Class Fragmentation in a Distributed Object Based System

by

Christiana I. Ezeife

A thesis
presented to the University of Manitoba
in partial fulfilment of the
requirements for the degree of
Doctor of Philosophy
in
Computer Science

Winnipeg, Manitoba, Canada, 1995

Abstract

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. Fragmentation enhances application performance by reducing the amount of irrelevant data accessed at a local site 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 are yet to appear in the literature. This thesis first presents a taxonomy of the fragmentation problem in a distributed object base. The thesis then contributes by presenting a comprehensive set of algorithms for (1) horizontally and (2) vertically, fragmenting the four realizable class models on the taxonomy. The fundamental approach is top–down, where the entity of fragmentation is the class object. With horizontal fragmentation, our approach consists of first generating primary horizontal fragments of a class based on only applications accessing this class, and secondly generating derived horizontal fragments of the class arising from primary fragments of its descendant classes, its complex attributes (contained classes), and/or its complex methods classes. Finally, we combine the sets of primary and derived fragments of each class to produce the best possible fragmentation scheme. With the vertical fragmentation, 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 descendants, its containing classes or its complex method classes. Thus, these algorithms account for inheritance and class composition hierarchies as well as method nesting among objects, and are shown to be polynomial time.

Acknowledgements

I am deeply grateful to my supervisor Dr. Ken Barker for his immense support and encouragement during this research. He was easy to work with because he gave me the freedom to explore my own ideas, reviewed my ideas thoroughly and critically, and recognized meaningful contributions with compliments. His encouraging style and directions have helped me develop my confidence and abilities in research and writing.

Many thanks also go to my committee members Dr. Sylvia Osborn, Dr. John Van Rees, Dr. Howard Card and Dr. Mark Evans for all their insights and comments, and for taking time to read this thesis.

I express my sincere gratitude to my husband Dr. Anthony Ezeife, my three lovely daughters Emilia, Doreen and Laura Ezeife for their love, understanding, patience and encouragement without which this research would not have been completed. Special thanks go to my husband for the great influence he has had on my entire academic career.

Thanks to the research group at the Advanced Database Systems Laboratory of the University of Manitoba for all the hot and and interesting discussions that contributed to the quality of this thesis.

I would like to further express my sincere gratitude and appreciation to Dr. Ken Barker, the Manitoba Hydro, Dr. Hugh Williams, the University of Manitoba and the Department of Computer Science for the funding they provided for this research project. Manitoba Hydro deserves a special mention for providing the major funding that made this project possible, and I thank Ivy Kwok, Jim Schellenberg and the entire Information Systems crew of Manitoba Hydro for being interested enough in the project to encourage this funding.

Finally, I thank my parents for their love, concern and prayers.

Contents

List of Figures

Chapter 1

Introduction

Many researchers have demonstrated the importance of entity fragmentation in distributed relational database design. Fragmentation enhances application performance by reducing the amount of irrelevant data accessed and the amount of data transferred unnecessarily between distributed sites. Algorithms that fragment relations horizontally and vertically exist but fragmentation techniques for class objects in a distributed object based system (DOBS) have not appeared in the literature.

A DOBS supports an object oriented data model including features of *encapsulation* and *inheritance*. Encapsulation requires methods (procedures) be bundled with the data values they manipulate. A class has a unique identifier and collects objects with common attributes and methods together. Inheritance allows reuse and incremental redefinition of new class structures in terms of existing ones. Parent classes are called *superclasses* while classes that inherit attributes and methods from them are called *subclasses*. The overall inheritance hierarchy of the database is captured in a *class inheritance hierarchy*. We look at a class as an ordered relation $\mathcal{C} = (\mathbf{K},\mathcal{A},\mathcal{M},\mathcal{I})$ where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[1]. There is an object identifying attribute *oid* which is a member of the set of attributes $\mathcal{A}$. The *oid* could be either a system defined object identifier or a user-defined key attribute.

Although many object oriented database systems exist today, only a few provide some form of support for distribution, including: ITASCA [23], ENCORE [22], GOBLIN [25], THOR [30], and EOS [37]. Some of these systems have efficient techniques for grouping and clustering objects of a class with objects of their most needed parent on the same disk unit (container) [37]. These techniques, however, do not obtain partitions of each database entity (like a class or its instances) which are distributed. Secondly, though they provide techniques for handling object migration, minimizing replication and migration of objects from the onset is not emphasized. Moreover, with these approaches, logical design and organization of data is not clearly independent of the physical organization of data. The work of Bertino and Kim [2] on index selection is also complementary to this work, but different in the sense they aim at efficient design at the physical level (local internal schema), while our design aims at efficient design at the logical level (local conceptual schema). To maximize the utility of these object-oriented systems, it is beneficial to understand the distributed design problems in this environment and provide solutions for them.

Distributed relational databases [38] benefit greatly from fragmentation and these benefits should be realized in a distributed object environment [24]. 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.

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

- Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base.
- Fragmentation reduces the amounts 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.

Two approaches are possible in distributed design – top-down and bottom-up. The top–down approach is useful when designing a system from scratch. The input to the design process is the global conceptual schema (GCS) and access pattern information while the output from the design process is a set of local conceptual schemas [38]. The fragments created are later *allocated* to distributed sites. The bottom–up approach is used when databases already exist at distributed sites so the process requires constructing a GCS from pre–existing local schemas. Since we are attempting to design an object base, the top-down approach is used and the entity of distribution is a class fragment.

Efficient storage and handling of large amount of data on secondary storage media is extremely important in a database management system [3]. Thus, considerable attention is usually given to such design issues as the management of metadata in the conventional system. These design issues also ought to be carefully addressed in the object based environment.

This thesis contributes to the broad objective of providing an efficient storage system for a DOBS in the following ways:

1. Provides a precise taxonomy of the possible DOBS models.

2. Provides algorithms for horizontally fragmenting the four realistic class models in a DOBS.

3. Provides algorithms for vertically fragmenting the four realistic class models in a DOBS.

4. Provides an intuitive argument to show that these algorithms generate best possible fragments that are also correct with the prime objectives of:

   (a) Minimizing the local retrieval costs of applications running at any local site by minimizing the amount of unneeded data stored at any local site, considering applications running at this site.

   (b) Minimizing the transmission costs incurred in the course of running applications at local sites by minimizing the amount of relevant data needed by local applications from remote sites.

Although algorithms currently exist for fragmenting relations, fragmentation and allocation of objects is still a relatively untouched field of study. This work contributes and is unique in the following ways: each object-oriented database entity (database class) is fragmented and can be distributed. Moreover, partitioning of each entity aims at minimizing the need for object replication, migration and redundancy.

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 [47]. 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 algorithms. Future research will investigate how these techniques can be incorporated into a dynamic system.

1.1 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 [38].

- 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. The architecture is summarized in Figure 1.1.

The global dictionary (GD) is used to provide required global mappings and provides the function of allowing user queries some location transparency. This means that local database management components are integrated by means of global DBMS functions as shown in Figure 1.2.

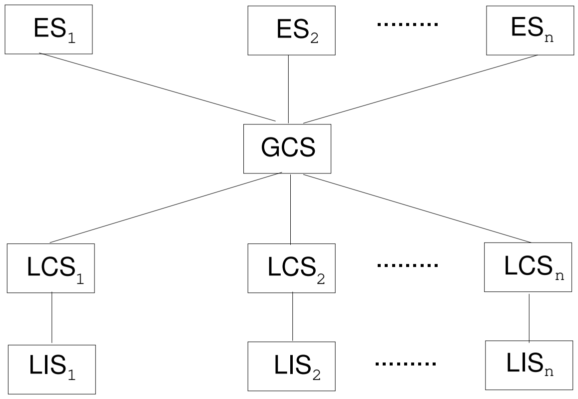


Figure 1.1: Distributed Database Reference Architecture

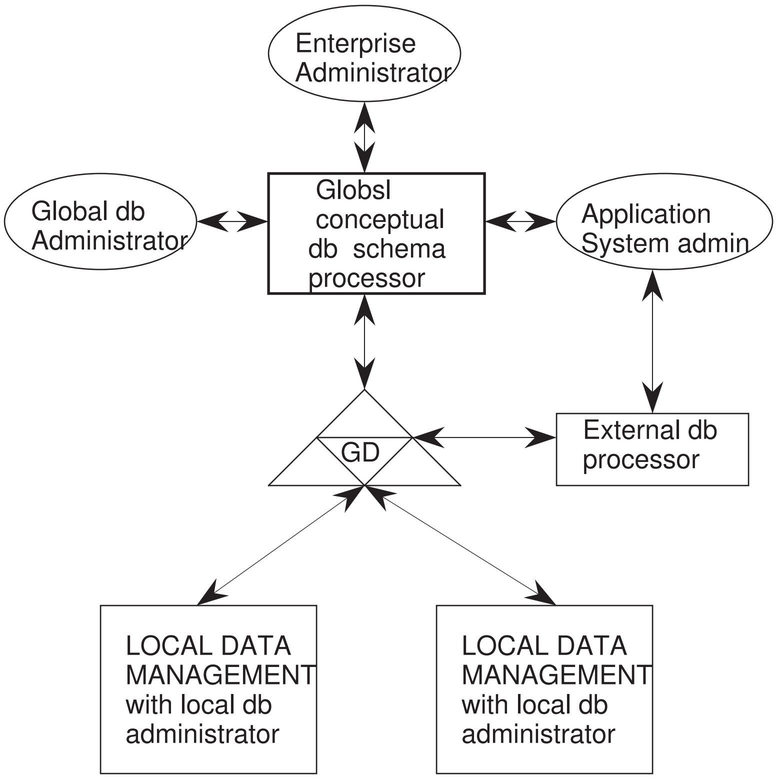


Figure 1.2: Functional Schematic of an Integrated Distributed DBMS

1.1.1 The Data Model

The data in a DOBS consist 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* which is an ancestor of every other class in the database. The overall inheritance hierarchy of the database is captured in a class inheritance hierarchy. A class is an ordered relation $\mathcal{C}$ = ($\mathbf{K}$,$\mathcal{A}$,$\mathcal{M}$,$\mathcal{I}$). 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 below.

Fragmentation Type: The three types of class fragmentation possible are horizontal, vertical and hybrid.

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,$\mathcal{A}'$,$\mathcal{M}'$,$\mathcal{I}'$).

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 thesis [1].

Complex hierarchy: The domain of an attribute may be 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.

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

Simple methods: 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: simple methods of a class that are part–of the invoking class.

Complex methods: those that can invoke methods of other classes. Further, it is possible for a complex method to return an object of a different type.

1.2 A Taxonomy

This section defines a class fragmentation taxonomy suitable for reasoning about object bases. The taxonomy illustrated in Figure 1.3 characterizes the object base model with respect to (1) the type of fragmentation (2) the type of attribute, and (3) the type of method in the object model. In defining the taxonomy, the fragmentation issues necessary in object bases, identified by Karlapalem, Navathe and Morsi [24] were considered and these are: How are subclasses of a fragment of a class handled? Which objects and attributes of objects are being accessed by methods? What type of methods are considered: simple methods or complex methods? We extend these issues to include:

- 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 axes of the taxonomy are the type of fragmentation, the type of attributes, and the type of methods in the object model.

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

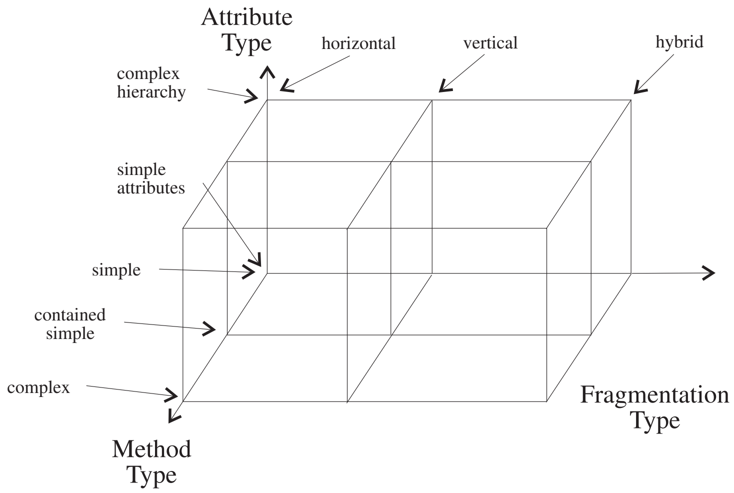


Figure 1.3: The Object Design Taxonomy

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 (Figure 1.3) enables us to accommodate all the necessary features of object orientation and provide solutions for object bases that are structured in various ways. Complete solutions to the design problem in distributed object based systems require a description of most points shown on this taxonomy. Note that some points on Figure 1.3 may not be realistic (for example, the model with a complex hierarchy and simple methods as well as the model with simple attribute and contained simple methods).

The thesis problem in summary is: Given an object based system consisting of a number of classes and a set of applications accessing instance objects of these classes at various sites, define schemes to fragment different class models both horizontally and vertically.

1.3 Outline of Thesis

The organization of the balance of the thesis is as follows. Previous work on distributed database design as well as some object-oriented systems are discussed in Chapter 2. Chapter 3 presents algorithms for horizontally fragmenting the four realistic class models in a DOBS, while Chapter 4 presents algorithms for vertically fragmenting the four realistic class models in a DOBS. Chapter 5 demonstrates that our algorithms generate the best possible fragments within the constraints of conflicting application requirements, minimized data replication and migration as well as redundancy. This chapter also demonstrates the practicality and correctness of the fragmentation schemes. Finally, Chapter 6 makes some concluding comments and suggests directions for future research.

Chapter 2

Related Work

This chapter reviews other work on distributed database design, object-oriented database systems as well as work on distributed object systems and/or Client/Server object-oriented databases. Section 2.1 presents a review of the previous work on fragmentation in relational database systems, Section 2.2 reviews some existing object-oriented systems. Section 2.3 reviews previous work on distributed object systems as well as Client/Server object-oriented databases. Section 2.4 discusses previous work on fragmentation in object-oriented systems leaving Section 2.5 to review some early work on clustering and indexing in object based systems.

2.1 Distributed Design (Relational)

In this section, previous work on horizontal fragmentation in relational databases is first reviewed, followed by a review of previous work on vertical fragmentation in relational databases.

2.1.1 Horizontal Fragmentation

Several researchers have worked on fragmentation in the relational data model including Ceri, Negri and Pelagatti [7], Özsu and Valduriez [38], Navathe *et al.* [34], Navathe and Ra [36], and Shin and Irani [8].

Ceri, Negri and Pelagatti [7] 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). They characterize the horizontal partitioning problem in distributed database design, demonstrate file partitioning on primary and secondary memory, and distribute data to different devices. They define applications in terms of boolean *predicates* and use access pattern information to achieve the design. Predicates are collected into sets. A set of simple predicates is *complete* if and only if the probability of accessing any two records belonging to a minterm fragment is the same. A complete set of predicates is *minimal* if and only if all its elements are relevant (needed). Further, implications exist between boolean expressions of simple predicates which eliminate meaningless minterms.

Navathe, Karlapalem and Ra [35] 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.* [34, 36] is used to produce horizontal fragments. They consider transactions with high access frequencies using simple or derived predicates to produce horizontal fragments. A predicate usage matrix is obtained from transaction information and leads to a predicate affinity matrix that clusters the predicate set into subsets. These are optimized using predicate inclusion and implications so the final horizontal fragments can be formed.

Özsu and Valduriez [38] 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* [9] model this relationship explicitly using directed links drawn between relations via equijoin operations. The relation at the tail of a link is called the *owner* of the link and the relation at the head is the *member* [9]. They use the same concepts and parameters as well as methodology identified in Ceri *et al.* [7, 9] to define horizontal fragmentation algorithms. Primary horizontal fragmentation is performed on all owner relations of the database schema and the steps for primary horizontal fragmentation are: (1) obtain the set of complete and minimal predicates as in Ceri *et al.* [7] (generated by the COM-MIN algorithm), (2) define the set of minterm predicates which are the conjuncts of the predicates, either in their natural or negated forms, and (3) eliminate meaningless minterm fragments using the implication set [7]. They define derived horizontal fragmentation on all member relations of a link according to a selection operation specified on its owner relation. The link between the owner and member relations is an equijoin implemented as a semijoin.

Shin and Irani [41] partition relations horizontally based on estimated user reference clusters (URCs). URCs are estimated from user queries [7, 38] but are refined using semantic knowledge of the relations. There is a knowledge–based system containing specific knowledge about the data and an inference mechanism applied to the knowledge–base generates new equivalent queries from which the URCs are derived.

2.1.2 Vertical Fragmentation

Hoffer and Severance [21] 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.* [33] is used to form these attribute clusters. Navathe *et al.* [34] 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 [11] optimized this work by developing an algorithm that obtains an optimal binary partitioning for relational databases. They use knowledge of physical factors[2] to decrease the number of disk accesses. Further refinement is accomplished by applying the binary vertical partition algorithm iteratively [34]. Navathe and Ra [36] 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 tree form fragments of the relation.

Özsu and Valduriez [38] 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 for splitting relations into fragments.

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

Chakravarthy *et al.* [10] 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 [33], binary vertical partition [34], Ra's graphical algorithm [36] and an exhaustive enumeration algorithm [10].

2.1.3 Hybrid

Özsu and Valduriez [38] argue that nesting of horizontal and vertical fragmentations in any order produces hybrid fragmentations of relations. However, Pernul, Karlapalem and Navathe [39] argue the user view is an important semantic not captured in distributed design of the relational data model. They argue that a view can encompass a set of transactions and different views may overlap to give rise to relationships between views. Then, fragments are derived from these view relationships by using the horizontal, vertical and derived fragmentation operators defined in [7, 34]. Their methodology involves decomposing relations into a set of disjoint fragments based on views where one or more fragments represents a view. To obtain mixed fragments, they find all overlapping views and for each pair of overlapping views perform a structured decomposition by applying vertical, primary and derived horizontal operators to acquire the smallest non-overlapping set.

2.2 Object-Oriented Database Systems

This section presents a review of features supported by some existing object-oriented systems. The systems reviewed are: GEMSTONE [4], EXODUS [5], ORION [29], ENCORE [42], POSTGRES [43, 44], STARBURST [31], IRIS [19] and O_2 [12].

GEMSTONE is an object-based system developed at Servio [4, 32] whose basic architecture distinguishes two main subsystems: the GEM Server process for query evaluation and the Stone Monitor for allocating object identifiers in blocks and coordinating commit activity. Its object-oriented database language (OPAL) is used for data definition, data manipulation and general computation. GEMSTONE provides support for encapsulation, both single and multiple inheritance as well as attribute-value inheritance. User access to data is permitted through both declarative and procedural language. An object is stored as a chunk of memory. It uses class defining objects for management of its metadata and objects communicate through message passing. GEMSTONE does not provide distributed services.

ORION [27, 28, 29] is built at Microelectronics and Computer Technology Corporation (MCC). ORION has three different systems ORION-1, ORION-1SX, and ORION-2. ORION-1 is a single user, multitask system, ORION-1SX is a client/server system on a local area network, while ORION-2 is a distributed object-management system in which all computers in the network participate in the management of the shared persistent database. The ORION-1 architecture consists of:

1. Object Subsystem: which provides high-level functions such as schema evolution, version control, query optimization, and multimedia information management.
2. Storage Subsystem: which provides access to objects on disk.
3. Transaction Subsystem: which handles transaction management.
4. Message Handler: which handles three types of messages - user-defined method, access message for accessing the value of an attribute of a class; and system-defined functions for schema definition, creation and deletion of instances; and transaction management.

ORION supports encapsulation, all forms of inheritance, and an access language that is both declarative and procedural. Objects storage requires full replication of superclasses' definitions in their subclasses, and objects are stored as recursive nested objects with pointers to the domain classes of the attributes. Metadata is managed by class objects at a higher level and objects communicate through message passing. ORION's client/server and distributed models are discussed in Section 2.3.

ENCORE [22, 42] is an OODBS from Brown University, Department of Computer Science. The database has two subsystems, a typeless backend ObServer that manages the persistent object store and a component that enforces the type system called ENCORE. ENCORE supports encapsulation, single and multiple inheritance. It stores an object as an aggregate of chunks of memory representing all superclasses up to the root. Metadata are managed by objects at a higher level and objects communicate through an exported interface. The Client/Server aspect of ENCORE is discussed in Section 2.3.

EXODUS [5, 6] is an extensible database management system developed at the University of Wisconsin with the goal of implementing a storage system to support the development of abstract data types, application-specific access methods, operations and version management, in addition to other features. Its objects are stored as uninterpreted byte sequences of unlimited size and a set of operations are defined for manipulating every large storage object and another set of operations is used for manipulating any file storage object. Thus, EXODUS provides weak support for encapsulation and there is no clear support for inheritance at this low level. EXODUS uses "hints" for the management of metadata. There is no direct communication between objects at this level.

POSTGRES [40, 43, 44, 45, 46] is a database system built at the University of California, Berkeley. It supports a data model which is relational but extended with abstract data types, attribute, and procedure inheritance. POSTGRES's novel 'no-overwrite' storage manager permits old records to remain in the database whenever an update occurs. There is a repository of procedures in the system which are bound to classes, attributes or tuples using the types of the arguments. POSTGRES supports both single and multiple inheritance. POSTGRES uses a system catalog for managing metadata.

STARBURST [20, 31] is an extensible system built at IBM's Almaden Research Centre and is aimed at extending the relational database system to capture object-oriented and knowledge-based system features. Access to objects and their components is only through operations defined on the table. It provides support for both single and multiple inheritance. User access to database objects is through a declarative language. Objects are stored as nested or ordinary relations and there is no direct communication between objects. STARBURST does not support distributed services and metadata is managed with tables.

IRIS [19, 48] DBMS is developed at Hewlett-Packard laboratories. The layered architecture of IRIS comprises the object manager on top of the storage manager. The IRIS storage manager is a conventional storage subsystem similar to system R's RSS. There is no explicit bundling of structure and behavior of data as an object. Objects are accessed, using functions. It supports both single and multiple inheritance. An object at a higher level is used for managing metadata and there is no direct communication between objects.

O_2 [12] is an OODBS with origin traced to ALTAIR, a project funded by 1N2 (a Siemens Subsidiary), INRIA (Institute National de Recherche en Informatique) and LRI (Laboratoire de Recherche en Informatique, University of Paris XI). The O_2 architecture consists of eight functional modules which include the object manager and the disk manager. O_2's disk manager is the Wisconsin Storage System (WiSS). It provides both declarative and procedural user-interface and objects are stored as WiSS records. Metadata is managed by objects at a higher level and objects communicate through message passing. O_2 provides strong support for encapsulation and supports both single and multiple inheritance but no support for attribute value inheritance.

2.3 Early Work on Distributed & Client/Server OODBs

This section presents a review of existing object based systems that are either distributed or support a client/server architecture. The OODB's reviewed are ITASCA [23], ENCORE [22, 42], GOBLIN [25], THOR [30], and EOS [37].

ITASCA [23] is a distributed active object database management system developed by Itasca Systems. Itasca emerged from the ORION prototypes. ORION2 has a distributed, object-oriented architecture for multi-client, multi-server architecture and ITASCA is an extension of ORION-2. The distributed environment of ITASCA allows shared and private databases and the shared database is distributed across workstations sites in the network. An ITASCA server controls the partitioning of the shared database at each site. Private databases allow private data not shared with other users of the database. The schema is stored redundantly at each site but each instance of data is stored at one site. There is no central data server nor central name server. The user or application does not need to know the location of a targeted object in the database. Location of data may be changed by the user or by the system moving data from one site to another. ITASCA uses one schema for the entire distributed database. Each site has a copy of the shared database schema, including code for methods. Therefore, only data needs to move among sites at execution time. There is an object directory at each site which maps unique identifications to physical locations on either the shared or private database partitions of the local site.

The ObServer component of ENCORE [22] reads and writes chunks of memory from secondary storage. The type level communicates with the Server through the UNIX remote procedure call (RPC) mechanism. The functions of ObServer include: (1) managing chunks of memory in secondary storage with a unique identifier (UID) attached to each chunk, and (2) maintaining correspondence between UIDs and chunks of memory. There is a network of workstations (nodes) running independent processes. A server and its data reside on a single node and processes on other nodes could access this server. To communicate with the server, the process binds a module called the client to its image so the client and the server can reside on different machines. The ENCORE module uses the object server as a backend and when two different processes on two different machines are using the ENCORE database, separate copies of ENCORE must reside on each machine. A binder process provides a client with a connection to the desired database. When a client wants to access a database, it issues a request to the binder which returns information identifying the server attached to this database, to get the client connected to the server. This enables access to multiple databases. Clustering of groups of related objects as a segment stored on disk is used to improve performance. A segment is of variable size and is the unit of transfer for objects between client and server and from secondary storage to main memory. Thus, the segments collectively provide a partition of the objects within a database and every database object is contained in at least one segment.

GOBLIN [25] is a distributed OODBS developed in Amsterdam. GOBLIN runs in an environment consisting of a network of workstations with memory-resident databases. The storage scheme used by GOBLIN is a three-level directory hierarchy for complex objects which uses attribute collections with multiple incarnations called Binary Association Tables (BAT). The Class Administration Table (CAT) describes the structural relationships between object components for a specific database schema. Finally, the mapping from schema to database instances is administered by a Redistribution Administration Table (RAT) including the partitioning and distribution information. The storage management functions are of three types – *schema functions*, *path functions* and *storage functions*. The schema functions describe the relationships to be maintained. For example, functions that map a class identifier (clid) to the set of object ids (oid) representing instance objects of this class. The counterpart of the oid or clid at the physical level is called the Physical Identifier (PID). Schema functions have corresponding path functions. For example, the schema function P_{person}: CLID $\rightarrow$ SET(OID) relates a class id to its set of instance objects and has the corresponding path function P_{person}: PID $\rightarrow$ SET(PID) which maps a class to its instance objects. The mapping between the extensional and intensional layers is captured by the storage functions, e.g, functions that relate OIDs and values at the schema level to PIDs and values at the implementation level. This mapping between OIDs and PIDs is achieved using an object table. The OIDs and PIDs can also be made identical. The path function definitions are administered in Binary Association Tables, the schema functions are captured in a Class Administration Table, and the storage functions are captured in a Redistribution Administration Table. Object persistence, stability and consistency are not addressed by the BAT, CAT or RAT but left for the distributed operating system or handled by the Class Manager which resides on each processor.

THOR [30] is a distributed system developed at Massachusetts Institute of Technology (MIT), Laboratory for Computer Science. THOR is a distributed system in which objects are stored at server nodes distinct from the machines where clients programs reside. Copies of THOR frontends run at client machines and perform the tasks of caching and prefetching objects and running operations on objects locally. Its implementation is fully distributed because clients are separated from servers which are themselves distributed. The system architecture comprises a set of computing nodes connected by a network. Some of these nodes are THOR servers that store objects in the THOR universe while others are client nodes, where users of THOR run their programs. It is possible to have a single node act as both a server and a client. The THOR system runs frontends (FEs) at client nodes, and backends (BEs) and object repositories (ORs) at the servers. While FEs and BEs handle type and operations on type, the ORs deal with managing the resilient storage for objects. Every resilient object resides at one of the ORs, usually the first OR selected when the object became permanent. However, objects can move from one OR to another under user control. An OR's persistent objects are stored on disk organized into variable-length segments. Each segment contains a group of related objects (a cluster). An object is identified by its *oref* which is local to the OR containing the object and is comprised of a segment id (sid) and an offset. An OR also keeps information about relationships between segments. Thus, an object identifier is made up of the pair (OR-id,oref). Each OR contains one of the second-level directories. There is also a top-level directory with entries as ORs. Each OR resides at a number of servers making multiple copies of their objects and for each object, one of the servers that has its copy acts as primary while others with copies act as backups. The primary handles all FE interactions for its objects.

EOS [37] is an object-oriented programming environment for distributed systems developed at INRIA, France. Leos, the language of EOS, is an object-oriented programming language that provides transparency for both distribution and persistence. EOS provides a flat, virtual and garbage collected object space which needs to be paginated and distributed. Object grouping is used to minimize cost of distribution. Object grouping includes both object declustering onto nodes and object clustering on disks. Object declustering is the partitioning of an object into subobjects and a mapping of these subobjects onto nodes. Object clustering is the process of grouping objects on disks. Objects accessed together are grouped closely on disks, usually determined by access patterns of computations which are in turn estimated by object aggregation. EOS's object model permits the definition of arbitrarily complex objects as aggregation of objects of arbitrary depth. An object is an instance of a class which holds a system wide identifier. Uniform persistence is supported so any object persists as long as it is reachable from a root of persistence. Objects are grouped in *harbor* and *pier* containers. The harbors are used to improve on node locality and is local to a single node. A harbor is divided into pier containers each of which is a set of disk tracks. Within the containers, objects are grouped with their most relevant parent.

2.4 Distributed Design (Object-Oriented)

This section first discusses previous work on horizontal fragmentation in the object-oriented database system, followed by a review of early work on vertical and hybrid fragmentations.

2.4.1 Horizontal Fragmentation

Karlapalem, Navathe and Morsi [24] identify some of the fragmentation issues in object bases They identify two types of methods - simple methods that access a set of attribute values of an object and complex methods that access a set of objects and instance variables?[3] Further, they argue that a precise definition of the 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.* [35] for horizontal fragmentation could be applied.

2.4.2 Vertical Fragmentation

Karlapalem *et al.* [24] argue that a model consisting of simple methods can be vertically partitioned using techniques described by Navathe *et al.* [34], 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

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

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 of 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.* [39] to fragment classes based on views.

2.4.3 Hybrid Fragmentation

Karlapalem *et al* [24] 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.

2.5 Early Work on Allocation of Fragments at Distributed Sites

Karlapalem *et al* [24] identify reduction of data transfer and communication between sites as the main objectives in allocation of fragments. They identify the need to consider the invocation sequence of complex methods as another issue in an object base. Other issues they identify include: references to other classes by attributes of a class, the inheritance hierarchy and references to different versions of an object. They note that the problem of allocation is not yet satisfactorily solved in the relational system and appears more difficult in an object base with these added issues to consider.

2.6 Physical Object Clustering and Indexing

Some systems like EOS [37], provide efficient techniques for grouping and clustering objects of a class with objects of their most needed parent on the same disk unit (container). This scheme is used to decide how best to group objects of related classes on secondary storage. For example, if an object of a class X is shared by objects of classes P and Q, the system uses relevance weighting to decide which object (that of P or Q) to group the shared object X with. Although the main objective of these systems is to give users distribution or clustering transparency and shield them from explicitly clustering their objects, their techniques do not obtain partitions of each database entity (like a class or its instances) which are later distributed. Secondly, though they provide techniques for handling object migration, minimizing replication and migration of objects from the onset is not emphasized. Moreover, with these techniques, logical design and organization of data is not clearly independent of the physical organization of data.

The work of Bertino and Kim [2] on index selection is also complementary to our work, but different in the sense that they aim at efficient design at the physical level (local internal schema), while our design aims at efficient design at the logical level (local conceptual schema). They define three index configurations that could be used to provide fast access to class instances using a variation of B-trees. Efficient designs at both physical (local internal schema) and logical (local conceptual schema) levels are important for an efficient distributed system. The design at one level could affect the performance of the design at the other level. For example, if the local conceptual schemas from our design leave fragments of classes not frequently needed at a local site, this may lead to storage of many indexes on objects that are not needed at this site but may be needed at remote sites, thus, retrieval costs of these objects will include transmission cost. Since there are many objects at this local site not frequently accessed, the retrieval cost (at the physical level) for any object, which depends on the size of the B-trees and the indexes, will be reduced by a good design at the logical level because the data search space is reduced. In effect, improper design at the logical level may cause a continual reorganization at the physical level. Thus, while the two designs are independent and on different levels, they are complementary.

Chapter 3

Horizontal Fragmentation

This chapter presents horizontal fragmentation algorithms for all four models defined in the taxonomy of Figure 1.3. This represents the first vertical layer on the figure. Section 3.1 presents a horizontal fragmentation algorithm for classes consisting of objects that have simple attributes using simple methods while Section 3.2 discusses the algorithm for fragmenting class objects composed of complex attributes and simple methods. Section 3.3 presents the algorithm for handling classes composed of simple attributes and complex methods while Section 3.4 discusses the algorithm for horizontally fragmenting classes composed of complex attributes and complex methods. Finally, Section 3.5 discusses other important characteristics of the algorithms including their time complexities.

We proceed by explicitly stating assumptions, definitions required, and then presenting the algorithms. We make the following assumptions:

1. Objects of a subclass physically contain only pointers to objects of 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. Secondly, there are no cycles in the dependency graphs discussed later.

This assumption makes for a more efficient storage structure because inherited attributes and methods are not replicated at subclasses. The structure is more realistic and reflects the actual object and specialization many object oriented database systems use, e.g., ENCORE [22]. The alternative storage structure also has its advantages and is fragmented as well by our approach without modification. The significance of this assumption is that providing a fragmentation scheme for the alternative storage structure (where inherited parts of superclasses are replicated at subclasses) would call for an extensive modification of the fragmentation algorithm to accommodate the storage structure in the first assumption. Thus, with the first assumption, we are in the position to define a more general fragmentation scheme.

2. There are no cycles in the dependency graphs discussed later.

   Some applications may require an object structure where provision for a cycle is necessary in the aggregation or class composition graph (e.g., when two classes are part of each other). Our approach is to define a uni-directional dependency graph from the aggregation hierarchy, and in this case, we select which of the two bi-directional links in the aggregation graph has higher application access frequency. The effect of the uni-directional dependency graph is provides a deterministic direction for propagation of primary fragments to derived fragments between two classes that depend on each other.

Primary Horizontal Fragmentation is the partitioning of a class based only on applications accessing the class directly. For example, an application running on a class **Student**, with an attribute **dept**, could have predicates $\{P_1$:dept="Math",

P_2:dept="Computer Sc",P_3:dept="Stats"}. *Derived horizontal fragmentation* occurs in three ways. (1) Partitioning of a class arising from the fragmentation of its subclasses, (2) fragmentation of a class arising from the fragmentation of some complex attribute (part–of) of the class, and (3) fragmentation of a class arising from the fragmentation of some complex method classes invoking methods of this class. *The cardinality of a class* is the number of instance objects in the class (denoted **card**(C_i)). Several more definitions are required.

Definition 3.1 *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 3.2 *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 3.3 *Object Pointer join ($\uparrow$)* between a fragment F_o^p of an owner class C_o and a member class C_m returns the set of objects in the member class which are pointed to by objects in the fragment of the owner class F_o^p. ■

Definition 3.4 *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) $\odot$ {Grad3,Mary Smith}. This means that the actual I_3 of *Grad* is the quantity representing the instance object I_5 of the superclass *Student* concatenated to the quantity {Grad3,Mary Smith}. *A method invocation* $M^{i.j}$ on the objects of a class C_i is represented as a set of simple predicates that describe which objects to access. These simple predicates are represented as $\{Pr_1^i, Pr_2^i, \ldots, Pr_n^i\}$ for class C_i. *Minterm selectivity* is the number of instance objects of the class accessed by a user query specified according to a given minterm predicate.

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

3.1 Simple Attributes and Methods

This section presents horizontal fragmentation algorithms for classes consisting of objects that have simple attributes using simple methods as discussed in [13, 15, 18]).

3.1.1 The Algorithm (HorizontalFrag)

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. The four-step algorithm required for horizontally fragmenting this first class model consisting of simple attributes and simple methods is called *HorizontalFrag*. Discussions of the four steps follow.

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. Generally, the object base information needed by the fragmentation procedure are of two types: *The class hierarchy* showing the superclass–subclass relationship between all classes, and *the dependency graph*[4] which captures the method link or attribute link between any two classes in the database. This means that, if the domain of an attribute of a class C_i is another class C_j, then there is an attribute link between classes C_i and C_j and is represented by an arc in the dependency graph ($C_j \rightarrow C_i$). We capture both the class hierarchy information and dependence information of the database schema using a link graph. Since only simple attributes and simple methods are used in this first model, the link graph is restricted to class hierarchy information and is generated as follows. Every subclass

[4] An example of a graph depicting dependency information is the class composition hierarchy shown in Kim [26].

C_j depends on its superclass C_i in the link graph and has an arc $(C_j \rightarrow C_i)$ inserted in the link graph. Starting from the leaf classes of the class hierarchy, we define a link from each class to its superclasses. 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 results that approach our intuitive notion of optimality. 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 $(C_j \rightarrow C_i)$. Algorithm *Linkgraph* of Figure 3.1 gives a formal presentation of this step. This algorithm starts from the leaf classes of the class hierarchy and defines a link from each class to its superclasses (lines 3-10).

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. A primary horizontal fragment of a class is a set of objects of this class accessed together by only applications running on this class. In the object oriented case, owner classes constitute 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 get the set of simple predicates on any class C_i, we form the union of all predicates of all methods from all the user queries operating on attributes of the class C_i that are defined over some domain D_i.

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

Algorithm Linkgraph

input: $\mathcal{C}_d$**:** set of classes in the database; $\mathcal{C}_l$**:** set of leaf classes and $\mathcal{C}_l \subseteq \mathcal{C}_d$.

LT: Class hierarchy of the database.

output: LG: The Link graph 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

$LG \leftarrow$ 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 $\lambda = \emptyset$;

$C_i \leftarrow$ 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)

for each $C_j \in superclass(C_i)$ (5)

$\lambda = \lambda \cup (C_i \rightarrow C_j)$ (6)

$\mathcal{C}_l = \mathcal{C}_l \cup \{C_j\}$ (7)

end; {for C_j}

end; {for C_i}

$\mathcal{C}_l = \mathcal{C}_l - \{C_i\}$; (8)

$C_i =$ a leaf node in $\mathcal{C}_l$ (9)

end; {while C_i}

return (LG); (10)

end;

Figure 3.1: The Linkgraph Generator

A simple predicate is defined as follows. Given a class C_i with the attributes (A_1,A_2,...,A_n) where A_i is an attribute defined over domain D_i, a simple predicate p_j defined on the class C_i has the form: p_j: A_i θ *value* where $\theta \in \{=, <, \neq, \leq, >, \geq\}$ and *value* is chosen from D_i. We use $\mathcal{P}_i$ to denote the set of all simple predicates defined on a class C_i. The members of $\mathcal{P}_i$ are denoted by P_{ij} in the object base. Following Özsu and Valduriez [38], given a class with the set $\mathcal{P}_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ 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 $\mathcal{P}_i$ and $MM_i = \{m_{ij} \mid m_{ij} = \bigwedge_{p_{ik} \in p_i} P^*_{ik}\}$, $1 \leq k \leq n, 1 \leq j \leq z$, where $P^*_{ik} = P_{ik}$ or $P^*_{ik} = \neg P_{ik}$.[5] We now use the semantics of the class to eliminate meaningless minterm predicates. Algorithms *SimplePredicates* of Figure 3.2 and *COM-MIN* of [38] are used in this step. COM-MIN is an algorithm that generates a complete and minimal set of predicates from a set of simple predicates. The objective of COM-MIN is to include into the set of predicates for a class C_i, every predicate that partitions the class into at least two parts accessed differently by at least one application. Any simple predicate that partitions C_i differently and which is relevant is included by COM-MIN. The Algorithm *SimplePredicates* of Figure 3.2 converts all user queries into a set of simple predicates for all classes in the database (lines 3-7).

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 and methods of the member class only. Therefore, given a link L where owner(L) = S and member(L)

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

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
  for each class $C_i \in \mathcal{C}_d$ **do** (1)
    $\mathcal{P}_i = \emptyset$ (2)
  for each $q_k \in \mathcal{Q}$ **do** (3)
    for each $M_i^j \in q_k$ **do** (4)
      for each $P_{i1}^{j1} \in M_i^j$ (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 3.2: Simple Predicate Generator

= 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 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. Derived fragmentation of member classes based on applications running on owner classes of links may follow the chain of member class links. A derived fragment of a member class based on a primary fragment of an owner class is a set of member class objects accessed together by applications running on the owner class. The derived horizontal fragments of member classes of links are generated by lines 5 through 7 of *HorizontalFrag* of Figure 3.4. On Line 7, $F_j^{d_k}$ is the kth derived fragment of class C_j and $F_o^{p_k}$ is the kth primary fragment of class C_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.

Step Four – Combining Primary and Derived Fragments

The final horizontal fragments of a class is composed of objects accessed together by both applications running only on this class and those running on its owner classes. Therefore, we must determine the most appropriate primary fragment to merge with each derived fragment of every member class. Several simple heuristics could be used such as selecting the smallest or largest primary fragment or the primary fragment 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. Algorithm *HorizontalMember* of Figure 3.3 gives a formal presentation of this step. *HorizontalMember* selects a primary fragment of a member class that is most appropriate to combine with a derived fragment of the same class. The primary fragment of choice is the one that has highest affinity with the current derived fragment by applying affinity Rule 3.1 (Lines 1-3). If the class has no primary fragments, the derived fragments are made the final horizontal fragments and instance objects not yet contained in a derived fragment are placed in one of the fragments (Lines 8-11).

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

Definition 3.5 *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. Thus, acco(I) = $\sum_i^q acc(q_i, I)$ for all q applications accessing the object. ■

This definition can be used to define the number of relevant and irrelevant accesses to an object with respect to another fragment.

Definition 3.6 *Relevant accesses(F_i^d,F_i^p)* are those made to local objects of both fragments F_i^d and F_i^p and are defined as all of the access frequencies of objects belonging to both the derived and primary fragments. Thus, Relevant accesses(F_i^d,F_i^p) = $\sum_{I_i}$ acco(I_i), for all $I_i \in (F_i^d \cap F_i^p)$. ■

Definition 3.7 *Irrelevant access(F_i^d,F_i^p)* are those made to instance objects which are in the primary fragment, F_i^p, or the derived fragment, F_i^d, but not in both. Irrelevant access(F_i^d,F_i^p) = $\sum_{I_j}$ acco(I_j), for all $I_j \in ((F_i^p \cup F_i^d)$ - $(F_i^p \cap F_i^d))$. ■

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

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

Algorithm HorizontalMember
input: $\mathcal{F}_i^p$: set of primary horizontal fragments of class C_i
$\mathcal{F}_i^d$: set of derived horizontal fragments of class C_i
output: $\mathcal{F}_i$: set of horizontal fragments of class C_i
begin
  for each $F_i^d \in \mathcal{F}_i^d$ (1)
    select F_i^p according to Affinity Rule 3.1 (2)
    $\mathcal{F}_i^p = \mathcal{F}_i^p \cup F_i^d$ (3)
        //Ensure disjointness //
    for each overlapping $I_i \in \mathcal{F}_i^p$ **do** (4)
      select F_{im}^p according to Affinity Rule 3.2 (5)
      for each $F_{in}^p, n \neq m$ **do** (6)
        $F_{in}^p = F_{in}^p - I_i$; (7)
      end; {for F_i^p}
    end; {for I_i}
  end; {for F_i^d}
  if card$(\mathcal{F}_i^p) = 1$ **then** (8)
  begin
    for each $F_i^d \in \mathcal{F}_i^d$ **do** (9)
      $F_i^p = F_i^d$ (10)
    end;   // Ensure Completeness //
    $\mathcal{F}_{ik}^p = \mathcal{F}_i^p - \bigcup_{k \neq j} \mathcal{F}_{ij}^p$ (11)
  end
  $F_i = \mathcal{F}_i^p$ (12)
end;

Figure 3.3: Horizontal and Derived Fragments Integrator

Algorithm 3.4 *(HorizontalFrag – Horizontal Fragments Generator)*

Algorithm HorizontalFrag
input: $\mathcal{Q}$: set of user queries; // where $\mathcal{Q}_i$ is the set accessing C_i; //
$\mathcal{C}_d$: set of database classes
L(C) : the class hierarchy
output: $\mathcal{F}_{c_i}$: set of horizontal fragments 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
linkgraph = LinkGraph($\mathcal{C}_i$,L(C)); (1)

$\mathcal{P}_i$ = **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)

for each member class $C_j \in$ L(C) **do** (5)
for every primary fragment in an owner class of C_j, C_o (6)
$F_j^{d_k} = C_j \uparrow F_o^{p_k}$ (7)
end; {for k}
end; {for C_j**}**

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 3.4: Class Horizontal Fragments Generator

Definition 3.8 *Affinity between a derived Fragments F_i^d and a primary fragment F_i^p of the same class, aff(F_i^d,F_i^p)* is a measure of how frequently objects of these two fragments are needed together by applications. Thus, aff(F_i^d,F_i^p) = Relevant access(F_i^d,F_i^p) − Irrelevant access(F_i^d,F_i^p). ■

Definition 3.9 *The Object Affinity $affo(I_i, I_j)$*, the affinity between two objects I_i and I_j is the sum of each object's accesses (affinity) for all queries that access both objects. Thus, $affo(I_i, I_j) = \sum_{\{q_k || q_k \in \mathcal{Q} \wedge q_k \text{access} I_i \wedge I_j\}}(acc(q_k, I_i) + acc(q_k, I_j))$ ■

Definition 3.10 *Affinity between an object I_i and a fragment F, faff(I_i,F)* is the sum of object affinities between this object I_i and all objects of the fragment F. Thus, 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 class that is most suitable to merge with 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_i^d, F_i^p) where F_i^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 an instance object appearing in more than one fragment is 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) is where I_i is placed. ■

The final step of this process converts derived fragments to primary for a class that has no primary fragments due to application access patterns.

3.1.2 An Example

This example uses the the sample data illustrated in Figure 3.6 [6] and the class hierarchy depicted in Figure 3.5 [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**.

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: supervisor = "Prof John West",P_2: supervisor = "Prof Mary Smith"}

[6]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, and this applies to all examples.

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

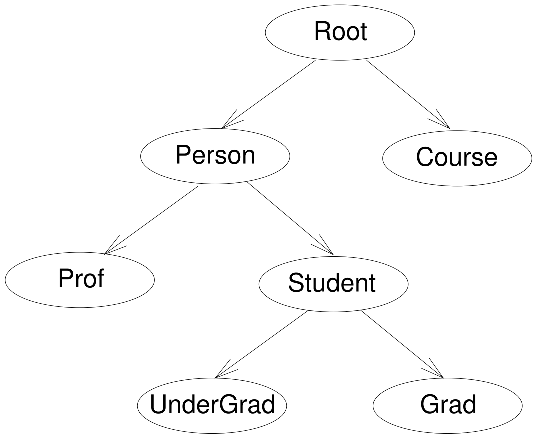


Figure 3.5: Class Hierarchy of Simple Sample Object Base

q_2: This application groups profs by their addresses. The methods used are defined on class **Prof** with the following predicates:

$\{P_1$: address = "Winnipeg",P_2: address = "Vancouver",P_2: address = "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:

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

q_4: Groups students by their departments. The methods are from class **Student** and the predicates used are:

$\{P_1$:dept="Math",P_2:dept="Computer Sc",P_3:dept="Stats"$\}$

Given this application information we form horizontal fragments of the class in our example database using the class inheritance hierarchy given in Figure 3.5. We

Person = {Person,{a.ssno,a.name,a.age,a.address},
{m.whatlast,m.daysold,m.newaddr},
{ I_1 {Person1,John James,30,Winnipeg}
I_2 {Person2,Ted Man,16,Winnipeg}
I_3 {Person3,Mary Ross,21,Vancouver}
I_4 {Person4,Peter Eye,23,Toronto}
I_5 {Person5,Bill Jeans,40,Toronto}
I_6 {Person6,Mandu Nom,32,Vancouver} } }
Prof = Person pointer $\odot$ {Prof,{a.empno,a.status,a.salary},
{m.coursetaught,m.whatsalary},
{ I_1 (person pointer5) $\odot$ {Prof1,full prof,45000}
I_2 (person pointer6) $\odot$ {Prof2,assoc prof,60000} } }
Student = Person pointer $\odot$ {Student,{a.stuno,a.dept,a.feespd},
{m.stuno-of,m.dept-of,m.owing},
I_1 (person pointer1) $\odot$ {Student,Math,Y}
I_2 (person pointer4) $\odot$ {Student,Computer Sc.,N}
I_3 (person pointer2) $\odot$ {Student,Stats,Y}
I_4 (person pointer3) $\odot$ {Student,Computer Sc.,N} } }
Grad = Student pointer $\odot$ {Grad,{a.gradstuno,a.supervisor},{m.whatprog}
{ I_1 (Student pointer1) $\odot$ {Grad1,John West}
I_2 (Student Pointer2) $\odot$ {Grad2,Mary Smith} } }
UnderG = Student
I_1 (Student pointer3)
I_2 (Student pointer4)

Figure 3.6: The Simple Sample Object Database Schema

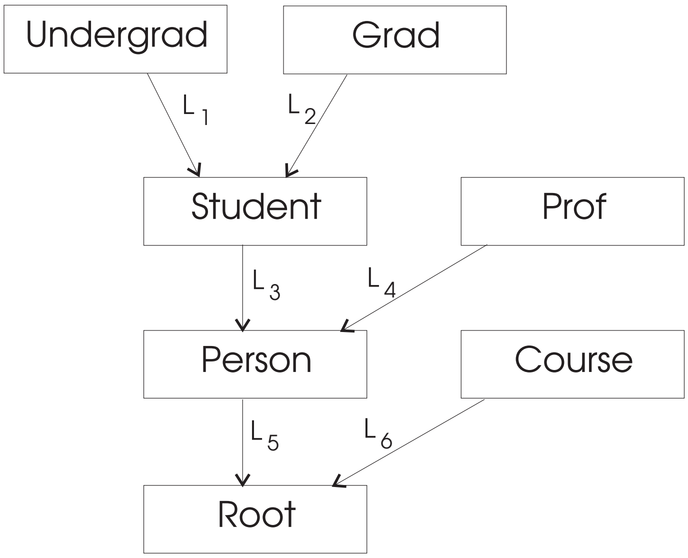


Figure 3.7: Class Hierarchy's Link Graph

apply the HorizontalFrag algorithm of Figure 3.4. Line 1 requires the construction of a linkgraph showing the dependence between classes due to inheritance (see Figure 3.7). 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 3.4 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 3.4) 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 3.4) 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].

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

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

F_3^h ={I_3}

Class Person

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

3.2 Complex Attributes and Simple Methods

This section presents an algorithm for horizontally fragmenting classes consisting of complex attributes that support a class composition hierarchy using simple method invocations. The algorithm first uses the attribute link information from the class composition hierarchy [26] to fragment classes according to the fragmentation of their contained classes. The class composition hierarchy is used to define the link graph showing the dependencies between any two classes. If class C_i has class C_j as part–of it there is a link from class C_j to class C_i in the link graph. The first iteration invokes *HorizontalFrag*, using the link graph from the class composition hierarchy, to produce an initial fragmentation. Generating derived fragments of a member class originating from class composition hierarchy entails finding the predicates of its owner class and defining derived fragments of the member class based on these predicates. The second iteration of the algorithm executes *HorizontalFrag* with the link graph defined using the inheritance hierarchy to produce fragments based on applications that preserve inheritance hierarchy too. The first iteration involves:

1. Form a class link graph from the class composition hierarchy. If class C_i is composed using class C_j (i.e. C_i contains C_j), form a link from class C_j to C_i.

2. Define *primary horizontal fragments* on owner classes of links which compose the contained classes.

3. Define *derived horizontal fragments* on member classes (containing classes). These are fragments of the containing class that point to only some instance objects of the contained classes.

4. Form the *union* of primary and derived fragments for member classes (containing classes) ensuring all fragments remain disjoint.

This iteration produces the primary horizontal fragments of the contained classes in the aggregation graph, which are propagated to the containing classes to produce their initial horizontal fragments (line (1), Figure 3.8). The final horizontal fragments of the containing classes are produced using the class inheritance hierarchy to obtain a "new" link graph. The *HorizontalFrag* algorithm uses this link graph and the fragments of containing class from the first iteration as its primary

Algorithm 3.5 *(Complex Attribute and Simple Method)*

Algorithm Hor_CA_SM
input: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$: set of database classes including classes with complex attribute and simple methods; $\mathbf{L(C)}$: the class hierarchy
$\mathbf{A(C)}$: class composition hierarchy showing attribute link
output: $\mathcal{F}_{c_i}$: set of horizontal fragments of the set of classes in the database
begin
$\mathcal{F}_{c_i}$ = **HorizontalFrag**($\mathcal{Q}_i$, $\mathcal{C}_d$, $\mathbf{A(C)}$) (1)
$\mathcal{F}_{c_i}$ = **HorizontalFrag**($\mathcal{Q}_i$, $\mathcal{F}_{c_i}$, $\mathbf{L(C)}$) (2)
end; {**Hor_CA_SM**}

Figure 3.8: Horizontal Fragmentation – Complex Attribute and Simple Method

fragments (line (2), Figure 3.8). It is possible that the second application of the algorithm results in the original class. This means the application requirements make fragmentation unnecessary.

3.3 Simple Attributes and Complex Methods

Horizontal fragmentation of classes consisting of objects with simple attributes using complex methods requires that we know *à priori* those objects accessed by a method invocation and that encapsulation is not violated. The former requirement is accomplished with static analysis and the latter is inherent in the object model. Achieving optimal fragmentation requires that objects that invoke methods on a set of other related objects should be contained within the same fragment. Obviously, if this can be achieved it will be possible to allocate highly related objects together. This goal is reflected in the following definitions because we can develop an algorithm that maximizes local relevant access and minimizes local irrelevant access.

Thus, we present definitions that enable us to define groups of objects of each class being fragmented that are needed together based on nested method invocations from other classes in the database.

Definition 3.11 *Object reference set of method* $M^{i.j}$ denoted oref($M^{i.j}$) contains all objects referenced by method $M^{i.j}$ of some object in class C_i. ■

We can enumerate the objects in oref($M^{i.j}$) by $\{I_1^k, I_2^l, \ldots, I_n^m\}$ where I_n^m is the n^{th} object needed by method $M^{i.j}$ and is also an object of class C_m.

Definition 3.12 *Class fragment object reference set coref(*F_i^h*,*C_m*)* are objects of class C_m referenced by methods of class fragment F_i^h of class C_i. This is derived from oref($M^{i.j}$) as : coref(F_i^h,C_m) $= \bigcup_{I_m \in oref(M^{i.j})} I_m$ for all $M^{i.j} \in F_i^h$. ■

If fragments generated with coref statistics contain overlapping objects, we use method object affinity value between each overlapping object and each of the fragments to decide which one fragment it is most beneficial to keep this object in, taking into consideration all application and nested method needs. This measure is also important in calculating the global affinity between two fragments (say a derived and primary fragment) and used during merging of such fragments. Thus, the method relevant and irrelevant access statistics in this model have been modified to include nested method accesses to these objects.

Definition 3.13 *Number of references* of method $M^{i.j}$ to an object I_m of class C_m is denoted numref($M^{i.j}$,I_m) which is the number of times method $M^{i.j}$ accesses object I_m during one invocation of $M^{i.j}$. ■

Definition 3.14 *Class number of reference cnumref(F_i^h,I_m)* of class fragment F_i^h on object I_m of class C_m, is the sum of all accesses made to this object by all methods of this class fragment. Thus, cnumref(F_i^h,I_m) = $\sum_{M^{i.j} \in F_i^h}$ numref($M^{i.j}$,I_m). ■

Definition 3.15 *The Method Object Affinity maffo(I_i,I_j)* is the sum of accesses made by any methods of a class fragment or applications to both objects I_i and I_j. maffo(I_i,I_j)= (application accesses to both objects) + (method accesses to both objects)

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

$\sum_{\{M^{k.j} || M^{k.j} \in C_k \wedge M^{k.j} \text{access} I_i \wedge I_j\}} (numref(M^{k.j}, I_i) + numref(M^{k.j}, I_j))$ ■

Definition 3.16 *Method Access frequency macco(I_m)* of an object (I_m) is the sum of the access frequencies of all the applications $\mathcal{Q}_i$ accessing the object plus the sum of all method references to the object. Thus, $macco(I_m) = \sum_i^q acc(q_i, I_m) + \sum_{\forall M^{i.j} | I_m \in oref(M^{i.j})} numref(M^{i.j}, I_m)$. ■

Definition 3.17 *Method relevant access MRA(F_i^d,F_i^p)* between a derived fragment F_i^d and primary fragment F_i^p of a class C_i, measures the number of times objects of both fragments are accessed by methods of other objects. Thus, MRA(F_i^d,F_i^p) = $\sum_{I_m \in \{F_i^d \cap F_i^p\}}$ numref($M^{i.j}$,I_m) for any $M^{i.j}$. ■

Definition 3.18 *Method Irrelevant access MIA(F_i^d,F_i^p)* between a derived fragment F_i^d and a primary fragment F_i^p, measures the number of times objects members of either of the fragments but not both are accessed by methods of other objects.

MIA(F_i^d,F_i^p) = $\sum_{I_m || I_m \in \{((F_i^p \cup F_i^d) - (F_i^d \cap F_i^p))\}} acc(q_p, I_m)$

+ $\sum_{I_m || I_m \in \{((F_i^p \cup F_i^d) - (F_i^d \cap F_i^p))\}} numref(M^{i.j}, I_m)$ for all applications q_p and methods $M^{i.j}$ accessing this object. ■

Definition 3.19 *Object Fragment Affinity* between an object I_m and a horizontal fragment F_i^h, denoted ofaff(I_m,F_i^h), is the amount of accesses made to fragment F_i^h *vis a vis* the object I_m.
ofaff(I_m,F_i^h) $= \sum_{I_j \in F_i^h}$ maffo(I_m,I_j), m $\neq$ j. ■

Definition 3.20 *Method Affinity between Fragments maff(F^d,F^p)* is the difference between relevant and irrelevant access: maff(F_i^d,F_i^p)= MRA(F_i^d,F_i^p) - MIA(F_i^d,F_i^p). ■

Affinity Rule 3.3 Select the primary fragment that maximizes the method affinity measure maff(F_i^d, F_i^p) where F_i^d is the derived fragment generated from method dependencies and F_i^p is a primary fragment in the class ranging over all candidate fragments. ■

Affinity Rule 3.4 The primary fragment F_i^p that maximizes the function ofaff(I_k, F_i^p) is where I_m is placed. ■

Thus three dependence types between classes are captured in a complete design with the most complex object model (Section 3.4). The inheritance relationship between classes is captured in the fragmentation scheme by propagating the fragmentation of a subclass to a superclass so that derived fragments of a superclass, based on the fragments of its subclasses, is obtained. The second type of dependence is from the complex hierarchy. If a class C_i contains another class C_j as part of it, the fragmentation of the contained class C_j is used to obtain derived fragments of the containing class C_i. The third is method dependence. If objects of a fragment of a class C_i invoke methods in objects of another class C_j, we want to group all objects of the class C_j invoked by all objects of the fragment of C_i into a derived fragment of C_j, so that these two fragments are allocated together.

3.3.1 The Algorithm

This section presents an algorithm where objects consist of simple attributes and complex methods. In this model, only two types of dependencies exist between classes – the inheritance hierarchy and method dependencies. Thus, algorithm *Hor_SA_CM* (Figure 3.11) first generates class fragments from the inheritance hierarchy information using *HorizontalFrag* (line (1)). The second iteration captures the method dependency information using object reference statistics. The objective of this iteration is to take each existing fragment of every class, say class C_k in the object base and produce a derived fragment F_i^d of the class being fragmented C_i, due to complex method dependencies of objects of another class fragment F_k^j. It further selects a primary fragment F_i^p of this class C_i most appropriate to merge with this derived fragment F_i^d and eventually produces a non-overlapping set of horizontal fragments that has incorporated method dependency information. This second iteration is formally defined as algorithm *Derived_From_CompM* of Figure 3.10. The first step of Figure 3.10 takes every class (line (1)) and creates a set of objects referenced by each class's fragments (line (3-5)), called the *derived fragment*. The algorithm then determines which primary fragment the derived fragment has the greatest affinity with (line (6-13)) and adds it to the appropriate primary fragment (line (14)). Finally, since it is possible to place an object in more than one primary fragment, Affinity Rule 3.4 is used to find the primary fragment with which it has the greatest affinity (line 15 of 3.10 and line (1-7) of 3.9) and it is removed from others (line (8-11) of 3.9).

Algorithm 3.6 *(Disallowing Overlapping of Objects in Fragments)*

Algorithm No_Overlap
input: $\mathcal{F}^h_{C_i}$: set of horizontal fragments of the classes.
output: $\mathcal{F}^i_h$: set of non-overlapping horizontal fragments of the classes.
var
    Prifragforobj : set of objects;
    k : integer;
begin
  // This **algorithm** ensures each object belongs to //
  // only one fragment.//
  for each overlapping object I_i of class fragments (F^p_{i1},...,F^p_{in}) (1)
    k = 1 (2)
    Prifragforobject= $F^p_k \in C_i$ (3)
    for every fragment $F^p_i \in$ class C_i (4)
      if ofaff(I_i,F^p_i) $>$ ofaff(I_i,F^p_k), (5)
        then begin
          Prifragforobject = F^p_i (6)
          k = k + 1 (7)
        end; {**begin**}
    end; {**for** F^p_i}
    F^p_i = Prifragforobject (8)
    for every fragment $F^p_k \in$ class C_i (9)
      if ($I_i \in F^p_k$) and ($F^p_k \neq F^p_i$) (10)
        then $F^p_k = F^p_k - I_i$ (11)
    end; {**for** F^p_k}
  end; {**for** I_i}
end; {**of No_Overlap**}

Figure 3.9: Disallow Overlap of Objects in Fragments

Algorithm 3.7 *(Generate Derived Fragment Based on Complex Methods)*

Algorithm Derived_From_CompM
input: $\mathcal{F}_i^h$: set of horizontal fragments of the classes
$oref(M^{i.j})$: set of object references for all $M^{i.j} \in C_i$.
$\mathcal{C}_d$: set of database classes
output: $\mathcal{F}_i^h$: set of horizontal fragments of the classes
var
$\mathcal{F}_i^d$: set of derived horizontal fragments for class C_i
Prifragforderived : set of objects;
k : integer;
begin
// This **algorithm** captures the method dependency information //
for every fragment $F_{ik} \in$ class $\mathcal{F}_{c_i}$ (1)
$F_i^d = \emptyset$ (2)
for each class $C_m \in \mathcal{C}_d$ **do** (3)
if $m \neq i$ **then** (4)
$F_i^d = F_i^d \cup$ coref(F_{ik},C_m) (5)
end; {**for** C_m}
end; {**for** F_{ik}}
for each primary fragment $F_i^p \in \mathcal{F}_{c_i}$ of class C_i, (6)
k = 1 (7)
Prifragforderived = $F_k^p \in \mathcal{F}_{c_i}$ (8)
for every fragment $F_i^p \in \mathcal{F}_{c_i}$ (9)
if maff(F_i^d,F_i^p) > maff(F_i^d,F_k^p) (10)
then begin
Prifragforderived = F_i^p (11)
k = i (12)
end; {**begin**}
end; {**for** F_i^p}
F_i^p = Prifragforderived (13)
$F_i^p = F_i^p \cup F^d$ (14)
end; {**for** F_i^d}
$\mathcal{F}_{c_i}$ = No_overlap(F_i^p) (15)
end; {**of Derived_From_CompM**}

Figure 3.10: Capturing the Complex Method dependency information Class Fragments

Algorithm 3.8 *(Simple Attribute and Complex Methods)*

Algorithm Hor_SA_CM
input: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$: set of database classes
L(C) : the class hierarchy
$oref(M_i^j)$: set of object references for all $M_i^j \in C_i$.
output: $\mathcal{F}_{c_i}$: set of horizontal fragments of the classes
var
$\mathcal{P}_i$: 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
$\mathcal{F}_{c_i}$ **= HorizontalFrag(**$\mathcal{Q}_i$, $\mathcal{C}_d$, **L(C))** (1)
for each class $C_i \in C_d$ **do** (2)
$\mathcal{F}_{c_i}$ = Derived_From_CompM($\mathcal{F}_{c_i}$,C_i,$oref(M_i^j)$,$\mathcal{C}_d$) (3)
end; {**for** C_i}
end; {**of Hor_SA_CM**}

Figure 3.11: Horizontal Fragmentation For Simple attributes and Complex methods

3.4 Complex Attributes and Complex Methods

This section presents an algorithm for horizontally fragmenting classes consisting of complex attributes and complex methods as discussed in [14, 18]. With this model, the database information that needs to be captured includes: the inheritance hierarchy, the attribute link to reflect the part-of hierarchy, and the method links to reflect the use of methods of other classes by fragments of a class. The algorithm is essentially the sum of the previous model's algorithms. The first and second iterations of this algorithm involve generating a set of horizontal fragments that capture the inheritance hierarchy and attribute link information using the same technique presented in Section 3.1. During the third iteration, the method link information is captured by generating a set of derived fragments using the same techniques as in the second iteration of the algorithm presented in Section 3.3 with algorithm *Derived_From_CompM* of Figure 3.10. These final derived fragments are merged with horizontal fragments from the second iterations to obtain the final fragments of the classes. The formal presentation of this algorithm *Hor_CA_CM* is given in Figure 3.12. We illustrate the application of the third iteration of this algorithm using an example given in the following section.

3.4.1 An Example

This example incorporates class models consisting of complex attributes and complex methods. The extended complex class object base is given in Figure 3.13.

The database schema information that is one input to this design process consists of the class hierarchy of the object base which is as given in Figure 3.14, and the class composition hierarchy in Figure 3.15. The other input is the application information q_1 to q_5 given below.

Algorithm 3.9 *(Complex Attribute and Complex Method)*

Algorithm Hor_CA_CM
input: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$: set of database classes
L(C) : the class hierarchy
A(C) : the class composition hierarchy with attribute link between classes
$oref(M_i^j)$: set of object references for all $M_i^j \in C_i$.
output: $\mathcal{F}_{c_i}$: set of horizontal fragments of the classes
var
$\mathcal{P}_i$: 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
$\mathcal{F}_{c_i}$ = **HorizontalFrag(**$\mathcal{Q}_i$, $\mathcal{C}_d$, **L(C))** (1)
$\mathcal{F}_{c_i}$ = **HorizontalFrag(**$\mathcal{Q}_i$, ($\mathcal{F}_{c_i}$, **A(C))** (2)
for each class $C_i \in C_d$ **do** (3)
$\mathcal{F}_{c_i}$ = Derived_From_CompM($\mathcal{F}_{c_i}$,C_i,$oref(M_i^j)$,$\mathcal{C}_d$) (4)
end; {**for** C_i}
end; {**Hor_CA_CM**}

Figure 3.12: Horizontal Fragmentation For Complex Attribute and Methods

Person = {Person,{a.ssno,a.name,a.age,a.address},
{m.ssno-of,m.whatname,m.daysold,m.newaddr},
{ I_1 {Person1,John James,30,Winnipeg}
I_2 {Person2,Ted Man,16,Winnipeg}
I_3 {Person3,Mary Ross,21,Vancouver}
I_4 {Person4,Peter Eye,23,Toronto}
I_5 {Person5,Mary Smith,40,Toronto}
I_6 {Person6,John West,32,Vancouver}
I_7 {Person7,Jacky Brown,35,Winnipeg}
I_8 {Person8,Sean Dam,27,Toronto}
I_9 {Person9,Bill Jeans,43,Vancouver}
I_{10} {Person10,Mandu Nom,30,Winnipeg} } }
Prof = Person pointer $\odot$ {Prof,{a.empno,a.status,a.dept,a.salary,a.student},
{m.empno,m.status-of,m.coursetaught,m.whatsalary},
{I_1 (person pointer5) $\odot$ {Prof1,asst prof,Computer Sc.,45000,students pointers}
I_2 (person pointer6) $\odot$ {Prof2,assoc prof,Math,60000,students pointers}
I_3 (person pointer9) $\odot$ {Prof3,full prof,Math, 80000,students pointers}
I_4 (person pointer10) $\odot$ {Prof4,full prof,Math,82000,students pointers} } }
Student = Person pointer $\odot$ {Student,{a.stuno,a.dept,a.feespd} ,
{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}
I_5 (person pointer7) $\odot$ {Student5,Math,Y}
I_6 (person pointer8) $\odot$ {Student6,Stats,N} } }
Grad = Student pointer $\odot$ {Grad,{a.gradstuno,a.supervisor},
{m.gradstuno-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 = {Dept,{a.code,a.name,a.profs,a.students},{m.whichdept,m.number-of-profs}},
{ 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 pointers}
I_4 {Dept4,{Stats,prof pointers,student pointers} } }

Figure 3.13: The Complex Sample Object Database Schema

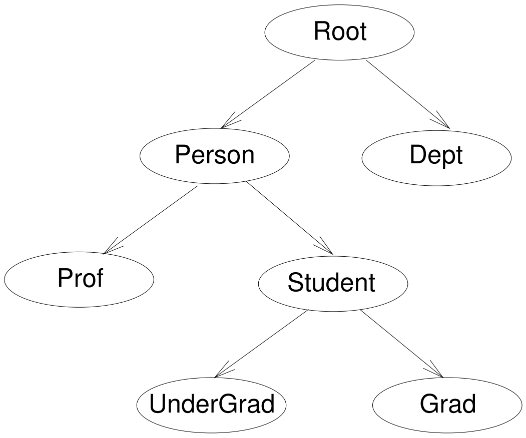


Figure 3.14: Complex Class Hierarchy

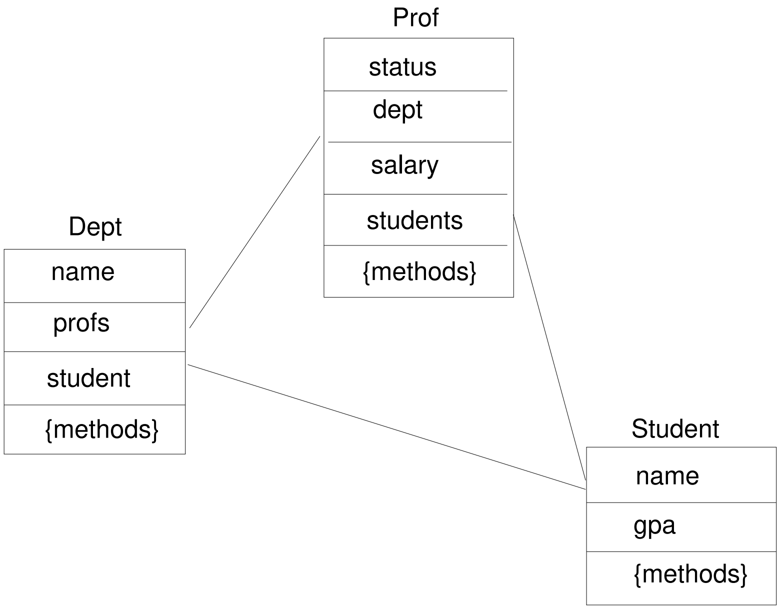


Figure 3.15: Class Composition Hierarchy

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:

$\{P_1$: supervisor = "Prof John West",P_2: supervisor = "Prof Mary Smith"$\}$

q_2: This application groups profs by their addresses. The methods used are defined on class **Prof** with the following predicates:

$\{P_1$: address = "Winnipeg",P_2: address = "Vancouver",P_2: address = "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:

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

q_4: Groups students by their departments. The methods are from class **Student** and the predicates used are:

$\{P_1$:dept="Math",P_2:dept="Computer Sc",P_3:dept="Stats"$\}$

q_5: This application groups departments by their general area determined by their names. The methods are for the class **Dept** and the predicates used are:

$\{P_1$:dept="Math",P_2:dept="Computer Sc",P_3:dept="Stats"$\}$

Fragmentation Process : Execution of Algorithm Hor_CA_CM

Line 1 of algorithm Hor_CA_CM (Figure 3.12) generates the horizontal fragment using the complex class hierarchy as follows:

1. Generates the complex class link graph shown in Figure 3.16 from Figure 3.14.

2. Defines primary horizontal fragments for all owner classes on the link graph to obtain:

   Class Grad

   Fragments F_1^p = instance object 1 $\{I_1\}$

   F_2^p = instance objects 2 and 3 $\{I_2,I_3\}$

   Class Student

   Fragments $F_1^p = \{I_1,I_5\}$, $F_2^p = \{I_2,I_4\}$, and $F_3^p = \{I_3,I_6\}$.

   Class Prof

   Fragments $F_1^p = \{I_4\}$, $F_2^p = \emptyset$, $F_3^p = \{I_2,I_3\}$, $F_4^p = \emptyset$, $F_5^p = \emptyset$, and $F_6^p = \{I_1\}$

   Class Dept

   Fragments $F_1^p = \{I_2,I_3\}$, $F_2^p = \{I_1\}$, and $F_3^p = \{I_4\}$.

3. Generates derived fragments of member classes to obtain:

   Class Student

   Using the primary fragments of the owner class Grad, the derived fragments are:

   $F_1^d = \{I_1\}$, and $F_2^d = \{I_2,I_5\}$.

   Derived fragments of the Class Person

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

   $F_1^d = \{I_1,I_7\}$, $F_2^d = \{I_4,I_3\}$, and $F_3^d = \{I_2,I_8\}$.

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

   $F_4^d = \{I_{10}\}$, $F_5^d = \{I_6,I_9\}$, and $F_6^d = \{I_5\}$.

4. Combines Primary and Derived fragments of member classes using the access

frequency information given below. 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.

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$(q_4, F_3) = 0$. For example, the first primary fragment of the class *Student* F_1^p corresponds to the minterm predicate {dept= "Math" $\wedge dept \neq$ "$ComputerSc" \wedge dept \neq$ "$Stats"$} and this fragment contains the set of instance objects $\{I_1, I_5\}$. Since the access frequency to objects of this first fragment by application Q_1 is 10, and no other application accesses these objects, the access frequency of each of these objects is 10. In the case where minterms of a class are defined from predicates arising from more than one application, access frequency of an object in each fragment is obtained by adding relevant access frequencies of the relevant predicates of the two applications. For example, instance object $\{I_4\}$ of F_1^p of the class *Prof* has access frequency of (5 + 20) from the two applications Q_2 and Q_3 accessing this class respectively.

Member Class Student

F_1^d has the maximum affinity with F_1^p of 0, and so we merge F_1^d and F_1^p to get :

$F_1^h = \{I_1, I_5\}$

Details of this computation is given in the next three lines

$(aff(F_1^d, F_1^p) = 10 - 10 = 0$

$aff(F_1^d, F_2^p) = 0 - (10 + 5 + 5) = -20$

$aff(F_1^d, F_3^p) = 0 - (10 + 0 + 0) = -10)$

F_2^d has the maximum affinity with F_1^p of -5, and so we merge F_2^d and F_1^p to obtain F_1^h ={I_1,I_5,I_2}

Details of this computation is given in the next three lines

$(aff(F_2^d, F_1^p) = 10 - (10+5) = -5$

$aff(F_2^d, F_2^p) = 5 - (5+10) = -10$

$aff(F_2^d, F_3^p) = 0 - (5+0+10+0) = -15)$

However, since overlapping object I_2 has maximum faff(I_2,F_2^h) of 5, we keep I_2 in F_2^h and delete it from F_1^h.

F_2^h ={I_2,I_4}

The third primary fragment remains the same as

F_3^h ={I_3,I_6}

Class Person

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

This completes the first iteration. The second iteration is the execution of line 2 of algorithm *Hor_CA_CM* (Figure 3.12) using the fragments from the first iteration and the class composition hierarchy. The link graph generated from the class composition hierarchy is given in Figure 3.17.

We generate derived fragments of the member classes *Dept and Prof* based on applications on owner class *Student*. This is done by generating fragments of Prof that satisfy the dominating predicates of the class Student which are: $P_{student}$: {dept="Math",dept="Computer Sc",dept="Stats"}. This leads to the following three derived fragments of the class Prof.

Class Prof

F_1^d = {I_2,I_3,I_4}, F_2^d = {I_1}, and F_3^d = ($\emptyset$)

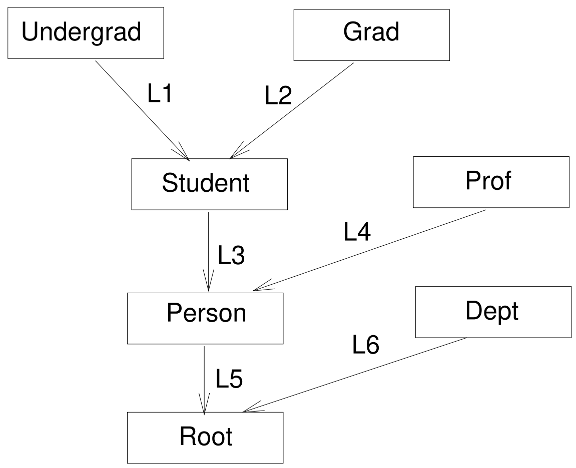


Figure 3.16: Complex Class Hierarchy's Link Graph

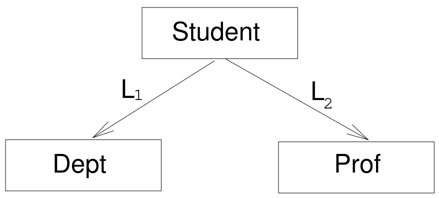


Figure 3.17: Class Composition Hierarchy Linkgraph

Derived Fragments for member class Dept

$F_1^d = \{I_2,I_3\}$, $F_2^d = \{I_1\}$, and $F_3^d = \{I_4\}$. Combining primary and derived fragments of member classes yields the following horizontal fragments:

Class Prof

$F_1^h = \{I_2,I_3,I_4\}$

$F_2^h = \{I_1\}$

This is because F_1^d has highest affinity of 35 with F_3^p of *Prof* and merges with it to generate F_1^h. Similarly, F_2^d merges with F_6^p to obtain F_2^h. Details of this computation are:

$$(aff(F_1^d, F_1^p) = 25 - (30 + 30) = -35$$
$$aff(F_1^d, F_3^p) = (30 + 30) - 25) = 35$$
$$aff(F_1^d, F_6^p) = 0 - (15 + 30 + 30 + 25) = -100)$$

$$(aff(F_2^d, F_1^p) = 0 - (15 + 25) = -40$$
$$aff(F_2^d, F_3^p) = 0 - (30 + 30) = -60$$
$$aff(F_2^d, F_6^p) = 25)$$

Class Dept

$F_1^h = \{I_2,I_3\}$

$F_2^h = \{I_1\}$

$F_3^h = \{I_4\}$

Lines 3 - 5 of the algorithm captures method dependence information by defining fragments of classes that correspond to the *coref* of its fragments from the second iteration. Assume that objects of the class *Student* invoke methods of objects of classes *Prof* and *Dept*. Since *Student* is not a subclass or contained class of any of

these two classes, then the use of methods of these classes by class *Student* can not be treated under inheritance or class composition hierarchies, but as nested or complex method hierarchy. The pattern of method invocation of objects of a fragment of the class *Student* is shown in Figure 3.18. Using *oref* statistics, the derived fragments of classes based on complex methods of the first fragment of the class *Student* is as given next. Derived fragment of the Class *Dept* is $F_1^d = \{I_1, I_2, I_3\}$. The derived horizontal fragment of *Prof* based on methods of the first fragment of the class (Student), as shown in Figure 3.18 is: $F_1^d = \{I_1, I_2, I_3\}$. For each new derived fragment of classes affected, we find a primary fragment it can combine with by applying method reference statistics to select the primary fragment with which it has highest $maff$. This computation requires more quantitative information about access patterns of methods. Assume the following quantitative statistics.
numref($M_3^{student}, I_1^{prof}$) = 15, numref($M_2^{student}, I_2^{prof}$) = 30,
numref($M_3^{student}, I_3^{prof}$) = 20, numref($M_1^{student}, I_1^{prof}$) = 40,
numref($M_1^{student}, I_3^{prof}$) = 20.

Class Prof

We are now ready to select a current horizontal fragment of *Prof* generated from the second iteration that is suitable to merge with each derived fragment of *Prof* generated according to method dependencies of class fragments. For purposes of simplicity, we show one such derived fragment $F_1^d = \{I_1, I_2, I_3\}$ and the current horizontal fragments of *Prof* are $F_1^h = \{I_2, I_3, I_4\}$ and $F_2^h = \{I_1\}$. Since F_1^h has highest maff(F_1^d, F_1^h) with the derived fragment F_1^d of 30, we merge F_1^d with F_1^h to obtain the fragment $F_1^h = \{I_1, I_2, I_3, I_4\}$

Class Student

Here, we assume that application information leads to the merge of F_1^d and F_2^h to generate $\{I_2, I_3, I_4, I_5\}$. This leads to I_3 and I_5 overlapping in both F_2^h and F_3^h as well as in F_1^h. Thus, we apply Affinity Rule 3.4 to the overlapping objects and the contending fragments to decide the best placement for them. The result is that I_5 gets removed from F_2^h and I_3 gets removed from F_3^h to produce the following final fragments for the class *Student*. $F_1^d = \{I_2, I_3, I_4, I_5\}$ and final fragments are: $F_1^h = \{I_1, I_5\}$, $F_2^h = \{I_2, I_4, I_3\}$ and $F_3^h = \{I_6\}$.

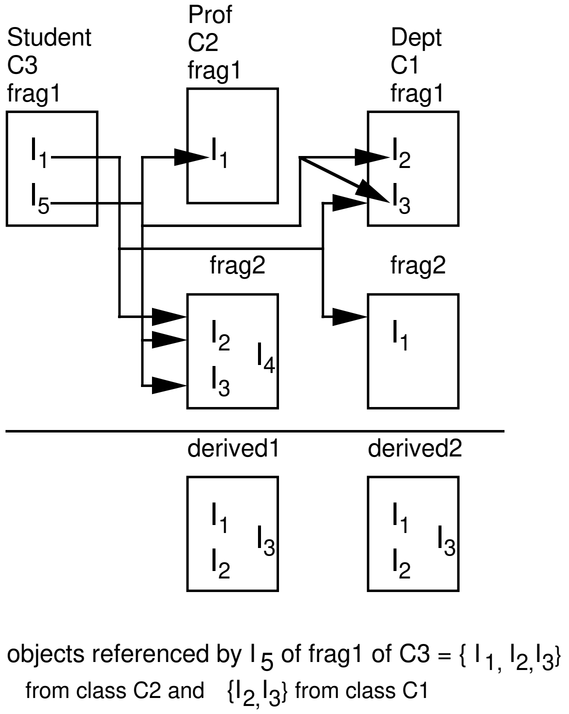


Figure 3.18: Generation of Derived Fragments of Classes based on Complex Methods

From Figure 3.18, the collection of all objects of C_2 (Prof) invoked by methods of objects in fragment F1 of C_3 (Student) is $\{I_1, I_2, I_3\}$ and the collection of all objects of C_1 (Dept) invoked by methods of fragment F1 of C_3 (Student) is $\{I_1, I_2, I_3\}$. These collections form derived fragments of these classes based on complex methods of a fragment of *Student*. Next, quantitative information about number of accesses made by these methods to these objects are used in affinity rules to merge the set of primary and derived fragments of each class.

3.5 The Structure Chart and Complexities of Algorithms

This section discusses other important characteristics of the algorithms presented in this chapter. In the first part, a structure chart that concisely defines the horizontal fragmentation scheme for the most complex class model consisting of complex attributes and methods is presented; and secondly the complexities of the horizontal fragmentation schemes are discussed.

3.5.1 The Structure Chart

An interesting feature of this design is reusability of design for a simpler class model in a more complex class model. This is illustrated concisely using the structure chart for the fragmentation scheme of the most complex class model - consisting of complex attributes and methods, shown in Figure 3.19. The structure chart also gives a summary of all the algorithms that are used in the design.

3.5.2 Complexities of Algorithms

The benefits of distributed database design generally outweigh the impending overheads involved in running the partitioning algorithms. This argument is supported by the time complexities of the fragmentation algorithms for the four class models presented in Sections 3 through 6 on the basis of the worst case analysis. Assume that f represents the largest number of fragments possible, o represents the largest

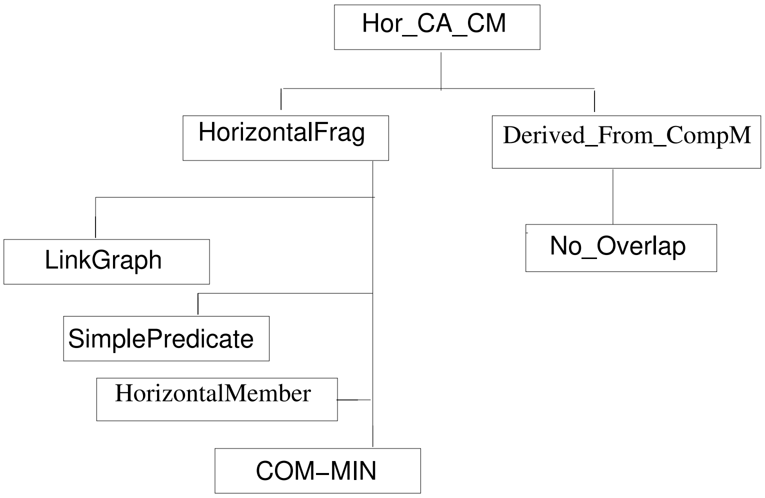


Figure 3.19: The Structure Chart For Model with Complex Attributes and Methods

number of instance objects in a class or a fragment, q is the number of user queries, c is the number of classes in the database, m is the largest number of method invocations in a user query, while p is the largest number of predicates in a method invocation and p_c is the largest number of predicates in a class. The horizontal fragmentation algorithm for the class model consisting of simple attributes and simple methods *HorizontalFrag* of Figure 3.4 is of time complexity O(c^3 + qmp + cf + c^2o + cf^4o + fc) which simplifies to O(c^3 + qmp + c^2o + cf^4o). This is because line 1 of this algorithm *HorizontalFrag* which is the *Linkgraph* algorithm is of O(c^3), and generation of simple predicates (line 2) is of O(qmp). Lines 5 through 7 of *HorizontalFrag* are of O(c^2o) because the object join operation is of time O(o); while lines 8 through 10 have time complexity O(cf^4o) since the algorithm *HorizontalMember* (line 9) is of time O(f^4o). The prevailing time complexity of the algorithm depends on the number of classes in the object base and the number of queries and objects in a class. In a system with many more queries than classes, it is likely that the term *qmp* becomes the dominant term and that makes the complexity of *HorizontalFrag*

O(qmp).

The time complexity of the algorithm *Hor_CA_SM* for the second model with complex attribute and simple method in Figure 3.8 is the same as that for HorizontalFrag and is O(c^3 + qmp + c^2o + cf^4o). Both the third and fourth class models consisting of simple attributes and complex methods; and complex attributes and complex methods respectively, have time complexity which is O(c^3 + qmp + c^2o + cf^4o + c^2fo) which simplifies to O(c^3 + qmp + cf^4o + c^2fo). This is because the algorithm *Derived_From_CompM* of Figure 3.10 is of time complexity O(cfo). Thus, these algorithms are polynomial in the sizes of these variable inputs.

Chapter 4

Vertical Fragmentation

This chapter presents vertical fragmentation algorithms for all models as defined in the taxonomy in Figure 1.3. This represents the second vertical layer on the figure. Section 4.1 presents a vertical fragmentation algorithm for classes consisting of objects that have simple attributes using simple methods while Section 4.2 discusses the algorithm for fragmenting class objects composed of complex attributes and simple methods. Section 4.3 presents the algorithm for handling classes composed of simple attributes and complex methods while Section 4.4 discusses the algorithm for vertically fragmenting classes composed of complex attributes and complex methods. Finally, Section 4.5 discusses the time complexities of the algorithms.

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 minimize user application execution time [38]. 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. 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 object's attribute values except through the object's 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 in [21, 34]. 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 [16, 18] 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 [34]. (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. As with horizontal fragmentation, vertical fragmentation should satisfy the correctness rules of completeness, disjointness and reconstructibility. The reconstruction of all vertical fragments to produce the original class is achieved with a fragment integrator operator ($\bigodot F_i$) which is defined as follows. The fragment integrator operator ($\bigodot F_i$)

joins objects in all vertical fragments that have the same object identifier or key attribute value. We first present some definitions and assumptions before presenting the algorithm.

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.

Definition 4.1 *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 4.2 *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 4.3 *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).

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

Definition 4.4 *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 4.5 *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 4.6 *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 4.7 *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 (complex method 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 4.8 *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 4.9 *Attribute/Method Affinity* AMA($A^{i.j}$,$M^{i.m}$) between an attribute and a method of the same class C_i is the sum of the access frequencies of all methods using this attribute and this method together at all sites. AMA$(A^{i.j}, M^{i.m}) =$

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

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

Definition 4.10 *Attribute Fragment Affinity AFA($A^{i.m}$,$F^{i.j}$)* is a measure of the affinity between attribute $A^{i.m}$ and vertical fragment $F^{i.j}$, and is the sum of all the attribute/attribute and attribute/method affinities between $A^{i.m}$ and all attributes and methods of the class fragment $F^{i.j}$.

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

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

Affinity Rule 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. ■

4.1 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 [34]:

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 that in Navathe *et al.* [34]. We use the same techniques for grouping attributes to group methods starting with the three matrices - method usage matrix, application frequencies, and method affinity matrices. Secondly, we cluster methods in much the same fashion as they cluster attributes, 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 that methods most frequently used together, are grouped together. In the relational system, the same attribute usage matrix used to generate the attribute affinity matrix is used for this process. With the inheritance link information taken into consideration, we group methods/attributes of a class accessed together at a site to reflect accesses by applications running on each subclass at a particular site. Thus, we need to modify the method usage matrix of the class to include application accesses to null methods of subclasses and the sites where they are used. Finally, unlike the relational case, partitioning of the methods of the class does not necessarily end the vertical fragmentation process. This is because we want to group both methods and attributes used together by applications in a fragment. To achieve this objective, we use method-attribute binding to group attributes of a class with methods that use them, and later obtain non-overlapping fragments using a set of attribute-fragment affinity rules.

The algorithms also require the following data structures and functions.
$\mathcal{M}^{c_i}$: set of all methods of class C_i.
$\mathcal{C}_i^{des}$: set of all descendant classes of C_i.

$\mathcal{C}_i^{cont}$: set of all containing classes of C_i.

$\mathcal{C}_i^{cmeth}$: set of all complex method classes of C_i.

EM-set(C_i^{des}) : set of extended methods of C_i and its descendant classes.

NM-set(C_i^{des}) : set of null methods of C_i and its descendant classes.

EMapplic-set(C_i^{des}) : set of applications accessing extended methods of C_i and its descendant classes.

$\mathbf{L}(C_i)$: 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$(\mathbf{L}(C_i^{des}))$: 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 descendants. The method usage matrices are generated by the Algorithm *UsageMtrx* defined in Figure 4.1. 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 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 4.1). 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 4.2. 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 Definition 4.5) between the two extended methods of the class in this (row,column) position (Lines 1-4 of Figure 4.2).

M3. Use the Bond Energy Algorithm (BEA) developed by [33] as presented in [34, 38] to generate clustered affinity matrix of the class. This algorithm accepts the method affinity matrix as input and permutes its rows and columns to generate a clustered affinity matrix. The clusters are formed so that methods with larger affinity values are collected together.

M4. Generate a modified method usage matrix of the class as described in the Algorithm *MUsageMtrx* of Figure 4.3. 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 4.3). Next the *Partition* algorithm (PARTITION) of [34, 38] takes the clustered affinity matrix and the modified method usage matrix to produce fragments of the methods. The *Partition* algorithm finds sets of methods that are mostly accessed by distinct sets of applications.

M5. Use method-attribute reference information of the methods in each method fragment (MAR of Definition 4.1) to include in each method fragment all

Algorithm 4.1 *(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.
begin
//Generate the original method usage matrix for class C_i //
// and other classes on the tree passed in as parameter.//
$\mathcal{C}_c = \{C_i\}$ (1)
while $\mathcal{C}_c \neq \emptyset$ **do** (2)
C_k = a class $\in \mathcal{C}_c$ (3)
if C_k is not a leaf class (4)
begin
$\mathcal{C}_c = \mathcal{C}_c \cup children(C_k)$ (5)
for each application $q_j \in$ EMapplic-set(C_i^{des}) **do** (6)
For each method, $M^{k.m} \in$ EM-set(C_k)) do (7)
if use(q_j,$M^{k.m}$) = 1 then (8)
MU(q_j,$M^{k.m}$) = 1 (9)
else MU(q_j,$M^{k.m}$) = 0 (10)
end; {for q_j}
end; {if C_k}
$\mathcal{C}_c = \mathcal{C}_c - \{C_k\}$ (11)
end; {while}
end {UsageMtrx}

Figure 4.1: Original method Usage Matrix Generator

Algorithm 4.2 *(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): 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 4.5. //
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 4.2: Method Affinity Matrix Generator

Algorithm 4.3 *(MUsageMtrx - Generate Modified method usage matrices)*

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

Figure 4.3: Modified Usage Matrix Generator

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 in 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 4.9 and 4.10, respectively. It places the attribute in the fragment with highest affinity measure and removes the attributes from every other.

The formal algorithm for vertically fragmenting a class consisting of simple attributes and simple methods is presented as Algorithm *VerticalFrag* of Figure 4.4. This algorithm takes as its inputs a set of user queries, class inheritance of the database, a class in the hierarcht 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[9]) before computing the method affinity matrix of the class (line 2). Next, the clustered method affinity matrix of the class is generated (line 3) before the class' modified method usage matrix (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.

[9]All line numbers in this paragraph refer to Figure 4.4

Algorithm 4.4 *(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 hierarchy; $\mathcal{C}_i^{des}$: 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 4.4: Class Vertical Fragments Generator

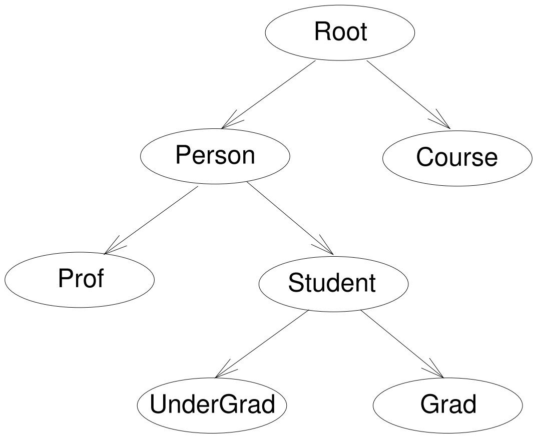


Figure 4.5: Class Hierarchy of Sample Object Base

4.1.1 An Example

We use the class hierarchy depicted in Figure 4.5 and the sample data illustrated in Figure 4.6 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 4.4) using algorithm *UsageMtrx* (Figure 4.1) is:

Person = {Person,{a.ssno,a.name,a.age,a.address},
{m.ssno-of,m.whatname,m.age-in-year,m.newaddr}
{ I_1 {Person1,John James,30,Winnipeg}
I_2 {Person2,Ted Man,16,Winnipeg}
I_3 {Person3,Mary Ross,21,Vancouver}
I_4 {Person4,Peter Eye,23,Toronto}
I_5 {Person5,Bill Jeans,40,Toronto}
I_6 {Person6,Mandu Nom,32,Vancouver} } }
Prof = Person pointer $\odot$ {Prof,{a.empno,a.status,a.salary,a.students,a.courses},
{m.empno-of,m.course-taught,m.whatsalary,m.status-of,m.students-of},
{ I_1 (person pointer5) $\odot$ {Prof1,asst prof,45000, {Ken,Peter,Maria},{111}}
I_2 (person pointer6) $\odot$ {Prof2,assoc prof,60000, {Janet,Larry},{236}} } }
Student = Person pointer $\odot$ {Student,{a.stuno,a.dept,a.feespaid},
{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 $\odot$ {Grad,{a.gradstuno,a.supervisor},
{m.gradno-of,m.whatprog}
{ I_1 (Student pointer1) $\odot$ {Grad1,John West}
I_2 (Student Pointer2) $\odot$ {Grad2,Mary Smith} } }
UnderG = Student
I_1 (Student pointer3)
I_2 (Student pointer4)

Figure 4.6: The Second Simple Sample Object Database Schema

	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

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 frequency matrix of this class are given in Figure 4.7. The second subclass of the class *Person* is the class *Student* and the applications running on this subclass are as follows:

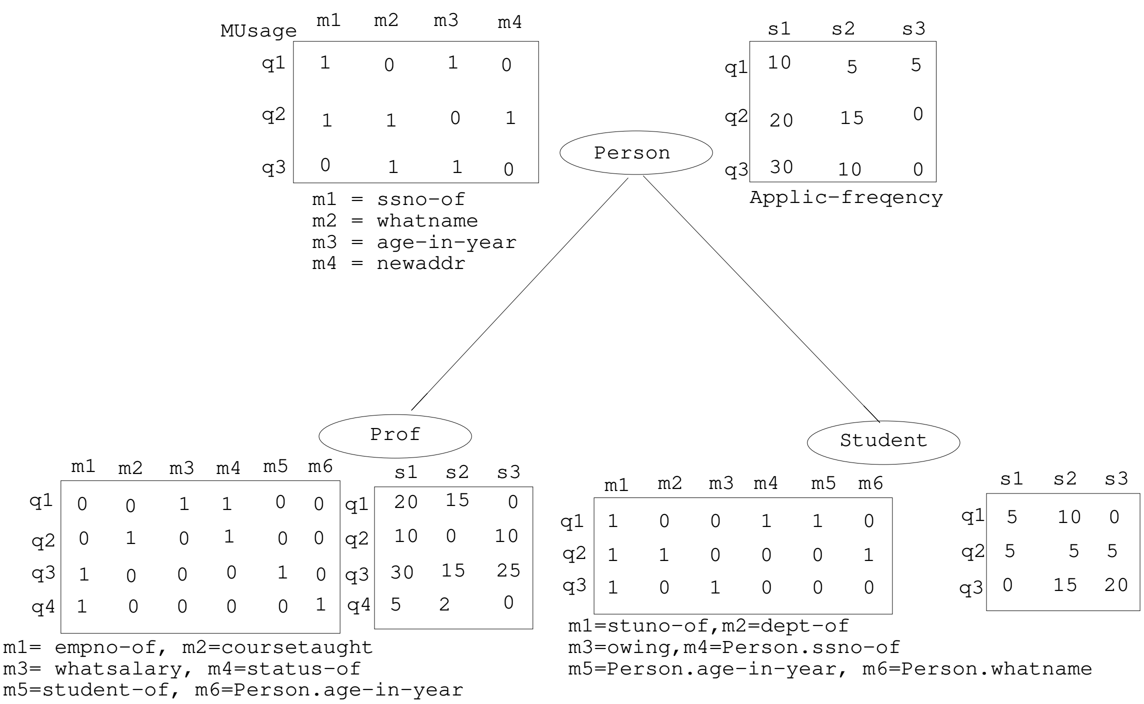


Figure 4.7: The Method Usage and Application Frequency matrices of the Classes

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.

Similarly, the method usage matrix of the subclass *Student* derived from these applications as well as the application frequency matrix of this class are given in Figure 4.7. 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.whatname.* All the matrices needed to compute the method affinity matrix of the class *Person* are given in Figure 4.7.

Executing line 2 of the algorithm *VerticalFrag* of Figure 4.4 using the method usage and application frequency matrices of the class *Person* and its descendant classes *Prof and Student* yields the method affinity matrix for the class *Person* given below. To illustrate the creation of the required method affinity matrix for the class *Person*, we give an example discussing how the values of two of its elements (M_1,M_3) and (M_2,M_2) are obtained.

$$\begin{aligned}
\mathrm{MA}(M^{person.1},M^{person.3}) &= \textstyle\sum_{k=1}^{1}\sum_{l=1}^{3} acc_l(q_1) + \\
&\quad saff_{prof}(M^{person.1}, M^{person.3}) + saff_{student}(M^{person.1}, M^{person.3}). \\
&= (10+5+5) + (0) + (5+10+0) = 35 \\
\mathrm{MA}(M^{person.2},M^{person.2}) &= \textstyle\sum_{k=1}^{1}\sum_{l=1}^{3}(acc_l(q_2) + acc_l(q_3)) + \\
&\quad saff_{prof}(M^{person.2}, M^{person.2}) + saff_{student}(M^{person.2}, M^{person.2}) \\
&= ((20+15+0)+(30+10+0)) + 0 + (5+5+5) \\
&= 35 + 40 + 0 + 15 = 90
\end{aligned}$$

The method affinity matrix for the class *Person* generated from the execution of the algorithm *MAMtrx* of Figure 4.2 as required by line 2 of algorithm *VerticalFrag* of Figure 4.4 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 4.4 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 the 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 methods 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 the use of the methods by applications running on its descendant classes. The modified method usage matrix generated from line 4 of algorithm VerticalFrag (Figure 4.4) 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 = whatname

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(whatname) = {name}, MAR(age-in-year) = {age}, and MAR(newaddr) = {address}. Then, the attribute/method fragments now become: $F_1 = \{ssno, address, ssno{-}of, newaddr\}$ and $F_2 = \{ssno, name, age, ssno{-}of, whatname, newage\}$. Since there are no overlaps of non-class identifier attributes/methods, step 6 of the algorithm VerticalFrag of Figure 4.4 (lines 10-13) leaves the fragments unchanged.

4.2 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 hierarchy 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 [21, 34]. 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 are their access frequencies as defined earlier.

4.2.1 The Algorithm

User applications that access attributes and methods in this class model are of three types: (1) those running directly on this class, (2) those running on descendants 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 4.1.
- **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 4.2.

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 4.1 and the *CCLinkgraph* from Figure 4.8. 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 4.8).
- **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

Algorithm 4.5 *(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.
$\mathcal{C}_i^{cont}$: 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 $\lambda = \emptyset$;
$\mathcal{C}_l = \{C_i\}$ (2)
while $C_i \neq$ Root **do** (3)
for each $C_i \in \mathcal{C}_d$ (4)
for each $C_j \in \mathcal{C}_i^{cont}$ (5)
$\lambda = \lambda \cup (C_i \rightarrow C_j)$ (6)
$\mathcal{C}_l = \mathcal{C}_l \cup \{C_j\}$ (7)
end; {for C_j}
end; {for C_i}
$\mathcal{C}_l = \mathcal{C}_l - \{C_i\}$; (8)
C_i = a class in $\mathcal{C}_l$ (9)
end; {while C_i}
return $(LG(C_i))$; (10)
end;

Figure 4.8: The Linkgraph Generator For Containing Classes of C_i

given in Figure 4.9. 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 [33] as presented in [38, 34] 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 4.3 except this time, we modify the method usage matrix to include a row for every application q_j that accesses a method of the class (lines 1-10, Figure 4.3) being fragmented either through its descendant or containing classes. Next the *Partition* algorithm [34, 38] 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 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 of the class.

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 4.10. This algorithm is the same as the VerticalFrag algorithm of the model with simple

Algorithm 4.6 *(CMAMtrx - includes complex attribute factor in method affinity matrix of class C_i)*

Algorithm CMAMtrx

input: C_i: the database class; $\mathcal{C}_i^{cont}$: set of containing classes of C_i.
EM-set($\mathcal{C}_i^{cont}$) : extended methods set of C_i and its containing classes.
EMapplic-set($\mathcal{C}_i^{cont}$) : applications accessing extended methods of C_i and its containing classes.
MU-set($\mathcal{C}_i^{cont}$): 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 4.6. //
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 4.9: Complex Attribute Modified Method Affinity Matrix Generator

attribute and simple method except that the method affinity matrix includes in addition to subclass affinity (saff) between the extended methods, complex class affinity (ccaff). Secondly, the modified method usage matrix used for the partitioning includes additional rows to account for method usage of the class being fragmented by methods of containing classes.

We simplify the presentation by defining the vertical fragmentation algorithm in terms of the two key matrices of the class being fragmented: the method affinity and the modified method usage matrices of the class. These two matrices constitute the major final inputs to the vertical fragmentation scheme before fragments are produced. Thus, the major difference in the various schemes for fragmenting various class models lies in how these two matrices are obtained. In describing the procedures involved in obtaining the matrices needed before running the vertical fragmentation algorithm, we shall attach the sequence of modifications performed on the matrix as its arguments. Thus, VerticalFrag(MA(C_i^{des}), $MU(C_i^{des})$ 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 4.10 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.

4.3 Simple Attributes and Complex Methods

This section presents a vertical fragmentation algorithm for classes consisting of objects that have simple attributes using complex methods. The two relationships between this class model consisting of simple attributes and complex methods are the inheritance and method link relationships. Vertical fragmentation of this class model requires that we know *à priori* those methods of other classes referenced by each method of the class we want to fragment and that encapsulation is not violated. 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 4.11

Algorithm 4.7 *(Vertical Fragments of Complex attributes and Simple Methods)*

Algorithm Vert_CA_SM
input: $\mathcal{Q}^{\mathcal{C}_i^{des}}$: set of user queries accessing C_i and its descendant classes.
$\mathcal{Q}^{\mathcal{C}_i^{cont}}$: set of user queries accessing C_i and its containing classes.
C_i : the database class to fragment
L(C) : the class hierarchy
$\mathcal{C}_i^{des}$: 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($\mathcal{C}_i^{des}$): application frequency matrices of C_i and descendants.
AF-set($\mathcal{C}_i^{cont}$): application frequency matrices of C_i and containing classes.
output: $\mathcal{F}^{c_i}$: set of vertical fragments of C_i.
var
MU-set($\mathcal{C}_i^{des}$) : method usage matrices for class C_i and its descendants.
MU-set($\mathcal{C}_i^{cont}$) : method usage matrices for class C_i and containing classes.
MA^i : method affinity matrix of C_i.
CA^i : clustered affinity matrix of 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($\mathcal{C}_i^{des}$, $\mathcal{C}_i^{cont}$), $MU(\mathcal{C}_i^{des}, \mathcal{C}_i^{cont})$) (1)
end {Vert_CA_SM}

Figure 4.10: Vertical Fragmentation – Complex Attributes and Simple Methods

Algorithm 4.8 *(icnm - Generates a set of intra class null methods for class C_i).*

Algorithm icnm
input: C_i: the database class; $\mathcal{M}_k^{cd}$: 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($\mathcal{C}_i^{des}$) : set of extended methods of C_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 $C_k \in \mathcal{C}_d$ **do** (2)
    For every method, $M^{k.n} \in$ EM-set($\mathcal{C}_i^{des}$)) 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 4.11: The Intra Class Null Method Generator

generates the set of methods (complex methods) of other classes using methods of this class C_i. Thus, in vertically fragmenting this class model, we first generate the method affinity matrix of the class that also accounts for the use of its methods by its descendant classes, then we modify the method affinity matrix to include use of the class's methods by other complex methods of other classes. This process uses the method usage and application frequency matrices of all descendants of the class as well as all classes whose complex methods 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 descendants as defined in algorithm *UsageMtrx* of Figure 4.1.

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 4.2. 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 $M^{j.n}$ represented in icnm(C_i), then there is a link created from class C_i to class C_j in the linkgraph as ($C_i \rightarrow C_j$). The linkgraph algorithm is as given in Figure 4.12. 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 4.9. This algorithm includes the complex method relationships 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 4.9).

P3. Use the Bond Energy Algorithm developed by [33] as presented in [34, 38] 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 4.3. 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 4.3). Next the *Partition* algorithm [34, 38] takes the clustered affinity matrix and the modified method usage matrix and produces method fragments.

Algorithm 4.9 *(CMLinkgraph - generates a Link tree rooted at C_i linking it to its complex method classes)*

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

Figure 4.12: The Linkgraph Generator for Complex Method Classes

P5. Use method-attribute reference information of the methods in each method fragment (MAR of Definition 4.1) to include in each method fragment all attributes of the class accessed by methods of the fragment.

P6. Since attributes may be in more than one fragment, use Attribute Placement Affinity Rule 1 to decide in which vertical fragment to place them.

The formal algorithm for vertically fragmenting class models consisting of simple attributes and complex methods (algorithm *Ver_SA_CM*) is given in Figure 4.13. This algorithm takes as its inputs a set of user queries, class hierarchy of the database, a class in the hierarchy, 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.

4.4 Complex Attributes and Complex Methods

This section presents an algorithm for vertically fragmenting classes consisting of complex attributes and complex methods. The database information that needs to be captured include: the inheritance hierarchy, the attribute link to reflect the part-of hierarchy and the method links to reflect the use of methods of objects of

Algorithm 4.10 *(Vertical Fragments of simple attributes and Complex Methods)*

Algorithm Vert_SA_CM

input: $\mathcal{Q}^{C_i^{des}}$: set of user queries accessing C_i and its descendant classes.
C_i : the database class to fragment
L(C) : the class hierarchy
$\mathcal{C}_i^{des}$: set of descendant classes of C_i
icnm(C_i) : set of intra class null methods of C_i.
$\mathcal{C}_i^{cmeth}$: set of complex method classes of C_i
$(EM)^{C_i}$: extended method set of C_i.
MAR-set(C_i) : method attribute reference set of methods of C_i
AF-set($\mathcal{C}_i^{des}$): application frequency matrices of C_i and descendants.
AF-set($\mathcal{C}_i^{cmeth}$): application frequency matrices of C_i
and 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($\mathcal{C}_i^{cont}$) : 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($\mathcal{C}_i^{des}, \mathcal{C}_i^{cmeth}$), $MU(\mathcal{C}_i^{des}, \mathcal{C}_i^{cmeth})$) (1)

end; {Vert_SA_CM}

Figure 4.13: Vertical Fragmentation – Simple Attributes and Complex Methods

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 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 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 class C_i, use of C_i's methods through the intra class null methods of the class from all classes in the object base generated with algorithm *icnm* of Figure 4.11. 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 descendants.

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

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

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

Steps

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

- **a.** Obtain the method usage and application frequency matrices of the class and its descendants as defined in algorithm *UsageMtrx* of Figure 4.1.
- **b.** Define the method affinity matrix of the class from step (R1a) using the usage matrices and application frequency matrices of the class as defined in algorithm *MAMtrx* of Figure 4.2. By doing this, we are producing an initial method affinity matrix of the class that includes use of its methods by methods of its descendants.

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

- **a.** We repeat the operations in step R1 above, using a different type of relationship. Obtain the method usage and application frequency matrices of the class and its containing classes using the algorithm shown in Figure 4.1 and using the Linkgraph from Figure 4.8 that returns a tree rooted at the class showing the attribute link between that class and other classes in the database.
- **b.** Modify method affinity matrix of the class from step (R1b) using the usage matrices and application frequency matrices of the class and its containing classes (from R2a). The algorithm for this process is given in Figure 4.9.

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

a. We repeat the operations in step R2 above, using a different type of relationship – the complex method link. We first produce a link graph which is a tree rooted at the class being fragmented C_i, that links it to all other classes in the object base whose complex methods are represented in the intra class null method set of this class. The linkgraph algorithm *CMLinkgraph* is as given in Figure 4.12. 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 (R2b) using the usage matrices and application frequency matrices of the class and its complex method classes (from R3a). The algorithm *CMAMtrx* for this process is given in Figure 4.9.

R4. Use the Bond Energy Algorithm developed by [33] as presented in [34, 38] to generate clustered affinity matrix of the class.

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

a. Generate a modified method usage matrix of the class as described in the algorithm *MUsageMrtx* of Figure 4.3 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 4.3 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 4.3 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.

R6. Use method-attribute reference information of the methods in each method fragment (MAR of definition 4.1) to include in each method fragment all attributes of the class accessed by methods of the fragment.

R7. Since there may be problems of overlapping attributes in more than one fragment, use Attribute Placement Affinity Rule 4.1 to decide which vertical fragment to keep each overlapping attribute.

The formal algorithm *Vert_CA_CM* is given in Figure 4.14. This algorithm takes as its inputs a set of user queries, class hierarchy of the database, a class in the hierarchy, 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.

4.4.1 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 4.15. The database schema information consists of the class hierarchy of the object base and is given in Figure 3.14, while the class composition hierarchy is as in Figure 3.15.

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 4.16. The method affinity value of a method pair in the matrix is computed as follows:

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

$= (40+0+20) + (0+60+10) = 130$ (initial MA).

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

$ccaff_{prof}(M^{student.1}, M^{student.2})$.

$= [(50 + 15 + 0) + (25 + 40 + 5)] + 0 = 135$

MA($M^{student.1}$,$M^{student.2}$) $= 130 + 135 = 265$ (after first modification)

The method affinity, clustered affinity and the modified method usage matrices of the class after executing line 1 of the algorithm 4.14 are given in Figure 4.17.

The vertical fragments from the execution of the partition algorithm are given below.

Algorithm 4.11 *(Vertical Fragments of Complex attributes and Complex Methods)*

Algorithm Vert_CA_CM

input: $\mathcal{Q}^{\mathcal{C}_i^{des}}$: set of user queries accessing C_i and its descendant classes.
C_i : the database class to fragment
L(C) : the class hierarchy
$\mathcal{C}_i^{des}$: set of descendant classes of C_i
icnm(C_i) : set of intra class null methods of C_i.
$\mathcal{C}_i^{cont}$: set of containing classes of C_i
$\mathcal{C}_i^{cmeth}$: set of complex method classes of C_i
$(EM)^{C_i}$: extended method set of C_i.
MAR-set(C_i) : method attribute reference set of methods of C_i
AF-set($\mathcal{C}_i^{des}$): application frequency matrices of C_i and descendants.
AF-set($\mathcal{C}_i^{cont}$): application frequency matrices of C_i and containing classes.
AF-set($\mathcal{C}_i^{cmeth}$): 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($\mathcal{C}_i^{des}$) : method usage matrices for class C_i and its descendants.
MU-set($\mathcal{C}_i^{cont}$) : method usage matrices for C_i and containing classes.
MU-set($\mathcal{C}_i^{cmeth}$) : method usage matrices for C_i and complex method classes.
MA^i : method affinity matrix of C_i.
CA^i : clustered affinity matrix of 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. //

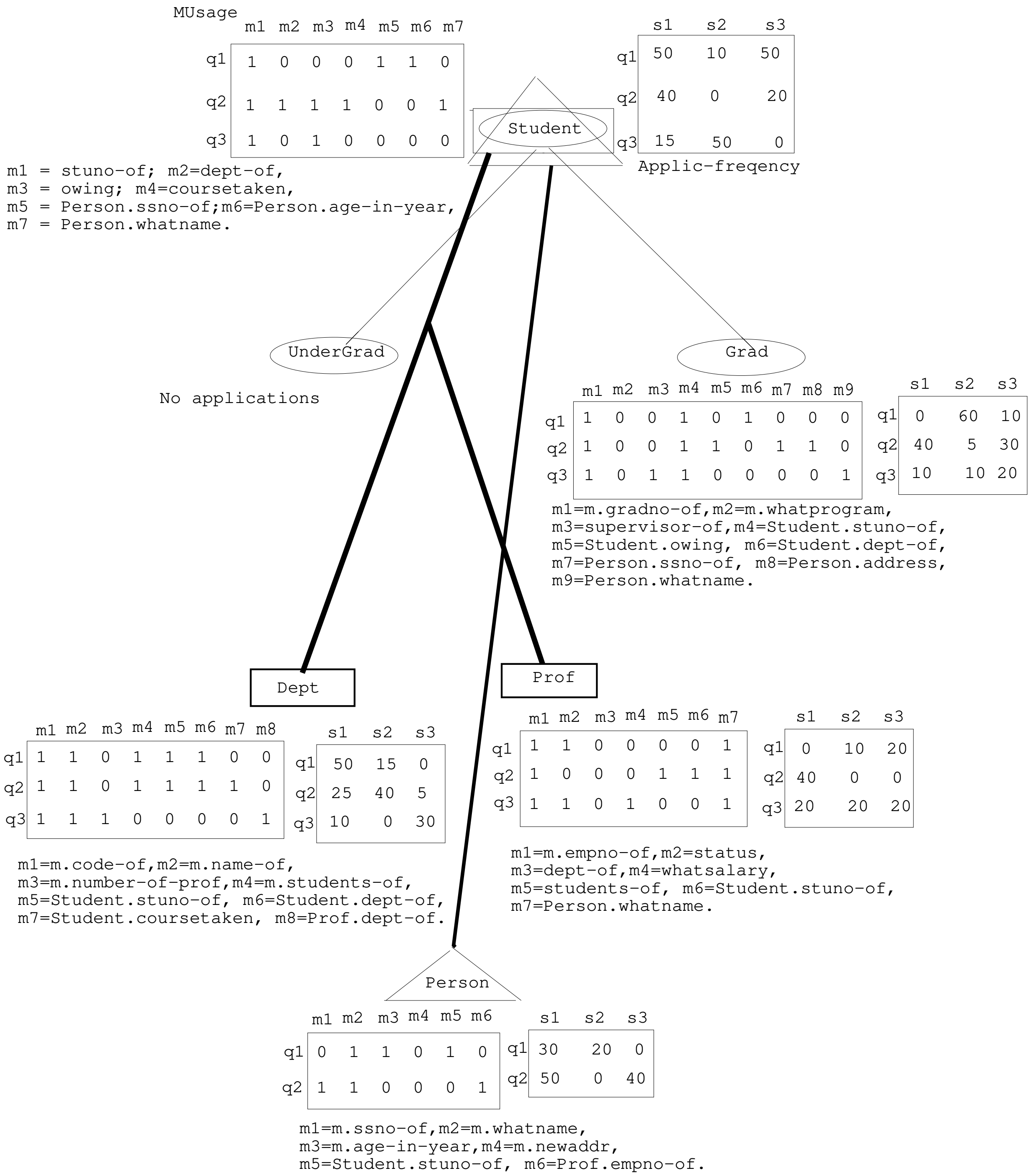
VerticalFrag(MA($\mathcal{C}_i^{des}, \mathcal{C}_i^{cont}, \mathcal{C}_i^{cmeth}$), $MU(\mathcal{C}_i^{des}, \mathcal{C}_i^{cont}, \mathcal{C}_i^{cmeth})$) (1)

end {Vert_CA_CM}

Figure 4.14: Vertical Fragmentation - Complex Attributes and Complex Methods

Person = {Person,{a.ssno,a.name,a.age,a.address},
{m.ssno-of,m.whatname,m.age-in-year,m.newaddr},
{ I_1 {Person1,John James,30,Winnipeg}
I_2 {Person2,Ted Man,16,Winnipeg}
I_3 {Person3,Mary Ross,21,Vancouver}
I_4 {Person4,Peter Eye,23,Toronto}
I_5 {Person5,Mary Smith,40,Toronto}
I_6 {Person6,John West,32,Vancouver}
I_7 {Person7,Jacky Brown,35,Winnipeg}
I_8 {Person8,Sean Dam,27,Toronto}
I_9 {Person9,Bill Jeans,43,Vancouver}
I_{10} {Person10,Mandu Nom,30,Winnipeg} } }
Prof = Person pointer $\odot$ {Prof,{a.empno,a.status,a.dept,a.salary,a.student},
{m.empno-of,m.status-of,m.students-of,m.whatsalary,m.dept-of},
{ I_1 (person pointer5) $\odot$
{Prof1,asst prof,Computer Sc.,45000,students pointers}
I_2 (person pointer6) $\odot$ {Prof2,assoc prof,Math,60000,students pointers}
I_3 (person pointer9) $\odot$ {Prof3,full prof,Math,80000,students pointers}
I_4 (person pointer10) $\odot$ {Prof4,full prof,Math,82000,students pointers} } }
Student = Person pointer $\odot$ {Student,{a.stuno,a.dept,a.feespd,a.coursetaken},
{m.stuno-of,m.dept-of,m.owing,m.course-taken}
{ I_1 (person pointer1) $\odot$ {Student1,Math,Y,[]}
I_2 (person pointer4) $\odot$ {Student2,Computer Sc.,N,[521,632]}
I_3 (person pointer2) $\odot$ {Student3,Stats,Y,[211]}
I_4 (person pointer3) $\odot$ {Student4,Computer Sc.,N,[111,211]}
I_5 (person pointer7) $\odot$ {Student5,Math,Y,[321,333]}
I_6 (person pointer8) $\odot$ {Student6,Stats,N,[111,211]} } } }
Grad = Student pointer $\odot$ {Grad,{a.gradstuno,a.supervisor},
{m.gradno-of,m.whatprog}
{ I_1 (Student pointer1) $\odot$ {Grad1,John West}
I_2 (Student Pointer2) $\odot$ {Grad2,Mary Smith}
I_3 (Student Pointer5) $\odot$ {Grad3,Mary Smith} } }
UnderG = Student
I_1 (Student pointer3), I_2 (Student pointer4), I_3 (Student pointer6)
Dept = {Dept,{a.code,a.name,a.profs,a.students},
{m.code-of,m.name-of,m.number-of-profs,m.students-of},
{ I_1 {Dept1,{Computer Science,prof pointers,student pointers}
I_2 {Dept2,{Math,prof pointers,student pointers}
I_3 {Dept3,{Actuary Science,prof pointers,student}
I_4 {Dept4,{Stats,prof pointers,student pointers} } }

Figure 4.15: The Second Complex Sample Object Database Schema



Note: methods preceded by m are methods of that particular class. Person is a superclass of the class Student. Undergrad and Grad are subclasses of Student while Dept and Prof are its containing classes; and class Person is its complex method class..

Figure 4.16: Method Usage/Application Frequencies for Student's Subclasses, Containing and Complexmethod Classes

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

(a) Method Affinity matrix

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

(b) Clustered Method Affinity matrix

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

(c) Modified Method Usage matrix

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

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

Figure 4.17: Method/Clustered Affinity and Modified Method Usage Matrices

$F_1 = \{m_1, m_3, m_4\}$

$F_2 = \{m_1, m_2, m_5, m_6, m_7\}$.

4.5 Complexities of Vertical Fragmentation Algorithms

The computation times of the vertical fragmentation algorithms for all four class models are based on that for the simple model consisting of simple attributes and methods. Assume f is the maximum number of fragments in a class, m is the maximum number of methods in a class, a is the maximum number of attributes in a class, q is the number of applications accessing a database class, while c is the number of classes in the database. The vertical fragmentation algorithm for the class model consisting of simple attributes and methods *VerticalFrag* of Figure 4.4 is of time complexity $O(cqm + m^2 + m^2 + cqm + m^2 + fma + af)$ which simplifies to $O(cqm + m^2 + fma)$. This is because line 1 of this algorithm *VerticalFrag* which is *UsageMtrx* algorithm is of O(cqm), and generation of method affinity matrix (line 2) has computation time of $O(m^2)$. The Bond Energy algorithm (line 3) has a computation time of $O(m^2)$, and modifying method usage matrices (line 4) has the same computation time of O(cqm) as generating original method usage matrices. Lines 6 though 9 used for attribute inclusion are of O(fma) while lines 10 through 13 for ensuring disjointness have time complexity O(af). The prevailing time complexity of the algorithm depends on the four parameters, number of classes, number of applications accessing a class, number of methods in a class and number of attributes in a class. However, since every method is expected to be accessing some attribute, then the number of attributes will be approximately the same as the number of methods allowing the term *fma* subsume m^2. If the number of classes and applications are much more than the number of methods, the complexity of *VerticalFrag* is O(cqm).

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

Chapter 5

Analysis

The objective of this chapter is to evaluate the performance of our fragmentation schemes by showing that (1) our fragmentation schemes are correct because they produce fragments that are complete, disjoint and reconstructible; (2) our schemes are good because they produce the lowest overall penalty access costs due to remote relevant and local irrelevant access to data with respect to applications when compared with two other schemes; and (3) our fragmentation schemes are practicable since some important components are already implemented while the rest can easily be implemented. Section 5.1 demonstrates the correctness of our fragmentation schemes by showing that the schemes are complete, disjoint and reconstructible. Section 5.2 measures the performance of our fragmentation scheme by comparing the total cost of remote relevant accesses, local irrelevant accesses and total penalty access costs (remote relevant and local irrelevant costs) made by all applications to our fragments, with access costs made to fragments generated using two other approaches when the same set of applications run on these different fragments. Section 5.3 finally demonstrates the practicality of this distributed database design technique by discussing the implementation status of the evolving testbed prototype.

5.1 Correctness of Our Fragmentation Schemes

It is imperative that fragments produced by an algorithm are not only good, but also correct. In this section, we check the fragmentation algorithms with respect to the three correctness criteria of completeness, reconstruction and disjointness.

5.1.1 Horizontal Fragmentation

Showing that our horizontal fragmentations are correct entails showing that they are (1) complete (2) reconstructible and (3) disjoint.

Completeness:

Our fragmentation scheme generates a set of primary fragments for every class and a set of derived fragments for every member class of a link in the object base. The basis of the primary horizontal fragmentation is the selection predicates from user queries. If the set of selection predicates used is complete, then every instance object is accounted for in some primary fragment.

Since our algorithms use a set of complete and minimal predicates generated with COM-MIN, then completeness of the primary horizontal fragmentation is guaranteed. A derived horizontal fragmentation is complete if every instance object in every primary fragment of an owner class is linked to an instance object in its member class. For example, completeness of derived fragmentation requires that every instance object in a subclass or contained class be linked to an instance object in a superclass or containing class. Since our LinkGraph and object join operations guarantee this condition, our derived horizontal fragmentation is complete.

Reconstruction

Since the primary fragments are generated from a set of complete and minimal predicates, and the final horizontal fragments of a class are the union of its set of primary and derived fragments, then for a class with fragmentation,

$F_c = \{F_1^h, F_2^h, \ldots, F_w^h\}$

$F_c = \bigcup F_i^h$ for all i = 1 ... w,

where reconstruction of a global class from its fragments is performed by the union operator.

Disjointness

COM-MIN generates the set of predicates used for primary fragmentation is minimal and thus generates non-overlapping primary fragments. The derived fragments may however introduce some overlapping if for example, one member class is linked to two or more owner classes, or an instance object of a member class is linked with more than one instance object of the owner class (this may be the case with complex method uses). Our algorithm uses affinity measure information to decide on which one fragment it is most profitable to keep every overlapping instance object and this is taken care of in the algorithm *HorizontalMember* and there is no overlap.

5.1.2 Vertical Fragmentation

The arguments here are similar to those for horizontal fragmentation.

Completeness

If all the methods of the class are used in defining the method affinity matrix, the Bond Energy algorithm will only cluster methods by re-arranging rows and columns of this matrix. Next, the *PARTITION* algorithm will assign each of the methods to one fragment. Thus, this guarantees that the method fragments are complete. Then, attributes are included in the method fragments to belong in the same fragment as the methods using them. Since every attribute in a class is accessed by some method in the class, this process guarantees that every method and attribute in a class belongs to some vertical fragment of the class.

Reconstruction

The reconstruction of the original global class is achieved with a vertical fragment integrator ($\bigodot F_i$) operation. Thus, the integrator takes a number of fragments of a class as its input and returns one fragment containing all the attributes and methods in each of the vertical fragments. Therefore, for a class C with vertical fragmentation,

$F_c = \{F_1^v, F_2^v, \ldots, F_w^n\}$

$C = \bigodot F_i, \forall F_i \in F_c$

This means that if each F_i is complete, the fragment integrator operation will properly reconstruct C. Note that each F_i should contain the class identifier and its method.

Disjointness

Fragments are disjoint except for the object identifier and its methods, and this is guaranteed by the *PARTITION* algorithm.

5.2 Goodness of Our Fragmentation Schemes

The major contribution of this thesis includes incorporating the inheritance and aggregation hierarchy as well as method nesting information into the process of fragmentation. For simplicity, with vertical partitioning, we restrict the number of partitions to be generated from a class to two. If the partition sizes are big, we can apply further binary partitioning algorithm iteratively on each vertical fragment. An ideal environment for this performance analysis requires a distributed object oriented system that supports distribution of data entity and running on either local area or wide area network. A performance measure can be obtained by gathering the cost of local and remote accesses to data on the network by all the applications over a period of time using our fragments. This is also run on different sets of applications, database schemas and instance objects. The same performance measure is obtained over the same period of time, the same sets of applications, database schemas and instance objects but with fragments from schemes that do not account for inheritance, aggregation hierarchy and method nesting. Since there are no existing real life systems to obtain test applications, database and instance objects from, for this analysis, we simulate an ideal environment with a set of synthetic data and applications. Our test data are obtained by defining an application domain, e.g., a university academic database system, and defining possible classes, attributes and methods in this domain. Then, we define example applications running on these classes. From these applications, the method usage and application frequency matrices are specified. Inheritance, attribute and method nesting relationships among classes are also specified. We compare our results with fragments generated with the horizontal and vertical fragmentation algorithms [7, 34] for relations. Efforts to obtain a performance analysis of our distributed database design algorithms has led to a vertical fragmentation testbed prototype with structure and components as shown in Figure 5.1.

The testbed prototype figure has two major parts (a and b). Part b gives details of what constitutes each component of Part a for all the fragmentation schemes to be evaluated. For example, input to our fragmentation algorithms are listed in Part b as extended method affinity matrix (EMAM), method usage matrix (MUM) and modified method usage matrix (MMU); while input to the alternative fragmentation

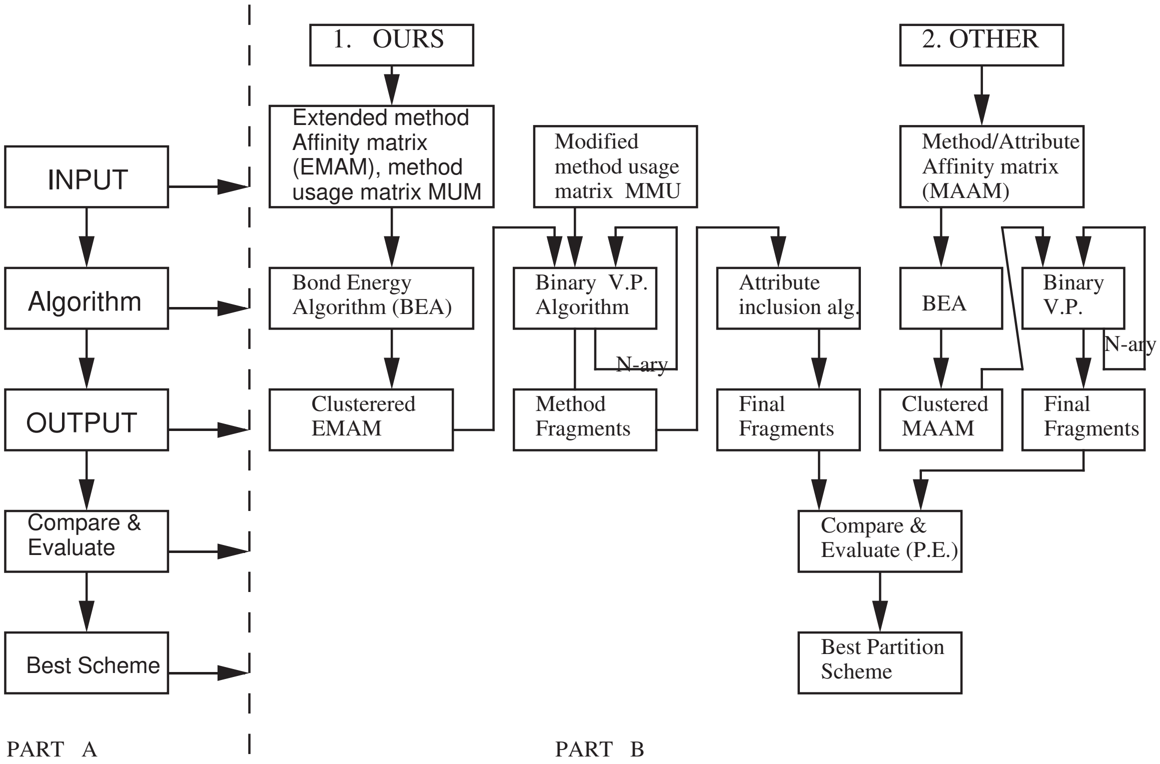


Figure 5.1: Vertical Fragmentation Testbed Prototype

algorithm is listed in Part b as method/attribute affinity matrix (MAAM).

We use the Partition Evaluator (P.E.) proposed by Chakravarthy *et al.* [10] to measure the costs of local and remote accesses incurred by the fragments from the two schemes. The objective of the partition evaluator is to measure the total costs of processing all applications at all distributed sites if each access to a data fragment amounts to a unit cost. Every application processing cost consists of its local processing and remote processing costs. It is assumed that there are no data redundancies and following fragmentation, the fragments have been allocated to sites where they are most needed. The P.E. has two components, namely:

1. *Irrelevant Local Method Access Cost* which minimizes the square error for a fixed number of fragments and assigns a penalty factor whenever irrelevant methods are accessed in a particular fragment. The mean vector V_i for fragment i is given by:
   $V_i = (1/n_i)\sum_{j=1}^{n_i} M_{ij}$
   where n_i is the number of methods in fragment i and M_{ij} is the method vector for the *jth* method in fragment i. This is the M_{ij}th column in the application method frequency matrix. The application method frequency matrix consists of the methods in a class as columns and the applications (q) as rows with the frequency of access to the methods for each application as the values in the matrix. The mean vector computes an average access pattern of the applications over all methods of fragment i. The difference vector for method vector M_{ij} is $(M_{ij} - V_i)$ and the square-error of the fragment i is the sum of the squares of the lengths of the difference vectors of all the methods in fragment i. This is given by:
   $e_i^2 = \sum_{j=1}^{n_i}(M_{ij} - V_i)^T(M_{ij} - V_i)$

where T is the total number of applications. The square error for all m fragments generated is:

$E_m^2 = \sum_{i=1}^{m} e_i^2$.

This equation simplifies to the following:

$E_m^2 = \sum_{i=1}^{m} \sum_{t=1}^{T} [q_t^2 * |S_{it}| * (1 - |S_{it}|/n_i^r])$

where S_{it} is the set of methods contained in fragment i that the transaction t accesses. It is empty if t does not need fragment i. A lower value of E_m^2 means a lower penalty cost of irrelevant local method access.

2. *Relevant Remote Method Access cost*: This second component of the P.E. computes for a set of applications running on a fragment, the ratio of the number of remote methods to be accessed to the total number of methods in each of the remote fragments. This is summed over all the fragments and applications to obtain: $E_R^2 = \sum_{t=1}^{T} minimum(\sum_{k \neq i} [q_t^2 * |R_{itk}| * |R_{itk}|/n_{itk}^r])$
where q_t is the frequency of application t for t = 1,2,...,T. R_{itk} is the set of relevant methods of fragment k remotely needed by application t while running on fragment i and $|R_{itk}|$ is its cardinality. n_{itk}^r is the total number of methods in remote fragment k accessed by application t while running on fragment i. Thus, $PE = E_m^2 + E_R^2$.

For simplicity, we consider only two fragments for each partitioning scheme. To show the performance of our vertical fragmentation scheme, we generate nine test cases with each test case involving:

1. Method Usage and Application Frequency matrices :

a. a method usage and an application frequency matrices for the class fragmented which include inheritance, aggregation, and method nesting information.

b. a method usage and an application frequency matrices for the class not including inheritance, aggregation and method nesting information.

2. **Modified Method Usage Matrix** :

   a. A modified method usage matrix of the class that includes usages by descendant classes, containing classes and complex method classes.

   b. A method usage matrix not including these dependencies.

3. **Generate Vertical Fragments** :

   a. Using the matrices in 1a and 2a above, we generate fragments with our fragmentation algorithms.

   b. Similarly, using matrices from 1b and 2b, we generate fragments using earlier approach that does not capture object oriented features [34].

4. **Generate Application Method Frequency Matrix (AMF)** :

   a. Convert the information in the two matrices in 1a above to generate application method frequency matrix. This is converted by multiplying each method usage value on the method usage matrix for an application q_i by the length of this application q_i vector on the application frequency matrix such that for example, the following two matrices:

	method usage matrix				application frequency		
	M_1	M_2	M_3	M_4	S_1	S_2	S_3
q_1	1	0	1	0	10	5	5
q_2	1	1	0	1	20	15	0
q_3	0	1	1	0	30	10	0

would convert to the following application method frequency matrix.

application method frequency matrix

	M_1	M_2	M_3	M_4
q_1	20	0	20	0
q_2	35	35	0	35
q_3	0	40	40	0

Thus, $\overline{q_i}$ in application method frequency matrix is similar to $|\overline{q_i}|$ in application frequency multiplied by the $\overline{q_i}$ in method usage matrix.

b. Generate the AMF matrix for the alternative approach using method usage and application frequency matrices from 1b above.

5. **Compute the PE values** :

   a. Compute the PE value for our fragments using the matrix in 4a and our fragments from 3a.

   b. Similarly, compute the PE value for the alternative set of fragments using the matrix in 4b and fragments from 3b.

6. **Handle all test cases:** Repeat steps 1 to 5 for eight more test cases (detailed in the next section).

7. Plot Graphs to show Results: Plot the set of PE values obtained from step 5a for all nine test cases against the set of PE values obtained from step 5b for the same nine test cases and the set of PE values obtained for the case supporting no fragmentation at all. Recall that lower PE value means better performance. Also plot graphs for the local irrelevant and remote relevant access costs of all three cases.

5.2.1 The Vertical Fragmentation Test Experiment

Using this objective function that measures the processing costs of applications running at two local sites where two different fragments generated by a fragmentation scheme are presumed allocated, we conducted an experiment that compared the processing costs of the two fragments from our vertical fragmentation algorithm with those from the general vertical fragmentation algorithm [34] that does not incorporate the relationships of inheritance, class aggregation and method nesting necessary in object oriented data model. We compared the costs incurred by these two schemes with those incurred by a scheme supporting no fragmentation at all, but which replicates all object methods and fragments at the two sites. The two component costs of local irrelevant access and remote relevant access costs were also gathered and compared for these three fragmentation approaches. The experiment was run on nine test cases which modify the data to reflect the following:

c1. Increasing the number of methods of this class accessed by methods of other classes through inheritance, aggregation, and method nesting relationships without changing the frequency of usage.

c2. Keeping the number of the methods accessed constant but increasing the frequency of usage of these methods by increasing the application access frequency values of the rows of the application method frequency matrix representing accesses by other classes.

c3. Changing the access pattern of methods of this class by other classes to show some variance from the original access pattern of these methods directly by applications.

c4. Other changes include increasing the application access frequency values of the rows of the application method frequency matrix which represent access by applications directly and not through other classes.

The experiment was conducted primarily with three test data, each modified later to reflect any or a combination of the changes listed above as c1 to c4. The first three test cases constitute the three main test data while the rest represent a modification of one of these three main cases. The nine test cases are described below:

T1: The first test case consists of a class with nine methods, eight original applications accessing these methods directly, and six additional applications accessing these methods indirectly through other classes.

T2: The second test case consists of a class with only four methods and three original applications accessing these methods directly, while three additional applications access these methods indirectly through other classes.

T3: The third test case consists of a class with seven methods, three original applications accessing these methods, and nine additional applications accessing these methods through other classes.

T4: This is a modification of T1 according to c1 because the number of methods accessed by other classes is increased.

T5: This modifies T1 according to c1 and c3 by both increasing the number of methods accessed by other classes and changing the pattern of access to these methods to be different from the original access pattern of these methods.

T6: This modifies T2 according to c3 by changing the application access pattern.

T7: This test case is a modification of T3 according to c2 by slightly increasing application access values of a few rows of the application method frequency matrix, and c3 by slightly changing their access pattern.

T8: This is a modification of T3 according to c2, c3 and c4. T7 is modified by also increasing the application access frequencies of applications directly on these methods.

T9: This modifies T1 according to c4 by increasing the application access frequency of original applications on the methods of this class.

The nine test cases are given in Figure 5.2, Figure 5.3 and Figure 5.4.

The next step entails generating two fragments from each of these test cases using (1) our approach that includes indirect access through other classes, (2) the general vertical fragmentation approach that considers only applications access directly on the methods as in [34], and (3) the approach supporting no fragmentation at all which has the complete set of methods in a fragment replicated at the two sites. Generating fragments from our approach entails using the modified method usage matrix and application access frequency which account for accesses through other classes. Generating fragments from the alternative approach entails using the original method usage matrix of the class and only the application frequency matrix of the class. The fragments generated by both our approach and the alternative approach for the nine cases are given as Figure 5.5. To compare the performance of

Applic\Methods

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
q1	25	0	0	0	25	0	25	0	0	0
q2	0	50	50	0	0	0	0	50	50	0
q3	0	0	0	25	0	25	0	0	0	25
q4	0	35	0	0	0	0	35	35	0	0
q5	25	25	25	0	25	0	25	25	25	0
q6	25	0	0	0	25	0	0	0	0	0
q7	0	0	25	0	0	0	0	0	25	0
q8	0	0	15	15	0	15	0	0	15	15
q9	0	0	0	0	25	0	25	0	0	0
q10	0	0	0	20	0	0	0	0	0	20
q11	0	0	0	0	0	0	50	50	0	50
q12	75	75	75	0	0	0	75	75	0	0
q13	0	0	0	0	0	0	0	5	0	0
q14	0	55	55	0	0	55	0	0	0	0

TEST 1

Applic\Methods

	m1	m2	m3	m4
q1	20	0	20	0
q2	35	35	0	35
q3	0	40	40	0
q4	0	0	30	0
q5	25	0	25	0
q6	0	25	0	0

TEST 2

Applic\Methods

	m1	m2	m3	m4	m5	m6	m7
q1	65	0	0	0	65	65	0
q2	60	60	60	60	0	0	60
q3	65	0	65	0	0	0	0
q4	70	70	0	0	0	0	0
q5	75	0	75	0	75	0	0
q6	40	0	0	0	0	0	40
q7	65	65	0	0	0	0	0
q8	70	70	0	70	0	0	0
q9	0	0	0	0	0	0	30
q10	40	0	0	0	0	0	40
q11	0	0	0	0	0	0	60
q12	50	0	0	0	0	0	0

TEST 3

Figure 5.2: Test Cases 1 To 3

Applic\Methods	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
q1	25	0	0	0	25	0	25	0	0	0
q2	0	50	50	0	50	0	0	50	50	0
q3	0	0	0	25	0	25	0	0	0	25
q4	0	35	0	0	0	0	35	35	0	0
q5	25	25	25	0	25	0	25	25	25	0
q6	25	0	0	0	25	0	0	0	0	0
q7	0	25	25	0	0	25	0	25	25	0
q8	0	0	15	15	0	15	0	0	15	15
q9	0	0	0	0	25	0	25	0	0	0
q10	20	20	0	0	0	0	0	20	0	20
q11	0	0	0	50	50	0	50	50	0	50
q12	75	75	75	0	0	0	75	75	0	0
q13	5	0	5	0	5	0	0	5	0	0
q14	55	55	55	0	0	55	0	0	0	0

TEST 4

Applic\Methods	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
q1	25	0	0	0	25	0	25	0	0	0
q2	0	50	50	0	50	0	0	50	50	0
q3	0	0	0	25	0	25	0	0	0	25
q4	0	35	0	0	0	0	35	35	0	0
q5	25	25	25	0	25	0	25	25	25	0
q6	25	0	0	0	25	0	0	0	0	0
q7	0	25	25	0	0	25	0	25	25	0
q8	0	0	15	15	0	15	0	0	15	15
q9	0	0	0	0	25	0	25	0	0	0
q10	0	20	20	0	0	0	0	20	0	20
q11	0	0	0	50	0	0	0	50	0	50
q12	75	0	75	0	0	0	0	0	0	0
q13	5	0	5	0	5	0	0	5	0	0
q14	55	55	55	0	0	0	0	0	0	0

TEST 5

Applic\Methods	m1	m2	m3	m4
q1	20	0	20	0
q2	35	35	0	35
q3	0	40	40	0
q4	0	0	0	30
q5	25	0	25	25
q6	20	0	0	20

TEST 6

Figure 5.3: Test Cases 4 To 6

Applic\Methods

	m1	m2	m3	m4	m5	m6	m7
q1	65	0	0	0	65	65	0
q2	60	60	60	60	0	0	60
q3	65	0	65	0	0	0	0
q4	0	90	90	0	90	0	0
q5	0	0	75	0	75	0	75
q6	70	0	0	0	0	0	70
q7	85	85	0	0	0	0	0
q8	70	70	0	70	0	0	0
q9	0	0	0	0	0	0	50
q10	40	0	0	0	0	0	40
q11	0	0	0	0	0	0	60
q12	50	0	0	0	0	0	0

TEST 7

Applic\Methods

	m1	m2	m3	m4	m5	m6	m7
q1	110	0	0	0	110	110	0
q2	60	60	60	60	0	0	60
q3	65	0	65	0	0	0	0
q4	90	90	0	0	0	0	0
q5	75	0	75	0	75	0	0
q6	40	0	0	0	0	0	40
q7	65	65	0	0	0	0	0
q8	70	70	0	70	0	0	0
q9	0	0	0	0	30	0	30
q10	40	0	0	0	0	0	40
q11	0	0	0	0	0	0	60
q12	50	0	0	0	0	0	0

TEST 8

Applic\Methods

	m1	m2	m3	m4	m5	m6	m7	m8	m9	m10
q1	25	0	0	0	25	0	25	0	0	0
q2	0	50	50	0	0	0	0	50	50	0
q3	0	0	0	25	0	25	0	0	0	25
q4	0	35	0	0	0	0	35	35	0	0
q5	25	25	25	0	25	0	25	25	25	0
q6	25	0	0	0	25	0	0	0	0	0
q7	0	0	25	0	0	0	0	0	25	0
q8	0	0	15	15	0	15	0	0	15	15
q9	0	0	0	0	25	0	25	0	0	0
q10	0	0	0	20	0	0	0	0	0	20
q11	0	0	0	0	0	0	50	50	0	50
q12	75	75	75	0	0	0	75	75	0	0
q13	0	0	0	0	0	0	0	5	0	0
q14	0	55	55	0	0	55	0	0	0	0

TEST 9

Figure 5.4: Test Cases 7 To 9

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

Figure 5.5: Fragments From the Three Approaches

these three approaches for fragmentation, using the application method frequency matrix of the class that reflects all uses of the methods of this class by all applications both directly and indirectly, we compute the partition evaluator values of the three schemes. Each PE value corresponds to the total penalty cost incurred by each scheme through both local irrelevant access costs and remote relevant access costs. A lower PE value means a better performance.

5.2.2 The Results of the Experiment

The comparative local irrelevant access costs, remote relevant access costs and the PE values computed using the objective functions of Chakravarthy *et al.* [10] for the nine test cases are as given in Figure 5.6. The graphs of the costs (represented on the y axis) against the nine test cases (represented on the x-axis) for the three fragmentation approaches are given in Figure 5.7 for local irrelevant costs, Figure 5.8 for remote relevant cost, and Figure 5.9 for total penalty costs.

Note that for many cases, our approach yielded lowest irrelevant processing cost (Figure 5.7) while the approach with no object oriented relationship accounted for, performs better than the approach supporting no fragmentation at all. In addition, since data is replicated with the approach supporting no fragmentation, it incurs the hidden cost (not shown in this analysis) of replication updates. For some cases (5 and 6), the second approach generated lower local irrelevant costs than our approach but much higher remote relevant and total penalty costs. With the Remote Relevant Costs of the three approaches (Figure 5.8), the approach supporting no fragmentation but with replicated data performs best with its constant remote relevant access costs of zero for all cases. Our approach performs better than the approach supporting no object oriented relationships for all test cases. Finally, a

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

Figure 5.6: Costs of Processing Fragments Using Three Approaches

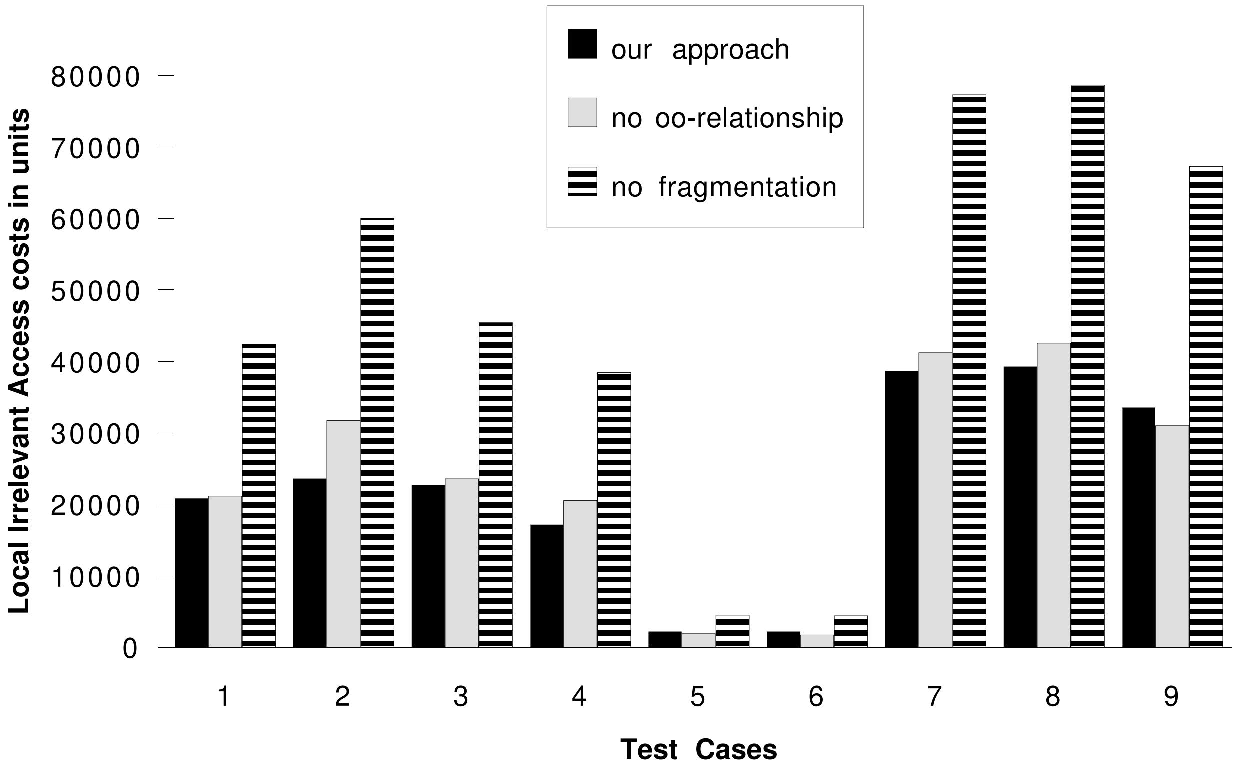


Figure 5.7: Local Irrelevant Costs of Three Approaches

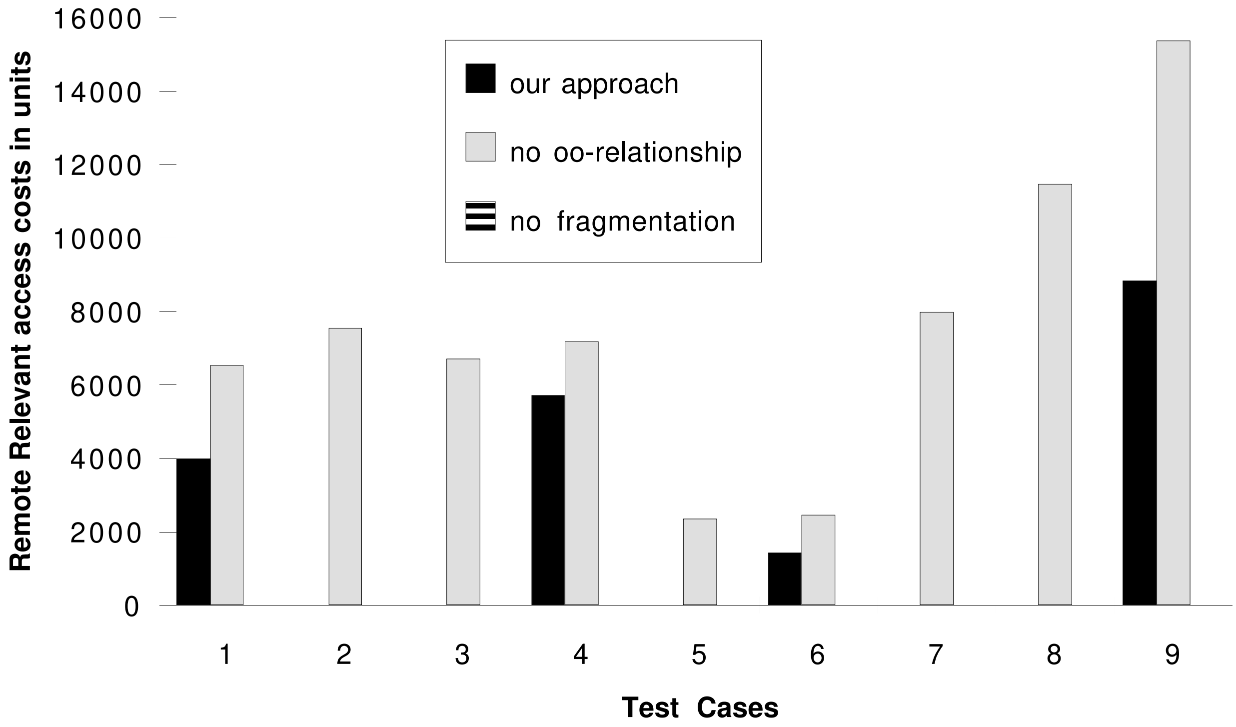


Figure 5.8: Remote Relevant Costs of Three Approaches

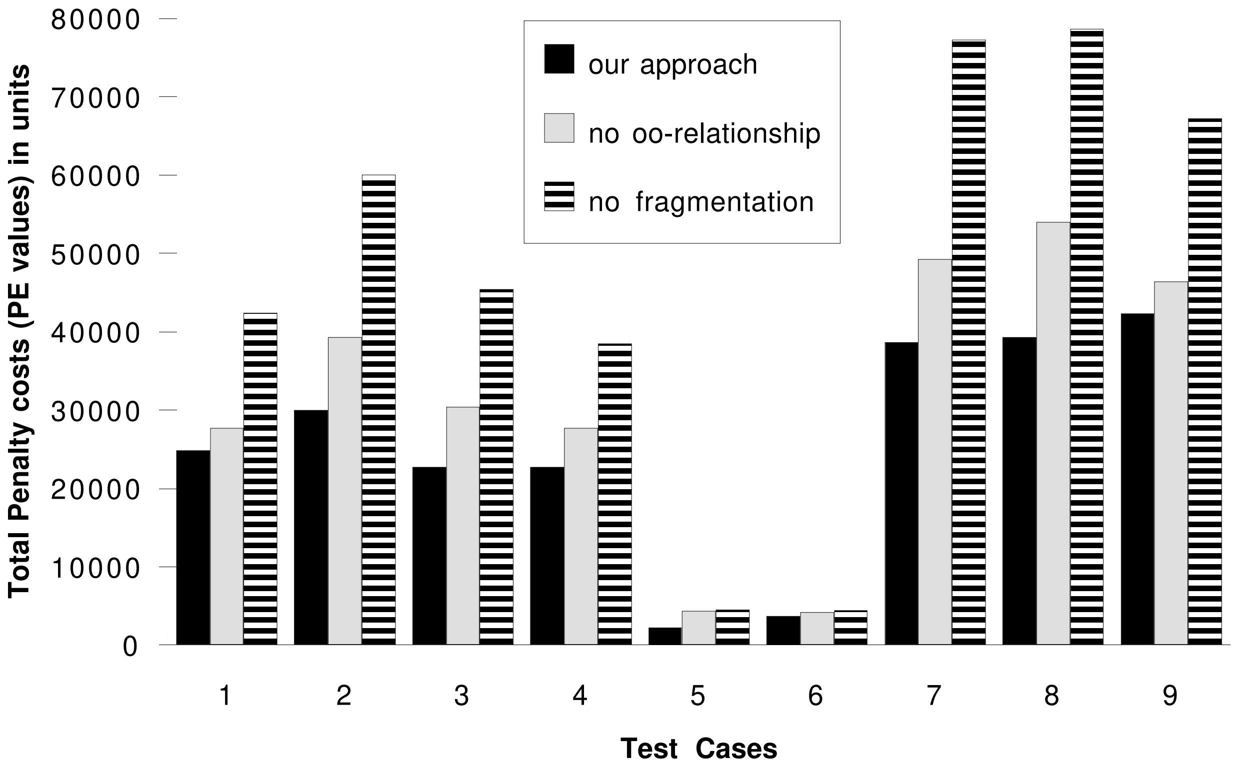


Figure 5.9: Total Penalty Costs of Three Approaches

comparative analysis of the Total Penalty Costs of the three approaches (Figure 5.9) shows that our approach performs best with the overall lowest total processing cost while the approach with no fragmentation performs worst.

5.2.3 Performance Analysis of the Horizontal Fragmentation

The performance of our horizontal fragmentation algorithms can be analyzed in a similar fashion. However, this requires implementation of both our horizontal fragmentation algorithms and those horizontal fragmentation algorithms that account for no object-oriented relationship as discussed in [7]. These implementations will enable fast generation of horizontal fragments by the two alternative approaches. The objective functions developed by Chakravarthy *et al.* [10] for measuring the goodness of the partitions from different approaches are only good for vertical fragmentation. It is possible to modify this function to enable measuring the goodness of the horizontal fragmentation algorithms. This can be done by using an application object frequency matrix in place of the application method frequency matrix used in the vertical fragmentation case, but it is an issue for future research.

5.3 Practicality of Our Fragmentation Algorithms

Efforts to demonstrate the practicality of our fragmentation algorithms have led to the project of developing a distributed database design testbed currently going on here at the Advanced Database Systems laboratory of the University of Manitoba. This testbed has the general structure as given in Figure 5.1 for the vertical fragmentation case. A similar testbed can be organized for the horizontal fragmentation case for future research. For vertical fragmentation testbed, so far, three vertical fragmentation schemes can be compared and evaluated. This has so far involved about three thousand lines of C code and the algorithms already developed from the testbed prototype of Figure 5.1 are as follows:

1. Extended Method Affinity Matrix generator [15, 17].
2. Bond Energy algorithm [33].
3. Modified Method Usage Matrix generator [15, 17].
4. Binary Vertical Partition algorithm [34].
5. Compare and Evaluate module using Partition Evaluator [10].

More algorithms can easily be attached to this design testbed. A quick comparison of the actual execution times of the three fragmentation schemes on any one class shows insignificant difference running on a Sun Sparc station.

Chapter 6

Conclusions

6.1 Summary and Contributions

In this thesis, a comprehensive study of issues in distributed object based design techniques is undertaken. In particular, the thesis involves a detailed study of fragmentation issues in distributed object based systems. This thesis has made a number of important contributions. Early research in this area addressed relations but failed to address the object-oriented data model. The first contribution of the research was to define a classification taxonomy for classes making up a DOBS which enables the accommodation of all the necessary features of object orientation. Classes are classified based on the nature of the attributes and methods they contain. The features of object orientation incorporated in the class model classification include encapsulation, inheritance and class aggregation hierarchies as well as method nesting among classes.

Two main types of fragmentation are commonly performed on database's entities, horizontal and vertical fragmentations. Early research efforts in fragmentation of database entities produced horizontal and vertical fragmentation algorithms for relations, but did not provide fragmentation algorithms for objects. The second major contribution of this thesis is the provision of a comprehensive set of algorithms for horizontally fragmenting the following four class models on our taxonomy:

1. a model consisting of simple attributes and methods,

2. a model consisting of complex attributes and simple methods,

3. a model consisting of simple attributes and complex methods,

4. a model consisting of complex attributes and complex methods.

In horizontally fragmenting class models, the approach adopted in this thesis consists of first generating primary horizontal fragments of a class based on only applications accessing this class, and secondly generating derived horizontal fragments of the class arising from primary fragments of its descendant classes, its complex attributes (contained classes), and/or its complex method classes. Finally, we combine the sets of primary and derived fragments of each class to produce the best possible fragmentation scheme.

The thesis also contributes by providing a comprehensive set of algorithms for vertically fragmenting the four class models on the taxonomy. With vertical fragmentation, the thesis groups into a fragment, all attributes and methods of the class frequently accessed together by applications running on either this class, its descendant classes, its containing classes, or complex method classes. This is achieved by including in the method affinity matrix of the class being fragmented, the usages of this class's methods by applications through its descendant classes, containing and complex method classes. The method usage matrix of the class is also extended to include uses of methods through other classes. This extended method affinity matrix is clustered and partitioned to produce method fragments. Attributes are later included in each method fragment to produce non-overlapping attribute/method fragments.

The thesis makes a significant contribution putting together a testbed prototype suitable for conducting a performance analysis of various fragmentation approaches for distributed object based systems. The evaluation and comparison of the goodness of each fragmentation approach is based on the total processing penalty cost that the system incurs. The two components of the total penalty cost are local irrelevant penalty cost and remote relevant penalty cost. This thesis provided the implementation of many components of the testbed that made possible the comparison of our approach with two other approaches. Our vertical fragmentation algorithms were shown to produce best performance for all the test cases used because they generated the overall lowest penalty processing costs. Both our set of horizontal and vertical fragmentation schemes were proven to be correct because they are complete, reconstructible, and disjoint. Furthermore, the practicality of our fragmentation algorithms were demonstrated. The time complexities of both the horizontal and vertical fragmentation algorithms show them to be polynomial.

This research makes an important contribution to database technology in general and object based systems in particular in the following ways. By using algorithms to generate and allocate optimal fragments to distributed sites, the amount of irrelevant data accessed by applications at distributed sites is reduced. This amounts to some performance gain which becomes even more significant in a system where objects of arbitrary sizes exist. Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base. When there is a change in the locations of applications, fragmentation reduces the amount of data transferred or that needs to migrate. This also amounts to an appreciable gain in transmission cost of both small and large objects. Fragmentation replication is more efficient than replicating the entire class because it reduces the update problem and saves storage.

6.2 Future Research

Although this research has made a number of significant contributions in this area, some open problems still exist. The next few paragraphs will attempt to present some of the open research problems related to this project.

The work described in this thesis is aimed at distributing the global conceptual schema to generate local conceptual schemas which constitutes logical data design. On the contrary, some earlier works exist on object-oriented database design and index selection [2] aimed at designing the local internal schema which is physical data design. Efficient design at both physical and logical levels are important for an efficient distributed database design. The design at one level could positively or negatively affect the performance of the design at the other level. Thus, while the two designs are independent and on different levels, they are complementary. For future research, the impact of varying indexing techniques available at the various local sites on class fragmentation should be investigated.

For this design, fragmentation is performed only statically. Ideally, these fragmentation algorithms can be modified so they can be used in a dynamic environment where data is added or removed. Unfortunately, such an environment is very complicated because supporting it involves not only the accurate placement of fragments but also the need to transparently migrate object fragments while the system is being accessed by users. The initial step in this research direction requires that we determine a performance threshold below which dynamic redesign is required so the system will continue to meet its performance goals. Future research should attempt 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.

Polymorphism, an object oriented database feature that allows overloading of an attribute or method definition of a class by its subclass, has not been considered in this work. For example, if a superclass *Person* has an attribute *salary*, then the subclass *Student* does not re-define this attribute *salary* because with our model, an instance object of class *Student* has a part of it that is an instance object of class *Person* through our object join operation. Accommodating polymorphism requires our object join operation identifying polymorphic attributes/methods at the subclass level and using them to overwrite their superclass equivalents rather than having the aggregation of both their values at both superclass and subclass levels. This work can easily be extended to accommodate this feature for future research.

In the current performance analysis, using partition evaluator, for the vertical fragmentation case, we assume only two fragments are generated for each class. Future research should incorporate into our testbed prototype, the component of the Partition Evaluator that determines the optimal number of fragments based on the class size and application uses. Other object based fragmentation approaches can also be hooked on to this testbed for evaluation purposes. Future research should also attempt to employ similar performance analysis techniques for our horizontal fragmentation algorithms.

The third type of fragmentation that may be necessary on a database entity is hybrid, which in our model generates fragments of a class C such that each hybrid fragment contains subsets of the class's attributes, methods and instance objects. In the relational case, iterative application of horizontal and vertical fragmentations yield hybrid fragmentation. Future research should attempt to define hybrid fragmentation schemes for object systems.

As the problem of distributed object based design has the two components of fragmentation and allocation of fragments, future research should attempt to discover suitable theoretical models for allocating fragments to distributed sites.

Finally, for efficient management of distributed object based design, future research should attempt to define a distributed directory that supports distributed class fragments and implement it on a heterogeneous network.

Bibliography

[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] Elisa Bertino and Won kim. Indexing Techniques for Queries on Nested Objects. *IEEE Transactions on Knowledge and Data Engineering*, 1(2), 1989.

[3] T. Bloom and S.B. Zdonik. Issues in the Design of Object-Oriented Database Programming Languages. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications*. SIGPLAN Notices, 1987. pp. 441-451.

[4] P. Butterworth, A. Otis, and J. Stein. The Gemstone Object Database Management System. *Communications of the ACM*, 34(10), Oct. 1991.

[5] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on VLDBs*, 1986.

[6] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Storage Management for Objects in EXODUS. In Kim Won and Lochovsky F.H., editors, *Object-Oriented Concepts, Databases, and Applications*. ACM Press, 1989.

[7] 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*. ACM, 1982.

[8] 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.

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

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

[11] 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.

[12] O. Deux et al. The O_2 System. *Communications of the ACM*, 34(10), Oct 1991.

[13] 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.

[14] C.I. Ezeife and Ken Barker. Horizontal Class Fragmentation for Advanced Object Models in a Distributed Object-Based System. In *Proceedings of the Ninth International Symposium on Computer and Information Sciences*. Turkey, 1994. Nov. 7-9.

[15] C.I. Ezeife and Ken Barker. Horizontal Class Fragmentation in Distributed Object Based Systems. In *Proceedings of the Second Biennial European Joint Conference on Engineering Systems Design and Analysis*. ASME Publications, 1994. London, England, July, Vol. 65, No.5, pp. 225-235.

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

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

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

[19] D.H. Fishman, D. Beech, J. Annevelink, E. Chow, T. Connors, J.W. Davis, W. Hasan, C.G. Hoch, W. Kent, S. Leichner, P. Lyngback, B. Mahbod, M.A. Neimat, T. Risch, M.C. Chan, and W.K. Wilkinson. Overview of the Iris DBMS. In W. Kim and F. Lochovsky, editors, *Object–Oriented Systems, Databases and Programming*, pages 174–199. Addison-Wesley Publishing Co., Inc., Reading, Massachusetts, 1989.

[20] L.M Haas, W.Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M.J. Carey, and E. Shekita. Starburst Mid-Flight: As the Dust Clears. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar. 1990.

[21] 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.

[22] 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.

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

[24] 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.

[25] 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.

[26] W. Kim. Object-Oriented Databases: Definition and Research Directions. *IEEE Transactions on knowledge and Data Engineering*, 2(3), Sept. 1990.

[27] W. Kim, N. Ballou, H. Chou J.F. Garza, and D. Woelk. Features of the Orion Object-Oriented Database System. In Kim Won and Lochovsky F.H., editors, *Object-Oriented Concepts, Databases and Applications*. ACM Press, 1989.

[28] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the Orion Next-Generation Database System. *IEEE Transactions on knowledge and Data Engineering*, 2(1), March 1990.

[29] Won Kim and H. Chou. Versions of Schema For Object-Oriented Databases. In *Proceedings of the 14th Very Large Data.* Base Conference, Los Angeles, California, 1988. pp.148-159.

[30] 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.

[31] G.M. Lohman, B. Lindsay, H. Pirahesh, and K.B.Schiefer. Extensions to Starburst: Object, Types, Functions and Rules. *Communications of the ACM*, 34(10), Oct 1991.

[32] D. Maier and J. Stein. Development and Implementation of an object-Oriented DBMS. In *Proceedings of the first ACM Conference on Object- Oriented Programming Systems, Langs., and Applications*, volume 21(11), pages 167–185. SIGPLAN Notices, 1986.

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

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

[35] 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.

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

[37] 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.

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

[39] 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.

[40] L. A. Rowe and M. Stonebraker. The Postgres Data Model. In *Proceedings of the 13th International Conference on VLDBs*, pages 83–96, 1987.

[41] D. Shin and K.B. Irani. Fragmenting Relations Horizontally Using a Knowledge-Based Approach . *IEEE Transactions on Software Engineering*, 17(9), Sept. 1991.

[42] A.H. Skarra and S.B. Zdonik. The Management of Changing Types in an Object-Oriented Database. In *Proceedings of the first ACM Conference on Object-Oriented Programming System, Languages and Applications*, volume 21(11), pages 483–495. SIGPLAN Notices, 1986. Vol.21, No.11, pp. 483-495.

[43] M. Stonebraker. The Postgres Storage Manager. In *Proceedings of the 13th International Conference on VLDBs*, pages 289–300, 1987.

[44] M. StoneBraker and G. Kemnitz. The Postgres Next- Generation Database Management System. *Communications of the ACM*, 34(10), 1991.

[45] M. Stonebraker, L.A. Rowe, and M. Hirohama. The Implementation of POSTGRES. *IEEE Transactions on knowledge and Data Engineering*, 2(1), March 1990.

[46] M. Sullivan and M. Olson. An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System. *IEEE Transactions on knowledge and Data Engineering*, 2(3), Sept. 1992.

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

[48] K. Wilkinson, P. Lyngbaek, and W. Hasan. The Iris Architecture and Implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar. 1990.