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

Tel.: (204) 474-8832

FAX: (204) 269-9178

{christie,barker}@cs.umanitoba.ca

Abstract

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

Similar to relations, objects can be fragmented both horizontally and vertically. This paper describes the formalisms necessary to reason about vertical fragmentation in a model consisting of simple attributes (no composition hierarchy is considered) and simple method invocations (nested transaction calls are not permitted). The paper then contributes by outlining an algorithm that fragments the classes into a set of "smaller" equivalent classes (actually fragments of the class' extent) that can be placed precisely where they are used. The algorithms will place data so network traffic is minimized thereby achieving optimal overall throughput while producing excellent response times for transactions executed locally on a distributed network.

1 Introduction

A distributed object based system (DOBS) is a collection of local object bases distributed among different local sites, interconnected by a communication network. A DOBS supports an object oriented data model including features of encapsulation and inheritance. The performance of a DOBS is greatly enhanced if data is stored at local sites in such a manner that many of the user applications running at each site get all needed data at that site without accessing irrelevant data. Further, the system must not incur unreasonable

*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.

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 data in a DOBS consists of a set of encapsulated objects. The data values (attribute values) are bundled together with the methods (procedures) for manipulating them to form an encapsulated object. Objects with common attributes and methods belong to the same class and every class has a unique identifier. Inheritance allows reuse and incremental redefinition of new classes in terms of existing ones. Parent classes are called *superclasses* while classes that inherit attributes and methods from them are called *subclasses*. The database contains a root class (*Root*), which is an ancestor of every other class. The overall inheritance hierarchy of the database is captured in a class lattice. The lattice presents the inheritance relationship between classes after the resolution of conflicts associated with multiple inheritance, and thus, a subclass in the lattice is linked to only one superclass. A class is an ordered relation $\mathcal{C} = (\mathbf{K},\mathcal{A},\mathcal{M},\mathcal{I})$ where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[1]. Other features of the model used to succinctly address the problem include the type of fragmentation that can be performed on a class, as well as the type of attributes and methods contained in the class. These features are discussed next. Each 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, $\mathcal{C}^v = (\mathrm{K},\mathcal{A}^{'},\mathcal{M}^{'},\mathcal{I})$. Two types of attributes in a class are possible (simple and complex). Simple attributes have only primitive attribute types that do not contain other classes as part of them. A complex attribute has a database class for its type. The composition relationship between a class (due to its complex attributes) and other classes in the database is captured using a "part–of" or class composition hierarchy. If class C_i has a complex attribute that uses class C_j for its type, then class C_j is a contained class of C_i while C_i is a containing class of C_j. Two possible method structures in a distributed object based system are simple and complex methods. Simple methods are those that do not invoke other methods of other classes. Complex methods are those that can invoke methods of other classes. The classes making up the DOBS are classified based on the nature of the attributes and methods they contain as discussed earlier in [2, 3]. 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.

The top-down design approach takes a global conceptual schema (GCS) describing the global database

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

entities and their relationships and combines it with the access pattern information to produce a set of local conceptual schemas (LCSs) describing database entities at each local site [9]. The input to the design process is obtained from an earlier static and system requirements analysis which define the environment of the system and collect an approximation of both the data and processing needs of all potential database users. 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 which will benefit from fragmentation [5]. A partial list of benefits include: Different applications access or update only portions of classes so fragmentation will reduce the amount of irrelevant data accessed by applications. Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base. Fragmentation allows parallel execution of a single query by dividing it into a set of subqueries that operate on fragments of a class. Fragmentation reduces the amount of data transferred when migration is required. Finally, fragment replication is more efficient than replicating the entire class because it reduces the update problem and saves storage.

The overhead and difficulty involved in implementing distributed design techniques include the generation of inputs from static analysis. Earlier work has argued that since 20% of user queries account for 80% of the total data accesses, this analysis is feasible [9]. However, major changes in a domain would entail a re-analysis of the system and re-running of the distributed design techniques. Future research will investigate how these techniques can be incorporated into a dynamic system.

This paper reviews possible DOBS models presented earlier [2, 3], and contributes by presenting algorithms for vertically fragmenting the first class model consisting of simple attributes and simple 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 fragmentation algorithms for an object model consisting of simple attributes and simple methods. Finally, Section 3 concludes and suggests future research directions.

1.1 Related Work

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

Vertical Fragmentation (relational): 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.* [6] is used to form these attribute clusters. Navathe *et al.* [8] extends Hoffer's work by defining algorithms for grouping attributes into overlapping and nonoverlapping fragments. Cornell and Yu [1] optimized this work by developing an algorithm that obtains an optimal binary partitioning for relational databases. Özsu and Valduriez [9] discuss this earlier work on vertical partitioning for distributed databases using the access frequency information and the Bond Energy Algorithm. Muthuraj *et al.* [7] argue that earlier algorithms for vertical partitioning are *ad hoc*, so they propose an objective function called the Partition Evaluator to determine the "goodness" of the partitions generated by various algorithms.

Vertical Fragmentation (objects): Karlapalem *et al.* [5] define issues involved in distribution design for an object oriented database system. They identify two types of methods – simple and complex methods. They argue that a model consisting of simple methods can be vertically partitioned using techniques described by Navathe *et al.* [8], while that of complex methods, requires a method-based view (MBV). The MBV identifies the set of objects accessed by a method and the set of attributes or instance variables accessed by the method. The sets are further grouped into sets of objects and instance variables based on the classes to which they belong. This generates the set pairs of objects and instance variables (O_i,I_i) accessed from a class C_i by a method. This is called method m_j's view of class C_i. They further suggest the use of concepts developed by Pernul *et al.* [10] to fragment classes based on views. Pernul, Karlapalem and Navathe [10] 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 [8].

2 Vertical Fragmentation of Class Model - Simple Attributes and Simple Methods

The objective of vertical fragmentation is to break a class model into a set of smaller classes (fragments) that permit user applications to execute using only one fragment. This means that optimal vertical fragments will minimize the execution time of user application [9]. Vertical fragmentation aims at splitting a class so all attributes and methods of the class most frequently accessed together by user queries 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, 8]. Secondly, we extend each method group (fragment) to incorporate all attributes accessed by methods in this group. However, some attributes may belong to the reference set of more than one method, and deciding which method group these attributes belong in so only non-overlapping vertical fragments are generated is required. Two alternative approaches for handling this conflict are: (1) use a set of affinity rules similar to those used in horizontal fragmentation schemes presented in [2, 3] to decide which fragment to place these overlapping attributes or (2) from the outset, group both attributes and methods using attribute/method affinity. Approach (1) is preferable for the following performance reasons:

It drastically reduces the size of the matrices that need to be input to the bond energy algorithm and partitioning algorithm [8]. Secondly, 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 related the methods are. We first present some definitions and assumptions before presenting the algorithm. The explicit assumptions are:

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

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

The major data requirements related to applications is their access frequencies. With Q = $\{Q_1, Q_2, \ldots, Q_q\}$, the set of user queries (applications) running on methods of objects of classes $\{M^{i1}_{j1}, M^{i2}_{j2}, \ldots, M^{in}_{jn}\}$. Then, for each query Q_i and each method M^{in}_{jn}, we associate a method usage value denoted as use(Q_i, M^{in}_{jn}) where use(Q_i, M^{in}_{jn}) = 1 if method M^{in}_{jn} is referenced by query Q_i, 0 otherwise. Thus, for each class, we define a method usage matrix.

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

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

Definition 2.3 *A null method of a class C_i, NM(C_i)* with respect to a superclass C_s is a representative place holder of a method of this superclass. A null method is named with a header that is the name of its original class followed by a period used to concatenate this class name to the name of the method. ■

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

In effect, the extended method set of a class is the union of its actual methods and its null methods.

Definition 2.5 *The cardinality of a class* is the number of instance objects in the class (denoted **card**(C_i)). ■

Definition 2.6 *Access frequency of a query* is the number of accesses a user application makes to data. If Q = $\{q_1, q_2, \ldots, q_q\}$ is a set of user queries, acc(q_i,d_j) indicates the access frequency of query q_i on data item

d_j where data item d_j can be a class, a fragment of a class, an instance object of a class, an attribute or method of a class. ■

Definition 2.7 *Instance Object join ($\odot$)* between a pointer to an instance object of a superclass and an instance object (I_j) of a class C_i returns the aggregate of instance objects of the two classes that represent the actual instance object of the class. ■

To illustrate the aggregate returned by the object join function, we consider the following example. In a database with *Student* a superclass of *Grad*, an instance object I_3 of *Grad* is represented as (Student pointer5) $\odot$ {537msc11,Mary Smith}. This means that the actual I_3 of *Grad* is the quantity representing the instance object I_5 of the superclass *Student* added to the quantity {537msc11,Mary Smith}.

Definition 2.8 *Attribute/Attribute Affinity AAA(A_i,A_j)between two attributes of the same class* is the sum of the access frequencies of all methods accessing these two attributes together at all sites.
AAA(A_i,A_j) = $\sum_{k=1}^{q} \sum_{l=1}^{m} \sum_{A_i \in MAR(M_k) \wedge A_j \in MAR((M_k)} acc_l(M_k, A_i) + acc_l(M_k, A_j)$. where $acc_l(M_k, A_i)$ is the number of accesses made to the attribute A_i by method M_k at site s_l. ■

Definition 2.9 *Attribute/Method Affinity AMA(A_i,M_j)* is the sum of the access frequencies of all methods accessing this attribute and this method together at all sites.
AMA(A_i,M_j) = $\sum_{k=1}^{q} \sum_{l=1}^{m} \sum_{A_i \in MAR(M_k) \wedge M_j \in MAR(M_k)} acc_l(M_k, A_i) + acc_l(M_k, M_j)$. ■

Definition 2.10 *Attribute Fragment Affinity AFA(A_i,F_j)* is a measure of the affinity between attribute A_i and vertical fragment F_j, and is the sum of all the attribute/attribute and attribute/method affinities between A_i and all attributes and methods of the class fragment F_j.
AFA(A_i,F_j) = $\sum_{k=1}^{q} \sum_{A_k, M_m \in F_j} AAA(A_i, A_k) + \sum_{m=1}^{n} AMA(A_i, M_m)$. ■

Definition 2.11 *Subclass affinity of two extended methods $(EM)_k^i$,$(EM)_j^i$ of a class C_i, saff($(EM)_k^i$,$(EM)_j^i$)* 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($(EM)_k^i$,$(EM)_j^i$) = $\sum_{o=1}^{w} \sum_{k \| use(q_k,(EM)_k^i)=1 \wedge use(q_k,(EM)_j^i)=1} \sum_{\forall S_l} ref_l(q_k) acc_l(q_k)$, where w is the number of subclasses, p is the class and S_l ranges over all sites. ■

Definition 2.12 *Method Affinity between two extended methods of a class C_i, MA($(EM)_j^i$,$(EM)_k^i$)* measures the bond between two extended methods of a class according to how they are accessed by applications.
MA($(EM)_j^i$,$(EM)_k^i$) = $(\sum_{p \| use(q_p,(EM)_j^i)=1 \wedge use(q_p,(EM)_k^i)=1} \sum_{\forall s_l} ref_l(q_p) acc_l(q_p)) + saff((EM)_j^i, (EM)_k^i)$ where $ref_l(q_p)$ is the number of accesses to methods ($(EM)_j^i$,$(EM)_k^i$) 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. ■

Affinity Rule 2.1 Place the overlapping attribute A_i in the fragment F_k^i with maximum AFA(A_i,F_k^i) and this is the vertical fragment with which attribute A_i 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.

2.1 The Algorithm

The inheritance relationship 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 all subclasses of the class at the same site. The vertical fragmentation process of a class requires three matrices [8]:

1. the method usage matrix indicates for each application Q_k in 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 that is 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 [8]. We use the same techniques for grouping attributes to group methods starting with the three matrices - method usage matrix, application frequencies matrix and method affinity matrix. Secondly, we cluster methods in much the same fashion as they cluster attributes, and we generate method affinity matrix. Finally, the same partition algorithm is used to generate partitioning of the class methods.

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 of the 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 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 Steps

The steps for vertically fragmenting a class consisting of simple attributes and simple methods are given below [2] while the formal algorithm *Verticalfrag* is presented in Figure 1:

1. Obtain the method usage and application frequency matrices of the class and its subclasses. The method usage matrices is generated by the Algorithm *UsageMtrx* [3]. 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 (subclasses of C_i in this case) on the tree from user applications. The application frequency matrices are part of input from pre-analysis of the system.

2. Define method affinity matrix of the class using the modified approach given in the algorithm *MAMtrx* [3]. 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 defined) between the two extended methods of the class in this (row,column) position. This includes all subclass affinities of the class.

3. Use the Bond Energy Algorithm (BEA) developed by [6] as presented in [8, 9] to generate clustered affinity matrix of the class. This algorithm accepts method affinity matrix as input and permutes its rows and columns to generate a clustered affinity matrix. The clusters are formed so that methods with larger affinity values are collected together.

4. Generate a modified method usage matrix of the class as described in the Algorithm *MUsageMtrx* [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. Next the *Partition* (PARTITION) algorithm of [8, 9] takes the clustered affinity matrix and the modified method usage matrix as its inputs and produces fragments of the methods. The *Partition* algorithm finds sets of methods that are mostly accessed solely by distinct sets of applications.

[2] Owing to paper length limitations, the formal algorithms representing these sub steps are not included in the body of the paper

Algorithm 2.1 *(VerticalFrag – Vertical Fragments Generator)*

Algorithm VerticalFrag
input: $\mathcal{Q}_i$: set of user queries; C_i : the database class to fragment
  L(C) : the class lattice; $\mathcal{C}_s$: set of subclasses of the class
  $\mathcal{M}_i$: set of methods of the class including null methods. C_l: set of leaf classes.
  $\mathbf{P}(\mathbf{MAR}(M_j^i))$: set of method attribute references of the set of all methods of the class C_i.
  $\mathbf{P}(AF^c)$: set of application frequency matrices of the class and its subclasses.
output: $\mathcal{F}_{c_i}$: set of vertical fragments of the database class C_i.
var
  $\mathbf{P}(MU^i)$: set of method usage matrices for class C_i and its subclasses.
  MA^i,CA^i : method affinity and clustered affinity matrices of the class C_i.
  MU^i : the modified method usage matrix of the class C_i.
begin
  $\mathbf{P}(MU^i) = \mathbf{UsageMtrx}(\mathbf{LT}(C_i))$ (1)
  $MA^i = \mathbf{MAMtrx}(\mathbf{P}(MU^i), \mathcal{Q}_{c_s}, C_s)$ (2)
  $CA^i = \mathbf{BEA}(MA^i)$ (3)
  $MU^i = \mathbf{MUsageMtrx}(C_s, \mathcal{M}_k^i, \mathbf{P}(MU^i))$ (4)
  $\mathcal{F}_{c_i} = \mathbf{PARTITION(CA)}$ (5)
  For each fragment $F_k \in \mathcal{F}_{c_i}$ do (6)
  begin
    For each method $M_k \in F_k$ do (7)
      For each attribute $A_k \in MAR(M_k)$ do (8)
        $F_k = F_k \cup A_k$ (9)
  end {of for F_k}
  Ensure disjointness
    for each overlapping $A_k \in \mathcal{F}_{c_i}$ **do** (10)
    select F_k according to Affinity Rule 2.1 (11)
    for each $F_m, m \neq k$ **do** (12)
      $F_m = F_m - A_k$; (13)
    end; {for F_m}
  end; {for A_k}
end {VerticalFrag}

Figure 1: Class Vertical Fragments Generator

5. Use method-attribute reference information of the class to elaborate each fragment to also include attributes of the class accessed by methods of the fragment.

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

2.2 An Example

We use the class lattice depicted in Figure 2 and the sample data [3] illustrated in Figure 3 for this example. Assume we want to obtain vertical fragments of the class *Person* based on the following application information.

[3] 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.

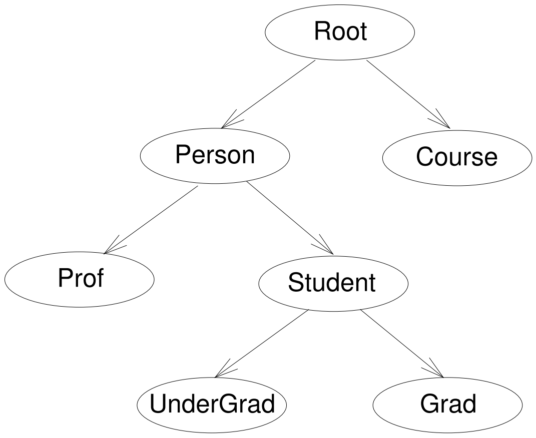

Figure 2: Class Lattice of Sample Object Base

Person = {k.ssno,{a.name,a.age,a.address},{m.ssno-of,m.whatlast,m.newage,m.n ewaddr}
{ I_1 {501-156-17,John James,30,Winnipeg}
I_2 {905-406-01,Ted Man,16,Winnipeg}
I_3 {658-157-09,Mary Ross,21,Vancouver}
I_4 {903-505-18,Peter Eye,23,Toronto}
I_5 {502-781-15,Bill Jeans,40,Toronto}
I_6 {669-882-08,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) $\odot$ {62-1736,asst prof,45000, {Ken,Peter,Maria},{111}}
I_2 (person pointer6) $\odot$ {60-1840,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) $\odot$ {5371019,Math,Y}
I_2 (person pointer4) $\odot$ {5370008,Computer Sc.,N}
I_3 (person pointer2) $\odot$ {4110000,Stats,Y}
I_4 (person pointer3) $\odot$ {4135709,Computer Sc.,N} } }
Grad = Student pointer $\odot$ {k.gradstuno,{a.supervisor},{m.whatprog}
{ I_1 (Student pointer1) $\odot$ {537phd1,John West}
I_2 (Student Pointer2) $\odot$ {537msc10,Mary Smith} } }
UnderG = Student
I_1 (Student pointer3)
I_2 (Student pointer4)

Figure 3: The Sample Object Database Schema

Q_1: Give the social security number of all persons with age greater 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.

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 subclasses. 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 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 as well as student numbers of all students who are less than 40 years old.

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.

Note that applications running on this subclass *Student* use three null methods with respect to superclass *Person*, which are *Person.ssno-of, Person.newage and Person.whatlast*. Since we want to obtain vertical fragments of the class *Person*, the method usage and application frequency matrices of this class and its two subclasses *Prof* and *Student* are as given in Figure 4. The complete picture of all the matrices needed to compute the method affinity matrix of the class *Person* is also given in Figure 4. Executing lines 2 - 13 of the algorithm VerticalFrag of Figure 1 yields the method affinity, clustered affinity and modified method usage matrices shown in Figure 5 for the class *Person*. The modified method usage matrix takes into account method usage of the methods of the class *Person* by applications running on its subclasses. Thus, Q_4 represents use of the method *newage* of class *Person* by the subclass *Prof*, while Q_5 represents use of the methods *ssno-of and newage* by the subclass *Student*. Finally, Q_6 represents the use of method *whatlast* by the subclass *Student*. The final method fragments as well as the elaborated fragments generated are also shown in this Figure [4]. Note that since m_1 is the class identifier method, it is contained in every fragment.

[4]There are no conflicts in the method/attribute references in this example.

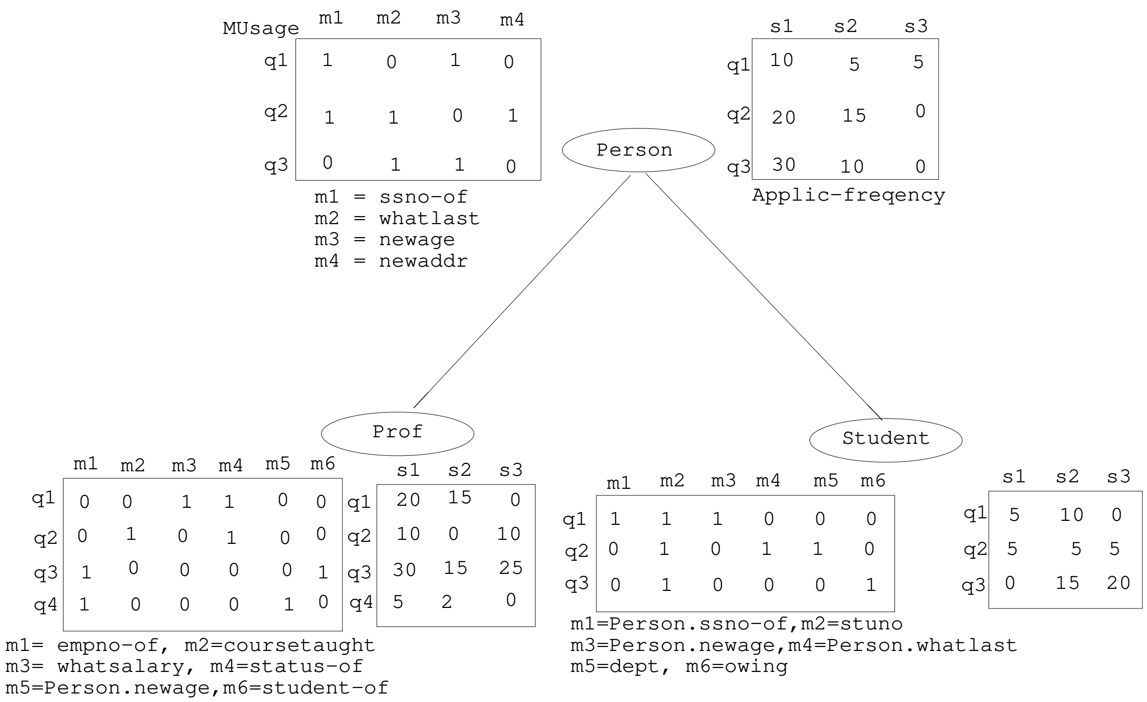

Figure 4: The Method Usage and Application Frequency matrices of the Classes

The elements of the method affinity matrix are computed as illustrated below:

MA(M_1,M_3) $= \sum_{k=1}^{1}\sum_{l=1}^{3} acc_l(q_1) + saff_{prof}(M_1^p, M_3^p) + saff_{student}(M_1^p, M_3^p)$.

MA(M_1,M_3)= (10+5+5) + (0) + (5+10+0) = 35

MA(M_2,M_2)= $\sum_{k=1}^{1}\sum_{l=1}^{3}(acc_l(q_2) + acc_l(q_3)) + saff_{prof}(M_2^p, M_2^p) + saff_{student}(M_2^p, M_2^p)$

MA(M_2,M_2)=((20+15+0)+(30+10+0)) + 0 + (5+5+5)

MA(M_2,M_2)=35 + 40 + 0 + 15 = 90

3 Conclusions

This paper describes class fragmentation in an object based system distributed on a network. 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 a description of the algorithms necessary to support the first simple model and an example of its application. Although results are required in all aspects of this problem before an optimal "fully functioning distributed object base" can be realized we have presented the necessary first steps that should

	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

Method Affinity matrix

	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

Clustered Affinity Matrix

m_1 = ssno–of, m_2 = whatlast
m_3 =newage, m_4 = newaddr

	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

Modified method usage matrix

Method Fragments are: $F_1 = \{ m_1 , m_4 \}$
$F_2 = \{ m_1, m_2, m_3 \}$

Elaborated Fragments including attributes are:
F_1 = {a.ssno,a.address,m.ssno–of,m.newaddr}
F_2 = {a.ssno,a.name,a.age,m.ssno–of,m.whatlast,m.newaddr}

Figure 5: The Method Affinity,Clustered Affinity and Modified Method Usage matrices of the Class Person

capture a large number of design application environment.

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 objectbased 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.

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

[3] C.I. Ezeife and K.E. Barker. Vertical class fragmentation in a distributed object based system. Technical Report TR 94-02, Univ. of Manitoba Dept of Computer Science, April 1994.

[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] 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.

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

[7] 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.

[8] S.B. Navathe, S. Ceri, G. Wiederhold, and J. Dou. Vertical partitioning algorithms for database design. *ACM Transactions on Database Systems*, 9(4), 1984.

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

[10] G. Pernul, K. Karalapalem, and S.B. Navathe. Relational database organization based on views and fragments. In *Proceedings of the Second Internationa Conference on Data and Expert System Applications*, Berlin, 1991.