Vertical Class Fragmentation in a Distributed Object Based System*

C.I. Ezeife and Ken Barker
Advanced Database Systems Laboratory
Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2
{christie,barker}@cs.umanitoba.ca
Technical Report: 94-03

April 1994

Abstract

Processing costs in distributed processing 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 behavioural 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.

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 marnner 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

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

incur unreasonable costs of data replication. This requires organizing database entities to reduce the amount of irrelevant data accessed by applications while 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 static and system requirements analysis which define 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 these design activities include the frequency of user applications and their reference patterns. The output of this static and system analysis are the GCS and the access pattern information which constitute inputs to the distribution design. 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 [6], ENCORE [5], GOBLIN [8], THOR [9], and EOS [14] which will benefit from fragmentation [7]. 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 [2, 3], 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. Section 2 presents vertical fragmentation algorithms for class models, including: a model consisting of simple attributes and simple methods (Section 2.1); a model where attributes are composed in a part-of hierarchy and accessed using simple methods (Section 2.2); a model consisting of simple attributes and complex methods (Section 2.3); and a model consisting of complex attributes and complex methods (Section 2.4). Finally, Section 3 concludes and suggests future research directions.

2 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 [4], Navathe *et al.* [12], Cornell and Yu [1], Navathe and Ra [13], Özsu and Valduriez [15] and Mutheraj *et al.* [11].

Hoffer and Severance [4] 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.* [10] 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. They apply vertical partitioning to three database types, namely, distributed databases, one memory level database, and databases arranged in a memory hierarchy. 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 [1] optimized this work by developing an algorithm that obtains an optimal binary partitioning for relational databases. They use knowledge of physical factors[1] to decrease the number of disk accesses. Further refinement is accomplished by applying the binary vertical partition algorithm iteratively [12]. Navathe and Ra [13] developed an algorithm that uses a graphical technique where the attribute affinity matrix is represented as a graph from which a linearly connected spanning tree is generated and all cycles on the spanning

[1] For example, the type of scan used or the rate at which data is supplied from secondary storage.

tree form fragments of the relation.

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.

Muthuraj *et al.* [11] 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. The vertical partition algorithms whose results could be evaluated using this objective function include: the bond energy algorithm [10], binary vertical partition [12], Ra's graphical algorithm [13] and an exhaustive enumeration algorithm.

Vertical Fragmentation (objects): Karlapalem *et al.* [7] 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].

3 The DOBS model

[2]

A distributed object based system is a collection of local object bases distributed at different sites, interconnected by a network. We assume the database management system (DBMS) is distributed and

[2] A note to the referee/editor - this section is quite complete and is included to provide a context for this research. If this is considered too verbose, it can be reduced or eliminated, but is included here in the interest of completeness

the objects making up the object base are placed around the network. The general architecture of a distributed DBMS is available in Özsu and Valduriez [15].

3.1 The Data Model

The data in a DOBS consists of a set of encapsulated objects. The data values (attributes) are bundled with the methods (procedures) for manipulating them to form an *encapsulated* object. Objects with common attributes and methods belong to the same class, and every class has a unique identifier. Inheritance allows reuse and incremental redefinition of new classes in terms of existing ones. Parent classes are called *superclasses* while classes that inherit attributes and methods from them are called *subclasses*. The database contains a root class called *Root* which is an ancestor of every other class in the database. The overall inheritance hierarchy of the database is captured in a class lattice. A class is an ordered relation $\mathcal{C}$ = ($\mathbf{K}$,$\mathcal{A}$,$\mathcal{M}$,$\mathcal{I}$) where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[3]. 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, $\mathcal{C}_h$ = (K,$\mathcal{A}$,$\mathcal{M}$,$\mathcal{I}'$). Each vertical fragment ($\mathcal{C}^v$) of a class contains its class identifier, and all of its instance objects for only some of its methods ($\mathcal{M}' \subseteq \mathcal{M}$) and some of its attributes ($\mathcal{A}' \subseteq \mathcal{A}$). Thus, $\mathcal{C}^v$ = (K,$\mathcal{A}'$,$\mathcal{M}'$,$\mathcal{I}$). 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

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

data accessed by applications is reduced while reducing the amount of data that needs to be transferred between sites.

4 Vertical Fragmentation

This section presents vertical fragmentation algorithms for all four class models, consisting of simple attributes using simple methods (Section 2.1), complex attributes using simple methods (Section 2.2), simple attributes using complex methods (Section 2.3), and complex attributes using complex methods (Section 2.4). 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 fragments 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 paper 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 application access pattern information using the same technique described earlier [4, 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 [2, 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 algorithm [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. We first present some definitions and assumptions before presenting the algorithm.

The explicit assumptions are:

1. Objects of a subclass physically contain only pointers to objects of its superclasses that are "logically" part of them. In other words, an object of a class is made from the aggregation of all those objects of its superclasses that are logically part of this object.

2. Application and database information are 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 fragment of this class. The significance of this assumption lies in the fact that providing a fragmentation scheme for the alternative storage structure would call for an extensive modification of the fragmentation algorithm in order to accommodate the storage structure in the first assumption.

The major data requirements related to applications is their access frequencies. Let Q = $\{Q_1, Q_2, \ldots, Q_q\}$ be the set of user queries (applications) running object methods from the set of all methods denoted $\{M^{i1.j}, M^{i2.k}, \ldots, M^{in.p}\}$. Then, for each query Q_k and each method $M^{i.j}$ (the jth method of class C_i), we associate a method usage value denoted as use(Q_k, $M^{i.j}$) where use(Q_k, $M^{i.j}$) = 1 if method $M^{i.j}$ is referenced by query Q_k, 0 otherwise. Thus, for each class, we define a method usage matrix. The cardinality of a class is the number of instance objects in the class (denoted **card**(C_i)).

Definition 4.1 *A user query* accessing database objects is a sequence of method invocations on an object or set of objects of classes. The invocation of method j on class C_i is denoted by $M^{i.j}$ and a user query Q_k is represented by $\{M^{i1.j}, M^{i2.k}, \ldots, M^{in.p}\}$ where each M in a user query refers to an invocation of a method of a class object. ■

Definition 4.2 *Method Attribute Reference MAR($M^{i.j}$)* of a method $M^{i.j}$ of a class C_i is the set of all attributes of C_i referenced by $M^{i.j}$. ■

Definition 4.3 *A null method of a class C_i, denoted NM(C_i)*, with respect to a superclass C_s is a place holder for the superclass's method. A null method is denoted by: (original class.method name) where original class is the name of the superclass and method name is the method in the subclass. ■

Definition 4.4 *An Extended Method of a class, C_i, $(EM)^{i.k}$* is either an original method of the class, $M^{i.j}$ or a null method of the class $NM(C_i)$. Thus, $(EM)^i = M^{c_i} | NM(C_i)$. ■

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

Definition 4.5 *Access frequency of a query* is the number of accesses a user application makes to "data". If Q = $\{q_1, q_2, \ldots, q_q\}$ is a set of user queries, acc(q_i,d_j) indicates the access frequency of query q_i on "data" item d_j where data item d_j can be a class, a fragment of a class, an instance object of a class, an attribute or method of a class. ■

Definition 4.6 *Instance Object join ($\odot$)* between a pointer to an instance object of a superclass and an instance object (I_j) of a class C_i returns the "complete" instance object consisting of the two classes that represent the actual instance object of the class. ■

To illustrate the aggregate returned by the object join function, we consider the following example. In a database with *Student* a superclass of *Grad*, an instance object I_3 of *Grad* is represented as (Student pointer5) $\odot$ {537msc11,Mary Smith}. This means that the actual I_3 of *Grad* is the quantity representing the instance object I_5 of the superclass *Student* added to the quantity {537msc11,Mary Smith} 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 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 4.7 *Subclass affinity of two extended methods $(EM)^{i.j}$,$(EM)^{i.k}$ of a class C_i, denoted saff($(EM)^{i.j}$,$(EM)^{i.k}$)* is a measure of how frequently methods/attributes of the subclasses of C_i and methods/attributes of the class are needed together by applications running at any particular site. saff($(EM)^{i.j}, (EM)^{i.k}$) =
$\sum_{o=1}^{w} \sum_{p \| use(q_p,(EM)^{i.j})=1 \wedge use(q_p,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_p) acc_l(q_p)$,
where w is the number of subclasses, p is some application and S_l ranges over all sites. ■

Definition 4.8 *Method Affinity between two extended methods of a class C_i, MA($(EM)^{i.j}$,$(EM)^{i.k}$)* measures the bond between two extended methods of a class according to how they are accessed by applications. MA($(EM)^{i.j}, (EM)^{i.k}$) =
$(\sum_{p \| use(q_p,(EM)^{i.j})=1 \wedge use(q_p,(EM)^{i.k}=1} \sum_{\forall s_l} ref_l(q_p) acc_l(q_p)) + saff((EM)^{i.j}, (EM)^{i.k})$
where $ref_l(q_p)$ is the number of accesses to methods ($(EM)^{i.j}$,$(EM)^{i.k}$) for each execution of application q_p at site s_l and $acc_l(q_p)$ is the application access frequency modified to include frequencies at different sites. This generates the method affinity matrix (MA), an n * n matrix. ■

Definition 4.9 *Containing Class affinity between two extended methods $(EM)^{i.j}$ and $(EM)^{i.k}$ of a class C_i, $ccaff((EM)^{i.j}, (EM)^{i.k})$* is a measure of how frequently methods/attributes of containing classes of C_i and methods/attributes of the class are needed together by applications running at any particular site.
ccaff($(EM)^{i.j}$,$(EM)^{i.k}$) = $\sum_{o=1}^{w} \sum_{k \| use(q_k,(EM)^{i.j})=1 \wedge use(q_k,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k)$,
where w is the number of containing classes, C_i is the class and S_l ranges over all sites. ■

Definition 4.10 *Complex Method affinity between two extended methods, $(EM)^{i.j}$ and $(EM)^{i.k}$ of a class C_i, cmaff($(EM)^{i.j}$,$(EM)^{i.k}$)* measures how frequently methods/attributes of other classes in the database and method/attributes of the class C_i are needed together by applications running at any particular site.
cmaff($(EM)^{i.j}$,$(EM)^{i.k}$) $= \sum_{o=1}^{d} \sum_{k \| use(q_k,(EM)^{i.j})=1 \wedge use(q_k,(EM)^{i.k})=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k)$,
where d is the number of database classes, C_i is the class and S_l ranges over all sites. ■

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

Definition 4.11 *Attribute/Attribute Affinity* AAA($A^{i.j}$,$A^{i.m}$) between two attributes of the same class C_i is the sum of the access frequencies of all methods accessing these two attributes together at all sites.
AAA($A^{i.j}$,$A^{i.m}$) $= \sum_{k|A^{i.j} \in MAR(M^{in.k}) \wedge A^{i.m} \in MAR((M^{in.k})} \sum_{l=1}^{m} acc_l(M^{in.k}, A^{i.j}) + acc_l(M^{in.k}, A^{i.k})$
where $acc_l(M^{in.k}, A^{i.j}$ is the number of accesses made to the attribute $A^{i.j}$ by method $M^{in.k}$ at site s_l. ■

Definition 4.12 *Attribute/Method Affinity* AMA($A^{i.j}$,$M^{i.m}$) between an attr ibute and a method of the same class C_i is the sum of the access frequencies of all methods using this attribute and this method together at all sites. AMA($A^{i.j}$, $M^{i.m}$) =

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

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

Definition 4.13 *Attribute Fragment Affinity AFA($A^{i.m}$,$F^{i.j}$)* is a measure of the affinity between attribute $A^{i.m}$ and vertical fragment $F^{i.j}$, and is the sum of all the attribute/attribute and attribute/method affinities between $A^{i.m}$ and all attributes and methods of the class fragment $F^{i.j}$. AFA($A^{i.m}$, $F^{i.j}$) = $\sum_{k|A^{i.m} \in F^{i.j} \wedge A^{i.k} \in F^{i.j})} AAA(A^{i.m}, A^{i.k}) + \sum_{k|A^{i.m} \in F^{i.j} \wedge M^{i.k} \subset n F^{i.j})} AMA(A^{i.m}, M^{i.k})$. ■

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 4.1 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 hierarchy. 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.

5 Simple Attributes and Methods

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 to this method's null equivalent at a class' subclass at the same site. 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.

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 attribute, and we generate a 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 subclasses 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 subclasses 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 generate 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 and group methods/attributes of a class 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}) : set applications accessing extended methods of C_i and its descendant classes.

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

MAR-set(C_i): set of method attribute references of 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 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 is generated by the Algorithm *UsageMtrx* defined in Figure 1. This algorithm accepts a tree rooted at a class C_i and generates 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

Algorithm 5.1 *(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.//
(1) $\mathcal{C}_c = \{C_i\}$
(2) **while** $\mathcal{C}_c \neq \emptyset$ **do**
(3) C_k = a class c $\in \mathcal{C}_c$
(4) **if** C_k is not a leaf class
begin
(5) $C_c = C_c \cup children(C_k)$
(6) **for each** application $q_j \in$ EMapplic-set(C_i^{des}) **do**
(7) For each method, $M^{k.m} \in$ EM-set(C_k)) do
(8) **if** use(q_j,$M^{k.m}$) = 1 then
(9) MU(q_j,$M^{k.m}$) = 1
(10) else MU(q_j,$M^{k.m}$) = 0
end; {for q_j}
end; {if C_k}
$C_c = C_c - C_k$
end; {while}
end {UsageMtrx}

Figure 1: Original method Usage Matrix Generator

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 1). The application frequency matrices are part of the input from pre-analysis of the system.

M2. Define method affinity matrix of the class using the modified approach given in the algorithm *MAMtrx* of Figure 2. 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 4.8) between the two extended methods of the class in this (row,column) position (Lines 1-4 of Figure 2).

M3. Use the Bond Energy Algorithm (BEA) developed by [10] 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.

Algorithm 5.2 *(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.
$\mathcal{C}_i^{des}$: 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 4.8. //
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 2: Method Affinity Matrix Generator

M4. Generate a modified method usage matrix of the class as described in the Algorithm *MUsageMtrx* of Figure 3. 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 3). 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 for most of the time, accessed solely by distinct sets of applications.

M5. Use method-attribute reference information of the methods in each method fragment (MAR of Definition 4.2) to include in each method fragment all attributes of the class accessed by 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 4.1 to decide which vertical fragment to keep each overlapping attribute. The Attribute Placement Affinity Rule 4.1 determines the affinity between the overlapping attribute and each of the fragments containing it using the AMA and AFA statistics of Definitions 4.12 and 4.13, respectively. It places the attribute in the fragment with highest affinity measure and removes the attributes from every other.

Algorithm 5.3 *(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 $q_j \in \mathrm{EMapplic-set}(\mathbf{C_i^{des}})$) **do** (2)
For each null method, $(NM)^{o.k} \in \mathrm{EM-set}(\mathbf{C_i^{des}})$ 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 3: Modified Usage Matrix Generator

The formal algorithm for vertically fragmenting a class consisting of simple attributes and simple methods is presented as Algorithm *VerticalFrag* of Figure 4. 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 method attribute reference 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 generating the method usage matrices of the class and its descendant classes (line 1[4]) before computing the method affinity matrix of the class (line 2). Next, the clustered method affinity matrix of the class generated (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.

5.1 An Example

We use the class lattice depicted in Figure 5 and the sample data [5] illustrated in Figure 6 for this example.

Assume 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 method usage matrix of the class *Person* obtained from Step 1 of algorithm *VerticalFrag*(Figure 4) using algorithm *UsageMtrx*(Figure 1) is:

	M_1	M_2	M_3	M_4
Q_1	1	0	1	0
Q_2	1	1	0	1
Q_3	0	1	1	0

M_1 = m.ssno-of M_2 = m.whatname

M_3=m.age-in-year M_4=m.newaddr

Application frequencies at the three sites obtained prior to this design from system analysis are:

$acc_1(q_1)$=10	$acc_2(q_1)$=5	$acc_3(q_1)$=5
$acc_1(q_2)$=20	$acc_2(q_2)$=15	$acc_3(q_2)$=0
$acc_1(q_3)$=30	$acc_2(q_3)$=10	$acc_3(q_3)$=0

[4]All line numbers in this paragraph refer to Figure 4

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

Algorithm 5.4 *(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.
MAR-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),$\mathcal{Q}^{C_i^{des}}$,$\mathcal{C}_i^{des}$) (2)
CA^i = **BEA**($(MA)^i$) (3)
MU^i = **MUsageMtrx**($\mathcal{C}_i^{des}$,$(EM)^{C_i}$, **MU-set**(C_i)) (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 $A^{i.n} \in MAR((EM)^{i.m})$ **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 4.1 (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 4: Class Vertical Fragments Generator

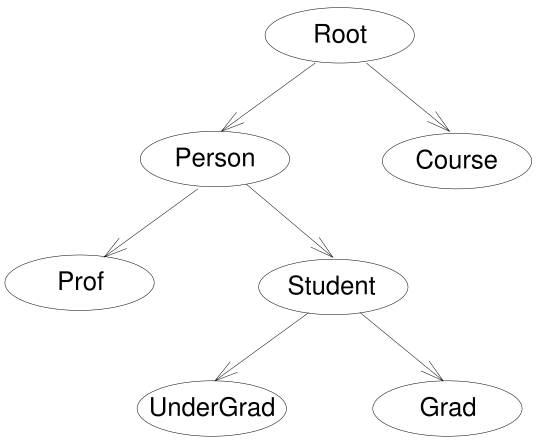

Figure 5: Class Lattice of Sample Object Base

Person = {k.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 $\odot$ {k.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 $\odot$ {k.stuno,a.dept,a.feespaid,a.coursetaken},
{m.stuno-of,m.dept-of,m.owing},
I_1 (person pointer1) $\odot$ {Student1,Math,Y}
I_2 (person pointer4) $\odot$ {Student2,Computer Sc.,N}
I_3 (person pointer2) $\odot$ {Student3,Stats,Y}
I_4 (person pointer3) $\odot$ {Student4,Computer Sc.,N} } }
Grad = Student pointer $\odot$ {k.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 6: The Sample Object Database Schema

To define the method affinity matrix used for vertical fragmentation of the class *Person*, we 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 frequeny matrix of this class are given in Figure 7. 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.

Q_3: List student numbers of those students who have not paid their term fees.

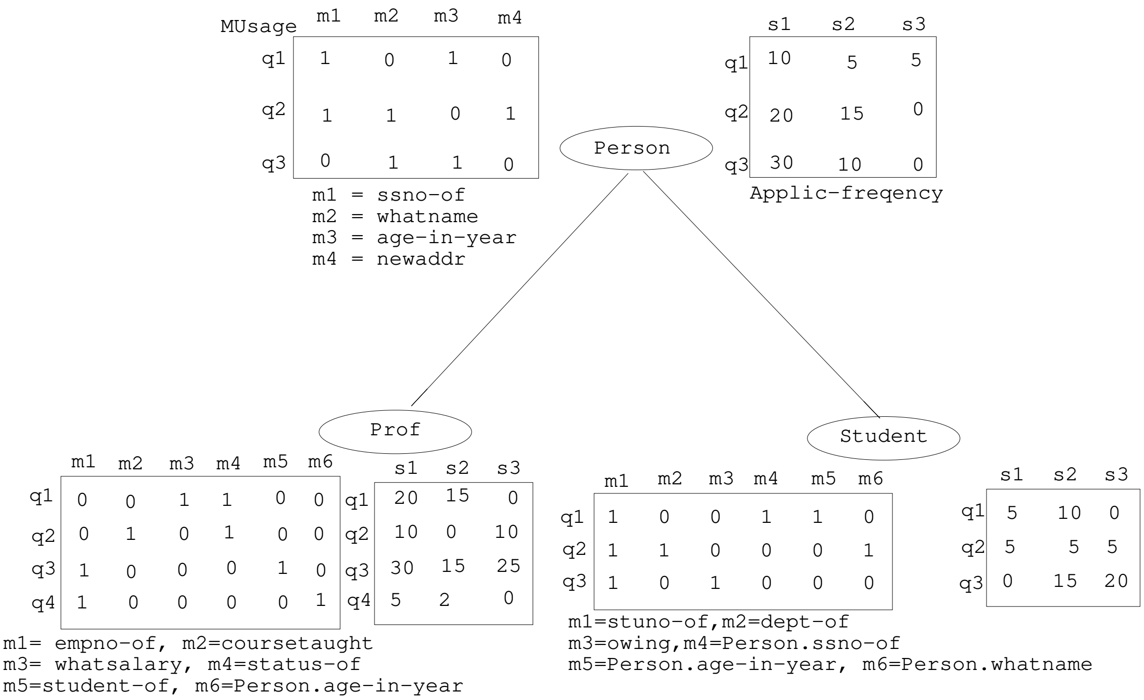

Figure 7: The Method Usage and Application Frequency matrices of the Classes

Similarly, the method usage matrix of the subclass *Student* derived from these applications as well as the application frequeny matrix of this class are given in Figure 7. 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.whatlast.* All the matrices needed to compute the method affinity matrix of the class *Person* are given in Figure 7.

Executing line 2 of the algorithm *VerticalFrag* of Figure 4 using the method usage and application frequency matrices of the class *Person* and its descendant classes *Prof and Student* yields the following method affinity matrix for the class *Person.* To illustrate the creation of the required method affinity matrix for the class *Person*, we give the following specific examples.

$\mathrm{MA}(M^{person.1},M^{person.3}) = \sum_{k=1}^{1}\sum_{l=1}^{3} acc_l(q_1) +$

$saff_{prof}(M^{person.1}, M^{person.3}) + saff_{student}(M^{person.1}, M^{person.3}).$

$= (10+5+5) + (0) + (5+10+0) = 35$

$\mathrm{MA}(M^{person.2},M^{person.2}) = \sum_{k=1}^{1}\sum_{l=1}^{3}(acc_l(q_2) + acc_l(q_3)) +$

$saff_{prof}(M^{person.2}, M^{person.2}) + saff_{student}(M^{person.2}, M^{person.2})$

$= ((20+15+0)+(30+10+0)) + 0 + (5+5+5)$

$= 35 + 40 + 0 + 15 = 90$

The method affinity matrix for the class *Person* generated from the execution of the algorithm *MAMtrx* of Figure 2 as required by line 2 of algorithm *VerticalFrag* of Figure 4 is given below:

	M_1	M_2	M_3	M_4
M_1	70	35	35	35
M_2	35	90	40	35
M_3	35	40	82	0
M_4	35	35	0	35

M_1 = m.ssno-of M_2 = m.whatname

M_3=m.age-in-year M_4=m.newaddr

The clustered affinity matrix for the class *Person* that arose from line 3 of algorithm *VerticalFrag* of Figure 4 is:

	M_4	M_1	M_2	M_3
M_4	35	35	35	0
M_1	35	70	35	35
M_2	35	35	90	40
M_3	0	35	40	82

M_1 = m.ssno-of M_2 = m.whatname

M_3=m.age-in-year M_4=m.newaddr

The clustered affinity matrix shows clustering together of methods with high affinity for each other and the next step is to split the methods of the class accordingly, called partitioning. Since the splitting point may not be obvious from the clustered affinity matrix, and a large matrix may have more than one cluster, a partitioning algorithm is employed to identify groups of methods that are accessed solely, for most of the time, by distinct sets of applications. Since the set of applications accessing method of this class includes those accessing them through methods of its descendant classes, our algorithm modifies the original method usage matrix of this class to include use of the methods by applications running on its descendant classes. The modified method usage matrix generated from line 4 of algorithm VerticalFrag (Figure 4) is:

	M_1	M_2	M_3	M_4
Q_1	1	0	1	0
Q_2	1	1	0	1
Q_3	0	1	1	0
Q_4	0	0	1	0
Q_5	1	0	1	0
Q_6	0	1	0	0

M_1 = ssno-of M_2 = whatlast

M_3=age-in-year M_4=newaddr

This accounts for class method usage for class *Person* by applications running on its descendant classes. Thus, 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*. Finally, Q_6 represents the use of method *whatname* by the descendant class *Student*. The method fragments produced from line 5 of the algorithm *VerticalFrag* after running the *Partition* algorithm are: $F_1 = \{m_1, m_4\}$ and $F_2 = \{m_1, m_2, m_3\}$. Note that since m_1 is the class identifier method, it is contained in every fragment. The fifth step of the algorithm *VerticalFrag* is incorporating attributes in these method fragments (lines 6-9) using method attribute reference information. Suppose the method/attribute reference (MAR) of these methods are as follows: MAR(ssno-of) = {ssno}, MAR(whatlast) = {name}, MAR(age-in-year) = {age}, and MAR(newaddr) = {address}. Then, the attribute/method fragments now become: $F_1 = \{ssno, address, ssno-of, newaddr\}$
$F_2 = \{ssno, name, age, ssno-of, whatlast, newaddr\}$ Since there are no overlaps of non-class identifier attributes/methods, step 6 of the algorithm VerticalFrag of Figure 4 (lines 10-13) leaves the fragments unchanged.

6 Complex Attributes and Simple Methods

This section presents an algorithm for vertically fragmenting classes consisting of complex attributes that support a class composition using simple method invocations. Thus, both the inheritance and the attribute link relationship 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. Encapsulation means user applications do not directly access object attribute values except through its methods. Since every method in the object accesses a set of class attributes, we group the class' methods based on application access patterns using the technique described earlier [4, 12]. We then extend each method group (fragment) to incorporate all attributes accessed by that group's methods. This places in one fragment those methods of a class usually accessed together by applications. The measure of togetherness is the affinity of methods which shows how closely related the methods are. The major data requirements related to applications is their access frequencies as defined earlier.

6.1 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 on a class' subclasses accomodates 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 matrix of the class. The steps for producing the desired vertical fragments are outlined below. **Steps**

N1. Generate initial Method Affinity matrix for the class as:

- **a.** Obtain the method usage and application frequency matrices of the class and its descendant classes using the Algorithm *UsageMtrx* defined in Figure 1.
- **b.** Define the method affinity matrix of the class from step (N1a) using the usage matrices and application frequency matrices of the class as defined in algorithm *MAMtrx* of Figure 2.

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 1 and the *CCLinkgraph* from Figure 8. CCLinkgraph returns a tree rooted at the class being fragmented (contained 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 8).
- **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 9. 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).

N3. Use the Bond Energy Algorithm developed by [10] as presented in [15, 12] to generate clustered affinity matrix of the class.

N4. . Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 3 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 3) 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

Algorithm 6.1 *(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 8: The Linkgraph Generator For Containing Classes of C_i

Algorithm 6.2 *(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 4.9. //
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(row,col) + ccaff($(EM)^{i.row}$,$(EM)^{i.col}$) (4)
end; {for col }
end; {for row}
end;

Figure 9: Complex Attribute Modified Method Affinity Matrix Generator

of the methods. The *Partition* algorithm finds sets of methods that are most of the time, accessed solely by distinct sets of applications.

N5. Use method-attribute reference information of the class to include attributes of the class used by methods of each fragment in the fragment.

N6. Since there may be problems of overlapping attributes in more than one fragment, use Attribute Placement Affinity Rule 4.1 to decide which vertical fragment to keep each overlapping attribute.

The formal algorithm for vertically fragmenting a class consisting of complex attributes and simple methods is presented as algorithm *Vert_CA_SM* of Figure 10. 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 include 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 scheme 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 10 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.

7 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 relationship and the method link relationship. 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. The former requirement is accomplished with static analysis and the latter is inherent in the object model. Vertical 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 11 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

Algorithm 6.3 *(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
$\mathbf{L(C)}$: 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.
MAR-set(C_i) : method attribute reference set of methods of C_i
AF-set(C_i^{des}): application frequency matrices of C_i and 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 10: Vertical Fragmentation – Complex Attributes and Simple Methods

Algorithm 7.1 *(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 $(M^{i.j} \in MMR(M^{k.n}))$ then
          icnm(C_i) = **icnm**(C_i) $\cup M^{k.n}$ (4)
    end; {for $M^{k.n}$}
  end; {for C_k}
end; {icnm}

Figure 11: The Intra Class Null Method Generator

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 use methods of this class (members of the intra class null method set 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 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

P1. Generate the original method usage matrix for the class.

a. Obtain the method usage and application frequency matrices of the class and its subclasses as defined in algorithm *UsageMtrx* of Figure 1.

Algorithm 7.2 *(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 $\mathcal{C}_l \subseteq \mathcal{C}_d$.
  $\mathbf{icnm}(C_i)$: set of intra class null methods of C_i; $\mathbf{P}(M^{c_d})$: methods of the set of database classes.
  $\mathbf{A}(C_i)$: 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 12: The Linkgraph Generator for Complex Method Classes

b. Define the method affinity matrix of the class from step (1a) using the usage matrices and application frequency matrices of the class as defined in algorithm *MAMtrx* of Figure 2. By doing this, we are producing an initial method affinity matrix of the class that includes use of its methods by methods of its descendant classes.

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 $M^{n.j}$ 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 12. 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 9. 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 9).

P3. Use the Bond Energy Algorithm developed by [10] as presented in [12, 15] to generate clustered affinity matrix of the class.

P4. Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 3. 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 3). Next the *Partition* algorithm [12, 15] takes the clustered affinity matrix and the modified method usage matrix and produces method fragments.

P5. Use method-attribute reference information of the methods in each method fragment (MAR of Definition 4.2) to include in each method fragment all attributes of the class accessed by methods of the fragment.

P6. Since there may be problems of overlapping attributes in more than one fragment, use Attribute Placement Affinity Rule 1 to decide in which vertical fragment to keep overlapping attributes.

The formal algorithm for vertically fragmenting class models consisting of simple attributes and complex methods (algorithm *Ver_SA_CM*) is given in Figure 13. 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 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 also includes a row for every application accessing the class being fragmented either through its descendant classes or its containing classes.

8 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

Algorithm 7.3 *(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
$\mathbf{L(C)}$: the class lattice
$\mathcal{C}_i^{des}$: set of descendant classes of C_i
$\mathbf{icnm}(C_i)$: 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.
MAR-set(C_i) : method attribute reference set of methods of C_i
AF-set(C_i^{des}): application frequency matrices of C_i and 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 13: Vertical Fragmentation – Simple Attributes and Complex Methods

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. The algorithm essentially reuses the algorithms for simpler models. 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 subclasses 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 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 11. 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 subclasses.
2. The method affinity matrix is modified using method usage and application frequency matrices of the class and its containing classes.
3. It is further modified using method usage and application frequency matrices of the class and its complex method classes.

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

Steps

Q1. Generate initial Method Affinity matrix for the class as:

- **a.** Obtain the method usage and application frequency matrices of the class and its descendants as defined in algorithm *UsageMtrx* of Figure 1.
- **b.** Define the method affinity matrix of the class from step (Q1a) using the usage matrices and application frequency matrices of the class as defined in algorithm *MAMtrx* of Figure 2. By doing this, we are producing an initial method affinity matrix of the class that includes use of its methods by methods of its subclasses.

Q2. Modify the method affinity matrix from step Q1 to include use of the methods through its containing classes.

- **a.** We repeat the operations in step Q1 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 1 and using the Linkgraph from Figure 8 that returns a tree rooted at the class that showing the attribute link between that class and other classes in the database.
- **b.** Modify method affinity matrix of the class from step (Q1b) using the usage matrices and application frequency matrices of the class and its containing classes (from N2a). The algorithm for this process is given in Figure 9.

Q3. Modify method affinity matrix from step Q2 above to include usage of methods through complex method classes.

- **a.** We repeat the operations in step Q2 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 12. 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 (Q2b) using the usage matrices and application frequency matrices of the class and its complex method classes (from Q3a). The algorithm *CMAMtrx* for this process is given in Figure 9.

Q4. Use the Bond Energy Algorithm developed by [10] as presented in [12, 15] to generate clustered affinity matrix of the class.

Q5. Modify the original method usage matrix to include method usages through complex class and complex method classes.

- **a.** Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 3 which modifies the method usage matrix of a class C_i to include a row for every application q_j that accesses a null method representative of this class at all its descendant classes.
- **b.** Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 3 which modifies the method usage matrix of a class C_i to include a row for every application q_j that accesses a null method representative of this class at all its containing classes

c. Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 3 which modifies the method usage matrix of a class C_i to include a row for every application q_j that accesses a null method representative of this class at all its complex method classes.

Q6. Use method-attribute reference information of the methods in each method fragment (MAR of definition 4.2) to include in each method fragment all attributes of the class accessed by methods of the fragment.

Q7. Since there may be problems of overlapping attributes in more than one fragment, use Attribute Placement Affinity Rule 4.1 to decide which vertical fragment to keep each overlapping attribute.

The formal algorithm *Vert_CA_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. 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.

8.1 An Example

This example incorporates class models consisting of complex attributes and complex methods. The extended complex class object base is as given in Figure 15 [6]. The database schema information consists of the class lattice of the object base and is given in Figure 16, while the class composition hierarchy is as in Figure 17.

Suppose we want to vertically fragment the class *Student*, the network of method usage and application frequency matrices needed to compute the method affinity matrix of this class that incorporates *saff*, *ccaff* and *cmaff* is given in Figure 18. The method affinity value of a method pair in the matrix is computed as follows:

MA($M^{student.1}$,$M^{student.2}$) $= \sum_{k=1}^{1} \sum_{l=1}^{3} acc_l(q_1) + saff_{grad}(M^{student.1}, M^{student.2})$.

MA($M^{student.1}$,$M^{student.2}$)= (40+0+20) + (0+60+10) = 130 (initial MA).

ccaff($M^{student.1}$,$M^{student.2}$)= $ccaff_{dept}(M^{student.1}, M^{student.2}) + ccaff_{prof}(M^{student.1}, M^{student.2})$.

ccaff($M^{student.1}$,$M^{student.2}$)= [(50 + 15 + 0) + (25 + 40 + 5)] + 0 = 135

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

Algorithm 8.1 *(Vertical Fragments of Complex attributes and Complex Methods)*

Algorithm Vert_CA_CM

input: $\mathcal{Q}^{C_i^{des}}$: set of user queries accessing C_i and its descendant classes.
C_i : the database class to fragment
$\mathbf{L(C)}$: the class lattice
$\mathcal{C}_i^{des}$: set of descendant classes of C_i
$\mathbf{icnm}(C_i)$: set of intra class null methods of C_i.
$\mathcal{C}_i^{cont}$: set of containing classes of C_i
$\mathcal{C}_i^{cmeth}$: set of complex method classes of C_i
$(EM)^{C_i}$: extended method set of C_i.
MAR-set(C_i) : method attribute reference set of methods of C_i
AF-set(C_i^{des}): application frequency matrices of C_i and its descendants.
AF-set(C_i^{cont}): application frequency matrices of C_i and its containing classes.
AF-set(C_i^{cmeth}): 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 14: Vertical Fragmentation - Complex Attributes and Complex Methods

Person = {k.ssno,{a.name,a.age,a.address}, {m.ssno-of,m.whatname,m.age-in-year,m.newaddr},
{ I_1 {Person1,John James,30,Winnipeg}
I_2 {Person2,Ted Man,16,Winnipeg}
I_3 {Person3,Mary Ross,21,Vancouver}
I_4 {Person4,Peter Eye,23,Toronto}
I_5 {Person5,Mary Smith,40,Toronto}
I_6 {Person6,John West,32,Vancouver}
I_7 {Person7,Jacky Brown,35,Winnipeg}
I_8 {Person8,Sean Dam,27,Toronto} } }
I_9 {Person9,Bill Jeans,43,Vancouver}
I_{10} {Person10,Mandu Nom,30,Winnipeg}
Prof = Person pointer $\odot$ {k.empno,{a.status,a.dept,a.salary,a.student},
{m.status-of,m.students-of,m.whatsalary,m.dept-of},
{ I_1 (person pointer5) $\odot$ {Prof1,asst prof,Computer Sc.,45000,students pointers}
I_2 (person pointer6) $\odot$ {Prof2,assoc prof,Math,60000,students pointers}
I_3 (person pointer9) $\odot$ {Prof3,full prof,Math,80000,students pointers}
I_4 (person pointer10) $\odot$ {Prof4,full prof,Math,82000,students pointers} } }
Student = Person pointer $\odot$ {k.stuno,{a.dept,a.feespd,a.coursetaken}},
{m.stuno-of,m.dept-of,m.owing,m.course-taken}
{ I_1 (person pointer1) $\odot$ {Student1,Math,Y}
I_2 (person pointer4) $\odot$ {Student2,Computer Sc.,N}
I_3 (person pointer2) $\odot$ {Student3,Stats,Y}
I_4 (person pointer3) $\odot$ {Student4,Computer Sc.,N} }
I_5 (person pointer7) $\odot$ {Student5,Math,Y}
I_6 (person pointer8) $\odot$ {Student6,Stats,N} } } }
Grad = Student pointer $\odot$ {k.gradstuno,{a.supervisor}, {m.gradno-of,m.whatprog}
{ I_1 (Student pointer1) $\odot$ {Grad1,John West}
I_2 (Student Pointer2) $\odot$ {Grad2,Mary Smith} } }
I_3 (Student Pointer5) $\odot$ {Grad3,Mary Smith} } }
UnderG = Student
I_1 (Student pointer3)
I_2 (Student pointer4)
I_3 (Student pointer6)
Dept = {k.code,{a.name,a.profs,a.students}, {m.code-of,m.name-of,m.number-of-profs,m.students-of}},
{ I_1 {Dept1,{Computer science,prof pointers,student pointers}
I_2 {Dept2,{Math,prof pointers,student pointers}
I_3 {Dept3,{Actuary science,prof pointers,student}
I_4 {Dept4,{Stats,prof pointers,student pointers} } }

Figure 15: The Sample Object Database Schema

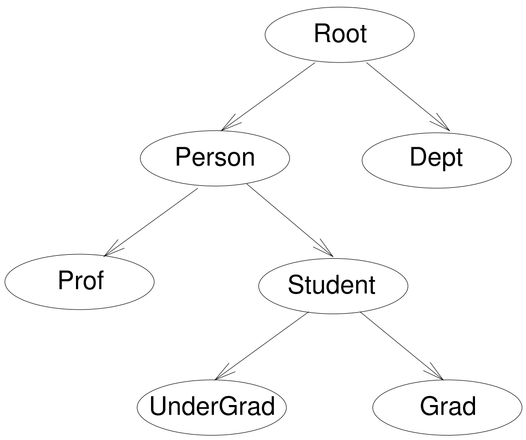

Figure 16: Complex Class Lattice

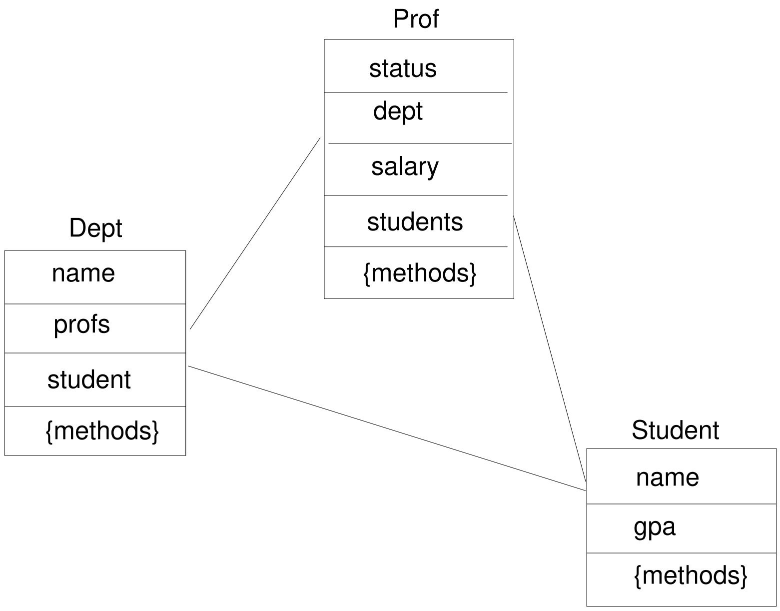

Figure 17: Class Composition Hierarchy

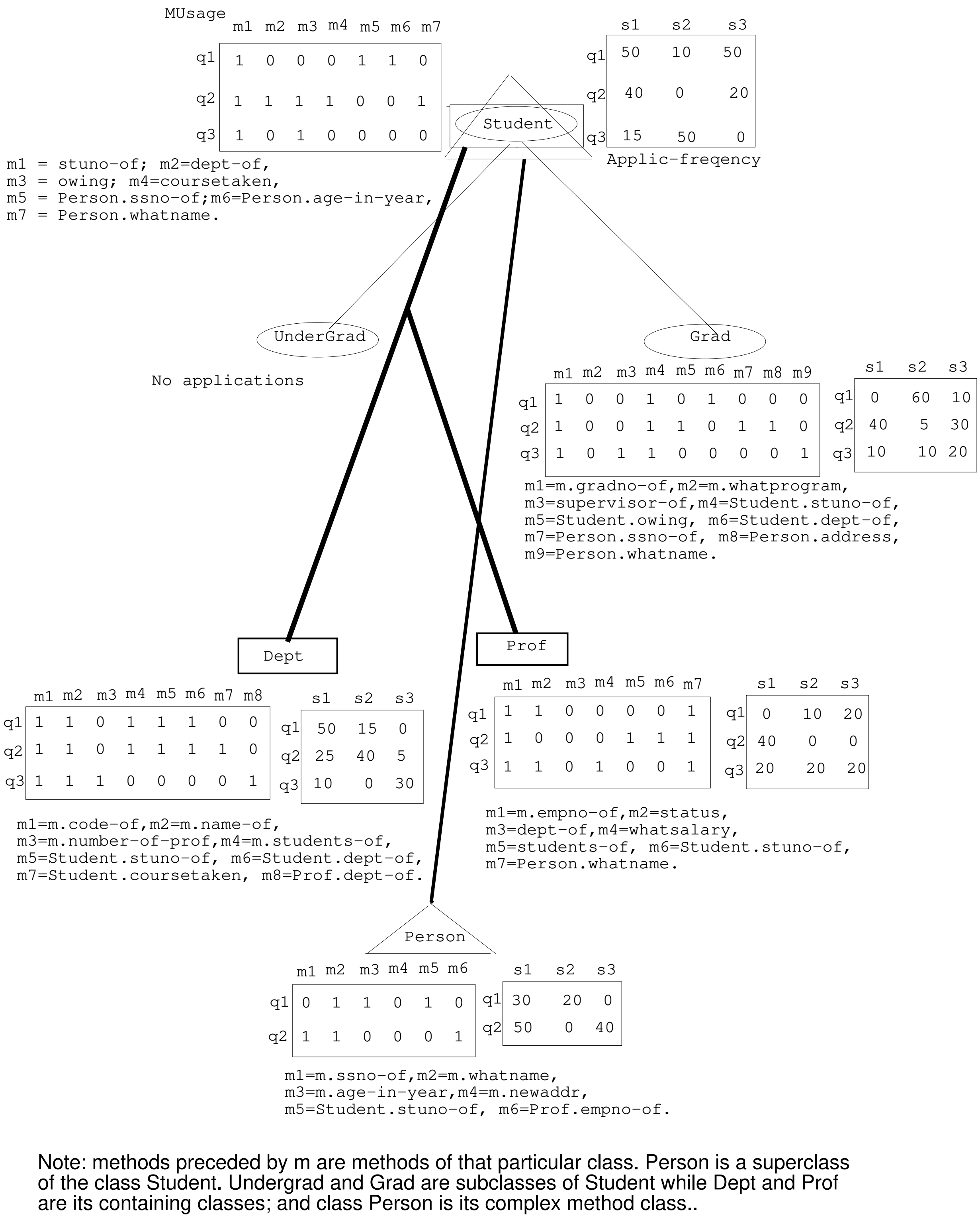

Figure 18: Method Usage/Application Frequencies for Student's Subclasses, Containing and Complexmethod Classes

MA($M^{student.1}$,$M^{student.2}$) = 130 + 135 = 265 (after first modification; line 5)
The method affinity, clustered affinity and the modified method usage matrices of the class after executing lines 2 through 12 of the algorithm 14 are given in Figure 19.

The vertical fragments from the execution of the partition algorithm on line 14 is given below. $F_1 = \{m_1, m_3, m_4\}$
$F_2 = \{m_1, m_2, m_5, m_6, m_7\}$

9 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 developing performance measurements to demonstrate the utility of our algorithms. Such performance analysis is difficult because very few of these systems (or more specifically, applications on these systems) have been developed. We are currently analyzing object-based applications to determine the way they are being used with the goal of determining metrices 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 acurate 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 vertical and hybrid fragmentation schemes for more complex class models.

References

[1] 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.

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

(a) Method Affinity matrix

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

(b) Clustered Method Affinity matrix

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

(c) Modified Method Usage matrix

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

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

Figure 19: Method/Clustered Affinity and Modified Method Usage Matrices

[2] C.I. Ezeife and Ken Barker. A distributed object based design technique. In *Proceedings of the First Annual Mid-Continent Information Systems Conference*, volume 1(1). Database Research Group, ND Fargo, 1993. Monterrey, Mexico, Vol.2, No.1, pp. 78-90.

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

[4] J.A. Hoffer and D.G. Severance. The use of cluster analysis in physical database design. In *Proceedings of the 1st International Conference on Very Large Databases*. Morgan Kaufmann Publishers, Inc, 1975. Vol 1, No.1.

[5] 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), Jan. 1987.

[6] Itasca Systems Inc. Itasca distributed object database management system. Technical Report Technical Summary Release 2.0, Itasca Systems Inc., 1991.

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

[8] 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.

[9] 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.

[10] W.T. McCormick, P.J. Schweitzer, and T.W. White. Problem decomposition and data reorganization by a clustering technique. *Operations Research*, 20(5), 1972.

[11] J. Muthuraj, S. Chakravarthy, R. Varadarjan, and S.B. Navathe. A formal approach to the vertical partitioning problem in distributed database design. In *Technical Report*. CIS Dept, Univ. of Florida, Gainesville, FL, 1992.

[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*. SIGPLAN Notices, 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. New York, 1982. Lecture Notes in Computer Science 132.