Horizontal Class Fragmentation in Distributed Object Based Systems[1]

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

Technical Report: 93-04

October 1993

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

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 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 paper first reviews a taxonomy of the fragmentation problem in a distributed object base. The paper then contributes by presenting a comprehensive set of algorithms for horizontally fragmenting the four realizable class models on the taxonomy. The fundamental approach is top–down, where the entity of fragmentation is the class object. 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 subclasses, 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 optimal fragments. Thus, these algorithms account for inheritance and class composition hierarchies as well as method nesting among objects, and are shown to be polynomial time.

Contents

List of Figures

Chapter 1

Introduction

Many researchers have demonstrated the importance of entity fragmentation in distributed relational database design. 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 [18]. A DOBS supports an object oriented data model including features of *encapsulation* and *inheritance*.

The problem of distributed database design comprises first, the fragmentation of database entities and secondly, the allocation of these fragments to distributed sites. Two approaches are possible in a distributed database design – top-down and bottom-up. With the top-down approach, the input to the design process is the global conceptual schema (GCS) and the access pattern information, while the output from the design process is a set of local conceptual schemas (LCSs) [33]. The input to the design process is obtained from an *a priori* static and system requirements analysis which defines the system environment and collects an approximation of both the data and processing needs of all potential database users [44]. The expectations of the system with respect to performance, reliability and availability, economics and flexibility are also specified from the analysis. The view and conceptual design activities produced from the requirements study defines interfaces for end users, entity types and their relationships. Statistical information collected from these design activities include the frequency of user applications and reference patterns of applications. The output of this static and system analysis are the GCS and the access pattern information which constitute inputs to the distribution design process described here.

This paper uses this top-down design approach and designs the LCSs by distributing the entities over distributed sites. In the relational database environment, the entity of distribution is a relation while in a distributed object based system (DOBS), our entity of distribution is a class. This work aims at dividing classes into class fragments, which are later distributed.

There are many existing object based systems that support some form of distribution including ITASCA [17], ENCORE [16], GOBLIN [19], THOR [24], and EOS [32]. Distributed relational databases [33] benefit greatly from fragmentation and these benefits should be realized in a distributed object environment [18]. A partial list of benefits include:

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

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 [42]. 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.

This research 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 classes in all necessary models of a DOBS.
3. Provides algorithms for vertically fragmenting classes in all necessary models of a DOBS.
4. Define criteria for measuring the optimality of the fragmentation schemes and prove the defined schemes meet these criteria by showing that:.
   (a) The fragments generated are correct.
   (b) The amount of remote access to data is minimized while relevant local access is maximized.

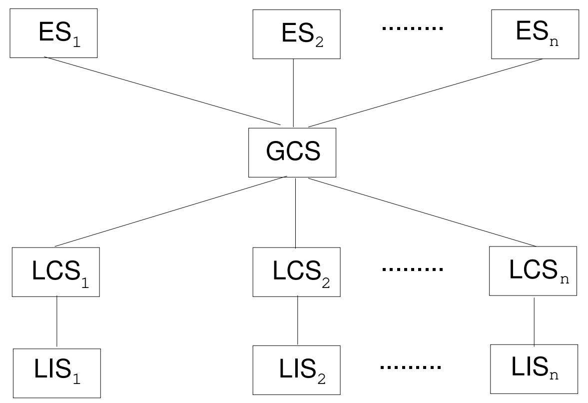


Figure 1.1: Distributed Database Reference Architecture

1.1 The DOBS model

A distributed object based system is a collection of local object bases distributed at different sites, interconnected by a network. We assume the database management system (DBMS) is distributed and the objects making up the object base are placed around the network. The general architecture of a distributed DBMS is summarized below [33].

- 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 directory/dictionary (GD/D) is used to provide required global mappings and allows user queries some location transparency.

1.1.1 The Data Model

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

The overall inheritance hierarchy of the database is captured in a class lattice. A class is an ordered relation $\mathcal{C} = (\mathbf{K},\mathcal{A},\mathcal{M},\mathcal{I})$ where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[1]. Three fragment types are defined on a class:

- Horizontal Fragmentation: Each horizontal fragment ($\mathcal{C}_h$) of a class contains all attributes and methods of the class but only some instance objects ($\mathcal{I}^{'} \subseteq \mathcal{I}$) of the class. Thus, $\mathcal{C}_h = (\mathrm{K},\mathcal{A},\mathcal{M},\mathcal{I}^{'})$.

- 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, $\mathcal{C}^v = (\mathrm{K},\mathcal{A}^{'},\mathcal{M}^{'},\mathcal{I})$.

- 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, $\mathcal{C}_h^v = (\mathrm{K},\mathcal{A}^{'},\mathcal{M}^{'},\mathcal{I}^{'})$.

1.1.2 A Taxonomy of Class Models

This section defines a class fragmentation taxonomy suitable for reasoning about object bases. The taxonomy describes what constitutes a class fragment (attributes or attributes and methods); the nature of the attributes (simple atomic data types or other objects); and how objects are accessed (a "traditional" transaction model or using a "nested" invocation strategy). The taxonomy (Figure 1.2) has three dimensions. The axes are the *type of fragmentation*, the *type of attributes*, and the *type of method invocations* in the object model. We present the taxonomy succinctly since a complete description is available in [12].

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

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

Simple Attributes: only primitive attribute types that do not contain other classes as part of them. The precise composition of the set of primitive types is an important issue but beyond the scope of this paper [1].

Complex hierarchy: The domain of an attribute may be another class. This is often referred to as a "part–of" or class composition hierarchy.

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

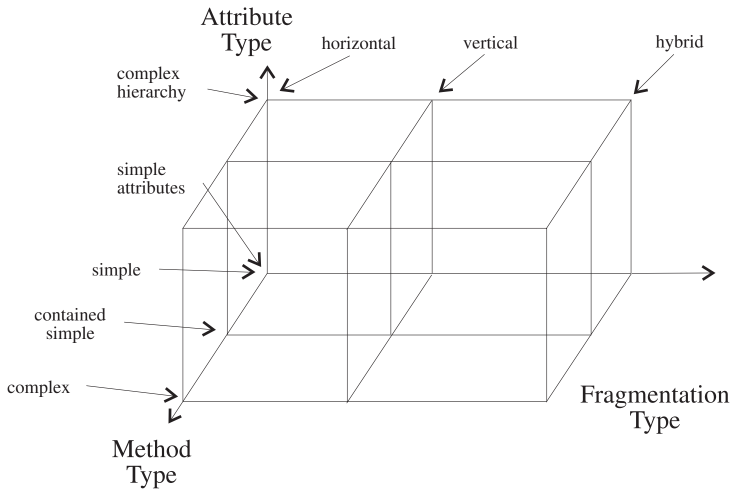


Figure 1.2: The Object Design Taxonomy

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

Simple methods: are those that do not invoke other methods of other classes. Further, simple methods only use local object data or data passed as parameters.

Contained Simple methods: are simple methods of a contained class that is part–of a containing class.

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

The contained simple methods are used to preserve the encapsulation features of objects of contained (part-of) classes and thus, the only four realistic scenarios are: simple attributes with simple methods, complex attributes with simple methods, simple attributes with complex methods, and complex attributes with complex methods. In effect, the point on the taxonomy with complex attribute and simple method is actually represented by the point having complex attribute with contained simple method, while the point having simple attribute with contained simple method is not realistic. This classification enables us accommodate all the necessary features of object orientation and provide solutions for object bases that are structured in various ways. Thus complete solutions to the design problem in distributed object based systems requires a description of the four points on the taxonomy.

1.2 Outline of The Document

The organization of the balance of this document is as follows. Previous work on distributed database design is reviewed in Chapter 2. Chapter 3 presents algorithms for horizontally fragmenting class objects in a DOBS while Chapter 4 presents future research in this area. Finally, Chapter 5 makes some concluding comments.

Chapter 2

Previous/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 system, 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 while Section 2.4 discusses previous work on fragmentation in object-oriented systems.

2.1 Distributed Design (Relational)

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

2.1.1 Horizontal Fragmentation

Several researchers have worked on fragmentation and allocation in the relational data model including Ceri, Negri and Pelagatti [5], Özsu and Valduriez [33], Navathe *et al.* [29], Navathe and Ra [31], and Shin and Irani [6].

Ceri, Negri and Pelagatti [5] 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. Further, implications exist between boolean expressions of simple predicates which eliminate meaningless minterms.

Navathe, Karlapalem and Ra [30] 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.* [29, 31] 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 [33] 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* [7] 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* [7]. They use the same concepts and parameters as well as methodology identified in Ceri *et al.* [5, 7] 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.* [5] (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 [5]. 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 [36] partition relations horizontally based on estimated user reference clusters (URCs). URCs are estimated from user queries [5, 33] 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 [15] 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.* [27] is used to form these attribute clusters. Navathe *et al.* [29] 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 [8] optimized this work by developing an algorithm that obtains an optimal binary partitioning for relational databases. They use knowledge of physical factors[1] to decrease the number of disk accesses. Further refinement is accomplished by applying the binary vertical partition algorithm iteratively [29]. Navathe and Ra [31] 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 [33] 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.

Muthuraj *et al.* [28] 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 [27], binary vertical partition [29], Ra's graphical algorithm [31] and an exhaustive enumeration algorithm.

2.1.3 Hybrid

Özsu and Valduriez [33] argue that nesting of horizontal and vertical fragmentations in any order produces hybrid fragmentations of relations. Pernul, Karlapalem and Navathe [34] 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. From these relationships fragments are derived by using the horizontal, vertical and derived fragmentation operators defined earlier [5, 29]. 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.

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

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 [2], EXODUS [3], ORION [23], ENCORE [37], POSTGRES [38, 39], STARBURST [25], IRIS [13] and O_2 [9].

GEMSTONE is an object-based system developed at Servio [2, 26] 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 [21, 22, 23] 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 definitions of superclasses 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 [16, 37] is an OODBS with origin at 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 [3, 4] 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 [35, 38, 39, 40, 41] 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.

STARBUST [14, 25] 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, multiple and attribute value 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 [13, 43] 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 but provides no support for attribute value inheritance. An object at a higher level is used for managing metadata and there is no direct communication between objects.

O_2 [9] 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 [17], ENCORE [16, 37], GOBLIN [19], THOR [24], and EOS [32].

ITASCA [17] 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 accross 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 [16] 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 [19] 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 [24] 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 [32] 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 need 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 a harbor 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 [18] identify some of the fragmentation issues in object bases including: How are subclasses of a fragment of a class handled? Which objects and attributes of the objects are being accessed by the methods? What type of methods are considered: simple methods that access a set of attribute values of an object or complex methods that access a set of objects and instance variables?[2] Further, they argue that

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

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.* [30] for horizontal fragmentation could be applied.

2.4.2 Vertical Fragmentation

Karlapalem *et al.* [18] define issues involved in distribution design for an object oriented database system. They identify two types of methods – simple and complex methods. Their first model consists of simple methods. They argue that a model consisting of simple methods can be vertically partitioned using techniques described by Navathe *et al.* [29], while that of complex methods, requires a method-based view (MBV). The MBV identifies the set of objects accessed by a method and the set of attributes or instance variables accessed by the method. The sets are further grouped into sets of objects and instance variables based on the classes to which they belong. This generates the set pairs of objects and instance variables (O_i,I_i) accessed from a class C_i by a method. This is called method m_j's view of class C_i. They further suggest the use of concepts developed by Pernul *et al.* [34] to fragment classes based on views.

2.4.3 Hybrid Fragmentation

Karlapalem *et al* [18] 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.

Chapter 3

Horizontal Fragmentation

This chapter presents horizontal fragmentation algorithms for all models as defined on the taxonomy on Figure 1.2. 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.

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 [10, 11]). We proceed by explicitly stating assumptions, definitions required, and then summarize the algorithm. The explicit assumptions are:

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

2. There are no cycles in the dependency graphs discussed later. This means that two classes cannot both depend on each other at the same time so if class C_i is part-of class C_j, class C_j cannot be part–of class C_i.

The second assumption is reasonable in the sense that a subclass of a class is never the class's superclass in the same object base. Similarly, it is highly unlikely that objects are structured in a way that two classes are part of each other (e.g., if class *Student* is part-of class *Dept*, for most cases, class *Dept* is not at the same time part-of class *Student*).

However, some applications may require an object structure where provision of this cycle is necessary, and in this case, the second assumption can be relaxed by selecting which of the two bi-directional links in the dependency graph has higher application access frequency. Several definitions are required:

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

Definition 3.2 *Derived horizontal fragmentation* occurs in two ways. (1) Partitioning of a class arising from the fragmentation of its subclasses, and (2) fragmentation of a class arising from the fragmentation of some complex attribute (part–of) of the class. ■

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

Definition 3.4 *The cardinality of a class* is the number of instance objects in the class (denoted $\mathbf{card}(C_i)$). ■

Definition 3.5 *Access frequency of a query* is the number of accesses a user application makes to data. Data in this context can be a class, a fragment of a class, an instance object of a class, an attribute or method of a class. If Q = $\{q_1, q_2, \ldots, q_q\}$ is a set of user queries, acc(q_i,d_j) indicates the access frequency of query q_i on data item j. We represent the access frequency of a minterm m_i as acc(m_i,d_j) since minterms formalize queries. ■

Definition 3.6 *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 3.7 *Instance Object join* ($\odot$) between a pointer to an instance object of a superclass and an instance object (I_j) of a class C_i returns the aggregate of instance objects of the two classes that represent the actual instance object of the class. ■

To illustrate the aggregate returned by the object join function, we consider the following example. In a database with *Student* a superclass of *Grad*, an instance object I_3 of *Grad* is represented as (Student pointer5) $\odot$ {537msc11,Mary Smith}. This means that the actual I_3 of *Grad* is the quantity representing the instance object I_5 of the superclass *Student* added to the quantity {537msc11,Mary Smith}. *A method invocation* M_j^i on the objects of a class C_i is a set of simple predicates that describe which objects to access. These simple predicates are represented as $\{Pr_{j1}^i, Pr_{j2}^i, \ldots, Pr_{jn}^i\}$ for 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 following correctness rules:

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

3.1.1 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 and also appear in [10, 11].

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:

1. **The class lattice** showing the superclass–subclass relationship between all classes.
2. **The dependency graph**[1] captures the method link or attribute link between any two classes in the database. This means that, if an attribute of a class C_i is another class C_j, then there is 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 lattice 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 lattice information and is generated as follows. Every subclass depends on its superclass in the link graph. This is a direct consequence of the inheritance hierarchy which implicitly partitions every superclass into subclasses. By starting fragmentation with the subclass and propagating intermediate results up the class hierarchy we preserve the implicit partitioning of the inheritance

[1] An example of a graph depicting dependency information is the class composition hierarchy shown in Kim [20].

hierarchy and produce optimal results. Directed links among database classes are used to show these relationships and if class C_j depends on class C_i, 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 lattice 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 in all the user queries from D_i.

A simple predicate is defined as follows. Given a class C 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 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 [33], 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 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 $P^*_{ik} = \neg P_{ik}$.[2] We now use the semantics of the class to eliminate meaningless minterm predicates. Algorithms *SimplePredicates* of Figure 3.2 and *COM-MIN* of [33] 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 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) = 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

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

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. 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 3.8 *Access frequency acco(I)* of an object (I) is the sum of the access frequencies of all the applications $\mathcal{Q}_i$ accessing the object. 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 3.9 *Relevant accesses(F^d,F^p)* are those made to local objects of both fragments F^d and F^p and are defined as all of the access frequencies of objects belonging to both the derived and primary fragments. Thus, Relevant accesses(F^d,F^p) = $\sum_{I_i}$ acco(I_i), for all $I_i \in (F^d \cap F^p)$. ■

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

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 lattice of the database showing subclass/superclass relationships.
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. $\Gamma \leftarrow \{C_k | C_k \in \mathcal{C}_d\}$
$\lambda = \emptyset$;
$C_i \leftarrow$ 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)
$C_l = C_l \cup C_j$ (7)
end; {for C_j}
end; {for C_i}
$C_l = C_l - C_i$; (8)
C_i = a leaf node in $\mathcal{C}_d$ (9)
end; {while C_i}
return (LG); (10)
end;

Figure 3.1: The Linkgraph Generator

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 $C_i \in C_d$ **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 $P_{i1}^{j\,1} \in M_i^j$ (5)
$\mathcal{P}_i = \mathcal{P}_i \cup P_{i1}^{j\,1}$ (6)
end; {for M_i^j}
end; {for q_k}
$\mathcal{P} = \bigcup \mathcal{P}_i$ (7)
end;

Figure 3.2: Simple Predicate Generator

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

Definition 3.11 *Affinity between a derived Fragments F^d and a primary fragment F^p of the same class, aff(F^d,F^p)* is a measure of how frequently objects of these two fragments are needed together by applications.
aff(F^d,F^p) = Relevant access(F^d,F^p) − Irrelevant access(F^d,F^p). ■

Definition 3.12 *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.

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

■

Definition 3.13 *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.

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 $I_i \in F^p$ **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: $\mathcal{Q}_i$: set of user queries
$\mathcal{C}_d$: set of database classes
L(C) : the class lattice
output: **P** ($\mathcal{F}_{c_i}$): set of horizontal fragments of
the set of classes in the database.
var
Linkgraph : tree
$\mathcal{P}_i$: set of simple predicates for class C_i.
$\mathcal{F}_i^p$: set of primary horizontal fragments for class C_i
$\mathcal{F}_i^d$: set of derived horizontal fragments for class C_i
$\mathcal{M}_c^L$: set of member classes for the link graph L(C)
begin
  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

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

This rule selects the most suitable primary fragment to merge with the derived fragment. The merging process results in some overlap of objects in a set of primary fragments. Disjointness requires that 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 pattern.

3.1.2 An Example

This example uses the class lattice depicted in Figure 3.5 [3] and the sample data illustrated in Figure 3.6 [4]. 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:

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

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

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

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:

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

[4] 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.

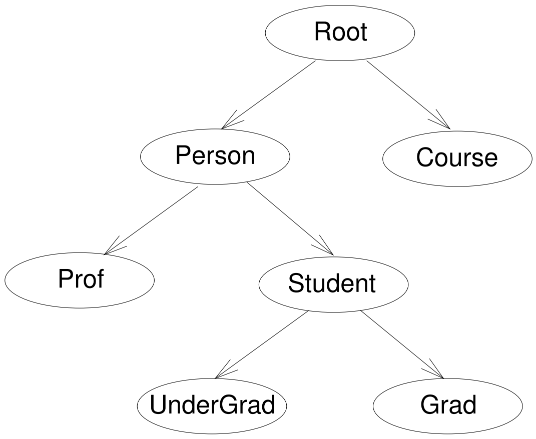


Figure 3.5: Class Lattice of Sample Object Base

Person = {k.ssno,{a.name,a.age,a.address},{m.whatlast,m.daysold,m.newaddr},
{ I_1 {501-156-17,John James,30,Winnipeg}
I_2 {905-406-01,Ted Man,16,Winnipeg}
I_3 {658-157-09,Mary Ross,21,Vancouver}
I_4 {903-505-18,Peter Eye,23,Toronto}
I_5 {502-781-15,Bill Jeans,40,Toronto}
I_6 {669-882-08,Mandu Nom,32,Vancouver} } }
Prof = Person pointer $\odot$ {k.empno,{a.status,a.salary},{m.coursetaught,m.whatsalary},
{ I_1 (person pointer5) $\odot$ {62-1736,full prof,45000}
I_2 (person pointer6) $\odot$ {60-1840,assoc prof,60000} } }
Student = Person pointer $\odot$ {k.stuno,{a.dept,a.feespd,a.coursetaken}, {m.stuno-of,m.dept-of,m.owing},
I_1 (person pointer1) $\odot$ {5371019,Math,Y}
I_2 (person pointer4) $\odot$ {5370008,Computer Sc.,N}
I_3 (person pointer2) $\odot$ {4110000,Stats,Y}
I_4 (person pointer3) $\odot$ {4135709,Computer Sc.,N} } }
Grad = Student pointer $\odot$ {k.gradstuno,{a.supervisor},{m.whatprog}
{ I_1 (Student pointer1) $\odot$ {537phd1,John West}
I_2 (Student Pointer2) $\odot$ {537msc10,Mary Smith} } }
UnderG = Student
I_1 (Student pointer3)
I_2 (Student pointer4)

Figure 3.6: The Sample Object Database Schema

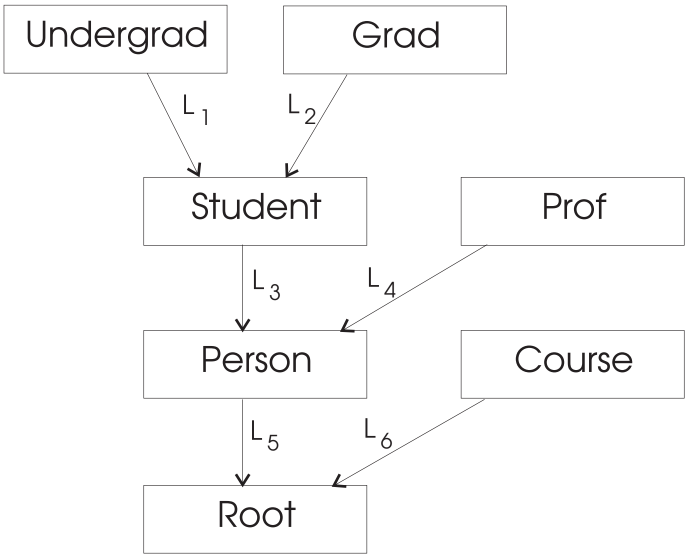


Figure 3.7: Class Lattice's Link Graph

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

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

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

Given this application information we form horizontal fragments of the class in our example database using the class lattice given in Figure 3.5. We apply the Horizontal-Frag 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"
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 [5].

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 [20] 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 *Horizontal-Frag*, using the link graph from the class composition hierarchy, to produce an initial fragmentation. Generating derived fragments of a member class originating from class

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

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
L(C) : the class lattice
A(C) : class composition hierarchy showing attribute link
output: **P** $(\mathcal{F}_{c_i})$: set of horizontal fragments of the set of classes in the database
begin
P $(\mathcal{F}_{c_i})$ = **HorizontalFrag**($\mathcal{Q}_i$, $\mathcal{C}_d$, **A(C)**) (1)
P $(\mathcal{F}_{c_i})$ = **HorizontalFrag**($\mathcal{Q}_i$, **P** $(\mathcal{F}_{c_i})$, **L(C)**) (2)
end; {**Hor_CA_SM**}

Figure 3.8: Horizontal Fragmentation – Complex Attribute and Simple Method

composition hierarchy entails finding the predicates of its owner class and defining fragments of the owner 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 containing class in the class lattice (line (1), Figure 3.8). The final horizontal fragments of the containing classes are produced using the class inheritance lattice 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 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. Our presentation reflects the approach used earlier where we present definitions, selection rules, and finally, the algorithm.

Definition 3.14 *Object reference set of method M_j^i* denoted oref(M_j^i) contains all objects referenced by method M_j^i of some object in class C_i. ■

We can enumerate the objects in oref(M_j^i) by $\{I_1^{k1}, I_2^{k2}, \ldots, I_m^{kn}\}$ where I_m^{kn} is the m^{th} object in class fragment C_{kn}.

Definition 3.15 *Class fragment object reference set coref($F_k^{c_i}$,C_m)* are objects of class C_m referenced by methods of class fragment F_k^i of class C_i. This is derived from oref(M_j^i) as :
coref(F_k^i,C_m) = $\bigcup_{I_k^m \in oref(M_{jn}^i)} I_k^m$ for all M_{jn}^i, $1 \leq n \leq$ card(M_{jn}^i). ■

Definition 3.16 *Number of references* of method M_j^i to an object I_n^k of class C_k is denoted numref(M_j^i,I_n^k) which is the number of times method M_i^i accesses object I_n^k. ■

Definition 3.17 *Class number of reference cnumref(F_k^i,I_n^k)* of class fragment F_k^i on object I_n of class C_k, is the sum of all accesses made to this object by all methods of this class fragment. Thus, cnumref(F_k^i,I_n^k) = $\sum_{M_j^i \in C_i}$ numref(M_j^i,I_n^k). ■

Definition 3.18 *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)

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

■

Definition 3.19 *Method Access frequency macco(I)* of an object (I) 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) = $\sum_i^q acc(q_i, I) + \sum_{\forall M_j^i | I \in oref(M_j^i)} numref(M_j^i, I)$. ■

Definition 3.20 *Method relevant access MRA(F^d,F^p)* measures the number of times objects of both fragments are accessed by methods of other objects. Thus, MRA(F^d,F^p) = $\sum_{I_k \in \{F^d \cap F^p\}}$ numref(M_j^i,I_k) for any M_j^i. ■

Definition 3.21 *Method Irrelevant access MIA(F^d,F^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^d,F^p) = $\sum_{I_k \in \{((F^p \cup F^d) - (F^d \cap F^p))\}}$ numref(M_j^i,I_k) for all M_j^i. ■

Definition 3.22 *Object Fragment Affinity* between an object I_i and a fragment F, denoted ofaff(I_i,F), is the amount of accesses made to a fragment *vis a vis* an object.
ofaff(I_i,F) = $\sum_{I_j \in F}$ maffo(I_i,I_j), i $\neq$ j. ■

Definition 3.23 *Method Affinity between Fragments maff(F^d,F^p)* is the difference between relevant and irrelevant access: maff(F^d,F^p)= MRA(F^d,F^p) - MIA(F^d,F^p). ■

Affinity Rule 3.3 Select the primary fragment that maximizes the method affinity measure maff(F^d, F_i^p) where F^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_j^p that maximizes the function ofaff(I_i, F_j^p) is where I_i is placed. ■

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 above definitions because we can develop an algorithm that maximizes local relevant access and minimizes local irrelevant access.

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 – inheritance hierarchy and method dependencies. Thus, algorithm *Hor_SA_CM* (Figure 3.10) 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 in the object base and produce a derived fragment of the class being fragmented due to complex method dependencies of objects of this other class fragment. It further selects a primary fragment of this class most appropriate to merge with this derived fragment 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 3.9. The first step of Figure 3.9 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 5.2 is used to find the primary fragment with which it has the greatest affinity (line (15-22)) and it is removed from all other (line (23-25)).

3.4 Complex Attributes and Complex Methods

This section presents an algorithm for horizontally fragmenting classes consisting of complex attributes and complex methods. With this model, 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 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. These final derived fragments are merged with horizontal fragments from the first and second iterations to obtain the final fragments of the classes. The formal presentation of this algorithm *Hor_CA_CM* is given in Figure 3.11. 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 3.12.

The database schema information that is one input to this design process consists of the class lattice of the object base which is as given in Figure 3.13, and the class composition hierarchy in Figure 3.14. The other input is the application information Q_1 to Q_5 given below.

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

Algorithm 3.6 *(Generate Derived Fragment Based on Complex Methods)*

Algorithm Derived_From_CompM
input: **P** ($\mathcal{F}_{c_i}$): set of horizontal fragments of the classes
P($oref(M_i^j)$): set of object references for all $M_i^j \in C_i$.
$\mathcal{C}_d$: set of database classes
output: **P** ($\mathcal{F}_{c_i}$): set of horizontal fragments of the classes
var
$\mathcal{F}_i^d$: set of derived horizontal fragments for class C_i
Prifragforderived, Prifragforobj : 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 $C_m \in C_d$ **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 $F^p \in \mathcal{F}_{c_i}$ 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^d,F_i^p) > maff(F^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}
for each overlapping object I_i of class fragments ($F_{i1}^p,\ldots,F_{in}^p$) (15)
k = 1 (16)
Prifragforobject= $F_k^p \in C_i$ (17)
for every fragment $F_i^p \in$ class C_i (18)
if ofaff(I_i,F_i^p) > ofaff(I_i,F_k^p), (19)
then begin
Prifragforobject = F_i^p (20)
k = i (21)
end; {**begin**}
end; {**for** F_i^p}
F_i^p = Prifragforobject (22)
for every fragment $F_k^p \in$ class C_i (23)
if ($I_i \in F_k^p$) and ($F_k^p \neq F_i^p$) (24)
then $F_k^p = F_k^p - I_i$ (25)
end; {**for** F_k^p}
end; {**for** I_i}
end; {**of Derived_From_CompM**}

Figure 3.9: Capturing the Complex Method dependency information Class Fragments

Algorithm 3.7 *(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 lattice
$\mathbf{P}(oref(M_i^j))$: set of object references for all $M_i^j \in C_i$.
output: **P** $(\mathcal{F}_{c_i})$: 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
P $(\mathcal{F}_{c_i})$ = **HorizontalFrag**($\mathcal{Q}_i$, $\mathcal{C}_d$, **L(C)**) (1)
for each class $C_i \in C_d$ **do** (2)
P $(\mathcal{F}_{c_i})$ = Derived_From_CompM(**P** $(\mathcal{F}_{c_i})$,C_i,P($oref(M_i^j)$),$\mathcal{C}_d$) (3)
end; {**for** C_i}
end; {**of Hor_SA_CM**}

Figure 3.10: Horizontal Fragmentation For Simple attributes and Complex methods

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"}

Algorithm 3.8 *(Complex Attribute and Compex Method)*

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

Figure 3.11: Horizontal Fragmentation For Complex Attribute and Complex Methods

Person = {k.ssno,{a.name,a.age,a.address},{m.whatlast,m.daysold,m.newaddr},
{ I_1 {501-156-17,John James,30,Winnipeg}
I_2 {905-406-01,Ted Man,16,Winnipeg}
I_3 {658-157-09,Mary Ross,21,Vancouver}
I_4 {903-505-18,Peter Eye,23,Toronto}
I_5 {502-781-15,Mary Smith,40,Toronto}
I_6 {669-882-08,John West,32,Vancouver}
I_7 {503-853-91,Jacky Brown,35,Winnipeg}
I_8 {617-589-77,Sean Dam,27,Toronto} } }
I_9 {515-200-91,Bill Jeans,43,Vancouver}
I_{10} {635-005-88,Mandu Nom,30,Winnipeg}
Prof = Person pointer $\odot$ {k.empno,{a.status,a.dept,a.salary,a.student},{m.coursetaught,m.whatsalary},
{ I_1 (person pointer5) $\odot$ {62-1736,asst prof,Computer Sc.,45000,students pointers}
I_2 (person pointer6) $\odot$ {60-1840,assoc prof,Math,60000,students pointers}
I_3 (person pointer9) $\odot$ {61-5006,full prof,Math, 80000,students pointers}
I_4 (person pointer10) $\odot$ {63-1843,full prof,Math,82000,students pointers} } }
Student = Person pointer $\odot$ {k.stuno,a.dept,a.feespd,a.coursetaken} ,
{m.stuno-of,m.dept-of,m.owing},
{ I_1 (person pointer1) $\odot$ {5371019,Math,Y}
I_2 (person pointer4) $\odot$ {5370008,Computer Sc.,N}
I_3 (person pointer2) $\odot$ {4110000,Stats,Y}
I_4 (person pointer3) $\odot$ {4135709,Computer Sc.,N} }
I_5 (person pointer7) $\odot$ {5370009,Math,Y}
I_6 (person pointer8) $\odot$ {4111005,Stats,N} } } }
Grad = Student pointer $\odot$ {k.gradstuno,{a.supervisor},{m.whatprog}
{ I_1 (Student pointer1) $\odot$ {537phd1,John West}
I_2 (Student Pointer2) $\odot$ {537msc10,Mary Smith} } }
I_3 (Student Pointer5) $\odot$ {537msc11,Mary Smith } } }
UnderG = Student
I_1 (Student pointer3)
I_2 (Student pointer4)
I_3 (Student pointer6)
Dept = {k.code,{a.name,a.profs,a.students},{m.number-of-profs}},
{ I_1 {074,{Computer science,prof pointers,student pointers}
I_2 {006,{Math,prof pointers,student pointers}
I_3 {006,{Actuary science,prof pointers,student pointers}
I_4 {005,{Stats,prof pointers,student pointers} } }

Figure 3.12: The Sample Object Database Schema

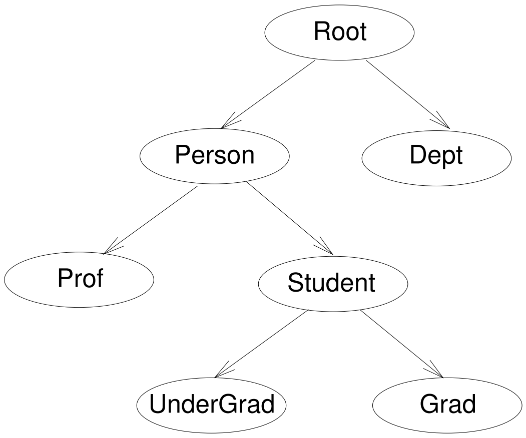


Figure 3.13: Complex Class Lattice

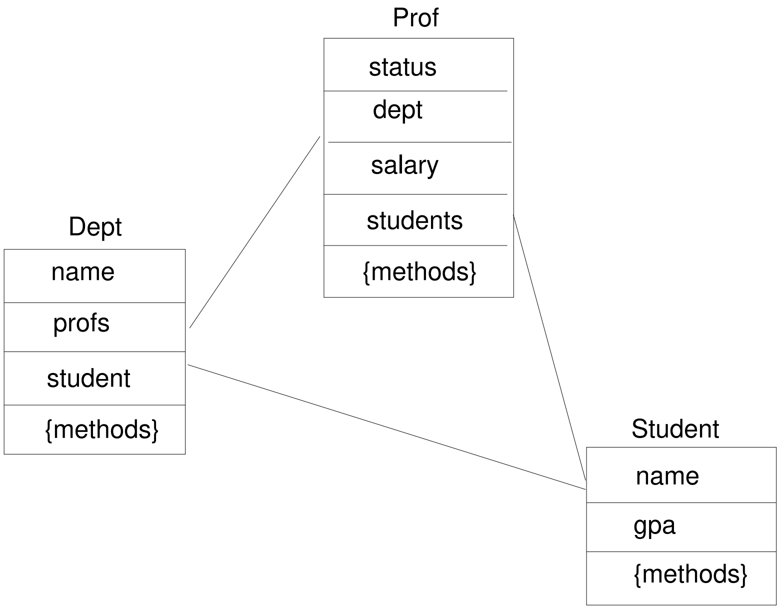


Figure 3.14: Class Composition Hierarchy

Q_5: This application groups departments by their names. The methods are for the class **Dept** and the predicates used are:

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

Fragmentation Process : Execution of Algorithm Hor_CA_CM
Line 1 of algorithm Hor_CA_CM (Figure 3.11) generates the horizontal fragment using the complex class lattice as follows:

- Generates the complex class lattice link graph shown in Figure 3.15.
- Defines primary horizontal fragments for all owner classes to obtain:

  Class Grad
  Fragments F_1^p = instance object 1 $\{I_1\}$
  F_2^p = instance object 2 $\{I_2,I_3\}$
  Class Student
  Fragments F_1^p : $\{I_1,I_5\}$
  F_2^p : $\{I_2,I_4\}$
  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 : $\emptyset$
  F_6^p : $\{I_1\}$
  Class Dept
  Fragments F_1^p : $\{I_2,I_3\}$
  F_2^p : $\{I_1\}$
  F_3^p : $\{I_4\}$

- Generates derived fragments of member classes to obtain:

  Class Student
  Using the primary fragments of the owner class Grad, the derived fragments are:
  $F_1^d = \{I_1\}$
  $F_2^d = \{I_2,I_5\}$
  Derived fragments of the Class Person
  The derived horizontal fragments of Person based on owner class (Student), are:
  $F_1^d = \{I_1,I_7\}$
  $F_2^d = \{I_4,I_3\}$
  $F_3^d = \{I_2,I_8\}$

Derived fragments of Person based on the owner class Prof are:
$F_4^d = \{I_{10}\}$
$F_5^d = \{I_6, I_9\}$
$F_6^d = \{I_5\}$

- Combines Primary and Derived fragments of member classes using the following access frequency information. 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[6]. 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 the 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.

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

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.

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

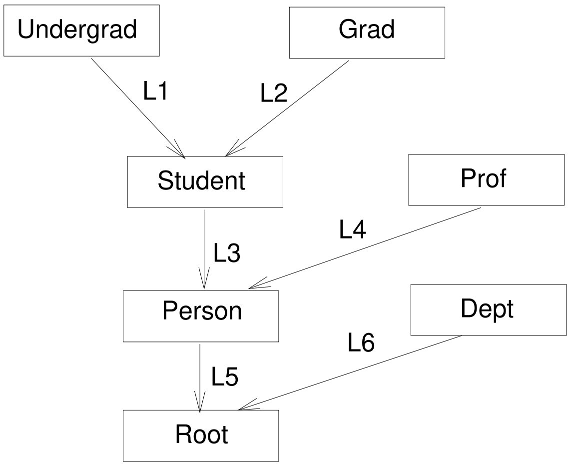


Figure 3.15: Complex Class Lattice's Link Graph

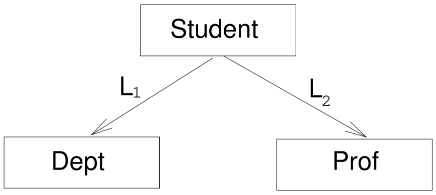


Figure 3.16: Class Composition Hierarchy Linkgraph

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.

This completes the first iteration. The second iteration is the execution of line 2 of algorithm *Hor_CA_CM* (Figure 3.11) 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 3.16.

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}

$F_3^d = (\emptyset)$
Derived Fragments for member class Dept
$F_1^d = \{I_2, I_3\}$
$F_2^d = \{I_1\}$
$F_3^d = \{I_4\}$

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 is given in the next six lines.

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

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

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 *Dept* invoke methods of objects of classes *Prof* and *Student*. The pattern of method invocation of objects of other classes is shown in Figure 3.17. Note that the attribute *student* of classes *Prof* and *Dept* is a complex attribute of type *student* and attribute *prof* of *Dept* is of type *Prof*. Also, some methods of *Dept* will need to invoke objects of *Student* and *Prof*, while some methods of *Prof* will need to invoke objects of *Student*. Using *oref* statistics, the derived fragments of classes based on complex methods of the first fragment of the class *Dept* is as given below:

Derived fragments of the Class Student
$F_1^d = \{I_2, I_3, I_4, I_5\}$

Derived fragments of the Class Prof
The derived horizontal fragments of Prof based on methods of the first fragment of the class (Dept), as shown in figure 3.17 are:

$F_1^d = \{I_1, I_2, I_3\}$

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 pattern of methods. Assume the following quantitative statistics.
numref(M_3^{dept}, I_1^{prof}) = 15
numref(M_2^{dept}, I_2^{prof}) = 30
numref(M_3^{dept}, I_3^{prof}) = 20
numref(M_1^{dept}, I_1^{prof}) = 40
numref(M_1^{dept}, I_3^{prof}) = 20

Class Prof

The task here is to determine the 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 $F_1^h = \{I_1, I_2, I_3, I_4\}$

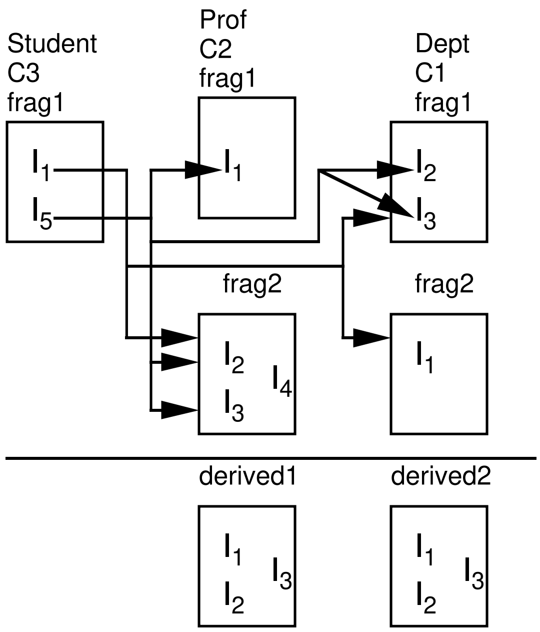
Class Student

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 $F_3^h = \{I_6\}$.

From Figure 3.17, the collection of all objects of C_2 (Prof) invoked by methods of objects in fragment F1 of C_3 (Dept) is $\{I_1, I_2, I_3\}$ and the collection of all objects of C_1 (Student) invoked by methods of fragment F1 of C_3 (Dept) is $\{I_2, I_3, I_4, I_5\}$. These collections form derived fragments of these classes based on complex methods of a fragment of *Dept*. 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 proposed in this paper. In the first part, a structure chart that concisely defines the fragmentation scheme for the most complex class model consisting of complex attributes and complex methods is presented; and secondly the complexities of the fragmentation schemes are discussed.



objects referenced by I_5 of frag1 of C3 = { I_1, I_2, I_3}
from class C2 and {I_2 I_3} from class C1

Figure 3.17: Generation of Derived Fragments of Classes based on Complex Methods

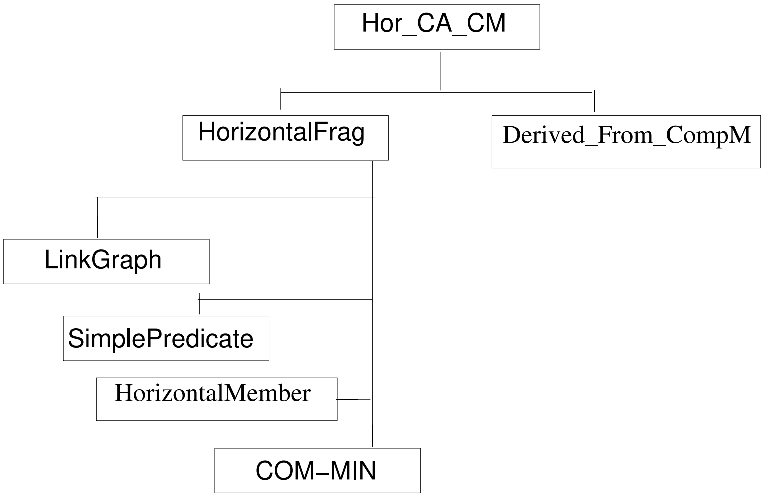


Figure 3.18: The Structure Chart For Model with Complex Attributes and Comlex Methods

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 complex methods, shown in Figure 3.18. 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 number of instance objects in a class or a fragment, q is the largest number of user queries, c is the largest 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 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 3.9 is of time complexity O(cfo). Thus, these algorithms are polynomial in the sizes of these variable inputs.

Chapter 4

Future Research

This chapter presents future research by discussing in each section a problem related to the broad topic of distributed design in an object based system that we intend to address as a continuation of the research results presented in Chapter 3.

4.1 Extensions to Support Vertical Fragmentation

The problem here has to do with defining vertical fragmentation and allocation schemes. The important issue here is that attributes and methods of a class are kept in separate fragments, and not only do we need to keep track of objects of other classes invoked by methods in a fragment, but we need to keep track of exactly what parts of these objects (attributes, methods) that these methods are using. We may need to first define vertical fragments of these classes using attribute and method affinities as done in Navathe *et al* [29]. Then, apply systematic procedure similar to those presented in Chapter 3 for capturing the inheritance hierarchy, the complex attribute and the method dependence information.

4.2 Analysis of Algorithms

This section addresses the problem of analysis of research results. The results of the fragmentation and allocation schemes we defined must be tested to show that:

- The fragments produced are correct. This could be done by showing that applying the algorithms produces a set of fragments that are complete, disjoint and reconstructible. Applying the algorithm to an example database produces a set of complete, disjoint and reconstructible fragments. And this can be proved by hand execution of the algorithms or implementation of the algorithms. Alternatively, a theoretical proof of correctness may be provided.

- The amount of remote access to data is minimized and also the amount of irrelevant access at local site is minimized. This can be done by manually recording the number of remote access as well as irrelevant local access made by applications after fragmentation and allocation using a test object base.

- System performance is reasonable. System performance is dependent on a lot of external factors outside the scope of this research. However, it is intended that if the second point above is proved, then, system performance from this design view point is proved.

Chapter 5

Conclusions

This document presents a precise taxonomy that includes all possible class models in an object based system necessary for distributed design. The classification is based on the type of attribute – simple or complex; type of method – simple or complex and the type of fragmentation – horizontal, vertical or hybrid.

A review of related work in distributed database design as well as early work in object oriented database systems and distributed and/or client/server object oriented systems is presented.

Algorithms for horizontally fragmenting four models of class objects are presented. The models handled are:

- Classes consisting of simple attributes and simple methods.
- Classes consisting of complex attributes and simple methods.
- Classes consisting of simple attributes and complex methods.
- Classes consisting of complex attributes and complex methods.

Precise definitions of problems to be addressed are also presented and problems to address for future research include:

- Developing vertical fragmentation schemes.
- Analysis to show that fragments produced by developed algorithms are optimal.

The balance of this chapter presents a summary of the importance of this work.

5.1 An Emphasis On the Importance of This Research

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 migrated. 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.

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] P. Butterworth, A. Otis, and J. Stein. The gemstone object database management system. *Communications of the ACM*, 34(10), Oct. 1991.

[3] 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.

[4] 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.

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

[6] 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.

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

[8] 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.

[9] O. Deux et al. The o_2 system. *Communications of the ACM*, 34(10), Oct 1991.

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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.

[22] 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.

[23] 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.

[24] 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.

[25] 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.

[26] 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.

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

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

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

[30] 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.

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

[32] 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.

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

[34] 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.

[35] L. A. Rowe and M. Stonebraker. The postgres data model. In *Proceedings of the 13th International Conference on VLDBs*, pages 83–96, 1987.

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

[37] 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.

[38] M. Stonebraker. The postgres storage manager. In *Proceedings of the 13th International Conference on VLDBs*, pages 289–300, 1987.

[39] M. StoneBraker and G. Kemnitz. The postgres next- generation database management system. *Communications of the ACM*, 34(10), 1991.

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

[41] 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.

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

[43] K. Wilkinson, P. Lyngbaek, and W. Hasan. The iris architecture and implementation. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), Mar. 1990.

[44] S.B. Yao, S.B. Navathe, and J.L. Weldon. An integrated approach to database design. New York, 1982. Lecture Notes in Computer Science 132.