Distributed Object Based Design: Vertical Fragmentation of Classes*

C.I. Ezeife, School of Computer Science,
University of Windsor, Windsor, Ontario, Canada N9B 3P4
cezeife@cs.uwindsor.ca
Tel: (519) 253-3000 ext 3012; FAX: (519) 973 - 7093
and
Ken Barker, Advanced Database Systems Laboratory,
Department of Computer Science, University of Manitoba,
Winnipeg, Manitoba, Canada R3T 2N2
barker@cs.umanitoba.ca
Tel: (204) 474-8832; FAX: (204) 269-9178

March 30, 1998

Abstract

Processing costs in distributed environments is most often dominated by the network communications required for interprocess communication. It is well-known from distributed relational database design research that careful placement of data "near" the users or processors where it is used is mandatory or system performance will suffer greatly. Data placement in relational database systems is comparatively simple because the data is flat, structured, and passive. Objects are characterized by an inheritance hierarchy (other hierarchies could also be considered including, class composition and execution), unstructured (possibly dynamic data), and contain a behavioral component that defines how the "data" is accessed by encapsulating it within the object *per se*. Algorithms currently exist for fragmenting relations, but the fragmentation and allocation of objects is still a relatively untouched field of study.

Similar to relations, objects can be fragmented both horizontally and vertically. 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 adopts a classification of classes into four main models, and contributes by proposing algorithms for vertically fragmenting the four realizable class models consisting of simple or complex attributes combined with simple or complex methods. Vertical fragmentation entails splitting classes into a set of "smaller" equivalent classes (actually fragments of the class' extent) that can later be placed precisely where they are used. 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 subclasses, 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 operating grants (OGP-0194134), (OGP-0105566); grants from University of Windsor and Manitoba Hydro.

List of Figures

1 Introduction

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 performance of a DOBS is greatly enhanced if data is stored at local sites in such a manner that many of the user applications running at each site get all needed data at that site without accessing irrelevant data. Further, the system must not incur unreasonable costs of data replication. This requires organizing database entities with the goal of reducing the amount of unneeded data accessed by applications as well as minimizing the amount of data that needs to be transferred between sites. Distributed object based design improves performance by fragmenting and subsequently allocating fragments to distributed sites.

The top-down design approach takes a global conceptual schema (GCS) describing the global database entities and their relationships and combines it with access pattern information to produce a set of local conceptual schemas (LCS) describing database entities at each local site [15]. The input to the design process is obtained from an à priori system requirements analysis which defines the environment of the system and collects an approximation of both the data and processing needs of all potential database users [18]. The expectations of the system with respect to performance, reliability and availability, economics and flexibility are also specified from the analysis. The view and conceptual design activities use output from the requirements study; while the former defines interfaces for end users, the latter defines entity types with relationships among them. Statistical information collected from the requirements analysis include the access frequencies of user applications and their reference patterns. The input to the distribution design are the GCS and the access pattern information from the requirements studies. The bottom-up approach constructs a GCS from preexisting local schemas. We use the top-down approach and distribute class fragments. Fragmentation breaks a class into a set of classes with only a subset of its components. Classes can be fragmented *horizontally* or *vertically*. Vertical fragmentation is the process of breaking a class into a set of smaller classes called vertical fragments. Each instance object in a vertical fragment is a portion of the original instance object in the original class.

Many distributed and client/server object based systems exist, including ITASCA [7], ENCORE [6], GOBLIN [9], THOR [10], and EOS [14] which will benefit from fragmentation [8]. A partial list of benefits include: (a) Different applications access or update only portions of classes so fragmentation will reduce the amount of irrelevant data accessed by applications. (b) Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base. (c) Fragmentation allows parallel execution of a single query by dividing it into a set of subqueries that operate on fragments of a class. (d) Fragmentation reduces the amount of data transferred when migration is required. (e) 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 and that this incomplete analysis is feasible [17]. 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 processes described here. Future research will investigate how these techniques can be incorporated into a dynamic system or determine the amount of change that can be accommodated before a re-analysis is required.

This paper reviews possible DOBS models presented earlier [3, 4], and contributes by presenting algorithms for vertically fragmenting the class models consisting of (simple attributes, simple methods), (complex attributes, simple methods), (simple attributes, complex methods) and (complex attributes, complex methods).

The balance of the paper is organized as follows. We complete this section by briefly reviewing previous work on distributed database design and our DOBS model. Section 2 presents vertical fragmentation algorithms for class model consisting of simple attributes and simple methods; Section 3 discusses extension to this work to capture a model where attributes are composed in a part-of hierarchy and accessed using simple methods; extension to handle a model consisting of simple attributes and complex methods is presented as Section 4 while a model consisting of complex attributes and complex methods is presented in Section 5. Section 6 discusses the time complexities of the algorithms and present experimental results comparing their performance with those of other approaches. Finally, Section 7 concludes and suggests future research directions.

1.1 Related Work

Algorithms that fragment relations horizontally and vertically exist in the distributed relational database environment. This section reviews previous work on vertical fragmentation in these systems and finally covers work on fragmentation in a DOBS.

Vertical Fragmentation (relational): Existing work on vertical fragmentation in the relational data model includes Hoffer and Severance [5], Navathe *et al.* [12], Cornell and Yu [2], Navathe and Ra [13], Özsu and Valduriez [15] and Chakravarthy *et al.* [1].

Hoffer and Severance [5] 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 developed by McCormick *et al.* [11] is used to form these attribute clusters. Navathe *et al.* [12] extends Hoffer's work by defining algorithms for grouping attributes into overlapping and nonoverlapping fragments. The approach is to minimize the number of fragments visited by a transaction and to refine fragments using cost factors that reflect the physical environment where the fragments are stored. Cornell and Yu [2] optimized this work by developing an algorithm that obtains an optimal binary partitioning for relational databases. Further refinement is accomplished by applying the binary vertical partition

algorithm iteratively [12]. Özsu and Valduriez [15] discuss this earlier work on vertical partitioning for distributed databases using the access frequency information and the Bond Energy Algorithm that groups attributes of a relation based on the attribute affinity values. Groups of attributes are clustered and cost equations are used to define the best position along the diagonal of this clustered affinity matrix to split relations into fragments. Charkravarthy *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. The Partition Evaluator has two terms; namely, irrelevant local attribute access cost and relevant remote attribute access cost. The irrelevant local attribute cost term measures the local processing cost of transactions due to irrelevant fragment attributes. The relevant remote attribute access term measures the remote processing cost due to remote transactions accessing fragment relevant attributes. The two components of the Partition Evaluator are responsive to partition sizes.

Vertical Fragmentation (objects): Karlapalem *et al.* [8] define issues involved in distribution design for an object oriented database system. They identify two method types – simple and complex. Their first model consists of simple methods. They argue that a model consisting of simple methods can be vertically partitioned using techniques described by Navathe *et al.* [12], while that of complex methods, requires a method-based view (MBV). The MBV identifies the set of objects accessed by a method and the corresponding set of attributes or instance variables. 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 m_j. This is called method m_j's view of class C_i. They further suggest the use of concepts developed by Pernul, Karlapalem and Navathe [16] to fragment classes based on views. Pernul *et al.* [16] argue that a user view can encompass a set of transactions and different views may overlap to give rise to relationships between views. From these relationships, fragments are derived using the vertical fragmentation operators defined earlier [12].

1.2 The DOBS Model

The data in a DOBS consists of a set of encapsulated objects belonging to classes that share an inheritance hierarchy captured in a class lattice. 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* which is an ancestor of every other class in the database. A class is an ordered relation $\mathcal{C} = (\mathbf{K},\mathcal{A},\mathcal{M},\mathcal{I})$ where[1] $\mathbf{K}$ 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 $\mathcal{M}$. 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. 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, $\mathcal{C}_h = (\mathrm{K},\mathcal{A},\mathcal{M},\mathcal{I}')$. Each vertical fragment ($\mathcal{C}^v$) of a class

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

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,$\mathcal{A}^{'}$,$\mathcal{M}^{'}$,$\mathcal{I}$). Each hybrid fragment ($\mathcal{C}_h^v$) of a class contains its class identifier, some of its instance objects ($\mathcal{I}^{'} \subseteq \mathcal{I}$) for only some of its methods ($\mathcal{M}^{'} \subseteq \mathcal{M}$), and some of its attributes ($\mathcal{A}^{'} \subseteq \mathcal{A}$). Thus, $\mathcal{C}_h^v$ = (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 the domain of an attribute as another class. The complex attribute relationship between a class and other classes in the database is usually defined using a class composition or aggregation 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. 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, 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.

2 Vertical Fragmentation: Simple Attributes and Methods

This section presents vertical fragmentation algorithms for class model consisting of simple attributes using simple methods. The objective of vertical fragmentation is to break a class into a set of smaller classes (fragments) that permit user applications to execute using only one fragment. This means that optimal vertical fragmentation minimizes user application execution time [15]. The top-down design approach, uses a set of user queries, the database schema consisting of a set of database classes, and the relationships between classes as input to the fragmentation procedure. The model described in this section only considers the inheritance relationship captured by the class lattice.

Vertical fragmentation aims at splitting a class so all attributes and methods of the class most frequently accessed together; are grouped together. *Encapsulation* means user applications do not directly access 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 ap-

plication access pattern information using the same technique described earlier [5, 12]. 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 [3] to decide which fragment to place these overlapping attributes or (2) from the onset, group both attributes and methods using attribute/method affinity. Approach (1) is preferable for the following performance reasons: (a) It drastically reduces the size of the matrices needed by the bond energy and partitioning algorithms [12]. (b) It 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, taking approach (1), 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 methods are related. The explicit assumptions made in this design 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 performed à priori to the fragmentation process as discussed in Section 1.

The first assumption supports the inheritance feature of object oriented systems without replication of inherited parts of an object at all its superclasses. This feature removes the problem of update of replicated parts of an object and exploits the natural fragmentation feature present in the inheritance hierarchy as a subclass of a class is a type of vertical fragment of this class. The significance of this assumption lies in the fact that providing a fragmentation scheme for the alternative storage structure (that replicates inherited parts of an object at subclasses) would call for an extensive modification of the fragmentation algorithm in order to accommodate the storage structure in the first assumption.

Section 2.1 presents a motivating example of an object database consisting of a set of classes and gives an elaborate and intuitive discussion of how the proposed scheme would vertically fragment a class of the database using the class lattice, applications access pattern and frequencies. Section 2.2 presents some definitions before discuss the algorithm in Section 2.3.

2.1 Motivating Example

The example object base has classes *Root*, *Person*, *Course*, *Prof*, *Student*, *UnderGrad* and *Grad* with an inheritance relationship between the classes as shown in the class lattice depicted in Figure 1. The

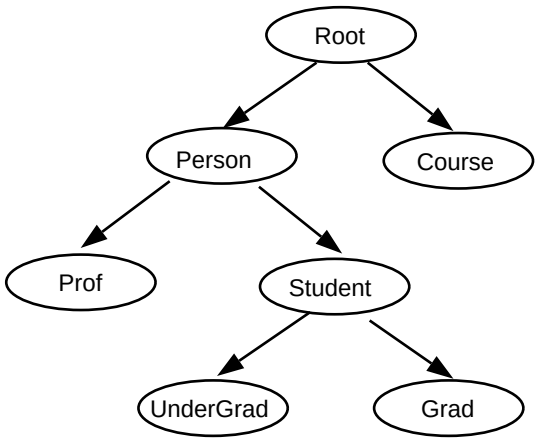

Figure 1: Class Lattice of Sample Object Base

object base sample data [2] for the example is illustrated in Figure 2.

Given this sample object base, we want to obtain vertical fragments of the class *Person* based on the following application information.

q_1: Give the social security number of all persons older than 65.

q_2: Give social security number, name and addresses of all Winnipeg clients.

q_3: List names of all clients who will be 65 in the year 2000.

The first step is to define the method usage matrix of the class *Person* from these three applications. It can be seen from the sample object base Figure 2, that the class *Person* has four methods, namely, *ssno-of* labeled method m_1, *whatname* labeled m_2, *age-in-year* labeled m_3 and *newaddr* labeled m_4. The methods of class *Person* accessed by application q_1 are m_1 (ssno-of) and m_3 (age-in-year). Application q_2 accesses m_1, m_2 and m_4, while application q_3 accesses m_2 and m_4. This information is captured using the method usage matrix of class *Person* shown on the left side of the class labeled *Person* in Figure 3. The frequencies of access of class *Person* at the three different existing sites by the three applications as collected from requirements analysis are defined with the application frequency matrix of the class (shown on the right side of the class labeled *Person* in Figure 3). Since subclasses of the class *Person* which are classes *Prof* and *Student* also receive a number of database applications running on them and which may require methods of the superclass *Person*, all methods of the superclass accessed by applications running on *Prof* and *Student* are represented in the method usage matrices of these subclasses as null method representatives of those methods of superclass *Person*. From Figure 3, m_6 in the method usage matrix of *Prof* is the same as the m_3 of *Person* and in the method usage matrix of *Student*, m_4 corresponds to m_1 of class *Person*, m_5 of *Student* is the same as m_3 of *Person* and m6 of

[2] For readability, we have preceded each attribute name of a class with an *a* and each method name with an *m*.

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,Bill Jeans,40,Toronto}
I_6 {Person6,Mandu Nom,32,Vancouver} } }
Prof = Person pointer $\odot$ {Prof,{a.empno,a.status,a.salary,a.students,a.courses},
{m.empno-of,m.course-taught,m.whatsalary,m.status-of,m.students-of},
{ I_1 (person pointer5) $\odot$ {Prof1,asst prof,45000, {Ken,Peter,Maria},[111]}
I_2 (person pointer6) $\odot$ {Prof2,assoc prof,60000, {Janet,Larry},[236]} } }
Student = Person pointer $\odot$ {Student,{a.stuno,a.dept,a.feespaid,a.coursetaken},
{m.stuno-of,m.dept-of,m.owing},
I_1 (person pointer1) $\odot$ {Student1,Math,Y,[511,601]}
I_2 (person pointer4) $\odot$ {Student2,Computer Sc.,N,[532,652]}
I_3 (person pointer2) $\odot$ {Student3,Stats,Y,[111,205]}
I_4 (person pointer3) $\odot$ {Student4,Computer Sc.,N,[236]} } }
Grad = Student pointer $\odot$ {Grad,{a.gradstuno,a.supervisor},{m.whatprog}
{ I_1 (Student pointer1) $\odot$ {Grad1,John West}
I_2 (Student Pointer2) $\odot$ {Grad2,Mary Smith} } }
UnderG = Student
I_1 (Student pointer3)
I_2 (Student pointer4)

Figure 2: The Simple Sample Object Database Schema

Student is the same as m_2 of *Person*. These null method representatives at the subclasses allow use of methods of superclasses by applications running on their subclasses to be accounted for.

To define the method affinity matrix used for vertical fragmentation of the class *Person*, we also need the method usage matrices and application frequency matrices of all its descendant classes that use its methods and attributes. The first subclass of the class *Person* is the class *Prof* and the following applications run on this subclass.

q_1: Report the salary of a professor given status.

q_2: Find the courses taught by all professors in a specific status.

q_3: Find the students supervised by a professor, given his employee number.

q_4: List employee numbers of all professors with age greater than 65.

The method usage matrix of the subclass *Prof* derived from these applications as well as the application frequency matrix of this class are given in Figure 3. The second subclass of the class *Person* is the class *Student* and the applications running on this subclass are as follows:

q_1: Find the social security numbers and student numbers of all students younger than 40.

q_2: List names and student numbers of all students of a specific department.

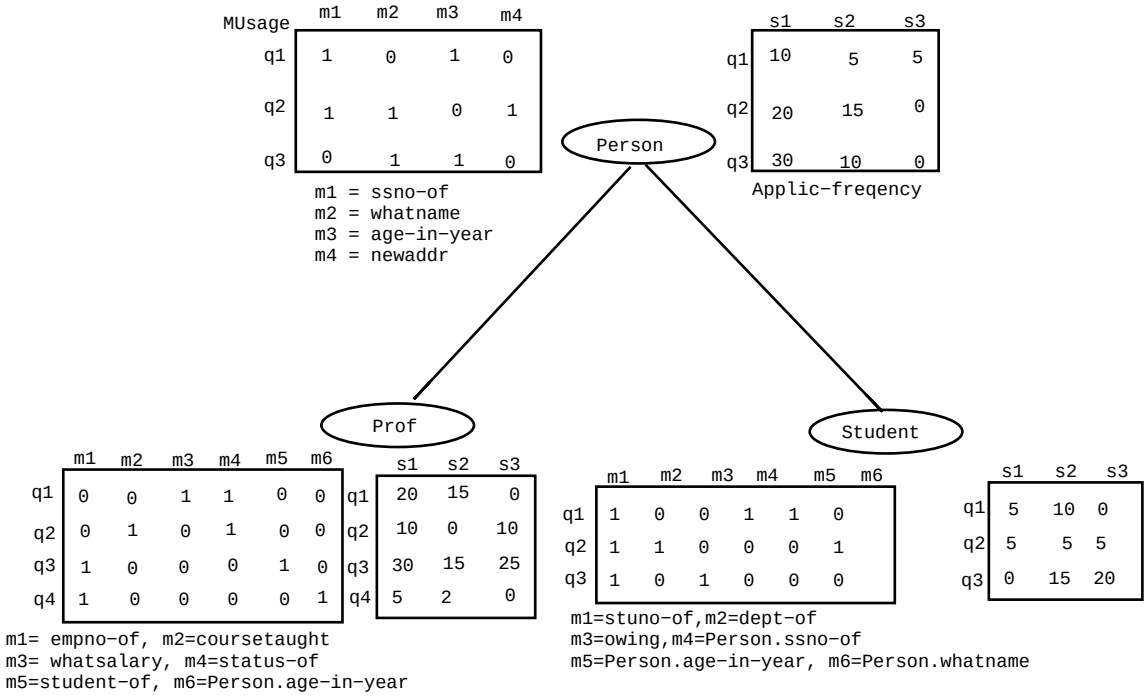

Figure 3: The Method Usage and Application Frequency matrices of the Classes

q_3: List student numbers of those students who have not paid their term fees.

Similarly, the method usage matrix of the subclass *Student* derived from these applications as well as the application frequency matrix of this class are given in Figure 3. Note that applications running on this subclass *Student* use three null methods with respect to superclass *Person* which are: *Person.ssno-of, Person.age-in-year and Person.whatname.* All the matrices needed to compute the method affinity matrix of the class *Person* are given in Figure 3. The second step is to define the method affinity matrix of the class *Person* being fragmented using the method usage and application frequency matrices of the class *Person* and its descendant classes *Prof and Student.* The method affinity of two methods m_i, m_j of class *Person* is obtained by obtaining the total of the application frequencies at all sites of every application q_k that accesses both methods. For example, since q_1 of class *Person* accesses both m_1 and m_3, the method affinity element (m_1, m_3) is the sum of accesses of q_1 which is $(10 + 5 + 5) = 20$. But also note that at the subclass level *Student*, m_1 and m_3 of *Person* are represented as m_4 and m_5 and thus, any access to both m_4 and m_5 of *Student* by any applications should be added to the affinity value of m_1 and m_3 of class *Person.* It can be seen from Figure 3 that q_1 of class *Student* accesses both m_4 and M_5, thus, the total access frequency of $(5 + 10 + 0) = 15$ has to be added to the original 20 from usage of class *Person* to give the method affinity value of 35 for the method pair m_1, m_3 of *Person.* The total frequency of 15 in this case is called the subclass affinity contribution from usage of subclass *Student.* The method affinity matrix for the class *Person* generated is as shown in Figure 4. The third step is to create the clustered affinity matrix for the class *Person* from its method affinity matrix using the bond

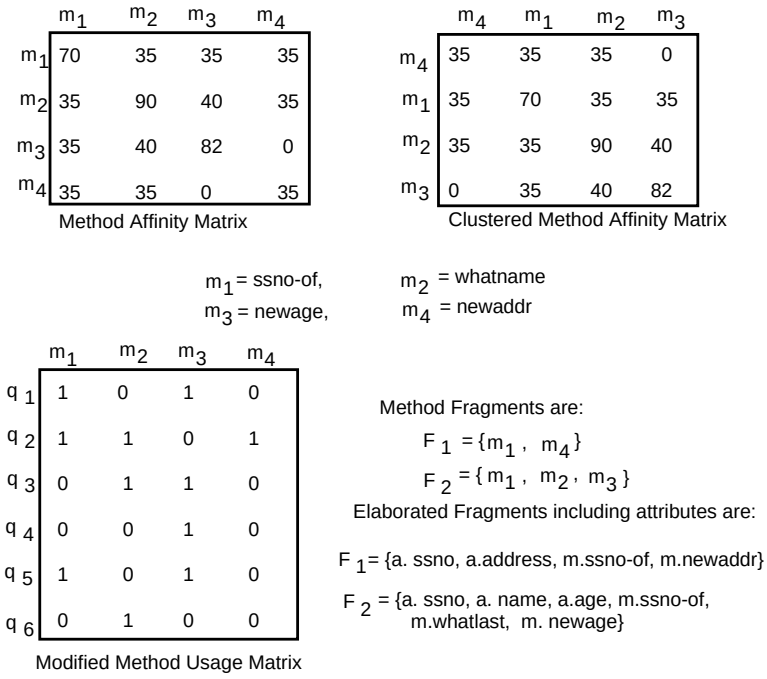

Figure 4: The Method Affinity,Clustered Affinity and Modified Method Usage matrices of the Class Person

energy algorithm discussed in [15]. The clustered affinity matrix shows clustering together of methods with high affinity for each other as shown in Figure 4. To create the clustered affinity matrix, the first two columns of the method affinity matrix are initially made the first two columns of the new clustered affinity matrix. Then, the algorithm proceeds to find the best placement for each remaining column of the method affinity matrix in the clustered affinity matrix being created. The chosen placement for each column is the one that makes the maximum contribution to a global affinity measure. For example, to find the best placement for column 3, the contending placements are 0-3-1, 1-3-2 and 2-3-4. This means we can place column 3 just before column 1 (0-3-1), or in between columns 1 and 2 (1-3-2) or after column 2 (2-3-4). The contribution of each placement is computed as 2 * (the sum of products of the each pair of adjacent columns) - 2 * (the sum of products of the two farthest columns). The sum of products of non-existent columns like 0 and 4 at this stage is 0. Thus, the contribution of placement 0-3-1 is computed as [2 * (sum of product of columns 0 and 3 + sum of product of columns 3 and 1) - (sum of product of columns 0 and 1)]. This gives 2 * (0 + [(35 * 70) + (40 * 35) + (82 * 35) + (0 * 35)] - 0) = 13440. It turns out that the contribution of position 2-3-4 is the highest and column 3 for m_3 is placed in the position. Later, column 4 for m_4 is placed in position 0-4-1 to give the final ordering of 4-1-2-3 in the columns of the clustered affinity matrix. A row ordering operation to place the rows of the clustered affinity matrix in the same order as the columns is the final operation needed to create the clustered affinity matrix shown in Figure 4. Then, we modify the original method usage matrix of

the class *Person* to include a row for all accesses to its method by all its descendant classes. In the modified method usage matrix of the example, q_4 represents use of the method *m.age-in-year* of class *Person* by the subclass *Prof*, while q_5 represents use of the methods *m.ssno-of and m.age-in-year* by the descendant class *Student*. In addition, q_6 represents the use of method *whatname* by the descendant class *Student*. Finallly, the partition algorithm of [15] is run using the modified method usage matrix and the clustered affinity matrix to determine where along the diagonal of the clustered affinity matrix it is most beneficial to partition. The purpose of the partitioning algorithm is to identify groups of methods that are accessed mostly, by distinct sets of applications. With the example in Figure 4, the partition is between m_4 and m_1 columns but with m_1 being the unique method id, it is kept in all fragments. The fragments are elaborated to include all attributes in the attribute sets of its methods. The method fragments produced after running the *Partition* algorithm are: $F_1 = \{m_1, m_4\}$ and $F_2 = \{m_1, m_2, m_3\}$. The fifth step of the algorithm is incorporating attributes in these method fragments using method attribute reference information. Suppose the method/attribute reference (AR) of these methods are as follows: AR(ssno-of) = {ssno}, AR(whatname) = {name}, AR(age-in-year) = {age}, and AR(newaddr) = {address}. Then, the attribute/method fragments now become: F_1 = {ssno,address,ssno-of,newaddr} and F_2 = {ssno,name,age,ssno-of,whatname,newage}. Since there are no overlaps of non-class identifier attributes/methods, step 6 of the algorithm leaves the fragments unchanged.

2.2 Definitions

Some definitions needed to present the formal algorithms for partitioning classes are presented in this section. 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 $\{M^{i1.j}, M^{i2.k}, \ldots, M^{in.p}\}$. The cardinality of a class is the number of instance objects in the class (denoted $\mathbf{card}(C_i)$).

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 $\{M^{i1.j}, M^{i2.l}, \ldots, M^{in.p}\}$ where each M in a user query refers to an invocation of a method of a class object. ■

Definition 2.2 *Attribute Reference Set AR($M^{i.j}$)* 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 *Method Reference Set MR($M^{i.j}$)* of a method $M^{i.j}$ of a class C_i is the set of all methods of any class C_k in the object base referenced by method $M^{i.j}$ of class C_i. ■

Definition 2.4 *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 2.5 *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 2.6 *Access frequency of a query* is the number of accesses a user application makes to "data". If Q = {q_1,q_2,...,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 2.7 *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* added to 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 2.8 *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 of C_i and methods/attributes of the class are needed together by applications running at any particular site.
$\text{saff}((EM)^{i.j}, (EM)^{i.k}) = \sum_{o=1}^{w} \sum_{p \| use(q_p,(EM)^{i.j})=1 \wedge use(q_p,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_p) acc_l(q_p)$,
where w is the number of subclasses, p is some application and S_l ranges over all sites. ■

Definition 2.9 *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. $\text{MA}((EM)^{i.j}, (EM)^{i.k}) =$
$(\sum_{p \| use(q_p,(EM)^{i.j})=1 \wedge use(q_p,(EM)^{i.k}=1} \sum_{\forall s_l} ref_l(q_p) acc_l(q_p)) + saff((EM)^{i.j}, (EM)^{i.k})$

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 2.10 *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. $\text{ccaff}((EM)^{i.j},(EM)^{i.k}) = \sum_{o=1}^{w} \sum_{k \| use(q_k,(EM)^{i.j})=1 \wedge use(q_k,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k)$

,

where w is the number of containing classes, C_i is the class and S_l ranges over all sites. ■

Definition 2.11 *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.

$\text{cmaff}((EM)^{i.j},(EM)^{i.k}) = \sum_{o=1}^{d} \sum_{k \| use(q_k,(EM)^{i.j})=1 \wedge use(q_k,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k)$,

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 2.12 *Attribute/Attribute Affinity* $\text{AAA}(A^{i.j},A^{i.m})$ 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. $\text{AAA}(A^{i.j},A^{i.m}) = \sum_{k|A^{i.j} \in AR(M^{in.k}) \wedge A^{i.m} \in AR((M^{in.k})} \sum_{l=1}^{m} acc_l(M^{in.k}, A^{i.j}) + acc_l(M^{in.k}, A^{i.k})$

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 2.13 *Attribute/Method Affinity* $\text{AMA}(A^{i.j},M^{i.m})$ 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. $\text{AMA}(A^{i.j}, M^{i.m}) =$

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

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

Definition 2.14 *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($A^{i.m}, F^{i.j}$) = $\sum_{k|A^{i.m} \in F^{i.j} \wedge A^{i.k} \in F^{i.j})} AAA(A^{i.m}, A^{i.k}) + \sum_{k|A^{i.m} \in F^{i.j} \wedge M^{i.k} in F^{i.j})} AMA(A^{i.m}, M^{i.k})$. ■

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, class composition 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.

2.3 The Algorithm

The only relationship to consider in this simple model is the inheritance hierarchy. This is incorporated in the vertical fragmentation process through *null methods* as follows. For every method of a class, the frequency of access of the method, includes accesses by all its subclasses and a method is represented at the subclass level as a null method on the parent class. The vertical fragmentation process of a class requires three matrices [12]:

1. The method usage matrix indicates for each application q_k on the global object base (the matrix's rows) and for each method M_j of the class (the columns) whether use(q_k,M_j) = 0 or 1. By definition, use(q_k,M_j) = 1 if method M_j is referenced by query q_k and is 0 otherwise.

2. The application frequencies matrix measures the number of accesses made by each of the applications q_k (the rows) at each of the sites s_l (the columns). This is part of the application information input to the design process.

3. The method affinity matrix for the n methods of the class is an n * n matrix measuring the number of accesses made to method pairs by all applications accessing them together at all sites. Both row and column headings of this matrix are the n ordered methods.

Our approach is similar in the following ways to earlier ones [12]. We use the same techniques for grouping attributes to group methods starting with the three matrices - method usage matrix, application

frequencies, and method affinity matrices. Secondly, we cluster methods in much the same fashion as they cluster attributes to obtain clustered method affinity matrix. Finally, the same partition algorithm is used to generate class method partitions.

The major differences and contributions of our approach are as follows. To capture the inheritance link between database classes, when computing the application frequencies of the methods of a class, for every method of the class, the access frequency of the method includes accesses made to its null method equivalents at all descendants of the class at all sites. This modifies the affinity measure between class methods to include all classes where these methods are used. Thus, in the object base, unlike in the relational case, we cannot use only those three matrices for a class in isolation to determine the vertical fragments of the class. Rather, we need to get the method usage matrices, as well as the application frequencies matrices of all the descendants of the class to compute the method affinity matrix for a particular class. Furthermore, the method usage matrix used by the partition algorithm for the final partitioning of the methods is modified in our approach. We partition so methods most frequently used together, are grouped together. In the relational system, the same method usage matrix used to create the method affinity matrix is used for this process. With the inheritance link information taken into consideration, we group methods/attributes of a class accessed together at a site to reflect accesses by applications running on each subclass at a particular site. Thus, we need to modify the method usage matrix of the class to include application accesses to null methods of subclasses and the sites where they are used. Finally, unlike the relational case, partitioning of the methods of the class does not necessarily end the vertical fragmentation process. This is because we want to group both methods and attributes used together by applications in a fragment. To achieve this objective, we use method-attribute binding to group attributes of a class with methods that use them, and later obtain non-overlapping fragments using a set of attribute-fragment affinity rules.

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}) : applications accessing extended methods of C_i and its descendant classes.

L(C_i) : a tree rooted at node (class) C_i.

AR-set(C_i): set of attribute references of the set of all methods of the class C_i.

AF-set(C_i^{des}): set of application frequency matrices of the class and its descendant classes.

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

Algorithm 2.1 *(UsageMtrx - Generate original method usage matrices for C_i and descendants)*

Algorithm UsageMtrx(L(C_i))
input: C_i the database class; L(C_i): a tree rooted at class C_i.
$\mathcal{M}^{C_i}$: set of methods of C_i; $\mathcal{C}_i^{des}$: set of descendant classes of C_i.
EM-set(C_i^{des}) : extended methods set of C_i and its descendant classes.
EMapplic-set(C_i^{des}) : applications accessing extended method set of C_i and its descendant classes.
output: **MU-set**(C_i): the original method usage matrices for class C_i and its descendant classes.
var
$\mathcal{C}_c$: set of classes; C_k: a class.
matrixrow : a sequence of n row values of a matrix.
begin
//Generate the original method usage matrix for class C_i //
// and other classes on the tree passed in as parameter.//
$\mathcal{C}_c = \{C_i\}$ (1)
while $\mathcal{C}_c \neq \emptyset$ **do** (2)
C_k = a class c $\in \mathcal{C}_c$ (3)
if C_k is not a leaf class (4)
begin
$C_c = C_c \cup children(C_k)$ (5)
for each application $q_j \in$ EMapplic-set(C_i^{des}) **do** (6)
For each method, $M^{k.m} \in$ EM-set(C_k)) do (7)
if use(q_j,$M^{k.m}$) = 1 then (8)
MU(q_j,$M^{k.m}$) = 1 (9)
else MU(q_j,$M^{k.m}$) = 0 (10)
end; {for q_j}
end; {if C_k}
$C_c = C_c - C_k$
end; {while}
end {UsageMtrx}

Figure 5: Original method Usage Matrix Generator

its descendant classes.

children(C_k) : a function that returns the immediate children of the node (class) C_k.

The Steps

The steps for vertically fragmenting a class consisting of simple attributes and simple methods are:

M1. Obtain the method usage and application frequency matrices of the class and its subclasses. The method usage matrices are generated by the Algorithm *UsageMtrx* defined in Figure 5. This algorithm accepts a tree rooted at a class C_i and creates the original method usage matrices for class C_i and other classes (descendant classes of C_i in this case) on the tree from user applications. 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 (Lines 4-10 of Figure 5). The application frequency matrices are part of the input from pre-analysis of the system.

Algorithm 2.2 *(MAMtrx - Generate method affinity matrix of class C_i)*

Algorithm MAMtrx
input: C_i the database class; $(EM)^{C_i}$: set of methods of C_i.
C_i^{des}: set of descendant classes of C_i.
EM-set(C_i) : set of extended methods of C_i and its descendant classes.
EMapplic-set(C_i) : set of applications accessing extended methods of C_i and its descendants.
MU-set(C_i): the set of method usage matrices of class C_i and its descendants.
output: MA^i: the method affinity matrix for class C_i.
var
matrixrow,matrixcol : a set of n row/column method headings for the matrix.
row,col : an extended method; n : integer.
begin
//Generate the method affinity matrix of the class C_i using Definition 2.9. //
matrixrow = $(EM)^{c_i}$
matrixcol = $(EM)^{c_i}$
n = card($(EM)^{c_i}$) (1)
for row = $(EM)^{i.1}$ to $(EM)^{i.n}$ **do** (2)
for col = $(EM)^{i.1}$ to $(EM)^{i.n}$ **do** (3)
MA(row,col) = MA($(EM)^{i.row}$,$(EM)^{i.col}$) (4)
end; {for col }
end; {for row}
end;

Figure 6: Method Affinity Matrix Generator

M2. Define method affinity matrix of the class using the modified approach given in the algorithm *MAMtrx* of Figure 6. This algorithm accepts a class C_i as an argument and generates the method affinity matrix of the class. It enters in each (row,column) position of the matrix, the method affinity (MA as given in Definition 2.9) between the two extended methods of the class in this (row,column) position (Lines 1-4 of Figure 6).

M3. Use the Bond Energy Algorithm (BEA) developed by [11] as presented in [12, 15] to generate clustered affinity matrix of the class. This algorithm accepts the 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 collected together.

M4. Generate a modified method usage matrix of the class as described in the Algorithm *MUsageMtrx* of Figure 7. The algorithm *MUsageMtrx* includes a row to the method usage matrix of a class C_i for every application q_j that accesses a method of this class C_i through any of its descendant classes (lines 1-10 of Figure 7). Next the *Partition* algorithm (PARTITION) of [12, 15] takes the clustered affinity matrix and the modified method usage matrix to produce fragments of the methods. The *Partition* algorithm finds sets of methods that are mostly accessed by distinct sets of applications.

M5. Use method-attribute reference information of the methods in each method fragment (AR of method Definition 2.2) to include in each method fragment all attributes of the class accessed by

Algorithm 2.3 *(MUsageMtrx - Generate Modified method usage matrices)*

Algorithm MUsageMtrx
input: C_i the database class; $\mathcal{M}^{C_i}$: set of methods of C_i.
$\mathcal{C}^{des}$: set of descendant classes of C_i.
EM-set(C_i^{des}) : set of extended methods of the descendant classes.
EMapplic-set(C_i^{des}) : set of applications accessing extended methods of C_i and its descendant classes.
numapplic : number of applications accessing objects of C_i
MU^i: the original method usage matrix for class C_i.
output: MU^i: the modified method usage matrix for class C_i.
var
matrixrow : a sequence of n row values of a matrix.
begin
//Modify method usage matrix to include a row for every application //
// that accesses a method of this class through its descendant classes.//
for each class $C_o \in \mathcal{C}^{des}$ **do** (1)
for each application $q_j \in$ EMapplic $-$ set$(\mathbf{C_i^{des}})$) **do** (2)
For each null method, $(NM)^{o.k} \in$ EM $-$ set$(\mathbf{C_i^{des}})$ do (3)
if use$(q_j,(NM)^{o.k}) = 1$ then
begin
numapplic = numapplic + 1 (4)
matrixrow$[(q_j,(EM)^i.1)) \ldots (q_j,(EM)^i.n))] = 0 \ldots 0$ (5)
for each extended method $(EM)^{i.k} \in (EM)^{C_i}$ **do** (6)
if $((EM)^{i.k} = (NM)^{o.k})$ or use$(q_j,(EM)^i.k = 1)$ **then** (7)
matrixrow$[(q_j,(EM)^i.k)] = 1$ (8)
end {for each}
matrixrow(numapplic) = matrixrow$[(q_j,(EM)^i.1) \ldots$ matrixrow$(q_j,(EM)^i.n)]$ (9)
$MU^i = MU^i \cup matrixrow(numapplic)$ (10)
end; {for q_j}
end; {for C_o}
end;

Figure 7: Modified Usage Matrix Generator

methods of the fragment.

M6. Since there may be problems of overlapping attributes in more than one fragment if the same attribute of a class belongs to the method attribute reference sets of two different methods in two separate fragments, we use Attribute Placement Affinity Rule 2.1 to decide which vertical fragment to keep each overlapping attribute. The Attribute Placement Affinity Rule 2.1 determines the affinity between the overlapping attribute and each of the fragments containing it using the AMA and AFA statistics of Definitions 2.13 and 2.14, respectively. It places the attribute in the fragment with highest affinity measure and removes the attributes from every other.

The formal algorithm for vertically fragmenting a class consisting of simple attributes and simple methods is presented as Algorithm *VerticalFrag* of Figure 8. This algorithm takes as its inputs a set of user queries, class lattice of the database, a class in the lattice to be fragmented, the attribute references of the methods of this class, and the application frequency matrices of this class and its descendant classes. It returns a set of vertical fragments of this class by first creating the method usage matrices of the class and its descendant classes (line 1[3]) before computing the method affinity matrix of the class (line 2). Next, the clustered method affinity matrix of the class is created (line 3) before the modified method usage matrix class (line 4) is used to partition the clustered method affinity matrix into fragments (line 5). Lines 6 through 9 incorporate attributes accessed by their methods in the fragments while lines 10 - 13 ensure that fragments contain non-overlapping attributes.

3 Complex Attributes and Simple Methods

This section presents an algorithm for vertically fragmenting classes consisting of complex attributes that support a class composition hierarchy using simple method invocations. Thus, both the inheritance and the attribute link relationships are considered. The inheritance relationship is captured with the class lattice while the attribute link is captured with the class composition hierarchy. Vertical fragmentation aims at splitting a class so attributes and methods of the class most frequently accessed together by user applications are grouped together. The measure of togetherness is the affinity of methods which shows how closely related the methods are. The major data requirements related to applications is their access frequencies as defined earlier.

3.1 The Algorithm

User applications that access attributes and methods of this type of class model are of three types: (1) those running directly on this class, (2) those running on subclasses of this class, and (3) those running on containing classes using this class as a type for its attributes. Information about applications running

[3]All line numbers in this paragraph refer to Figure 8

Algorithm 2.4 *(VerticalFrag – Vertical Fragments Generator)*

Algorithm VerticalFrag
input: $\mathcal{Q}^{C_i^{des}}$: set of user queries; C_i : the database class to fragment
L(C) : the class lattice; $\mathcal{C}_i^{des}$: set of descendant classes of C_i
$(EM)^{C_i}$: extended method set of C_i.
AR-set(C_i) : method attribute reference set of methods of C_i
AF-set(C_i): application frequency matrices of C_i and and its descendants.
output: $\mathcal{F}^{c_i}$: set of vertical fragments of C_i.
var
MU-set(C_i) : method usage matrices for class C_i and its descendants.
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
MU-set(C_i) = **UsageMtrx**(**LT**(C_i)) (1)
MA^i = **MAMtrx**(**MU-set**(C_i),$\mathcal{Q}^{C_i^{des}}$,$\mathcal{C}_i^{des}$) //compute method affinity matrix // (2)
CA^i = **BEA**($(MA)^i$) //create clustered affinity matrix // (3)
MU^i = **MUsageMtrx**($\mathcal{C}_i^{des}$,$(EM)^{C_i}$, **MU-set**(C_i)) //get modified method use matrix // (4)
$\mathcal{F}^{c_i}$ = **PARTITION**($(CA)^i$) (5)
For each fragment $F^{i.k} \in \mathcal{F}^{c_i}$ **do** (6)
begin
For each method $(EM)^{i.m} \in F^{i.k}$ **do** (7)
For each attribute $A^{i.n} \in AR((EM)^{i.m})$ **do** (8)
$F^{i.k} = F^{i.k} \cup A^{i.n}$ (9)
end {of for $F^{i.k}$}
// Ensure disjointness //
for each overlapping $A^{i.n} \in \mathcal{F}^{c_i}$ **do** (10)
select $F^{i.k}$ to place $A^{i.n}$ according to Affinity Rule 2.1 (11)
for each $F^{i.p}, p \neq k$ **do** (12)
$F^{i.p} = F^{i.p} - A^{i.n}$; (13)
end; {for $F^{i.p}$}
end; {for $A^{i.n}$}
end {VerticalFrag}

Figure 8: Class Vertical Fragments Generator

Algorithm 3.1 *(CCLinkgraph - generates a Link tree rooted at C_i)*

Algorithm CCLinkgraph
input: $\mathcal{C}_d$**:** set of database classes; $\mathcal{C}_l$**:** set of classes on link paths $\mathcal{C}_l \subseteq \mathcal{C}_d$.
A(C_i): class composition hierarchy rooted at class C_i.
C_i^{cont}: set of containing classes of C_i.
output: The Link graph (LG) tree rooted at C_i.
LG = (Γ,λ) where Γ is a set of nodes
λ is a set of arcs connecting nodes in Γ.
begin
// Starting from the class C_i, for every class C_j, that contains C_i as //
// part-of it, define a link from that class C_i to C_j.//
$LG \leftarrow$ initialized with a node $\forall C_i \in \mathcal{C}_d$; (1)
s.t. $\Gamma \leftarrow \{C_k | C_k \in \mathcal{C}_d\}$ and $\lambda = \emptyset$;
$C_l = C_i$ (2)
while $C_i \neq$ Root **do** (3)
for each $C_i \in \mathcal{C}_d$ (4)
for each $C_j \in C_i^{cont}$ (5)
$\lambda = \lambda \cup (C_i \rightarrow C_j)$ (6)
$C_l = C_l \cup C_j$ (7)
end; {for C_j}
end; {for C_i}
$C_l = C_l - C_i$; (8)
C_i = a class in $\mathcal{C}_l$ (9)
end; {while C_i}
return ($LG(C_i)$); (10)
end;

Figure 9: The Linkgraph Generator For Containing Classes of C_i

on a class' descendant classes accommodates the inheritance relationship between classes in the object base. Similarly, propagating the effects of applications running on a contained class to a containing class, during the process of fragmentation reflects the attribute link relationship between the classes in the object base. We incorporate these effects in the definition of the method affinity and the modified method usage matrices of the class. The steps for producing the desired vertical fragments are thus, similar to those for simple attributes and methods presented as steps **M1** to **M6** except that steps 2 and 4 here are modified as follows.

Steps

Note. Steps N1, N3, N5 and N6 are similar to M1, M3, M5 and M6 respectively.

N2. Modify the method affinity matrix from step N1 to include use of the methods through its containing classes.

a. We repeat the operations in step N1 above, using a different type of relationship - the class composition hierarchy. Obtain the method usage and application frequency matrices of the class and its containing classes using the algorithm *UsageMtrx* of Figure 5 and the *CCLinkgraph* from Figure 9. CCLinkgraph returns a tree rooted at the class being fragmented (contained

Algorithm 3.2 *(CMAMtrx - includes complex attribute factor in method affinity matrix of class C_i)*

Algorithm CMAMtrx
input: C_i: the database class; C_i^{cont}: set of containing classes of C_i.
EM-set(C_i^{cont}) : set of extended methods of C_i and its containing classes.
EMapplic-set(C_i^{cont}) : set of applications accessing extended methods of C_i and its containing classes.
MU-set(C_i^{cont}): the set of method usage matrices of class C_i and its containing classes.
MA^i: the method affinity matrix for class C_i.
output: MA^i: the modified method affinity matrix for class C_i.
var
matrixrow,matrixcol : a set of n row/column method headings for the matrix.
row,col : an extended method; n : integer.
begin
//include complex attribute use to method affinity matrix of the class C_i using Definition 2.10. //
matrixrow = $(EM)^{c_i}$
matrixcol = $(EM)^{c_i}$
n = card($(EM)^{c_i}$) (1)
for row = $(EM)^{i.1}$ to $(EM)^{i.n}$ **do** (2)
for col = $(EM)^{i.1}$ to $(EM)^{i.n}$ **do** (3)
MA(row,col) = MA(row,col) + ccaff($(EM)^{i.row}$,$(EM)^{i.col}$) (4)
end; {for col }
end; {for row}
end;

Figure 10: Complex Attribute Modified Method Affinity Matrix Generator

class in this case) and shows the attribute link between it and other classes in the database (containing classes) that use this class being fragmented as a type for their attributes. The algorithm starts from the class C_i, and for every class C_j that contains C_i as part-of it, it defines a link from that class C_i to C_j (Lines 3-10 of Figure 9).

b. Modify method affinity matrix of the class from step (1b) using the usage matrices and application frequency matrices of the class and its containing classes (from N2a). The algorithm *CMAMtrx* for this process is given in Figure 10. This algorithm includes the complex attribute factor in method affinity matrix of class C_i. It accepts the method affinity matrix computed earlier as input and for each pair of extended methods of the class, it adds the complex class affinity value of the two extended methods to their current method affinity value (Lines 1-3).

N4. . Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 7 except this time, we modify the method usage matrix to include a row for every application q_j that accesses a method of this class (lines 1-10, Figure 7) being fragmented either through its descendant or containing classes. Next the *Partition* algorithm [12, 15] takes the clustered affinity matrix and the modified method usage matrix as its inputs and produces fragments of the methods.

The formal algorithm for vertically fragmenting a class consisting of complex attributes and simple methods is presented as algorithm *Vert_CA_SM* of Figure 11. This algorithm is the same as the VerticalFrag algorithm 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). 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.

We simplify the presentation by defining the vertical fragmentation algorithm in terms of the two key matrices of the class 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 were 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}), $MU(C_i^{des})$) means running the VerticalFrag algorithm with method affinity matrix generated using only method usage and application frequency matrices of the class and its descendant classes. Similarly, the method usage matrix input to this algorithm is produced by including all applications accessing this class through its descendant classes. VerticalFrag(MA(C_i^{des}, C_i^{cont}), $MU(C_i^{des}, C_i^{cont})$) means that 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. The vertical fragmentation algorithm for the class model consisting of complex attributes and simple methods *Vert_CA_SM* of Figure 11 is obtained by running the algorithm *VerticalFrag* using method affinity matrix that uses information (method usage and application frequency matrices) from both descendant classes and containing classes of the class being fragmented. The modified method usage matrix used also includes a row for every application accessing the class being fragmented either through its descendant classes or its containing classes.

4 Simple Attributes and Complex Methods

This section presents a vertical fragmentation algorithm for classes consisting of objects that have simple attributes using complex methods. The two relationships between this class model consisting of simple attributes and complex methods are the inheritance and the method link relationships. Vertical fragmentation of this class model requires that we know *à priori* those methods of other classes referenced by each method of the class we want to fragment and that encapsulation is not violated. Vertical

Algorithm 3.3 *(Vertical Fragments of Complex attributes and Simple Methods)*

Algorithm Vert_CA_SM
input: $\mathcal{Q}^{C_i^{des}}$: set of user queries accessing C_i and its descendant classes.
$\mathcal{Q}^{C_i^{cont}}$: set of user queries accessing C_i and its containing classes.
C_i : the database class to fragment
L(C) : the class lattice
$\mathcal{C}_i^{des}$: set of descendant classes of C_i
$\mathcal{C}_i^{cont}$: set of containing classes of C_i
$(EM)^{C_i}$: extended method set of C_i.
AR-set(C_i) : attribute reference set of methods of C_i
AF-set(C_i^{des}): application frequency matrices of C_i and and its descendants.
AF-set(C_i^{cont}): application frequency matrices of C_i and and its containing classes.
output: $\mathcal{F}^{c_i}$: set of vertical fragments 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.
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. //

VerticalFrag(MA(C_i^{des}, C_i^{cont}), $MU(C_i^{des}, C_i^{cont})$) (1)
end {Vert_CA_SM}

Figure 11: Vertical Fragmentation – Complex Attributes and Simple Methods

Algorithm 4.1 *(icnm - Generates a set of intra class null methods for class C_i).*

Algorithm icnm
input: C_i: the database class; $\mathcal{M}_k^{c_d}$: methods of a set of database classes.
$\mathcal{C}_d$: set of database classes; $\mathcal{M}^{C_i}$: methods of class C_i.
EM-set(C_i^{des}) : set of extended methods ofC_i and its descendant classes.
output: **icnm**(C_i): a set of intra class null methods of the class C_i.
var
begin
icnm(C_i) = $\emptyset$ (1)
for each class $C_k \in C_d$ **do** (2)
For every method, $M^{k.n} \in$ EM-set(C_i^{des})) do (3)
For every method, $M^{i.j} \in \mathcal{M}^{C_i}$ do (3)
if ($M^{i.j} \in MR(M^{k.n})$) then
icnm(C_i) = **icnm**(C_i) $\cup$ $M^{k.n}$ (4)
end; {for $M^{k.n}$}
end; {for C_k}
end; {icnm}

Figure 12: The Intra Class Null Method Generator

fragmentation in this model splits a class so 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 three types namely: (1) those running directly on this class, (2) those running on descendants of this class, and (3) those running on methods of other classes in the database that use methods of this class. As in simpler models, the inheritance relationship between object base classes is accommodated by including in the method usage of methods of a class C_i, all usages by applications on their null method representatives at descendants of this class. Similarly, the method link is accommodated by including in the class' (C_i) method usage, uses by all the intra class null methods. The algorithm *icnm* of Figure 12 generates the set of methods (complex methods) of other classes using methods of this class C_i. Thus, in vertically fragmenting this class model, we first generate the method affinity matrix of the class that also accounts for the use of its methods by its descendant classes, then we modify the method affinity matrix to include use of the class's methods by other complex methods of other classes. This process uses the method usage and application frequency matrices of all descendants of the class as well as all classes whose complex methods (members of the intra class null method set of this class) use methods of this class.

The use of the intra class null methods of the class in the vertical fragmentation of the class makes the following contributions to the process. The affinities between actual methods of the class that arise from use by null methods from other classes are accounted for while computing the method affinity matrix of the class. The information captured by extended vertical fragments which include the intra class null methods is very useful during the allocation stage. During allocation, the cross class fragment affinity information of the extended fragments could be used to determine where it is optimal to place

Algorithm 4.2 *(CMLinkgraph - generates a Link tree rooted at C_i linking it to its complex method classes)*

Algorithm CMLinkgraph
input: $\mathcal{C}_d$**:** set of database classes; $\mathcal{C}_l$**:** set of classes on link paths $\mathcal{C}_l \subseteq \mathcal{C}_d$.
icnm(C_i) : set of intra class null methods of C_i; **P**(M^{c_d}) : methods of the set of database classes.
A(C_i): class composition hierarchy with a node for class C_i.
output: The Link graph (LG) tree rooted at C_i.
LG = (Γ,λ) where Γ is a set of nodes
λ is a set of arcs connecting nodes in Γ.
begin
// Starting from the class C_i, for every class C_j, whose method M_n^j appears //
// in the extended method set of the class, define a link from that class C_i to C_j. //
$LG \leftarrow$ initialized with a node $\forall C_i \in \mathcal{C}_d$; (1)
s.t. $\Gamma \leftarrow \{C_k | C_k \in \mathcal{C}_d\}$ and $\lambda = \emptyset$;
$C_l = C_i$ (2)
for each $C_j \in C_d$ **do** (3)
for each $(EM)_k^i \in$ icnm(C_i) **do** (4)
if $(EM)_k^i \in \mathcal{M}^{c_j}$ then (5)
begin
$\lambda = \lambda \cup (C_i \rightarrow C_j)$ (6)
$C_l = C_l \cup C_j$ (7)
end {if}
end; {for $(EM)_k^i$}
end; {for C_j}
return $(LG(C_i))$; (10)
end;

Figure 13: The Linkgraph Generator for Complex Method Classes

the actual fragments arising from these extended fragments.

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

Steps

Note. Steps P1, P3, P5 and P6 are same as steps M1, M3, M5 and M6 respectively.

P2. Modify method affinity to include usage of methods through complex method classes.

a. We repeat the operations in step P1 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 methods are represented in the intra class null method set of this class. Input to this Linkgraph generator is the intra class null method set of the class icnm(C_i). This algorithm starts from the class C_i as the root of the linkgraph and if any C_j has a method $M^{j.n}$ represented in icnm(C_i), then there is a link created from class C_i to class C_j in the linkgraph as $(C_i \rightarrow C_j)$. The linkgraph algorithm is as given in Figure 13. Then, with the linkgraph, we obtain the method usage and application frequency matrices of all the classes on this linkgraph.

b. Modify method affinity matrix of the class from step (P1b) using the usage matrices and application frequency matrices of the class and its complex method classes (from P2a). The algorithm *CMAMtrx* is given in Figure 10. This algorithm includes the complex method factor in method affinity matrix of class C_i. It accepts the method affinity matrix computed earlier and for each pair of extended methods of the class, it adds the complex method affinity value of the two extended methods to their current method affinity value (Lines 1-3 of Figure 10).

P4. Compute a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 7. The algorithm *MUsageMtrx* modifies method usage matrix of a class C_i to include a row for every application q_j that accesses a method of this class (lines 1-10 of Figure 7). Next the *Partition* algorithm [12, 15] takes the clustered affinity matrix and the modified method usage matrix and produces method fragments.

The formal algorithm for vertically fragmenting class models consisting of simple attributes and complex methods (algorithm *Vert_SA_CM*) is given in Figure 14. This algorithm takes as its inputs a set of user queries, class lattice of the database, a class in the lattice, the set of complex methods of other classes using methods of this class (intra class null methods), the method attribute reference of the methods of this class and the application frequency matrices of the class, its descendant classes and its complex method classes. Then, it returns a set of vertical fragments of this class. This algorithm is obtained by running the simple algorithm *VerticalFrag* using the method affinity matrix that contains information (method usage and application frequency matrices) from both descendant classes and complex method classes of the class being fragmented. The modified method usage matrix also includes a row for every application accessing the class being fragmented either through its descendant classes or its containing classes.

5 Complex Attributes and Complex Methods

This section presents an algorithm for vertically fragmenting classes consisting of complex attributes and complex methods. The database information that needs to be captured include: the inheritance hierarchy, the attribute link to reflect the part-of hierarchy and the method links to reflect the use of methods of objects of class C_i by objects of other classes. With this class model, 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. 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. As in simpler models, the inheritance relationship between object base classes is accommodated by including in the method usage of methods of a class C_i, all usages by applications of their null method

Algorithm 4.3 *(Vertical Fragments of simple attributes and Complex Methods)*

Algorithm Vert_SA_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
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.
$\mathcal{C}_i^{cmeth}$: set of complex method classes of C_i
$(EM)^{C_i}$: extended method set of C_i.
AR-set(C_i) : attribute reference set of methods of C_i
AF-set(C_i^{des}): application frequency matrices of C_i and and its descendants.
AF-set(C_i^{cmeth}): application frequency matrices of C_i and and its complex method classes.
output: $\mathcal{F}^{c_i}$: set of vertical fragments 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.
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. //

VerticalFrag(MA(C_i^{des}, C_i^{cmeth}), $MU(C_i^{des}, C_i^{cmeth})$) (1)
end; {Vert_SA_CM}

Figure 14: Vertical Fragmentation – Simple Attributes and Complex Methods

representatives at descendants of the class. Similarly, the attribute link between classes is accommodated by including in the method usage of methods of the contained class C_i (being fragmented), all usages by applications of their null method representatives at containing classes of this class. Finally, the method link is accommodated by including in the method usage of the methods of the class C_i, use by all the intra class null methods of the class from all classes in the object base generated with algorithm *icnm* of Figure 12. 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 descendants.

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 creating vertical fragments of classes consisting of complex attributes and complex methods are given below.

Steps

Note. Steps R1, R4, R6 and R7 are similar to steps N1, N3, N5 and N6 respectively.

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 shown in Figure 5 and using the Linkgraph from Figure 9 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). The algorithm for this process is given in Figure 10.

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. The linkgraph algorithm *CMLinkgraph* is as given in Figure 13. Then, with the linkgraph, we obtain the method usage and application frequency matrices of all the classes on this linkgraph.

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). The algorithm *CMAMtrx* for this process is given in Figure 10.

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

a. Create a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 7 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. Continue to modify method usage matrix of the class C_1 (algorithm *MUsageMrtx* of Figure 7) to include a row for every application q_j that accesses a null method representative of this class at all its containing classes

c. Finally, modify method usage matrix of the class C_i (algorithm *MUsageMrtx* of Figure 7) to include a row for every application q_j that accesses a null method representative of this class at all its complex method classes.

The formal algorithm *Vert_CA_CM* is given in Figure 15. This algorithm takes as its inputs a set of user queries, class lattice of the database, a class in the lattice, the set of complex methods of other classes using methods of this class (intra class null methods), the method attribute reference of the methods of this class and the application frequency matrices of the class, its descendant classes and its complex method classes. It returns a set of vertical fragments of this class. This algorithm is obtained by running the simple algorithm *VerticalFrag* using method affinity matrix that uses information (method usage and application frequency matrices) from both descendant classes, containing classes and complex method classes of the class being fragmented. The modified method usage matrix used also includes a row for every application accessing the class being fragmented either through its descendant classes, its containing classes or complex method classes.

6 Analysis

This section discusses time complexities of these algorithms and the results of our experiments analyzing their goodness and performance in comparison to an approach ignoring the object oriented features.

Algorithm 5.1 *(Vertical Fragments of Complex attributes and Complex Methods)*

Algorithm Vert_CA_CM
input: $\Pi^{C_i^{des}}$: set of user queries accessing C_i and its descendant classes.
C_i : the database class to fragment
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.
AR-set(C_i) : 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}): application frequency matrices of C_i and its 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. //

VerticalFrag(MA(C_i^{des}, , C_i^{cont}, C_i^{cmeth}), $MU(C_i^{des}, C_i^{cont}, C_i^{cmeth})$) (1)
end {Vert_CA_CM}

Figure 15: Vertical Fragmentation - Complex Attributes and Complex Methods

6.1 Complexities of Vertical Fragmentation Algorithms

The computation times of the vertical fragmentation algorithms for all four class models are based on that for the simple model consisting of simple attributes and methods. Assume f is the maximum number of fragments in a class, m is the maximum number of methods in a class, a is the maximum number of attributes in a class, q is the maximum number of applications accessing a database class, while c is the maximum number of classes in the database. The vertical fragmentation algorithm for the class model consisting of simple attributes and methods *VerticalFrag* of Figure 8 is of time complexity $O(cqm + m^2 + m^2 + cqm + m^2 + fma + af)$ which simplifies to $O(cqm + m^2 + fma)$. This is because line 1 of this algorithm *VerticalFrag* which is *UsageMtrx* algorithm is of O(cqm), and generation of method affinity matrix (line 2) has computation time of $O(m^2)$. The Bond Energy algorithm (line 3) has a computation time of $O(m^2)$, and modifying method usage matrices (line 4) has the same computation time of O(cqm) as generating original method usage matrices. Lines 6 though 9 used for attribute inclusion are of O(fma) while lines 10 through 13 for ensuring disjointness have time complexity O(af). The prevailing time complexity of the algorithm depends on the four parameters, number of classes, number of applications accessing a class, number of methods in a class and number of attributes in a class. However, since every method is expected to be accessing some attribute, then the number of attributes will be approximately the same as the number of methods allowing the term fma subsume m^2. If the number of classes and applications are much more than the number of methods, the complexity of *VerticalFrag* is O(cqm).

The time complexities of the other vertical fragmentation algorithms for more complex class models are the same as for *VerticalFrag* and are $O(cqm + m^2 + fma)$. However, the sizes of c and q increase as the complexity of the class model increases. Thus, these algorithms are polynomial in the sizes of these variable inputs.

6.2 Goodness of Our Fragmentation Schemes

We ran an experiment that gathers the costs of local and remote accesses to data on the network by applications over a period of time using fragments from our schemes and comparing it with those generated (1) with a scheme that does not account for inheritance, aggregation and method nesting hierarchies [12], and (2) with a scheme that supports no fragmentation at all but where methods are replicated at all sites. The costs of local and remote accesses incurred by the fragments from the three schemes were obtained using the Partition Evaluator (P.E.) as discussed in Chakravarthy *et al.* [1]. The two components of the P.E. are:

1. Irrelevant local attribute access cost which minimizes the square error for a fixed number of fragments and assigns a penalty factor whenever irrelevant methods are accessed in a particular fragment.

Test Cases	Our Approach With OO relationships	No OO-relationship Approach	No Fragmentation Approach
Test 1 Frag 1	{1,2,5,7,8,9,10}	{1,2,5,8,9}	{1,2,3,4,5,6,7,8,9,10}
Frag 2	{1,3,4,6}	{1,3,4,6,7,10}	{1,2,3,4,5,6,7,8,9,10}
Test 2 Frag 1	{1}	{1,5,6}	{1,2,3,4,5,6,7}
Frag 2	{1,2,3,4,5,6,7}	{1,2,3,4,7}	{1,2,3,4,5,6,7}
Test 3 Frag 1	{1}	{1,5}	{1,2,3,4,5,6,7,8,9,10}
Frag 2	{1,2,3,4,5,6,7,8,9,10}	{1,2,3,4,6,7,8,9,10}	{1,2,3,4,5,6,7,8,9,10}
Test 4 Frag 1	{1,3,2}	{1,5}	{1,2,3,4,5,6,7,8,9,10}
Frag 2	{1,4,5,6,7,8,9,10}	{1,2,3,4,6,7,8,9,10}	{1,2,3,4,5,6,7,8,9,10}
Test 5 Frag 1	{1}	{1,4}	{1,2,3,4}
Frag 2	{1,2,3,4}	{1,2,3}	{1,2,3,4}
Test 6 Frag 1	{1,3,4}	{1,4}	{1,2,3,4}
Frag 2	{1,2}	{1,2,3}	{1,2,3,4}
Test 7 Frag 1	{1}	{1,5,6}	{1,2,3,4,5,6,7}
Frag 2	{1,2,3,4,5,6,7}	{1,2,3,4,7}	{1,2,3,4,5,6,7}
Test 8 Frag 1	{1}	{1,3,4,7}	{1,2,3,4,5,6,7}
Frag 2	{1,2,3,4,5,6,7}	{1,2,5,6}	{1,2,3,4,5,6,7}
Test 9 Frag 1	{1,5,7,9,10}	{1,5,7}	{1,2,3,4,5,6,7,8,9,10}
Frag 2	{1,2,3,4,6,8}	{1,2,3,4,6,8,9,10}	{1,2,3,4,5,6,7,8,9,10}

Figure 16: Fragments From the Three Approaches

2. Relevant Remote Method Access Cost which computes for a set of applications running on a fragment, the ratio of the number of remote methods to be accessed to the total number of methods in each of the remote fragment summed over all fragments and applications.

The total penalty cost (P.E. value) for any scheme is the sum of its local irrelevant and remote relevant costs. Comparative analysis of the total penalty costs of the three approaches shows our approach performs best with the overall lowest total processing cost while the approach with no fragmentation at all performs worst. With nine test cases, the experiment yielded the fragments shown in Figure 16 and the costs shown in Figure 17 [4], while the graph of the total penalty cost is given as Figure 18.

7 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 algorithms necessary to support these four class models.

Current research efforts include extending our performance measurements to demonstrate gains

[4]These two tables are squeezed into one page to save space and can be kept on separate pages if needed.

Test Cases	Approach	Local Irrelevant Access Costs(units)	Remote Relevant Access Costs(units)	Total Penalty Costs(units)
Test 1	Ours	20847	3991	24838
	No-OO	21173	6537	27710
	No frag	42396	0	42389
Test 2	Ours	30032	0	30032
	No-OO	31791	7546	39337
	No frag	600064	0	60064
Test 3	Ours	22737	0	22737
	No-OO	23663	6718	30381
	No frag	45474	0	45474
Test 4	Ours	17172	5712	22737
	No-OO	20549	7187	27736
	No frag	38504	0	38504
Test 5	Ours	2259	0	2259
	No-OO	1971	2355	4326
	No frag	4518	0	4518
Test 6	Ours	2263	1439	3702
	No-OO	1740	2466	4206
	No frag	4461	0	4461
Test 7	Ours	38660	0	38660
	No-OO	41298	7977	49275
	No frag	77321	0	77321
Test 8	Ours	39324	0	39324
	No-OO	42580	11468	54049
	No frag	78649	0	78649
Test 9	Ours	33531	8839	42371
	No-OO	31052	15365	46418
	No frag	67290	0	67290

Figure 17: Costs of Processing Fragments Using Three Approaches

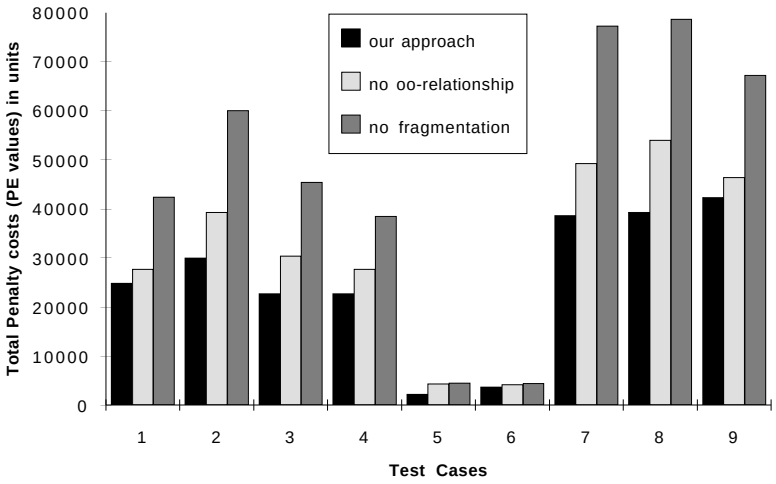

Figure 18: Total Penalty Costs of Three Approaches

in terms of both processing costs and response time and/or throughput. We are also interested in identifying issues involved in applying these techniques to real life applications. 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 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 and attempting to discover suitable theoretical models for allocating fragments to distributed sites.

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. A Comprehensive Approach to Horizontal Class Fragmentation in a Distributed Object Based System. *International Journal of Distributed and Parallel Databases*, 1, 1995. Kluwer Academic Publishers.

[4] C.I. Ezeife and Ken Barker. Vertical Class Fragmentation for Advanced Object Models in a Distributed Object-Based System. In *Proceedings of the 7th International Conference on Computing and Information, Trent, Canada*, volume 1, pages 613–632. Ieee Publication, July 1995.

[5] 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*, volume 1. Morgan Kaufmann Publishers, Inc, 1975.

[6] M.F. Hornick and S.B. Zdonik. A Shared, Segmented Memory System for an Object-Oriented Database. *ACM Transactions on Office Information Systems*, 5(1), January 1987.

[7] Itasca Systems Inc. *Itasca Distributed Object Database Management System*, 1991. Technical Summary Release 2.0.

[8] 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.

[9] M.L. Kersten, S. Plomp, and C.A Van Den Berg. Object Storage Management in Goblin. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[10] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed Object Management in Thor. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[11] W.T. McCormick, P.J. Schweitzer, and T.W. White. Problem Decomposition and Data Reorganization by a Clustering Technique. *Operations Research*, 20(5), 1972.

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

[13] S.B. Navathe and M. Ra. Vertical Partitioning for Database Design: A Graphical Algorithm. In *Proceedings of the ACM SIGMOD*. ACM, 1989.

[14] Gruber Oliver and Amsaleg Laurent. Object Grouping in EOS. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

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

[16] 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.

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

[18] S.B. Yao, S.B. Navathe, and J.L. Weldon. An Integrated Approach to Database Design. In *Data Base Design Techniques I: Requirements and Logical Structures*, pages 1–30. Springer-Verlag, New York, 1982. Lecture Notes in Computer Science 132.