage matrices of the class after executing line 1 of the algorithm 1 are given in Figure 6.

The vertical fragments from the execution of the partition algorithm are $F_1 = \{m_1, m_3, m_4\}$ and $F_2 = \{m_1, m_2, m_5, m_6, m_7\}$.

3 Conclusions

This paper reviews issues involved in class fragmentation in a distributed object based system. The model characteristics incorporated include: the inheritance hierarchy, the nature of attributes of a class, and the nature of methods in the classes. The paper argues that vertical fragmentation algorithms of four types of class object models is required, namely, classes with simple attributes and methods, classes with attributes that support a class composition hierarchy using simple methods, classes with complex attributes using simple methods, and finally classes with complex attributes and complex methods. We provide descriptions and formal algorithm necessary to support the most complex class model.

Current research efforts include developing performance measurements to demonstrate the utility of our algorithm. Such performance analysis is difficult because very few of these systems (or more specifically, applications on these systems) have been developed. We are currently analyzing objectbased applications to determine the way they are being used with the goal of determining metrics for such a performance analysis. Ideally these techniques can be modified so they can be used in a dynamic environment where data is added and removed. Unfortunately, such an environment is very complicated because supporting it involves not only the accurate placement of fragments but also the need to transparently migrate object fragments while the system is being accessed by users. The initial step in this research requires that we



Note: Person is a superclass of the class Student. Undergrad and Grad are subclasses of Student while Dept and Prof are containing classes. Person is also its complex method class..

Figure 5: Method Usage/Application Frequencies for Related Classes

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
m_1	645	265	200	170	255	110	190
m_2	265	265	60	130	70	0	60
m_3	200	60	200	60	75	0	60
m_4	170	130	60	260	70	0	190
m_5	255	70	75	70	255	110	0
m_6	110	0	0	0	110	110	0
m_7	140	60	60	190	0	0	230

(a) Method Affinity matrix

	m_3	m_4	m_5	m_7	m_1	m_2	m_6
m_3	200	60	75	0	200	60	60
m_4	60	260	70	0	170	190	130
m_5	75	70	255	110	255	0	70
m_7	0	0	110	110	110	0	0
m_1	200	170	255	110	645	190	265
m_2	60	190	0	0	140	230	60
m_6	60	130	70	0	265	60	265

(b) Clustered Method Affinity matrix

	m_1	m_2	m_3	m_4	m_5	m_6	m_7
q_1	1	0	0	0	1	1	0
q_2	1	1	1	1	0	0	1
q_3	1	0	1	0	0	1	0
q_4	1	1	0	0	0	0	0
q_5	1	0	1	0	1	0	0
q_6	1	0	0	0	0	0	1
q_7	1	1	0	0	0	0	0
q_8	1	1	0	1	0	0	0
q_9	0	0	0	0	0	0	1
q_{10}	1	0	0	0	0	0	1
q_{11}	0	0	0	0	0	0	1
q_{12}	1	0	0	0	0	0	0

(c) Modified Method Usage matrix

For matrices (a) to (c);
m_1 = m.stuno−of, m_2 = m.dept−of, m_3 = m.owing,
m_4 = m.course−taken, m_5 = Person.ssno−of, m_6 = Person.age−in−year,
m_7 = Person.whatname.

For matrix (c);
(q_1 to q_3) represent application accesses as in the original method usage matrix of class (Student)
(q_4 to q_6) represent application accesses through methods of subclass (Grad).
(q_7 to q_8) represent application accesses through methods of containing class (Dept)
(q_9 to q_{11}) represents application accesses through methods of containing class (Prof)
q_{12} represents application accesses through methods of complex method class (Person).

Figure 6: Method/Clustered Affinity and Modified Method Usage Matrices

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 hybrid fragmentation schemes for more complex class models.

References

[1] S. Chakravarthy, J. Muthuraj, R. Varadarajan, and S. B. Navathe. An Objective Function for Vertically Partitioning Relations in Distributed Databases and its Analysis. *Distributed and Parallel Databases*, 2(1):183–207, 1993.

[2] D. Cornell and P.S. Yu. A Vertical Partitioning Algorithm for Relational Databases. In *Proceedings of the Third International Conference on Data Engineering.* IEEE, 1987.

[3] C.I. Ezeife and Ken Barker. Vertical Class Fragmentation in a Distributed Object-Based System. In *Proceedings of the Second International Symposium on Applied Corporate Computing, ISACC*, volume 2(1). Texas A & M University, 1994. Monterrey, Mexico, October,Vol.2, No.1, pp. 43-52.

[4] C.I. Ezeife and Ken Barker. Vertical Class Fragmentation in a Distributed Object Based System. Technical Report TR 94-02, Univ. of Manitoba Dept of Computer Science, April 1994.

[5] C.I. Ezeife and Ken Barker. A Comprehensive Approach to Horizontal Class Fragmentation in a Distributed Object Based System. *International Journal of Distributed and Parallel Databases*, 1, 1995.

[6] J.A. Hoffer and D.G. Severance. The Use of Cluster Analysis in Physical Database Design. In *Proceedings of the 1st International Conference on Very Large Databases.* Morgan Kaufmann Publishers, Inc, 1975. Vol 1, No.1.

[7] 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.

[8] S.B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical Partitioning Algorithms for Database Design. *ACM Transactions on Database Systems*, 9(4), 1984.

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

[10] G. Pernul, K. Karalapalem, and S.B. Navathe. Relational Database Organization Based on Views and Fragments. In *Proceedings of the Second Internationa Conference on Data and Expert System Applications*, Berlin, 1991.