Horizontal Class Fragmentation in Distributed Object Based Systems*

Christie I. Ezeife and Ken Barker

Department of Computer Science
University of Manitoba
Winnipeg, Manitoba, Canada R3T 2N2

{christie,barker}@cs.umanitoba.ca

Abstract

Many researchers have demonstrated the importance of entity fragmentation in distributed relational database design. *Database design* will be essential in the "next-generation" engineering design environment that exploits object-oriented technologies. Fragmentation enhances application performance by reducing the amount of irrelevant data accessed and the amount of data transferred unnecessarily between distributed sites. Algorithms for effecting horizontal and vertical fragmentation of relations exist, but fragmentation techniques for class objects in a distributed object based system have not appeared in the literature. This paper first presents a taxonomy of the fragmentation problem in a distributed object based system capable of supporting systems engineering applications. Detailed horizontal fragmentation algorithms are then presented for one of these class models using a top–down approach where the entity of fragmentation is the class object. The algorithms described incorporate the inherent features of the object model including both inheritance and class composition hierarchies.

1 Introduction

Optimal application performance on a Distributed Object Based System (DOBS) requires class fragmentation and the development of allocation schemes to place fragments at distributed sites so data transfer is minimized [8]. A DOBS supports an object oriented data model including features of *encapsulation* and *inheritance*.

The problem of distributed database design comprises first, the fragmentation of database entities and secondly, the allocation of these fragments to distributed sites. Two approaches are possible in a distributed database design – top-down and bottom-up. With the top-down approach, the input to the design process is the global conceptual schema (GCS) and the access pattern information, while the output from the design process is a set of local conceptual schemas [18]. The bottom-up approach constructs a GCS from pre-existing local schemas. In the relational database environment, the entity of

*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.

distribution is a relation while in a distributed object based system (DOBS), our entity of distribution is a class.

There are many existing object based systems that support some form of distribution and these include ITASCA [7], ENCORE [6], GOBLIN [9], THOR [11], and EOS [17]. Distributed relational databases [18] benefit greatly from fragmentation and these benefits should be realized in a distributed object environment [8]. A partial list of benefits include:

- Different applications access or update only portions of classes so fragmentation will reduce the amount of irrelevant data accessed by applications.
- Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base. Fragmentation allows parallel execution of a single query by dividing it into a set of subqueries that operate on fragments of a class.
- Fragmentation reduces the amount of data transferred when migration is required.
- Fragment replication is more efficient than replicating the entire class because it reduces the update problem and saves storage.

This paper defines possible DOBS models and contributes by presenting algorithms for horizontally fragmenting a realizable class model.

The balance of the paper is organized as follows. We complete this section by briefly reviewing previous work on distributed database design. In Section 2, we define the DOBS model used in this paper and a taxonomy for the various possible class models. Section 3 presents the fragmentation algorithm for an object model with simple attributes and simple methods. Although this is not a complete description of possible engineering application environments it captures a significant subset of possible scenarios. Further, understanding the "simple case", in the way described here, leads to a deeper understanding of the very complex design environments that could be supported by other points on the design taxonomy presented in Section 2. Finally, Section 4 makes some concluding remarks and suggests future research directions.

1.1 Related Work

Three fragment types are defined on a database entity:

- Horizontal Fragmentation: Breaking up of a relation/class into a set of horizontal fragments with only subsets of its tuples/instance objects.
- Vertical Fragmentation: Breaking up of a class into a set of vertical fragments with only subsets of its attributes and methods.

- Hybrid Fragmentation: Breaking up of a class into a set of hybrid fragments with both subsets of its tuples/instance objects as well as subsets of its attributes and methods.

This section reviews previous work on fragmentation in distributed database systems. Previous work on horizontal fragmentation of relational database entities is first reviewed, followed by a review of previous work on vertical fragmentation in the relational databases. Finally, previous work on fragmentation in DOBS is reviewed.

Horizontal Fragmentation (relational): Several researchers have worked on fragmentation and allocation in the relational data model including Ceri, Negri and Pelagatti [2], Özsu and Valduriez [18], Navathe *et al.* [14], Navathe and Ra [16], and Shin and Irani [3]. Ceri, Negri and Pelagatti [2] show that the main optimization parameter needed for horizontal fragmentation is the number of accesses performed by the application programs to different portions of data (file of records). Navathe, Karlapalem and Ra [15] define a scheme for simultaneously applying the horizontal and vertical fragmentation algorithms on a relation to produce a grid. A technique similar to the vertical fragmentation schemes discussed in Navathe *et al.* [14, 16] is used to produce horizontal fragments. Özsu and Valduriez [18] define the database information needed for horizontal fragmentation of the universal relation and show how the database relations are reconstructible using joins. Ceri *et al.* [4] model this relationship explicitly using directed links drawn between relations via equijoin operations. Shin and Irani [20] partition relations horizontally based on estimated user reference clusters (URCs). URCs are estimated from user queries [2, 18] but are refined using semantic knowledge of the relations.

Vertical Fragmentation (relational): Hoffer and Severance [5] define an algorithm that clusters attributes of a database entity based on their affinity. Attributes accessed together by applications have high affinity so the Bond Energy Algorithm developed by McCormick *et al.* [12] is used to form these attribute clusters. Navathe *et al.* [14] extends Hoffer's work by defining algorithms for grouping attributes into overlapping and nonoverlapping fragments. Muthuraj *et al.* [13] 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.

Horizontal Fragmentation (objects): Karlapalem, Navathe and Morsi [8] identify some of the fragmentation issues in object bases including: How are subclasses of a fragment of a class handled? Which objects and attributes of the objects are being accessed by the methods? What type of methods are considered: simple methods that access a set of attribute values of an object or complex methods that access a set of objects and instance variables?[1] Further, they argue that a precise definition of the

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

processing semantics of the applications is necessary. They do not present solutions for horizontally fragmenting class objects but argue that techniques used by Navathe *et al.* [15] for horizontal fragmentation could be applied.

Vertical Fragmentation (objects): Karlapalem *et al.* [8] define issues involved in distribution design for an object oriented database system. They identify two types of methods – simple and complex methods. 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.* [14], while that consisting 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.* [19] to fragment classes based on views.

Hybrid Fragmentation (objects): Karlapalem *et al.* [8] propose forming groups of objects $(\mathcal{O}_i)$ and their attributes $(\mathcal{A}_i)$ of class C accessed by each method $(\mathcal{M}_i)$ in the database to obtain pair sets $(\mathcal{O}_i,\mathcal{A}_i)$ for each $(\mathcal{M}_i)$. Each $(\mathcal{O}_i,\mathcal{A}_i)$ defines the mixed class-fragment of class C accessed by method $(\mathcal{M}_i)$ and is method $\mathcal{M}_i$'s view of class C.

2 The DOBS model

A distributed object based system is a collection of local object bases distributed among different local sites, interconnected by a communication network. We assume the database management system (DBMS) is distributed and the objects making up the object base are placed around the network. The general architecture of a distributed DBMS is summarized below [18].

- At each site, there is an individual internal schema called the local internal schema (LIS) used to describe the physical data organization on that machine.
- The global view of the enterprise data is described by the global conceptual schema (GCS) which describes the logical structure of the data at all sites.
- Since the enterprise data is fragmented and replicated at local sites, the logical organization of data at each site is described using the local conceptual schema (LCS).
- User applications and user access to the database is supported by external schemas (ESs), defined above the global conceptual schema [18].

The global directory/dictionary (GD/D) is used to provide required global mappings and provides the function of allowing user queries some location transparency.

2.1 The Data Model

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

The overall inheritance hierarchy of the database is captured in a class lattice. A class is an ordered relation $\mathcal{C} = (\mathbf{K},\mathcal{A},\mathcal{M},\mathcal{I})$ where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[2]. Three fragment types are defined on a class:

- Horizontal Fragmentation: 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,$\mathcal{A}$,$\mathcal{M}$,$\mathcal{I}^{'}$).

- Vertical Fragmentation: 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,$\mathcal{A}^{'}$,$\mathcal{M}^{'}$,$\mathcal{I}$).

- Hybrid Fragmentation: 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,$\mathcal{A}^{'}$,$\mathcal{M}^{'}$,$\mathcal{I}^{'}$).

2.2 A Taxonomy of Class Models

This section defines a class fragmentation taxonomy suitable for reasoning about object bases. Issues raised by Karlapalem *et al.* [8] can be extended and classified as:

- What constitutes a fragment of a class? – Are only attributes of a class fragmented, or is it possible or necessary to fragment methods too?

- How are object versions fragmented and allocated? – Are they placed in the same fragment as the original version or separate fragments? In the latter case, what criteria are used?

- What type of attribute need to be considered? – Simple attributes that have simple domains or complex attributes that have domains in another class?.

The taxonomy illustrated in Figure 1 is composed of three dimensions. The axes of the taxonomy are the type of fragmentation, the type of attributes, and the type of methods in the object model. We present the taxonomy succinctly and without justification due to space limitations.

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

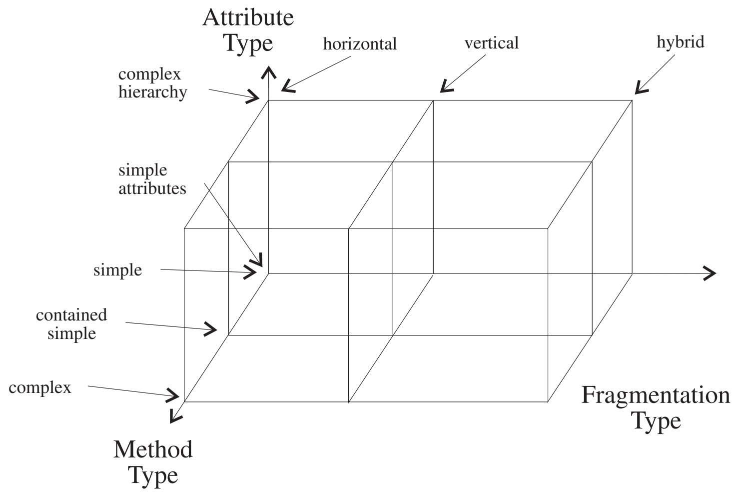

Figure 1: The Object Design Taxonomy

Fragmentation Type: The three types of class fragmentation possible are horizontal, vertical and hybrid. Since the case where no fragmentation occurs is irrelevant we omit it from the taxonomy.

Attribute Structures: Two types of attributes in a class are possible:

- **Simple Attributes:** only primitive attribute types that do not contain other classes as part of them. The precise composition of the set of primitive types is an important issue but beyond the scope of this paper [1].
- **Complex hierarchy:** The domain of an attribute may be another class. This is often referred to as a "part–of" or composition hierarchy.

Method Structures: Three possible method structures in a distributed object based system are:

- **Simple methods:** are those that do not invoke other methods of other classes. Further, simple methods only use local object data or data passed as parameters.
- **Contained Simple methods:** are simple methods of a class that are part–of the invoking class or a subclass of the invoking class. In other words, these are simple methods of a related class that can be invoked by a containing class.
- **Complex methods:** are those that can invoke methods of other classes. Further, it is possible for a complex method to return an object of a different type.

Complete solutions to the design problem in distributed object based systems requires a description of most points shown on this taxonomy. Note that some points on Figure 1 may not be realistic (for example, the model with a complex hierarchy and simple methods).

3 Horizontal Fragmentation – Simple Attributes and Methods

This section presents a horizontal fragmentation algorithm for classes consisting of objects that have simple attributes using simple methods. We proceed by explicitly stating the assumptions made, definitions required and then present 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. There are no cycles in the dependency graphs (discussed later). This means that two classes cannot both depend on each other at the same time so if class C_i is part-of class C_j, class C_j cannot be part–of class C_i.

A formal presentation of the algorithms requires a precise definition of several terms and definitions suitable for fragmentation in the first two models consisting of simple methods are presented here.

Definition 3.1 *Primary Horizontal Fragmentation* is the partioning of a class based on applications accessing the class. ■

Definition 3.2 *Derived horizontal fragmentation* occurs in two ways. (1) A class may be fragmented because of fragmentation in its subclasses, and (2) fragmentation may occur because some complex attribute is fragmented (part–of). ■

Definition 3.3 *A user query* accessing database objects is a sequence of method invocations on an object or set of objects of classes. The invocation of method j on class i is denoted by M_j^i and a user query Q_k is represented by $\{M_{j1}^{i1}, M_{j2}^{i2}, \ldots, M_{jn}^{in}\}$ where each M in a user query refers to an invocation of a method of a class object. ■

Definition 3.4 *A method invocation* M_j^i on the objects of a class C_i is viewed as a set of simple predicates that describe which objects to access. These simple predicates are represented as $\{Pr_{j1}^i, Pr_{j2}^i, \ldots, Pr_{jn}^i\}$ for some class. ■

Definition 3.5 *The cardinality of a class* is the number of instance objects in the class (denoted **card**(C_i)). ■

Definition 3.6 *Minterm selectivity* is the number of instance objects of the class accessed by a user query specified according to a given minterm predicate. ■

Definition 3.7 *Access frequency of a query* is the number of accesses a user application makes to data. Data in this context can be a class, a fragment of a class, an instance object of a class, an attribute or method of a class. If Q = $\{q_1, q_2, \ldots, q_q\}$ is a set of user queries, acc(q_i, d_j) indicates the access frequency of query q_i on data item j. We represent the access frequency of a minterm m_i as acc(m_i,d_j) since minterms formalize queries. ■

The proposed algorithm is guided by the intuition that an optimal fragmentation keeps those instance objects accessed frequently together while preserving and exploiting the inheritance hierarchy. Secondly, the fragments defined are guaranteed correct by ensuring they satisfy the following correctness rules :

- Completeness: every instance object belongs to a class fragment.
- Disjointness: every instance object belongs to only one class fragment.
- Reconstructibility: the union of all class fragments should reproduce the original class.

3.1 The Algorithm

This section presents the horizontal fragmentation algorithm for the class model consisting of simple attributes and simple methods. We assume that the database and application information required for distributed design are pre–defined and application information definitions are given as required in the following narrative. Input to the fragmentation process consists of the set of applications or user queries, the set of database classes and the inheritance hierarchy information. The output expected from this fragmentation process is a set of horizontal fragments for all classes in the database. Four steps are required.

3.2 Horizontal Fragmentation Algorithm

Step One – Define the Link Graph For all Classes in the Database
The object base information required is the global conceptual schema. In the relational model, this shows how database relations are implicitly connected to one another through joins. The object base information needed by the fragmentation procedure are of two types:

1. **The class lattice** showing the superclass–subclass relationship between all classes.
2. **The dependency graph**[3] captures the method link, attribute link or object link between any two classes in the database. This means that, if an attribute of a class C_i is another class C_j, then there is a an attribute link between classes C_i and C_j and is represented by an arc in the dependency graph ($C_i \rightarrow C_j$).

We capture both the class lattice information and dependence information of the database schema using a link graph. Since only simple attributes and simple methods are used the link graph is restricted to

[3] An example of a graph depicting dependency information is the class composition hierarchy shown in Kim [10].

Algorithm 3.1 *(Linkgraph - captures the inheritance hierarchy)*

Algorithm Linkgraph
input: $\mathcal{C}_d$**:** set of class in the database.
$\mathcal{C}_l$**:** set of leaf classes and $\mathcal{C}_l \subseteq \mathcal{C}_d$.
LT: Class lattice of the database showing subclass/superclass relationships
output: The Link graph (LG) for the database schema
LG = (Γ,λ) where Γ is a set of nodes
λ is a set of arcs connecting nodes in Γ.
begin
Starting from the leaf classes of the class lattice, for
every class, define a link from that class to its superclass.
Stop when the class is the *Root* class.
$LG \leftarrow$ initialized with a node $\forall C_i \in \mathcal{C}_d$; (1)
$\Gamma \leftarrow \{C_k | C_k \in \mathcal{C}_d\}$
$\lambda = \emptyset$;
$C_i \leftarrow$ leaf class in $\mathcal{C}_l$
$C_i = \{C_k | C_k \in \mathcal{C}_l\}$ (2)
while $C_i \neq$ Root **do** (3)
for each $C_i \in \mathcal{C}_d$ (4)
if $\exists (C_j | C_j =$ superclass $(C_i))$ (5)
then $\lambda = \lambda \cup (C_i \rightarrow C_j)$ (6)
end; {for C_i}
$C_d = C_d - C_i$; (7)
C_i = leaf node in $\mathcal{C}_d$ (8)
end; {while C_i}
return (LG); (9)
end;

Figure 2: The Linkgraph Generator

class lattice information and is generated as follows. Every subclass depends on its superclass in the link graph. This is a direct consequence of the inheritance hierarchy which implicitly partitions every superclass into subclasses. By starting fragmentation with the subclass and propagating intermediate results up the class hierarchy we preserve the implicit partitioning of the inheritance hierarchy and produce optimal results. Directed links among database classes are used to show these relationships and if class C_i depends on class C_j, an arc is inserted ($C_i \rightarrow C_j$). The link graph generation algorithm is given in Figure 2.

Step Two - Define Primary Horizontal Fragments of Classes

The quantitative database information required is the cardinality of each class. Both qualitative and quantitative application information is required. The fundamental qualitative information consists of the predicates used in external methods (messages or procedure calls) for user queries. A primary horizontal fragmentation is defined by the effects of user queries on objects of the owner classes. In the object oriented case, this constitutes all classes in the object base whose objects are accessed by user queries. Recall that each user query is a sequence of method invocations on objects of a class and each method of a class is represented as a set of predicates defined on values of attributes of that class. Therefore, to

Algorithm 3.2 *(SimplePredicates - Generate simple predicates from user queries)*

Algorithm SimplePredicates
input: $\mathcal{Q}$: set of all user queries (ie a set of sequences of methods)
$\mathcal{C}_d$: set of all database classes
output: $\mathcal{P}$**: set of set of simple predicates for all classes in the database.**
where the set of simple predicates for class C_i is $\mathcal{P}_i$.
begin
Initialize the set of simple predicates P_i for each class.
for each class $C_i \in C_d$ **do** (1)
$\mathcal{P}_i = \emptyset$ (2)
for each $q_k \in \mathcal{Q}$ **do** (3)
For every method, M_i^j in the user query, q_k,
place the predicates in the appropriate class's predicate set
for each $M_i^j \in q_k$ **do** (4)
if $\exists(P_{i1}^{j1} \in M_i^j)$ **then** (5)
$\mathcal{P}_i = \mathcal{P}_i \cup P_{i1}^{j1}$ (6)
end; {for M_i^j}
end; {for q_k}
$\mathcal{P} = \bigcup \mathcal{P}_i$ (7)
end;

Figure 3: Simple Predicate Generator

get the set of simple predicates on any class C_i, we form the union of all predicates of all methods in all the user queries. We summarize the algorithm for generating the set of simple predicates needed for primary partitioning of classes in Figure 3.

A simple predicate is defined as follows. Given a class C with the attributes $(A_1,A_2,\ldots,A_n)$ where A_i is an attribute defined over domain D_i, a simple predicate P_j defined on the class C has the form: P_j: A_i θ *value* where $\theta \in \{=,<,\neq,\leq,>,\geq\}$ and *value* is chosen from the domain of A_i (i.e; *value* $\in D_i$). We use P_i to denote the set of all simple predicates defined on a class C_i. The members of P_i are denoted by P_{ij} in the object base. Following Özsu and Valduriez [18], given a class with the set $P_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ of simple predicates we generated for the class from applications, we generate the set of minterm predicates $MM_i = \{m_{i1}, m_{i2}, \ldots, m_{iz}\}$. MM_i is formed using P_i and M_i by $MM_i = \{m_{ij} \mid m_{ij} = \bigwedge_{p_{ik} \in p_i} P_{ik}^*\}$, $1 \leq k \leq n, 1 \leq j \leq z$, where $P_{ik}^* = P_{ik}$ or $P_{ik}^* = \neg P_{ik}$.[4] We now use the semantics of the class to eliminate meaningless minterm predicates.

Step Three – Derived Horizontal Fragmentation on Member Classes

A derived horizontal fragmentation is defined on the member classes of links according to its owner. We partition a member class according to the fragmentation of its owner class and define the resulting fragment on the attributes of the member class only. Therefore, given a link L where owner(L) = S and member(L) = R, the derived horizontal fragments of R are defined as: $R_i = R \uparrow S_i$, where $1 \leq i \leq w$ and w is the number of fragments defined on R based on the owner. The link between owner and

[4] In other words, each simple predicate can occur in minterm predicates either in its natural form or negated form.

member classes is a pointer reference ($\uparrow$) which generates those instance objects of a member class that are pointed to by instance objects in an owner fragment. For example, an instance object of a subclass points to an instance object of its superclass which is logically part of this subclass instance object.

Step Four – Combining Primary and Derived Fragments

Since derived fragments of a member class has objects that overlap with one or more of its primary fragments, it is necessary to determine the most appropriate primary fragment to merge with each derived fragment of the member class. Several simple heuristics could be used such as selecting the smallest or largest primary fragment or the primary fragments that overlaps the most with the derived fragment. Although these heuristics are simple and intuitive they do not capture any quantitative information about the distributed object base. Therefore, a more precise approach is described that captures the environment and attempts to use it in merging derived fragments with primary fragments.

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

Definition 3.8 *Access frequency acco(I)* of an object (I) is the sum of the access frequencies of all the applications $\mathcal{Q}_i$ accessing the object. ■

This definition can be used to define the number of relevant and irrelevant accesses to an object. Relevant accesses are all of the access frequencies that are in the intersection of both the derived and primary fragments.

Definition 3.9 *Relevant accesses(F^d,F^p)* are those made to local objects of both fragments F^d and F^p as follows: $\sum_{I_i}$ acco(I_i), for all $I_i \in (F^d \cap F^p)$. ■

Irrelevant access of a derived fragment with respect to a prima ry fragment are accesses made to instance objects which are in the primary fragment but not in the derived fragment.

Definition 3.10 *Irrelevant access(F^d,F^p)* are those made to instance objects which are in the primary fragment, F^p, but not in the derived fragment, F^d. Irrelevant access(F^d,F^p) $= \sum_{I_j}$ acco(I_j), for all $I_j \in (F^p - F^d)$. ■

We now define the affinity between a derived fragment (F^d) and a candidate primary fragment (F^p).

Definition 3.11 *Affinity between Fragments* aff(F^d,F^p) = Relevant access(F^d,F^p) − Irrelevant access(F^d,F^p). ■

Two additional definitions are required. First, the affinity between two objects is the sum of each objects accesses (affinity), for all queries that access both objects.

Definition 3.12 *The Object Affinity* $affo(I_i, I_j)$ is the affinity between two objects I_i and I_j and

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

Finally, the affinity between an object and a fragment is the sum of all object affinities in the fragment.

Definition 3.13 *Affinity between an object I_i and a fragment F* is: faff(I_i,F) $= \sum_{I_j \in F}$ affo(I_i,I_j), $i \neq j$. ∎

We are now in a position to determine the primary horizontal fragment in the member that is most suitable for a particular derived fragment in the member. This is summarized in the following rule.

Affinity Rule 3.1 Select the primary fragment that maximizes the affinity measure aff(F^d, F_i^p) where F^d is the derived fragment and F_i^p is a primary fragment in the class ranging over all candidate fragments. This is the primary fragment that has the highest affinity with this derived fragment.

This rule selects the most suitable primary fragment to merge with the derived fragment. The merging process results in some overlap of objects in a set of primary fragments. Disjointness requires that instances appearing in two fragments are not permitted so a technique is required to determine the object's best location. One approach would be to eliminate the instance from all primary fragments other than the one selected using Affinity Rule 3.1. This is likely to be suboptimal. Our algorithm uses an object's affinity to its fragments to determine the best final placement for the object. This is accomplished with the following rule.

Affinity Rule 3.2 The primary fragment F_j^p that maximizes the function faff(I_i, F_j^p) where I_i is placed.

The final step of this process places fragments when there is no suitable primary fragments by making the derived fragments primary. This process is formally summarized by the *HorizontalMember* algorithm depicted in Figure 4.

Figure 5 precisely defines the four steps described above by forming the link graph (line 1)[5], producing the simple predicate (line 2), generating the minterms (lines 3–4), defining the primary and derived fragments (lines 5–7) and finally, integrating the derived fragments with the primary ones (lines 8–11).

3.3 An Example

This example uses the class lattice depicted in Figure 6 [6] and the sample data illustrated in Figure 7 [7]. The classes of the sample database are fragmented based on the following application requirements. Primary fragmentation is performed on the owner classes: **UnderG**, **Grad**, **Prof**, **Student** and **Person**.

[5] Line numbers refer to Figure 5.

[6] To keep the example simple we assume there is no application requirements for class *Course*.

[7] The ⊙ symbol denotes the integration of chunks or parts of a contained class within its superclass.

Algorithm 3.3 *(HorizontalMember - Primary and Derived Fragments Integrator)*

Algorithm HorizontalMember
input: $\mathcal{R}_i^p$: set of primary horizontal fragments of R
$\mathcal{R}_i^d$: set of derived horizontal fragments of R
output: $\mathcal{R}_i$: set of horizontal fragments of R
begin
For each derived horizontal fragment, find a primary fragment it will form
a union with by selecting the primary fragment with highest affinity.
Apply Affinity Rule 1.
for each $R_i^d \in \mathcal{R}_i^d$ (1)
select R_j^p according to Affinity Rule 3.1 (2)
$\mathcal{R}_j^p = \mathcal{R}_j^p \cup R_i^d$ (3)
Ensure disjointness
for each $I_i \in R^d$ **do** (4)
select R_m^p according to Affinity Rule 3.2 (5)
for each $R_n^p, n \neq m$ **do** (6)
$R_n^p = R_n^p - I_i;$ (7)
end; {for R_i^p}
end; {for I_i}
end; {for R_i^d}
If there are no primary fragments, the union operation leaves the
derived fragments as final fragments and other instance objects
not contained in a derived fragment are placed in a fragment.
if card$(\mathcal{R}_i^p) = 1$ **then** (8)
begin
for each $R_i^d \in \mathcal{R}_i^d$ **do** (9)
$R_j^p = R_i^d$ (10)
end; Ensure Completeness
$\mathcal{R}_k^p = \mathcal{R}_j^p - \bigcup_{k \neq j} \mathcal{R}_j^p$ (11)
end
The set of all primary fragments including those modified
is now the set of horizontal fragments.
$\mathcal{R}_i = \mathcal{R}_i^p$ (12)
end;

Figure 4: Horizontal and Derived Fragments Integrator

Algorithm 3.4 *(HorizontalFrag – Horizontal Fragments Generator)*

Algorithm HorizontalFrag
input: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$: set of database classes
$\mathbf{L(C)}$: the class lattice
output: $\mathbf{P}$ $(\mathcal{F}_{c_i})$: set of horizontal fragments of
the set of classes in the database.
var
Linkgraph : tree
$\mathcal{P}_i$: set of simple predicates for class C_i.
$\mathcal{F}_i^p$: set of primary horizontal fragments for class C_i
$\mathcal{F}_i^d$: set of derived horizontal fragments for class C_i
$\mathcal{M}_c^L$: set of member classes for the link graph L(C)
begin
Step One is to define the link graph using the inheritance hierarchy
linkgraph = LinkGraph($\mathcal{C}_i$,**L(C)**); (1)

Step Two is to define primary horizontal fragments on all classes
$\mathcal{P}_i$ **= SimplePredicates**($\mathcal{Q}_i$,$\mathcal{C}_i$) (2)
for each class $C_i \in$ **owner(L(C))** (3)
$\mathcal{F}_i^p$ **= minterms of [COM-MIN(**$\mathcal{P}_i$**)]** (4)

Step Three defines derived horizontal fragments on a member class
for each member class $C_j \in$ L(C) **do** (5)
for every primary fragment in class C_o (6)
(C_o is an owner class that shares a link with class C_j)
$F_j^{d_k} = C_j \uparrow F_o^{p_k}$ (7)
where
$F_j^{d_k}$ is the kth derived fragment of class j and
$F_o^{p_k}$ is the kth primary fragment of class o. the
symbol $\uparrow$ denotes the object pointer join and
$F_j^{d_k}$ is the set of objects in the member class
that are pointed to by objects in a fragment of the
owner class.)
end; {**for k**}
end; {**for** C_j}

Step Four is to integrate the sets of primary and derived
fragments for all member classes of a link
for each $C_j \in \mathcal{M}_c^L$ **do** (member class) (8)
$\mathcal{F}_{c_j}$ **= HorizontalMember**($\mathcal{F}_j^p$,$\mathcal{F}_j^d$) (9)
for each $C_i \in (\mathcal{C}_d - \mathcal{M}_c^L)$ **do** (nonmember class) (10)
$\mathcal{F}_{c_i} = \mathcal{F}_i^p$ (11)
end;

Figure 5: Class Horizontal Fragments Generator

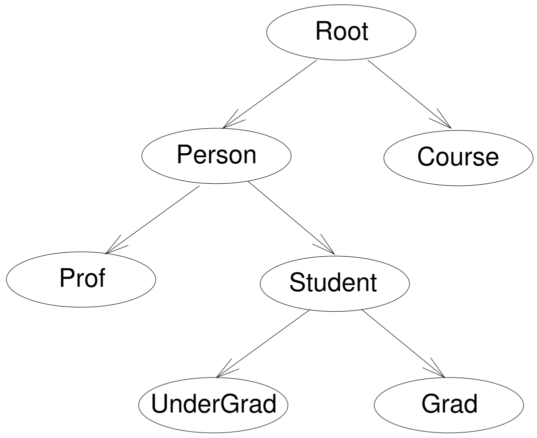

Figure 6: Class Lattice of Sample Object Base

Person = {ssno,{name,age,address},{whatlast,daysold,newaddr}
{ 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 $\odot$ {empno,{status,salary},{coursetaught,whatsalary}
{ I_1 (person pointer5) $\odot$ {62-1736,asst. prof,45000}
I_2 (person pointer6) $\odot$ {60-1840,assoc prof,60000} } }
Student = Person pointer $\odot$ {stuno,dept,feespd,coursetaken}
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 $\odot$ {gradstuno,{supervisor},{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 7: The Sample Object Database Schema

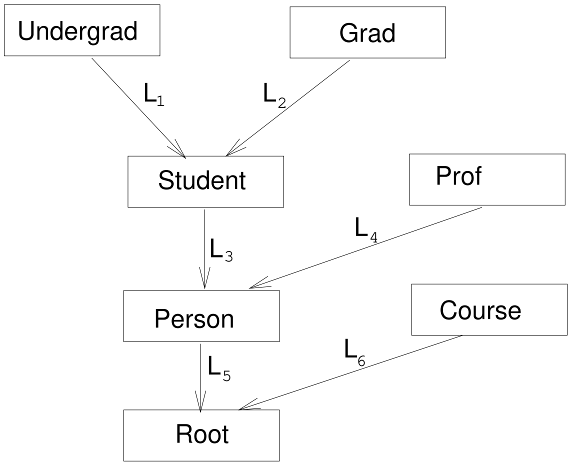

Figure 8: Class Lattice's Link Graph

Q_1: This application groups grads according to their area of specialization which is determined by the name of their supervisor. The methods used are defined on class **Grad** and the predicates are:

$$\{P_1\text{: supervisor} = \text{“Prof John West”}, P_2\text{: supervisor} = \text{“Prof Mary Smith”}\}$$

Q_2: This application groups profs by their addresses. The methods used are defined on class **Prof** with the following predicates:

$$\{P_1\text{: address} = \text{“Winnipeg”}, P_2\text{: address} = \text{“Vancouver”}, P_2\text{: address} = \text{“Toronto”}\}$$

Q_3: This application separates profs with salaries greater than or equal to \$60,000 from those with salaries less than \$60,000. The methods are in the class **Prof** with the following predicates:

$$\{P_4\text{:salary} \geq 60000, P_5\text{:salary} < 60000\}$$

Q_4: Groups students by their departments. The methods are from class **Student** and the predicates used are:

$$\{P_1\text{:dept=“Math”}, P_2\text{:dept=“Computer Sc”}, P_3\text{:dept=“Stats”}\}$$

Given this application information we form horizontal fragments of the class in our example database using the class lattice given in Figure 6. We apply the HorizontalFrag algorithm of Figure 5. Line 1 requires the construction of a linkgraph showing the dependence between classes due to inheritance (see Figure 8). Line 2 generates simple predicates for all classes from the user queries to give the following predicates:

P_{grad} : {supervisor = "Prof John West",supervisor="Prof Mary Smith"}

P_{prof} : {address = "Winnipeg",address="Vancouver",
address = "Toronto",salary $\geq$ 60000,salary $<$ 60000}

$P_{student}$: {dept="Math",dept="Computer Sc",dept="Stats"}

$P_{person} = \emptyset$

$P_{underg} = \emptyset$

Line 3 of Figure 5 generates the primary horizontal fragments of all classes by forming minterms of the complete and minimal set of predicates for each class. The minterms and primary fragments generated at this stage are:

Class Grad

Minterms MM_1 : supervisor = "Prof John West" $\wedge$ supervisor $\neq$ "Prof Mary Smith"

MM_2 : supervisor $\neq$ "Prof John West" $\wedge$ supervisor = "Prof Mary Smith"

Fragments F_1^p = instance object 1 (I_1)

F_2^p = instance object 2 (I_2)

Class Student

Minterms MM_1 : dept = "Math" $\wedge$ dept $\neq$ "Computer Sc" $\wedge$ dept $\neq$ "Stats"

MM_2 : dept $\neq$ "Math" $\wedge$ dept = "Computer Sc" $\wedge$ dept $\neq$ "Stats"

MM_3 : dept $\neq$ "Math" $\wedge$ dept $\neq$ "Computer Sc" $\wedge$ dept = "Stats"

Fragments F_1^p : $\{I_1\}$

F_2^p : $\{I_2,I_4\}$

F_3^p : $\{I_3\}$

Class Prof

Minterms MM_1 : address = "Winnipeg" $\wedge$ salary $\geq$ 60000

MM_2 : address = "Winnipeg" $\wedge$ salary $<$ 60000

MM_3 : address = "Vancouver" $\wedge$ salary $\geq$ 60000

MM_4 : address = "Vancouver" $\wedge$ salary $<$ 60000

MM_5 : address = "Toronto" $\wedge$ salary $\geq$ 60000

MM_6 : address = "Toronto" $\wedge$ salary $<$ 60000

Fragments F_1^p : $\emptyset$

F_2^p : $\emptyset$

F_3^p : $\{I_2\}$

F_4^p : $\emptyset$

F_5^p : $\emptyset$

F_6^p : $\{I_1\}$

The balance of the classes do not have predicates defined on them so no primary fragments are defined. Line 5 (of Figure 5) generates derived horizontal fragments on member classes of the linkgraph as shown below. Since **Grad** has two fragments, we have two derived fragments of the member class Student based on the two fragments of the owner class.

Class Student

$F_1^d = \{I_1\}$

$F_2^d = \{I_2\}$

Derived fragments of the Class Person

The derived horizontal fragments of Person based on owner class (Student), are:

$F_1^d = \{I_1\}$

$F_2^d = \{I_4, I_3\}$

$F_3^d = \{I_2\}$

Derived fragments of Person based on the owner class Prof are:

$F_4^d = \{I_5\}$

$F_5^d = \{I_6\}$

The next step is to apply the *HorizontalMember* algorithm (line 8–9 of Figure 5) to all member classes which integrates primary and derived fragments to generate the final horizontal fragments of these member classes. The member classes that need this algorithm applied to them are **Student** and **Person**. The quantitative application information needed concern the access frequencies of applications to different groups of data in each class and we assume the following access frequency information [8].

Class Grad : acc(Q_1, F_1) = 20, acc(Q_1, F_2) = 10.

Class Prof : acc(Q_2, F_1) = 5, acc(Q_2, F_2)=10, acc(Q_2, F_3)=10

acc(Q_3, F_1) = 20, acc(Q_3, F_2) = 5

Class Student : acc(Q_4, F_1) = 10, acc(Q_4, F_2) = 5, acc(Q_4, F_3) = 0

Further, apply the Affinity Rules to fragments of **Student** to produce the following:

Class Student

F_1^d has the maximum affinity with F_1^p of 10, and so we merge F_1^d and F_1^p to get :

$F_1^h = \{I_1\}$

F_2^d has the maximum affinity with F_2^p of 0, and so we merge F_2^d and F_2^p to get :

$F_2^h = \{I_2, I_4\}$

The third primary fragment remains the same as

[8] The fragments referred to in the access statistics are primary fragments generated from minterm predicates.

$F_3^h = \{I_3\}$

Class Person

Since the class **Person** has no primary fragments so the derived fragments become primary.

4 Conclusions

This paper defines 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 horizontal 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 detailed description of the algorithms necessary to support the the first simple model and an extended example of its application. Although results are required in all aspects of this problem before a optimal "fully functioning distributed object base" can be realized we have presented the necessary first steps that should capture a large number of design application environment. Ongoing research includes defining vertical and hybrid fragmentation schemes for these class models.

References

[1] K. Barker, M. Evans, R. McFadyen, and K. Periyasamy. A formal ontological object-oriented model. In *Technical Report*. Computer Sc. Dept, Univ. of Manitoba, Canada, 1992. Tr 92-02.

[2] S. Ceri, M. Negri, and G. Pelagatti. Horizontal data partitioning in database design. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. SIGPLAN Notices, 1982.

[3] S. Ceri and S.B.Navathe. A comprehensive approach to fragmentation and allocation of data in distributed databases. In *Proceedings of the IEEE COMPCON Conference*, 1983.

[4] S. Ceri, S.Navathe, and G. Wiederhold. Distributed design of logical database schemas. *IEEE Transactions on Software Engeneering*, 9(4), 1983.

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

[6] M.F. Hornick and S.B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Office Information Systems*, 5(1), Jan. 1987.

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

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

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

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

[11] 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.

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

[13] 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.

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

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

[16] S.B. Navathe and M. Ra. Vertical partitioning for database design: A graphical algorithm. In *Proceedings of the ACM SIGMOD*. SIGPLAN Notices, 1989.

[17] 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.

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

[19] 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.

[20] D. Shin and K.B. Irani. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Transactions on Software Engineering*, 17(9), Sept. 1991.