1999

Towards making object horizontal fragmentation dynamic.

Jian. Zheng
University of Windsor

Towards Making Object Horizontal Fragmentation Dynamic

by

Jian Zheng

A Thesis
Submitted to the College of Graduate Studies and Research
through the School of Computer Science in Partial
Fulfillment of the Requirements for the Degree of
Master of Science at the
University of Windsor

Windsor, Ontario, Canada
1999

0-612-52688-7

199775

Abstract

Distributed object database design enhances application performance by reducing data communication cost incurred while accessing nonlocal data. The distributed object database design problem is accomplished through fragmentation of database classes and allocation of these fragments to distributed sites. Fragmentation reduces the amount of irrelevant data accessed by applications locally as well as the amount of data transferred to remote sites when migration is required. Existing object fragmentation algorithms use inputs from static requirements analysis. Major changes in a domain would entail a re-analysis of the system and re-running of the distributed design algorithms. In order to make these system changes more acceptable by user, fragmenting database objects dynamically is desired. This thesis aims at defining techniques for initiating dynamic horizontal fragmentation of objects in an object oriented database system. It first presents an architecture that triggers dynamic object horizontal fragmentation and is capable of measuring the performance of existing object horizontal fragmentation schemes. Then it proposes a set of algorithms for (1) measuring the performance of object horizontal fragmentation schemes, (2) determining system performance threshold and (3) monitoring changes in system inputs.

To the people who inspire me…

Acknowledgements

There are many people I would like to thank for their help in writing this thesis. Thanks go to a great supervisor, Dr. Christie I. Ezeife. Her comments, encouragement, and patience were invaluable to the completion of this thesis. Thanks go to my other master's thesis committee members, Dr. J.M. Morrissey and Prof. Phil. Alexander for their comments on my thesis. I am thankful to my committee for being very accommodating. Thanks go to my parents and my brother for their continued support. I wish to thank Yi for encouragement, Yiqin, Mei, and Susan for helpful comments. Thank you all very much.

List of Contents

List of Figures

List of Tables

1 INTRODUCTION

The relational database management systems do not support complex, strongly structured and multimedia data very well. Object oriented database management systems (OODBMS) meet the requirements of these new applications [RZ95]. Distribution of data across a distributed system is a major design problem that impacts on system performance directly. A good distribution design enhances application performance by reducing communication cost overhead due to references to nonlocal data [BB95].

The distributed database design problem consists of fragmentation of database entities followed by allocation of these fragments to distributed sites. There are two ways to design a distributed database—top-down and bottom-up approaches [EB95a]. With the top-down approach, input to the design process is the global conceptual schema (GCS), which is obtained from the conceptual design step. The GCS describes the global database entities and their relationships, e.g., the database classes, the inheritance, class composition and method nesting hierarchies between them. The conceptual design is the process by which the enterprise is studied to determine entity types like class types and relationships among these entities. The second input consists of the access pattern information, which shows the queries accessing database classes and how frequently they access them. The output from the distributed design process is a set of local conceptual schemas (LCSs) [OV91]. LCSs describe database entities at each local site. With the bottom-up approach, since a number of databases already exist, the design task involves integrating them into one global database.

There are different ways to perform fragmentation and allocation. Apers in [Ape88] considers data fragmentation and allocation as a single problem. Another approach is fragmentation followed by an allocation phase. The allocation scheme distributes fragments in such a way that overall system throughput is maximized while minimizing the overall cost of maintaining the system. The work in this thesis treats fragmentation and allocation of database objects as separate design steps.

An object base is a collection of database objects classified into a number of classes. Objects with common attributes and methods belong to the same class. An object based system supports an object oriented data structure including features of encapsulation and inheritance [EB95a]. Encapsulation bundles together the methods and the attributes of an object. Inheritance allows reuse and incremental re-definition of new classes in terms of existing ones. A distributed object based system is a collection of local object bases located at different local sites, interconnected by a communication network [EB94].

With a relational database system, the locality of accesses of database applications is usually defined not on the entire relations but on their subsets because the application views are subsets of relations. Relation instances are essentially tables, and there are three methods for dividing (fragmenting) a table into smaller ones, namely, horizontal, vertical and hybrid fragmentation [OV91]. Horizontal fragmentation partitions a relation along its tuples such that each horizontal fragment has a subset of the tuples in the relation. Vertical fragmentation partitions a relation along its attributes such that each vertical fragment has a subset of attributes in the relation. Hybrid fragmentation partitions a relation along its tuples and then along its attributes or partitions a relation along its attributes and then along its tuples such that each hybrid fragment belongs to only one horizontal and one vertical fragment. In the case of object databases, horizontal fragmentation keeps subsets of instance objects of classes in horizontal fragments, while vertical fragments contain subsets of class attributes and methods, hybrid fragments contain grid of only one horizontal fragment and only one vertical fragment. Fragmentation of data improves overall performance by reducing the amount of data that needs to be moved between sites or replicated in order to answer user queries. Thus, distributed databases benefit greatly from fragmentation because of the following reasons:

(1) Fragmentation reduces the amount of irrelevant data accessed by applications.

(2) It allows greater concurrency and parallel execution of a single query.

(3) It reduces the amount of data transferred when migration is required.

(4) It allows replicating fragments rather than replicating the entire class.

Existing algorithms dealing with fragmentation obtain their inputs only from static requirements analysis. Major changes in a domain would entail a full re-analysis of the system and re-running of the distributed design algorithms. For reduced cost of system input, re-analysis and increased user confidence in distributed design techniques, a dynamic fragmentation system is required. This thesis work proposes an approach for measuring the system performance of a distributed object based system that is horizontally fragmented. Changes in the global conceptual schema and access pattern information are periodically monitored and used to assess the performance of the system based on the current horizontal fragments. The system performance is measured in terms of (1) the amount of irrelevant local access, which yields a measure of the local processing cost of transactions due to irrelevant instance objects of fragments, and (2) the amount of relevant remote access due to relevant instance objects of fragments that are accessed remotely by transactions. The thesis uses this measure to determine system performance threshold and when to trigger a re-fragmentation.

In a relational database system, transactions often involve operations on several relations, possibly located at different sites. However, object oriented systems include modelling of application behaviour and relationships between objects, which introduces more complexities. The object data model is capable of capturing the inheritance, class composition and method nesting relationships or hierarchies between objects. The object data model is discussed in the next section. The objective function defined in [CMV+93] for measuring the performance of relational vertical fragmentation is reviewed before presenting the objective of this thesis. This chapter ends with a presentation of the outline of the thesis document.

1.1 The Object Data model

A distributed object based system (DOBS) consists of a set of local object bases distributed at different sites, interconnected by a communication network. Each local object base contains fragments (horizontal in this case) of classes in the database as well as the relationships between these class fragments. A central data dictionary is used to describe the relationship between the local object bases at different sites. The general architecture of DOBS is available in [OV91].

The fundamental terms of an OODB include class, inheritance, and object identifier (Oid). An object represents an encapsulation of the attributes that describe data and operation or methods that manipulate the data. A class describes a set of objects/instances with similar characteristics, attributes and a set of procedural methods that embody the behaviour of its objects. Classes are established into an inheritance hierarchy, in which a subclass inherits the attributes and methods defined in the superclass for its objects. Objects are uniquely distinguished by their object identifiers (Oids). In addition, there is another useful hierarchy in OODB, which is called the class composition hierarchy. It captures the "is-part-of" relationships between a parent class and its component classes. A composite object (CO) is defined as an object with a set of non-atomic attributes, each of which refers to one or more component objects of the CO; the hierarchy of classes is called a class composition hierarchy. For example, the attribute *student* in object of class *Prof* has its domain as class *Student*. Furthermore, the methods defined in a class can be classified into simple methods and complex methods. Simple methods are those that do not invoke other methods of other classes. Complex methods are those that can invoke methods of other classes. Complex methods invocations form a method nesting hierarchy. For example, there are three classes, *Dept*, *Student* and *Prof*. Class *Dept* has attributes dname, profs and students; methods number-of-profs() and course-offer(). Class *Prof* has attributes status, dept, salary and students; method course-taught(). Class *Student* has attributes name and gpa; method owing(). Here, simple attributes are dname, status, salary, name, gpa. Complex attributes are profs, students and dept. Simple methods are number-of-profs(), course-taught() and owing(). Complex method is course-offer() which invokes method course-taught() in class *Prof*.

The object base is a collection of encapsulated objects. To form an encapsulated object we need to bundle the attributes and methods together. Objects belonging to a class have all attributes and methods that manipulate them in common. Inheritance allows objects in different classes but on the same inheritance path to share attributes and methods. The overall inheritance hierarchy information of an object base is expressed in a class inheritance hierarchy that has a 'root class' as the ancestor of every other class. Each class has a class identifier (CI) which contains objects that are identified by Oids. Every class has a set of attributes (A), method (M) and instance objects (I). A class can be represented as a tuple <CI, A, M, I>. For example, Person = {Csin, {a.name, a.age, a.address}, {m.whatlast, m.daysold, m.newaddr}, { I_1 {501-156-171, John James, 30, Winnipeg}}}. Each horizontal fragment of a class contains its class identifier, all attributes and methods of the class but only some instance objects ($I' \subseteq I$) of the class. Thus, it can be represented by (CI, A, M, I'). Each vertical fragment of a class contains its class identifier, all of its instance objects for only some of its attributes ($A' \subseteq A$) and some of its methods ($M' \subseteq M$). Thus, it can be represented by (CI, A', M', I). Each hybrid fragment of a class contains its class identifier, some of its instance objects ($I' \subseteq I$) for only some of its attributes ($A' \subseteq A$) and some of its methods ($M' \subseteq M$). Thus, it can be represented by (CI, A', M', I').

1.2 Partition evaluation function

The partition evaluation function is an objective function used to evaluate the "goodness" of a particular partitioning scheme. It can be used to measure the performance of a dynamic object horizontal fragmentation system whenever a major change in access pattern information or global conceptual schema occurs. However, defining the function suitable for object horizontal fragments and when to measure the performance are some of the contributions of this thesis.

In [CMV+93], Chakravarthy *et al.* presents a general approach to the vertical partitioning problem for relations. The paper proposes an objective function to evaluate any relational vertical partitioning algorithm based on an attribute usage matrix. The work formulates an objective function quantitatively using the intuitive execution of transactions and measures the performance of the system with a value called partition evaluator (PE). Finally, the partition evaluation (PE) has the flexibility of incorporating additional design information such as type of queries (retrieval/update), transmission cost, and replication. Replication of data provides local availability and reduces the costs of processing transactions. However, replication increases the expense of maintaining the consistency of data. The objective function proposed for measuring the performance of a vertically fragmented relational database is given as PE= $E_L^2 + E_R^2$ where the component E_L^2 represents the cost due to access of irrelevant local attributes and component E_R^2 represents the cost due to access of relevant remote attributes. The formulas for both E_L^2 and E_R^2 are given next.

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

$$E_R^2 = \sum_{t=1}^{T} \sigma_{i=1}^{M} \sum_{k \neq i} [q_t^2 * |R_{itk}| * |R_{itk}|/n_{itk}^{rem}]$$

Here,

M: the number of the fragments of a partition.

T: Total number of transactions that are under consideration.

q_t: Frequency of transaction t for t=1, 2, ..., T.

n_i: Number of attributes in fragment i.

n_{itk}^{rem}: Total number of attributes that are in fragment k accessed remotely with respect to fragment i by transaction t.

$|S_{it}|$: Number of attributes in fragment i that the transaction t accesses.

$|R_{itk}|$: Number of relevant attributes in fragment k accessed remotely with respect to fragment i by transaction t.

σ: An operator that computes either an average, minimum or maximum relevant remote attribute cost over all i.

Ezeife in [EB95b][EB98] extends the application of the PE function to distributed vertically fragmented object classes by including heuristics to capture the three object relationships of inheritance, class composition and method nesting. Detailed discussion is given in chapter 2.

1.3 The objective of this thesis

This thesis contributes to the objective of implementing a dynamic object horizontal fragmentation by first designing an objective function capable of measuring the performance of a distributed horizontally fragmented object based system. Measuring system performance is necessary for determining when to trigger a re-fragmentation. Thus, the thesis contributions include:

1) Designing an architecture for dynamic object horizontal fragmentation. Including four new major algorithms, namely, object horizontal partition evaluator (OHPE), get application instance set algorithm (GetAISet), compute threshold (OO threshold) algorithm and object monitor algorithm (OO monitor).
2) Designing a partition evaluation function and algorithm (OHPE) and GetAISet algorithm used for measuring the performance of the object horizontal fragmentation schemes.
3) Providing a system threshold PE value that is periodically compared with current PE value. If the current PE value at any time goes beyond the determined system threshold PE value, a re-fragmentation is triggered. Designing the algorithm for computing system threshold (OO threshold) is one contribution.
4) Providing a monitor module algorithm (OO monitor) that monitors changes in access frequency of applications, application queries, class inheritance, class composition, method nesting hierarchies and numbers of instance objects. Any change greater than a predefined average change will cause a current PE value check.
5) Discussing how the function can be used to compare object horizontal fragmentation algorithms to determine which algorithm is most suitable for an application domain.
6) Some of the proposed algorithms are also implemented.

These contributions are also presented in [EZ99a] and [EZ99b].

1.4 Outline of the Thesis

The organization of the rest of this document is as follows. Chapter 2 reviews previous work on distributed database design and the extended work on object vertical partition evaluator. Chapter 3 presents object dynamic horizontal fragmentation architecture, the partition evaluation function and algorithm (OHPE) and GetAISet algorithm used to measure the performance of horizontal fragmentation. The monitor algorithm (OO monitor) and the compute threshold algorithm (OO threshold) are presented in chapter 4. Chapter 5 discusses implementations and demonstrates the working of the system with an elaborate example while chapter 6 presents conclusions and future work.

2 Previous/Related Work

This chapter reviews earlier works on distributed database design, object-oriented database systems and works on distributed object systems and/or client/server object-oriented databases. Section 2.1 reviews previous work on fragmentation in relational database systems. Section 2.2 reviews some existing OO systems. Section 2.3 reviews previous work on distributed object systems and client/server object-oriented databases. Section 2.4 reviews previous work on fragmentation in object-oriented systems and section 2.5 reviews extended application of partition evaluation function developed by Chakravarthy *et al.* [CMV+93] in measuring object vertical fragmentation by Ezeife *et al.* [EB98].

2.1 Distributed Design (Relation)

This section reviews three types of fragmentation schemes in relational databases, namely, horizontal fragmentation, vertical fragmentation and hybrid fragmentation.

2.1.1 Horizontal Fragmentation

Ceri, Negri and Pelagatti [CNP82] define applications in terms of boolean predicates and use access pattern information to achieve the design. In their approach, predicates are collected into sets of minterms that form the horizontal fragments. Predicates are generated from user queries. For example, an application groups graduate students according to their area of specialization that is determined by the name of their supervisor. This means a method is defined to list all graduate students supervised by a named professor. The method used is defined on a class called *Grad* and the predicates generated from this method if the only professors are John West and Mary Smith are { P_1: supervisor = " Prof John West", P_2: supervisor = " Prof Mary Smith"}. The set of minterms from these predicates are, MM_1 : supervisor = "Prof John West" $\wedge$ supervisor $\neq$ "Prof Mary Smith"; MM_2: supervisor $\neq$ "Prof John West" $\wedge$ supervisor = "Prof Mary

Smith". They demonstrate that the main optimization needed for horizontal fragmentation is the number of accesses performed by application programs to different parts of data. They also present a method to determine the access parameters and formulate the optimal partitioning problem for several application environments.

Ceri *et al.* in [CNW83] uses directed links drawn between relations. The relation at the tail of a link is called the owner of the link and the relation at the head is called the member. The link between the owner and the member relations is an equijoin implemented as a semijoin. Primary fragmentation is performed on all owner relations and derived fragmentation is performed on all member relations of links. The derived fragmentation is defined on all member relations of a link according to a selection operation specified on its owner relation. For example, owner (L) = S, member (L) = E

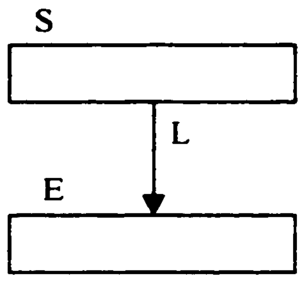


Figure 1: Link graph

E

ENO	ENAME	TITLE
E1	J. Lee	DBA
E2	M. Chu	Sr. Eng.
E3	B. Smith	Asst.
E4	L. Doe	Programmer
E5	R. Miller	P. Leader
E6	A. Jones	Programmer

Table 1: Employee

S

TITLE	SALARY
DBA	95000
Programmer	38000
Asst.	30000
P. Leader	99000
Sr. Eng.	57000

Table 2: Salary

We can group employees into two groups HE1 and HE2 according to their salary. (<50000 or >= 50000). The two horizontal fragments HE1 and HE2 are:

HE1

ENO	ENAME	TITLE
E3	B. Smith	Asst.
E4	L. Doe	Programmer
E6	A. Jones	Programmer

HE2

ENO	ENAME	TITLE
E1	J. Lee	DBA
E2	M. Chu	Sr. Eng.
E5	R. Miller	P. Leader

Table 3: Fragments of Employee

Özsu and Valduriez [OV91] define the database information needed for horizontal fragmentation of the universal relation. They address the techniques that can be used for distributed database design with special emphasis on the fragmentation and allocation issues and focus on distribution of data. They define a simple predicate to be in the form $p_j : A_i \theta$ Value (e.g., PNAME = "Administration"). Given a set of simple predicates $Pr_i = \{p_{i1}, p_{i2}, \ldots, p_{im}\}$, the set of minterm predicates $M_i = \{ m_{i1}, m_{i2}, \ldots, m_{iz} \}$ is defined as $M_i = \{ m_{ij} \mid m_{ij} = \wedge\, p^{*}_{ik} \}, p_{ik} \in Pr_i, 1 \le k \le m, 1 \le j \le z$ where $p^{*}_{ik} = p_{ik}$ or $p^{*}_{ik} = \neg p_{ik}$. A horizontal fragment R_i of relation R consists of all the tuples of R that satisfy a minterm predicate m_i. The complete and minimal predicates are used to generate minterm predicates. For a set of simple predicates Pr, completeness means the probability of access by every application to any tuples belonging to any minterms fragment is the same. Pr is called minimal if all of its predicates are relevant, which means, there exists applications that would access the resulting fragments differently. They propose an iterative algorithm named COM_MIN that would generate a complete and minimal set of predicates Pr' given a set of simple predicates Pr. The algorithm begins by finding a predicate that is relevant and partitions the input relation, and then derives the set of minterm predicates that can be defined on the predicates in set Pr'. Finally it eliminates some minterm fragments that may be meaningless. For example, if $Pr' = \{p_1, p_2\}$ where p_1: att=value_1, p_2: att=value_2 and domain of att is {value_1, value_2}. The algorithm generates the following minterm predicates. m_1: (att=value_1) $\wedge$ (att=value_2), m_2: (att=value_1) $\wedge$ $\neg$(att=value_2), m_3:$\neg$(att=value_1) $\wedge$(att=value_2), m_4: $\neg$(att=value_1) $\wedge$ $\neg$(att=value_2). Minterm predicates m_1 and m_4 are eliminated because they are contradictory to the implications from the knowledge of the data.

Shin and Iran [SI91] address a knowledge based approach to fragment relations horizontally, which allows revision of typical user queries into more precise form. User reference clusters (URCs) are estimated from user queries but are refined using semantic knowledge of the relations. User access and usage patterns of data at each site are considered the predominant factor. However, accurate user reference cluster information may not be obtained by brute force analysis of the typical user queries at each site [SI91].

2.1.2 Vertical Fragmentation

Hoffer and Severance [HS75] propose an algorithm that clusters attributes of database entities based on their affinities. Attributes accessed together by applications have high affinity. McCormick *et al.* [MS72] defines a Bond Energy Algorithm that is used to form these attribute clusters. Navathe *et al.* [NCW84] extends Hoffer's work by defining algorithms for grouping attributes into overlapping and nonoverlapping fragments. They apply vertical partitioning to three database types--distributed database, one memory level database and database 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 [CY87] define an algorithm that optimizes this work. It obtains an optimal binary partitioning for relational databases. They use knowledge of physical factors to decrease the number of disk accesses. Further refinement is accomplished by applying the binary vertical partitioning algorithm iteratively [NCW84]. Navathe and Ra [NR89] define 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 [OV91] examine earlier work on vertical partitioning for distributed databases using the access frequency information and the Bond Energy Algorithm which 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.

Chakravarthy *et al.* [CMV+93] discusses that earlier algorithms for vertical partitioning are ad hoc, so that they define an objective function called the Partition Evaluator to determine the "goodness" of the partitions generated by various algorithms. The Partition Evaluator has two terms, 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 [MS72], binary vertical partition [NCW84], Ra's graphical algorithm [NR89] and an exhaustive enumeration algorithm.

2.1.3 Hybrid Fragmentation

Özsu and Valduriez [OV91] discuss that nesting of horizontal and vertical fragmentation in any order produces hybrid fragmentation of relations. Pernul, Karlapalem and Navathe [PKN91] point out 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 [CNP82][NCW84]. Their methodology involves decomposing relations into a set of disjoint fragments based on views where one or more fragments represent 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.

Karlapalem *et al.* [NKR95] proposes a methodology for generating a mixed fragmentation scheme for the initial distributed database design phase by extending work in [NR89]. A graph theoretic algorithm clusters a set of attributes and predicates into a set of vertical and horizontal fragments. For the case of vertical fragmentation, the algorithm starts from the attribute affinity matrix by considering it as a complete graph called the "affinity graph" in which an edge value represents the affinity between two attributes, and then forms a linearly connected spanning tree. It is constructed by including one edge at a time such that only the "first" and the "last" node of the tree would be considered for inclusion. In the attribute affinity matrix, element (i, j) equals the total number of accesses of transactions referencing both attributes i and j. The algorithm then forms an affinity circle in the spanning tree by including the edges of high affinity value around the nodes and grows these circles as large as possible. After all the cycles are formed, fragments are generated through cutting the cycles apart at "Cut edges". "Cut edges" are those edges in the spanning tree that do not meet the condition of "extension of a cycle" which refer to a cycle being extended by pivoting at the cycle node. A cycle node is that node of the cycle completing edge, which was selected earlier. The same algorithm may be applied to horizontal fragmentation. By applying these two algorithms together, they form a grid on a relation that gives rise to a set of grid cells. Each grid cell belongs to only one horizontal and only one vertical fragment. Then they merge these grid cells to form a set of mixed fragments that minimizes the numbers of disk accesses needed to process the transactions. They consider only the sets of grid cells that are accessed by at least one transaction. They check whether they can be merged to form a regular fragment for every such set. If so, the sets of grid cells can be merged to form a mixed fragment potentially.

2.2 Object Oriented Database Systems

This section presents a review of existing object base systems including those that either support a client/server architecture or support distribution. We first review object oriented database systems Shore [CDN+94], O_2 [Deu91], STARBUST [LHC+90] [LLP+91] and POSTGRES [SRH90][SK91].

Shore [CDN+94] has a layered architecture that allows users to choose the level of support appropriate for a particular application. The Shore Storage Manager (SM) is a persistent object storage engine that supports creation of persistent files of records. Each record can be any size, with efficient storage of records from a few bytes to megabytes or larger. Records may be retrieved by object identifier or by scanning files. The SM provides full concurrency control and recovery (the so-called ACID properties) with two-phase locking and write-ahead logging. It also provides robust implementations of btrees and rtrees and record access through logical object identifiers. The SM, which is used as a library, is designed to create value-added servers tailored to specific applications. The

Shore Value-Added Server (SVAS) builds on the functionality of the SM to provide typed objects, a Unix-like directory namespace, access control, and a client-server architecture supporting object-level caching, transactional semantics, and security at the server boundary. The NFS value-added server fully implements the standard NFS (Network File Server) protocol, allowing legacy applications to access Shore objects as if they were Unix files. The Shore Data Language (SDL), which is based on the Object Database Management Group (ODMG) ODL language, supports language-independent description of object-oriented data types. The SDL compiler compiles definitions into type objects stored in the database and C++ language stubs. The combination of the SDL compiler and an extensive run-time library allows programmers to write applications that manipulate objects through type-safe object references. The library takes care of fetching objects on demand to an LRU client-level object cache, flushing changes to the server on transaction commit, and swizzling and unswizzling references as necessary. Shore serves as the basis of other database projects, including Paradise at the University of Wisconsin and PREDATOR at Cornell University.

O. Deux *et al.* [Deu91] presents an object-oriented database management system O_2 with the ideas of merging user interface, programming language and database technologies, using object-oriented technologies and conforming to standards. Structured and multimedia objects are stored in the O_2 Engine which handles disk management, distribution transaction management, concurrency, recovery, security and data administration. Metadata is managed through objects at high level and objects communicate through message passing. O_2 provides strong support for encapsulation and supports both single and multiple inheritance except for attribute value inheritance.

POSTGRES [SRH90][SK91] has object and rule management capabilities besides data management in a traditional data manager. It supports a data model that is relational but extended with abstract data types, classes, inheritance and functions. It supports both single and multiple inheritance and uses a system catalog for managing metadata. POSTQUEL is the POSTGRES query language. It is a set oriented query language that allows queries to be nested and has operators that have sets of instances as operands. For example, retrieve (DEPT.dname) where DEPT.floor NOT-IN {D.floor from D in DEPT where D.dname != DEPT.dname}. POSTGRES has a no-overwrite storage manager. Using this technique, the old record remains in the database whenever an update occurs, and serves the purpose normally performed by a write-ahead log.

STARBUST [LHC+90][LLP+91] extends relational database system to capture object-oriented and knowledge-based system features. It consists of a query language Corona and a data manager Core. Corona compiles queries in an extended version of the SQL language into calls on the underlying Core services to fetch and modify data. Parsing, query rewrite optimization, plan optimization, query refinement and query evaluations are five main phases in Corona. Core stores and retrieves data, maintains access paths to the data, and ensures transaction consistency and data integrity in the face of multiple users and failures. Vectors of functions and values have been used extensively in Starbust as a mechanism to permit easy extensibility. Rules provide a high-level and flexible interface for modifying the DBMS's logic. The only way to access objects and their components is through operations defined on the table. It supports both single, multiple and attribute value inheritance. STARBUST is not a distributed system and metadata is managed with a table.

2.3 Early Work on Distributed and Client/Server OODBS

This section presents a review of existing object base systems that are either distributed or support a client /server architecture. The OODB's reviewed are ITASCA [Ibe95], ENCORE [HZ87], GOBLIN [KPV94], THOR [LAD+96], and EOS [OL94].

ITASCA [Ibe95] is a distributed active object database management system developed by Itasca Systems. ITASCA is an extension of ORION-2. ORION2 has a distributed, object-oriented architecture for multi-client, multi-server architecture. The distributed environment of ITASCA allows shared and private databases and the shared database is distributed across workstations sites in the network. An ITASCA server controls the partitioning of the shared database at each site. Private databases allow private data not shared with other users of the database. The schema is stored redundantly at each site but each instance of data is stored at one site. There is no central data server and no central name server. The user or application does not need to know the location of a targeted object in the database. Data location 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. Thus, the only thing that needs to move among sites at execution time is data. There is an object directory at each site that maps unique identifications to physical locations on either the shared or private database partitions of the local site.

ENCORE [HZ87] contains a component named ObServer that reads and writes chunks of memory from secondary storage. The type level communicates with the Server through the UNIX remote procedure call (RPC) mechanism. ObServer has two functions: (1) manages chunks of memory in secondary storage with a unique identifier (UID) attached to each chunk, and (2) maintains 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 server can reside on different machines. The ENCORE module uses the object server as a backend. When two different processes on two different machines are using the ENCORE database, each machine has a complete copy of ENCORE. A binder process provides a client with a connection to the desired database. When a client wants to access a database, it requests 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. To improve performance, groups of related objects are clustered as a segment stored on disk. 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. Therefore, the segments collectively provide a partition of the objects within a database and every database object is contained in at least one segment.

GOBLIN [KPV94] is a distributed OODBS developed in Amsterdam. It runs in an environment consisting of a network of workstations with memory-resident databases. GOBLIN use a three-level directory hierarchy storage scheme 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, a Redistribution Administration Table (RAT) including the partitioning and distribution information administrates the mapping from schema to database instances. The storage management functions consist 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 (CID) to the set of object ids (OID) represent instance objects of this class. The counterpart of the OID or CID at the physical level is called the physical identifier (PID). Schema functions have associate path functions. For example, the schema function P_{person}: CID $\rightarrow$ 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 storage functions capture the mapping between the extensional and intensional layers, 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 Binary Association Tables administrate the path function definitions, the Class Administration Table captures the schema functions, and the Redistribution Administration Table captures the storage functions. 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 [LAD+96] is a distributed system developed at Massachusetts Institute of Technology (MIT), Laboratory for Computer Science. In THOR objects are 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 that are distributed. The architecture of THOR 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 clients, 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 [OL94] is an object-oriented programming environment for distributed systems developed at INTIS, France. Leos is the language of EOS. It is an object-oriented programming language that provides transparency for both distribution and persistence. EOS provides a flat, virtual and garbage collected object space that needs to be paginated and distributed. Object grouping is used to minimize the 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 that 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 class that holds a systems 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 discusses previous work on horizontal fragmentation in the object-oriented database system followed by a review of early work on vertical and hybrid fragmentation.

2.4.1 Horizontal Fragmentation

Recent work on fragmentation and allocation in the relational and object databases have been conducted by Ezeife and Barker [EB95a][EB98], Savonnet and Yetongnon [SY95] [STY98], Bellatreche *et al.* [BKS97], Ravat and Zurfluh [RZ95], Özsu and Valduriez [OV91], Shin and Irani [SI91], G.Wiederhold and J.Dou [NCW84], Ceri *et al.* [CNP82].

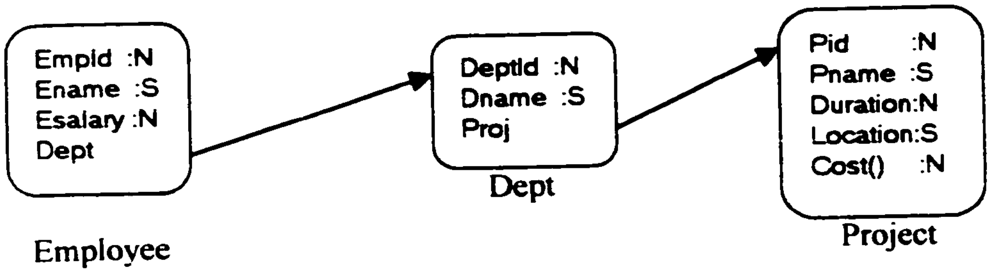
Karlapalem *et al.* [KNM92] identifies some of the fragmentation issues in object bases as techniques for handling inheritance, class composition and method nesting hierarchies. Further, they argue that a precise definition of the processing semantics of the applications is necessary. They do not propose solutions for horizontally fragmenting class objects but argue that techniques used by Navathe *et al.* [NKR95] for horizontal fragmentation could be applied.

Ezeife and Barker [EB95a] present a set of horizontal fragmentation algorithms (OOHF), for four types of class models including those 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. The algorithm for class consisting of simple attributes and methods is summarized next. The assumptions made are, first, objects of a subclass contain only pointers to objects of its superclasses that are logically part of them. Secondly, there are no cycles in the dependency graphs discussed later. Primary horizontal fragmentation is the partitioning of a class based on only applications accessing the class. Derived horizontal fragmentation is the partitioning of a class arising from the fragmentation of its subclasses. The input to the algorithm consists of sets of applications or user queries, the set of database classes and the inheritance hierarchy information. The output we expect from this algorithm is a set of horizontal fragments for all classes in the database. First, the algorithm defines the link graph for all classes in the database using the inheritance hierarchy to capture the class lattice information. From the inheritance hierarchy, starting from the leaf classes (most specialized subclasses) which are at the head of links and called owners of the links, the appropriate links are connected to their immediate superclasses and the process repeats until the root class is encountered. Secondly, the algorithm defines primary horizontal fragments of all classes. A primary horizontal fragment of a class is a set of objects of this class accessed together by applications that run on this class only. Each user query is a sequence of method invocations on objects and each method of a class is represented as a set of predicates defined on values of attributes of that class. Thirdly, the algorithm defines derived horizontal fragmentation on member classes. Derived fragmentation is defined on member classes of links according to its owner classes. It partitions a member class according to the fragmentation of its owner class and defines the resulting fragment on the attributes and methods of member class only. A derived fragment of a member class based on a primary fragment of an owner class is a set of member class objects accessed by applications running on the owner class. For example, class *Person* is a superclass (member) of class *Student* (owner) and from applications running on *Person*, two primary horizontal fragments are created as Frag1= {person1, person4, person5}, Frag2={person2, person3, person6}. Class *Student* has the following two primary horizontal fragments: Frag1(*Student*)={student3, student4}, Frag2(*Student*)={student1, student2}. Since class *Person* is a member class of *Student*, a derived fragmentation of class *Person* arises from primary fragments of its owner class *Student*. The persons corresponding to student3 and student4 in the first primary fragment of *Student* gives the first derived fragment of *Person*: these are person4 and person5. Finally, the algorithm combines primary and derived fragments. The final horizontal fragments of a class is composed of objects accessed by both applications running on this class and those running on its owner classes. The choice of the primary fragment is the one that has the highest affinity with the current derived fragment by maximizing the affinity measure aff(F^d, F_i^p) which is a measure of how frequently objects of these two fragments are needed together by applications. If a class has no primary fragments, the final horizontal fragments are made from derived fragments and instance objects not contained in a derived fragment are placed in one of the fragments. For more complex class models consisting of complex attributes and complex methods, the OOHF iteratively incorporates the class composition dependence and method nesting dependence through corresponding link graphs and definition of derived fragments of owner classes of links.

Marinette Savonnet and Kokou Yetongnon [SY95] present a qualitative approach for class fragmentation in distributed object oriented databases. The algorithm starts from the collection of partition trees that come from the class dependency graph (CDG). A class dependency graph captures the composition (part of) links, the generalization (is-a) links and method invocation links between object classes. A set of partition trees is called a partition forest that represents a covering set of class dependency graph. The algorithm constructs from a class dependency graph of the object classes, a set of partition trees where each class of the original OO schema belongs to only one partition tree. The horizontal fragmentation of classes of each partition tree is achieved by fragmenting the root class and inducing the resulting fragments on the member classes. The fragmentation of the member classes is called derived fragmentation. The class dependency graph has three structural links between object classes: composition, generalization and method invocation links. A composition link between a class C_i and a class C_j represents the fact that objects of C_i are defined in terms of one or more component objects belonging to C_j. To improve the access of composite objects, fragments of the composite class C_i must be grouped with the corresponding fragments of C_j. A generalization/specialization link for this algorithm is an object stored in the most specialized class. A complex method M_i of a class C_i invokes one or more methods M_j of other classes C_j. Each class of the original schema appears in only one partition tree of the forest that is a collection of partition trees. A partition tree is constructed from the CDG, by selecting a root node R (object class) and by removing from the graph, the connected subgraph rooted at R. At each step of the construction of a partition forest, the node with highest weight is selected as the root of a partition tree. This process is repeated until the CDG is empty. In summary, the fragmentation of the classes of each partition tree is accomplished through the following steps. First, fragmenting the root class (primary fragments). Second, inducing the resulting fragments on member classes (derived fragments). Let DL be a dependency link; let source (DL)=S and target (DL)=T be the initial and terminal classes of L respectively; if DL represents generalization link then $<S_i>$= Select from <S> where <predicates of T_i>. If DL represents a composition link then $<S_i>$=<S>↑$<T_i>$, (↑) generates the instance objects of source class S contained in instance objects of the fragments of the target class T. If DL is originated from a method invocation link then $<S_i>$ = orefm (<S>, $<T_i>$). The operation (orefm) generates the set of instance objects of source class S needed by complex methods of class fragment T_i of target class T. Non disjoint fragments of the subclass may be produced by the derived fragmentation. An additional "dangling" fragment may also be produced through the derived fragmentation process, which contains all instances that do not participate in the relationship on which the fragmentation is based. By allocating each root fragment and the corresponding derived fragments of member classes to the same site of the distributed system, the allocation of fragments of the classes of a partition tree is done.

Yetongnon *et al.* [STY98] demonstrates Fragtique—a graph based methodology for the distribution design of Object Oriented databases by using both conceptual information represented by structural links in a schema and quantitative information representing method usage or frequency of application queries. Their Fragtique OO distribution methodology has the following steps. First, they select a set of methods that provide a foundation for the partitioning strategy and they formalize the idea that a method can be executed in parallel with minimum overhead cost with the property of local sufficiency. A method is locally sufficient if it is possible to divide it into partial methods for parallel execution before execution starts. Second, the class dependency graph (CDG) is partitioned into a set of partition trees by identifying root classes and by extracting from the subgraphs connected to the root based on the selected method in the first step. A partition tree represents a group of co-referenced classes that must be partitioned together. Third, the classes of a partition tree are partitioned interdependently by using primary fragmentation techniques for the root class and by deriving the primary root class fragments on non-root classes. Finally, flow measures are defined for evaluating the quality of the results of the fragmentation process and for allocating class fragments among sites.

Bellatreche *et al.* [BKS97] proposes two algorithms for horizontally fragmenting object classes, primary and derived algorithms. A query in OODBs has three components—target clause, range clause and qualification clause. The target clause specifies some of the attributes of an object or the complete object of the class that is returned. The range clause contains the declaration of all object variables that are used in the qualification clause. The qualification clause defines a boolean combination of predicates by using the logical connectives ($\wedge, \vee, \neg$). For example, the query q_a for retrieving the name of all parts with price greater than \$100 and located at "New warehouse" is formulated as: q_a={v.pname; v/Part; v.price()>\$100 $\wedge$ v.location = "New warehouse"}. A path represents a branch in a class composition hierarchy and is specified by $C_1 . A_1 . A_2 \ldots A_n (n \geq 1)$ where C_1 is a class in the database schema, A_1 is an attribute of class C_1, A_i is an attribute of class C_i such that C_i is the domain of the attribute A_{i-1} of class C_{i-1}, ($1 < i \leq n$). A component predicate is a predicate defined on a path. A component query is a query that involves component predicates. Primary horizontal fragmentation of a class is performed using predicates of queries accessing this class. The primary horizontal class partition algorithm has seven steps. First, it builds the predicate usage matrix of class C, which contains queries as rows and predicates as columns. Second, it constructs the predicate affinity matrix from the predicate usage matrix. The element of predicate affinity matrix can either be numerical or non numerical representing the sum of the frequencies of queries which access two predicates simultaneously, the implication or similarity between two predicates, respectively. Third, it groups the predicates to form clusters where the predicates in each cluster have high affinity. Fourth, it optimizes the predicates contained in each subset by using predicate implication. Fifth, it splits every subset got from step 4 to include the missing part of simple predicates. Sixth, it links the predicates in each subset by OR or AND connector to generate the class fragment. Finally it refines these fragments to obtain nonoverlapping fragments if the algorithm gives rise to overlapping fragments. An example of a class composition hierarchy is given in Figure 2 where *Employee* has one complex attribute *Dept* and three simple attributes: EmpId, Ename and Esalary. Class *Project* will be fragmented according to query Q_1 to Q_4 given below.



Legend:

⟶ Aggregation relationship

S: String

N: Numeric

Figure 2: The Class Composition hierarchies of class Employee

Q_1:{v.Pid; v/Project; v.Duration $\le 3 \wedge$ v.Cost() >7000}. Q_2:{v.Pname; ; v/Project; v.Duration $\le 4 \wedge$ v.Cost() >7000}. Q_3:{v.Dname; v/Department; v.Proj.Duration $= 2 \wedge$ v.Proj.Cost() $\le 7000 \wedge$ v.Proj.Location = "HongKong"}. Q_4:{v.Esalary; v/Employee; 5 $\le$v.Dept.Proj.Duration $\le 6 \wedge$ v.Dept.Proj.Cost() $\le$ 7000 }. There are seven predicates in the qualification clause of these queries. P_1: Duration ≤ 3, P_2: Duration ≤ 4, P_3: Duration = 2, P_4: 5 $\le$Duration ≤ 6, P_5: Cost() >7000, P_6: Cost() $\le$ 7000, P_7: Location = "HongKong". The access frequency of Q_1 is 20, Q_2 is 35, Q_3 is 30 and Q_4 is 15. So we get the predicate usage matrix as Figure 3.

$$\begin{array}{c|cccccccc} & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 & \text{acc} \\ \hline Q_1 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 20 \\ Q_2 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 35 \\ Q_3 & 0 & 0 & 1 & 0 & 0 & 1 & 1 & 30 \\ Q_4 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 15 \end{array}$$

Figure 3: Predicate Usage Matrix

The attributes and methods used by these predicates are Duration, Location and Cost() which will be renamed as a_1, a_2 and m. Next, we calculate the sum of the frequencies of the queries which accesses those two predicates together to get the predicate affinity matrix Figure 4.

$$\begin{array}{c|ccccccc} & P_1 & P_2 & P_3 & P_4 & P_5 & P_6 & P_7 \\ \hline P_1 & 20 & \Rightarrow,* & \Leftarrow & 0 & 20 & 0 & 0 \\ P_2 & \Leftarrow,* & 35 & \Leftarrow & 0 & 35 & 0 & 0 \\ P_3 & \Rightarrow & \Rightarrow & 30 & * & 0 & 30 & 30 \\ P_4 & 0 & 0 & * & 15 & 0 & 15 & 0 \\ P_5 & 20 & 35 & 0 & 0 & 55 & 0 & 0 \\ P_6 & 0 & 0 & 30 & 15 & 0 & 45 & 30 \\ P_7 & 0 & 0 & 30 & 0 & 0 & 30 & 30 \end{array}$$

Figure 4: Predicate Affinity Matrix

We now form clusters in which predicates have high affinity with one another as Figure 5 shows and get three partitions Pa ={ P_1 , P_2, P_5 }, Pb = { P_3, P_6, P_7 }, Pc = { P_4 }.

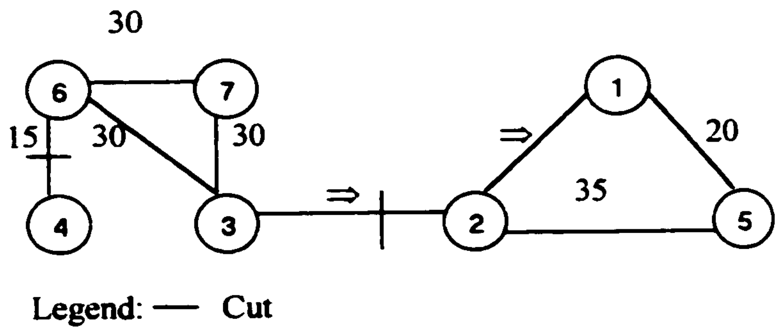


Figure 5: Predicates sets generated by primary Algorithm

We then optimize these subset by using implication to Pa' = { P_2, P_5 }, Pb' ={ P_3, P_6, P_7 }, Pc' = { P_4 }. After that, we go to the fifth step to split the optimized partitions to cover the missing part of a_1, a_2 and m. We get Pa1 = { P_2, P_5, P_7 }, Pb1 ={ P_3, P_6, P_7 }, Pc1 = { P_4, P_5, P_7 }, Pc2 ={ P_4, P_6, P_7 }. Finally, producing the following horizontal fragments:

Project1 given by: (Duration≤4)∧(Cost()>7000) ∧(Location="HongKong").

Project2 given by: (Duration=2)∧(Cost()≤7000) ∧(Location="HongKong").

Project3 given by: (5≤Duration≤6)∧(Cost()>7000) ∧(Location="HongKong").

Project4 given by: (5≤Duration≤6)∧(Cost()≤7000) ∧(Location="HongKong").

Project5 given by: else.

Derived horizontal fragmentation of a class is the partitioning of a class based on the horizontal fragmentation of another class. The fragments defined by the derived horizontal algorithm can overlap, so they have processed a method in order to eliminate overlapping instances. They argue the impact of queries on horizontal fragmentation and analyze the complexity of their solution in terms of the number of queries and predicates used by the fragmentation. The complexity of primary horizontal fragmentation is O($l * n^2$), n and l represent the number of queries and their predicates.

Ravat and Zurfluh [RZ95] introduce predicate affinity that is similar to the attribute affinity and group these predicates with higher affinity in a horizontal fragment using Bond Energy Algorithm [MS72]. The work considers the selection methods and projection/selection methods to create the usage matrix. The selection predicates of these methods are a conjunction or disjunction of simple predicates, not jointure predicates such as att_X > att_Y. The selection method manipulates all the attributes of a non-empty subset of objects of the class. Each row represents a method used for horizontal fragmentation and each column represents a simple predicate in the usage matrix, the "1" at the row "i" and at the column "j" indicates that the method "i" uses the simple predicate "j". The "0" at the row "i" and at the column "j" indicates that the method "i" does not use the simple predicate "j". The predicate affinity matrix can be generated from the predicate usage matrix in a similar way to generate the attribute affinity matrix. The number in the matrix at the row "i" and at the column "j" represents the combined frequency of the methods accessing both predicates i and j. They use the Bond Energy Algorithm to cluster the predicates in the matrix and apply the binary partitioning algorithm to partition the matrix. Every sub matrix contains a set of simple predicates which must be linked using OR or AND to generate a predicate term. Simple predicates with the same attribute are "ORed", and the predicates with different attributes are "ANDed". At last, they add a remaining fragment that is the negation of disjunction of predicates terms. Predicates inclusions and predicates implications could be considered to reduce the number of predicates in every fragment. The fragmentation also consists of four steps of obtaining the predicate usage matrix, taking account of access frequencies to calculate the predicate affinity values, clustering the affinity matrix with Bond Energy Algorithm and determining the fragment.

2.4.2 Vertical Fragmentation

Ezeife and Barker [EB98] present the vertical fragmentation algorithms of four types of class models. The following are the steps for vertically fragmenting a class consisting of simple attributes and methods. First, obtain the method usage and application frequency matrices of the class and its subclasses. It accepts a tree rooted at a class C_i and generates the original method usage matrices for class C_i and descendant classes of C_i on the tree from user applications. Secondly, define the method affinity matrix of the class from the method usage matrix. The method affinity matrix is square and symmetric, element (i, j) represents the sum of the frequencies of queries which access method i and j simultaneously. Thirdly, use the Bond Energy algorithm (BEA) [MS72] to generate the clustered affinity matrix of the class. The algorithm accepts the method affinity matrix as input and permutes its row and columns to generate a clustered affinity matrix in which methods with large affinity values are collected together. Fourthly, generate a modified method usage matrix of the class by including a row to the method usage matrix of a class C_i for every application q_j that accesses a method of this class C_i through any of its descendant class. The *Partition* algorithm [NCW84][OV91] next takes the clustered affinity matrix and modified method usage matrix to generate fragments of the methods. Fifthly, include all attributes of the class accessed by methods of the fragment in each method fragment using method-attribute reference information of the methods in each method fragment. Finally, eliminate overlapping attributes in more than one fragment according to the Attribute Placement Affinity Rule. It places the attribute in the fragment with the highest affinity measure together and removes the overlapping attributes from each other.

Bellatreche *et al.* [BSS96] proposes an algorithm for vertical fragmentation in a model consisting of complex attributes and complex methods. Fragments are generated through forming the class link graph, building the method usage matrix of class, constructing the affinity matrix, regrouping the methods to form clusters where methods in each cluster demonstrate high affinity and including all the attributes referenced by their methods in each fragment. This type of fragmentation facilitates query decomposition, optimization, and parallel treatment for distributed Object Oriented Database Systems. They consider that every method in the object model accesses a set of attributes of the class and group the methods of the class using the technique presented in [NCW84] and extend each fragment of methods to incorporate all the attributes manipulated by these methods. They provide an additional algorithm for eliminating the overlapping fragments problem in order to satisfy the disjointness property.

2.4.3 Hybrid Fragmentation

Fernanda *et al.* [BM98] proposes a new strategy to the fragmentation phase of the distributed design of OODBs that uses heuristics. Their three-step algorithm is given below. The first step analyzes operations and semantic information from an interface module and uses heuristics to decide on the most adequate fragmentation technique (horizontal and/or vertical) for each class in the database schema. The output of this step is a set of classes to be horizontally fragmented and a set of classes to be vertically fragmented. The second step defines the vertical fragments of the classes indicated in the previous step. This step adapts the algorithm developed in [NKR95] to generate vertical fragments. The last step defines the horizontal fragments of classes indicated in the first step. Both primary and derived horizontal fragments of classes must be defined. They define an algorithm that is an extension of the one used in the second step to produce primary fragments so that the same concepts and data structures can be used. Since the class paths are received as inputs, the definition of derived fragmentation is straightforward. The distributed system designer must define derived horizontal fragments of each non root class according to its preceding class in order to group in one horizontal fragment all the objects from different classes referenced by the same navigation operation.

Karlapalem *et al.* [KLV96] proposes method-induced partitioning schemes for object-oriented databases. They focus on articulating the concepts of method induced partition schemes in object-oriented databases by classifying the object behavior embodied by the method and provide a solution for supporting fragmentation transparency. For simple methods (methods that do not call other methods), the method dependency graph (MDG) contains a single node. If methods access value based instance variables (VBIVs) then only vertical or horizontal partitioning scheme are possible alternatives. If methods access object based instance variables (OBIVs) then vertical, horizontal or path partitioning are the possible alternatives. A path partition consists of grouping the objects of all the domain classes that correspond to all the instance variables in the class composition hierarchy rooted at the composite object. The algorithm for generating method induced partitions is described as follow. First, select the initial methods relevant for a method partition. Second, generate the method dependency graph for selected method, which represents the initial method induced partition and forms the basis of the further design steps. Third, for simple or complex methods, if the methods access VBIVs then only vertical or horizontal partitioning scheme are the possible alternatives, if the methods access OBIVs then vertical, horizontal or path partitioning schemes are the possible alternatives. Fourth, redefine the system basic methods for database interaction (e.g., constructor, destructor, read and write) in order to guarantee fragmentation transparency.

2.5 Partition Evaluation Function for Object Vertical Fragmentation (OVPE)

Ezeife and Barker [EB98] extend the relational vertical Partition Evaluator (PE) [CMV+93] to measure the costs of local and remote accesses incurred by the object vertical fragments. The objective of partition evaluator (OVPE) is to measure the total costs of processing all applications at all distributed sites if each access to a data fragment amounts to a unit cost. Every application processing cost consists of its local processing and remote processing costs. It is assumed that there are no data redundancies and following fragmentation, the fragments have been allocated to sites where they are most needed. The first component of PE is the irrelevant local method access cost that minimizes the square error for a fixed number of fragments and assigns a penalty factor whenever irrelevant methods are accessed in a particular fragment. The mean vector V_i for fragment i is given by $V_i = (1/n_i)\sum_{j=1}^{n_i} M_{ij}$ where n_i is the number of methods in fragment i and M_{ij} is the method vector for the jth method in fragment i. The difference vector for method vector M_{ij} is ($M_{ij} - V_i$) and the square error of the fragment i is the sum of squares of the lengths of the difference vectors of all the methods in fragment i given by $e_i^2 = \sum_{j=1}^{n_i} (M_{ij} - V_i)^T (M_{ij} - V_i)$. The square error for all m fragments generated is $E_m^2 = \sum_{i=1}^{M} e_i^2$ and this equation simplifies to $E_m^2 = \sum_{i=1}^{M} \sum_{t=1}^{T} [q_t^2 * |S_{it}|(1-|S_{it}|/n_i^r)]$ where S_{it} is the set of methods contained in fragment i that transaction t accesses. It is empty if t does not need fragment i. A lower value of E_m^2 means a lower penalty cost of irrelevant local method access. The second component of PE computes for a set of applications running on a fragment, the ratio of the number of remote methods to be accessed to the total number of the methods in each of the remote fragments. This is summed over all the fragments and applications to obtain

$E_R^2 = \sum_{t=1}^{T} \min(\sum_{k \neq i} [q_t^2 * |R_{itk}| * |R_{itk}| / n_{itk}^{rem}])$ where q_t is the frequency of application t for t=1, 2, ..., T. R_{itk} is the set of relevant methods of fragment k remotely needed by application t while running on fragment i and $|R_{itk}|$ is its cardinality. n_{itk}^{rem} is the total number of methods in remote fragment k accessed by application t while running on fragment i. Thus PE= $E_m^2 + E_R^2$.

3 Dynamic Horizontal Fragmentation Design

This chapter presents the main contribution of this thesis, architecture of dynamic horizontal fragmentation and its components. Section 3.1 discusses the architecture of dynamic horizontal fragmentation and its components. Section 3.2 presents algorithm GetAISet. Section 3.3 presents the new objective function OHPE and the algorithm. Section 3.4 uses the OHPE to compare the performance of object horizontal fragmentation schemes.

3.1 Architecture of Dynamic Horizontal Fragmentation

Existing object fragmentation algorithms use inputs from static requirement analysis. Major changes in a domain would entail a reanalysis of the systems and rerunning of the distributed design algorithms. In order to make the distributed design process more acceptable by users, fragmenting database objects dynamically is desired. This thesis aims at defining techniques for initiating dynamic horizontal fragmentation of objects in an object oriented database system. It first presents an architecture that triggers dynamic object horizontal fragmentation and is capable of measuring the performance of existing object horizontal fragmentation schemes. Figure 6 shows an initial design of an object dynamic horizontal fragmentation system. The dynamic distributed object horizontal design steps include:

(1) Data inputs of access frequency of queries (AF), application queries (AQ) and the sites where they run, class instance hierarchy (CH), object database (IS), class composition hierarchy (CCH), and method nesting hierarchy (MNH).

(2) Application of object horizontal fragmentation scheme such as OOHF in [EB95a] or Bell's in [BKS97] to generate horizontal fragments that are allocated to distributed sites.

(3) Using the input from step one to compute the application instance set which is the sets of all instance objects accessed by applications running at all sites on each database class through algorithm GetAISet.

(4) Computing the best system PE value using object horizontal partition evaluator (OHPE algorithm) immediately following a fragmentation and allocation, the result of step three is used as input to the OHPE algorithm.

(5) Computing the system performance threshold using the best system PE value and system change variable (S) which is accomplished by algorithm OO threshold. The system change variable measures how frequently the system inputs that may affect performance change. The S is computed as the inverse of the average percentage change in system input, multiplied by 50 to allow a reasonable margin of degrading performance, plus 1. The S value is always greater than 1 to allow the threshold value to always be bigger than the computed best PE value. A threshold value less than the computed best PE value results in the system always triggering a re-fragmentation, which is not desirable. A frequently changing system would require more frequent re-fragmentation and thus inverse of average percentage change is used. A low S value means a threshold is close to the best PE value. The lower the threshold, the more likely it is that the current computed PE value is higher than threshold and would call for a re-fragmentation. The margin of poor performance allowed by the system before triggering a re-fragmentation is the difference of the threshold and the best system PE value. This margin is close to 0 if S is close to 1 as the system average input change increases. The more frequently a system changes, the lower the S value; while a high S value represents a slowly changing system with a low average percentage change in its input. A high PE value represents a lower performance by the system.

(6) Comparing the current PE with system threshold such that if it is greater than system threshold, then it triggers a re-fragmentation. This means that if the lower performance exceeds system tolerance there is a re-fragmentation, otherwise it goes to monitor module.

(7) The Monitor module (OO Monitor) keeps track of all changes in the input data and runs periodically to determine if the change is big enough to call a re-computation of PE value.

(8) The system computes the current PE value if step seven determines a re-computation is needed and goes back to step six to check system performance.

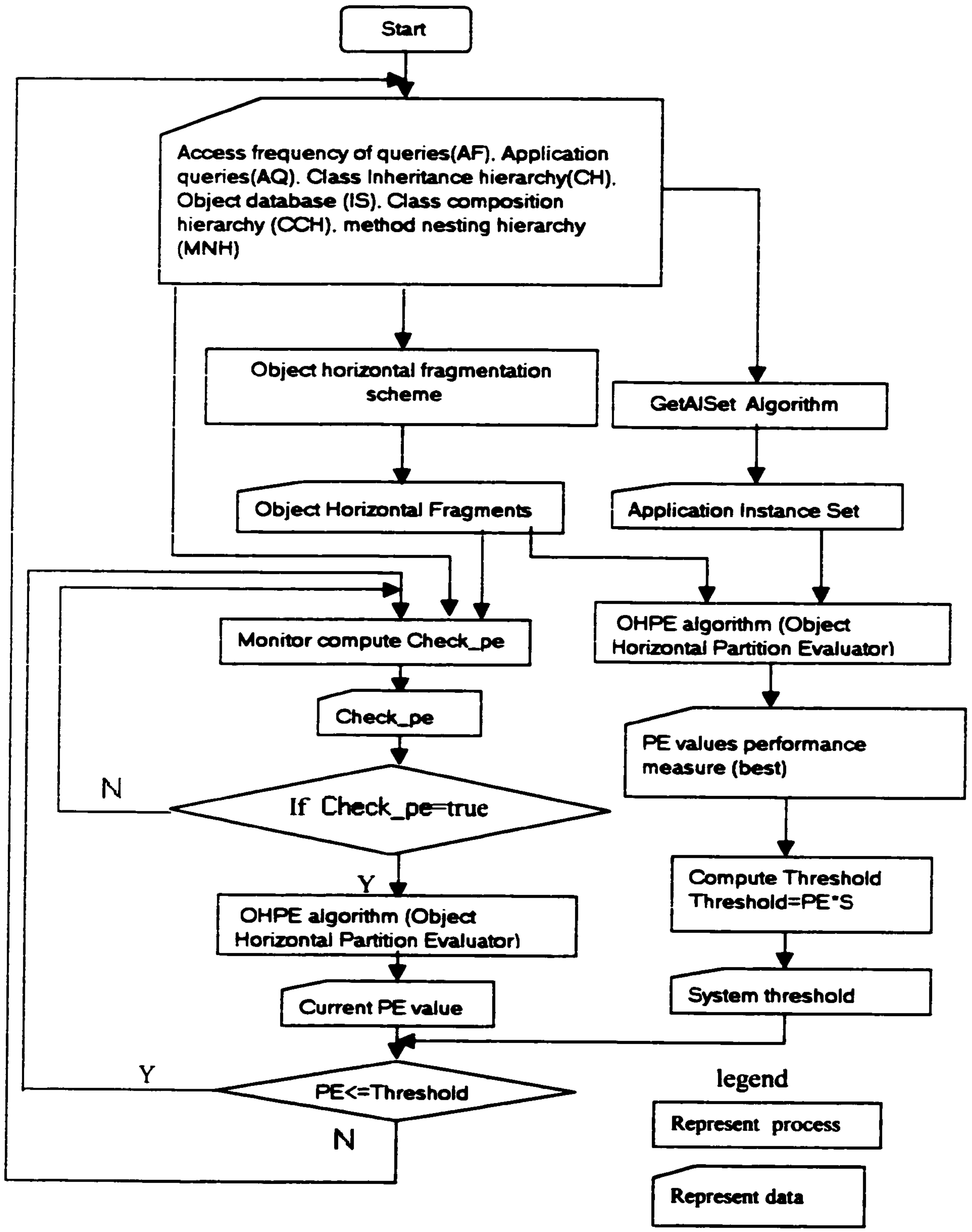

Start
Access frequency of queries(AF). Application queries(AQ). Class Inheritance hierarchy(CH). Object database (IS). Class composition hierarchy (CCH). method nesting hierarchy (MNH)
Object horizontal fragmentation scheme
GetAISet Algorithm
Object Horizontal Fragments
Application Instance Set
Monitor compute Check_pe
OHPE algorithm (Object Horizontal Partition Evaluator)
Check_pe
PE values performance measure (best)
N
If Check_pe=true
Y
Compute Threshold Threshold=PE*S
OHPE algorithm (Object Horizontal Partition Evaluator)
Current PE value
System threshold
Y
PE<=Threshold
N
legend
Represent process
Represent data

Figure 6: Dynamic distributed object horizontal database design architecture

```
Algorithm GetAISet
Input: queries on each class { $Q^{C_i}$ }, queries on $C_i$'s descendant class { $Q^{C_i}_{des}$ }
       queries on $C_i$'s containing class { $Q^{C_i}_{cont}$ }, queries on $C_i$'s complex method classes
       { $Q^{C_i}_{cmeth}$ }, access frequency queries on $C_i$ { $AF^{C_i}_{m}$ }, instance objects of $C_i$ { $I^{C_i}$ }
Output: AISET (application instance sets for applications on $C_i$)
begin
  for each q$\in Q^{C_i}$      // compute all predicates $P^{C_i}$ from q as //
     $P^{C_i} = P^{C_i} \cup P^q$
  for each q $\in Q^{C_i}_{des}$ // predicates accessing descendant classes //
     begin
       if access (q, $C_i$) = 1 then
         begin
           $P^{C_i} = P^{C_i} \cup P^q$, $Q^{C_i} = Q^{C_i} \cup$q
         end // if//     end //for//
for each q $\in Q^{C_i}_{cont}$     // predicates accessing containing classes //
     begin
       if access (q, $C_i$) = 1 then
         begin
           $P^{C_i} = P^{C_i} \cup P^q$, $Q^{C_i} = Q^{C_i} \cup$q
         end //if//
     end //for//
for each q $\in Q^{C_i}_{cmeth}$  // predicates accessing complex method classes //
     begin
       if access (q, $C_i$) = 1 then
         begin
           $P^{C_i} = P^{C_i} \cup P^q$, $Q^{C_i} = Q^{C_i} \cup$q
         end //if//
     end //for//
for each q$\in Q^{C_i}$ // now compute application instance sets matrix//
     for each p $\in P^{C_i}$ and p $\in$ q
       for each site c
         begin
           if access (q,p) at c =1 then
           begin
             for each I $\in C_i$ at site c
               begin
                  if I $\in$ P then $[AI]_q = [AI]_q \cup$ I
               end // for each I//
           end // if//
         end // for each c//
end { GetAISet}
```

Figure 7: The Get Application/Instance Sets Algorithm

3.2 Application Instance Sets Algorithm

From the dynamic object horizontal fragmentation architecture, the OHPE algorithm needs the input of application object instance set and this information may be too detailed to be available during fragmentation. It requires that the needed information be computed from the available data sets. In this section, we develop an algorithm named *GetAISet* (get application object instance sets) as shown in Figure 7. It is responsible for defining the application instance sets given the set of user queries accessing class C_i and its descendant classes, all of the classes containing this class in the class composition hierarchies and all of the classes using their complex methods to access this class' methods. The input to the GetAISet algorithm also includes an application frequency matrix that contains the frequency of each application at each distributed site. To compute the application object instance matrix of a class, we normally define for each query accessing the class as matrix rows and each instance object of the class as matrix column, assign 1 to matrix element (q, i) if application q accesses object i. However, classes may have hundreds of instance objects, this approach is not realistic. As an alternative, we can begin by computing application predicate matrix. There are only a countable number of predicates accessing a class and the matrix has applications as rows and predicates as columns. A value of "1" is assigned to matrix cell of application q and predicate p if application q accesses predicate p and "0" otherwise.

The approach entails defining an application predicate matrix by specifying the application Q^{C_i} as row labels and the predicates P^{C_i} as column labels. A value of "1" is assigned to element (q, p) if query q accesses predicate p and "0" otherwise. Then, for each application q and for each site c, the application /instance set $[AI]_{qc}$, which is the set of instance objects accessed by application q, is obtained as follows:

$I \in [AI]_{qc}$ iff $I \in C_i \wedge p \in P^{C_i} \wedge q \in Q^{C_i} \wedge \exists$ (p| access(q, p)=1 $\wedge$ I$\in$p).

This means that the application instance set for a class C_i is the union of the sets of all object instances of the class accessed by queries running directly on the class and those instances accessed by queries running on this C_i's descendant, containing and complex method classes.

3.3 Extended Partition evaluation function for object horizontal Fragmentation

Attributes of relations are the members of relational vertical fragments. On the other hand, methods and attributes of object classes are the members of object vertical fragments. Instance objects of classes are the members of object horizontal fragments. The set of instance objects in fragments created by different fragmentation schemes may be different. It is necessary to develop a modified partition evaluator to deal with object horizontal fragmentation schemes. This partition evaluator counts the number of local irrelevant accesses made to fragments by queries as well as the numbers of remote accesses made to fragments for each set of horizontal fragments created with a scheme. Therefore, in order to measure the system performance each fragment is assumed placed at one of the distributed sites, while the PE function is used to measure the partition evaluation value (the PE value) for the system. The higher the PE value of the system, the worse the system performance. Computing the PE value of horizontal fragments requires an input matrix that shows for each application the number of times an instance object is accessed. This is called the application instance matrix. In [CMV+93], Chakaravarthy *et al.* computes the processing costs of distributed fragments using an application/attribute matrix that counts the number of times applications access attributes at distributed sites. This thesis computes the processing costs of horizontally distributed objects by beginning with application instance sets computed by the GetAISet algorithm. It is assumed there are no data redundancies and following fragmentation, the fragments have been allocated to sites where they are most needed. The partition evaluator value (PE) is given by:

PE= $E_M^2 + E_R^2$, where $E_M^2 = \Sigma_{i=1}^{M} \ \Sigma_{t=1}^{T} [q_{tc}^2 * |S_{itc}| * (1 - |S_{itc}| / |n_i|)]$ and M is the number of fragments, S_{itc} is the set of instance objects contained in fragment F_i that transaction t accesses at site c. It is empty if t does not need fragment F_i and q_{tc} is the frequency of application t for t=1, 2, ..., T and n_i is the total number of instance objects contained in fragment F_i. Relevant remote instance access cost of PE is given by:

$E_R^2 = \Sigma_{t=1}^{T} \min(\forall_i^M \Sigma_{\forall k \neq i}\, q_{tc}^2 * |R_{itk}| * |R_{itk}| / |n_{itk}^{rem}|)$ where $|R_{itk}|$ is the number of relevant instances in fragment k accessed remotely with respect to fragment i by transaction t at site c, n_{itk}^{rem} is the total number of instances that are in fragment k accessed remotely with respect to fragment i by transaction t.

The algorithm OHPE implements the objective function defined above. For measuring the performance of a partition scheme, algorithm OHPE needs two inputs. One is every horizontal fragment from the partition scheme; the other is the application object instance sets. Then OHPE produces the PE value of this partition scheme. Each PE value corresponds to the total penalty cost incurred by each scheme through both local irrelevant access costs and remote relevant access costs. A lower PE value means less performance penalty and thus, a better performance. The algorithm OHPE is shown as Figure 8. This algorithm accepts the application object instance sets of all applications, the set of horizontal fragments allocated to distributed sites and application access frequencies at distributed sites. It then computes the local irrelevant access cost of each class as the sum of local irrelevant access cost of all applications for each fragment of the class. Similarly, it computes the remote relevant access cost as the sum of remote access cost made by all applications to fragments of this class at remote sites. The sum of these two costs make up the PE value of the class.

3.4 Measuring Performance of the OO Horizontal Fragmentation Schemes

Figure 9 shows an outline for comparing the performance of different object horizontal fragmentation schemes. *GetAISet* is an algorithm defined in previous section, which generates application instance sets. Measuring the performance of a set of object horizontal fragmentation schemes can be achieved by measuring their PE values computed by the partition evaluator. The scheme that produces the lowest partition evaluator value is the best partition scheme. Figure 9 shows the comparison of two object horizontal fragmentation schemes namely: Ezeife *et al.* [EB95a] called OOHF algorithm and Bellatreche *et al.* [BKS97] called Bell's algorithm.

```
Algorithm OHPE
Input: $[AI]_{tc}$ : application object instance set of application t at site c
  $F^{C_i}$ : set of horizontal fragments of class $C_i$, $Q^{C_i}$ : set of transaction or queries on class $C_i$
  $AF_{tc}^{C_i}$ : access frequencies of transactions on $C_i$ at site c
Output: $E_M^2$ : irrelevant local method access cost
        $E_R^2$ : relevant remote access cost
        PE: partition evaluator value
Begin
  $E_M^2 = E_R^2 = 0$        //initialize//
      for i=1 to number of fragments     // compute $E_M^2$ //
      begin
        for t=1 to number of transactions
          for c=1 to number of sites
            begin
              $S_{itc} = [AI]_{tc} \cap [Fragment]_i$   //set of instances in fragment i, $[AI]_{tc}$ //
              $n_i = |[Fragment]_i|$   //number of objects in fragment I //
              $E_M^2 = E_M^2 + (AF_{tc})^2 * |S_{itc}| * (1 - |S_{itc}| / |n_i|)$
            end  //for c, $AF_{tc}$ is the application frequency of transaction t at site c //
      end  // for i//
    for t=1 to number of transactions     // compute $E_R^2$ //
      begin
        for i=1 to number of fragments
          begin
            for k=1 to number of fragments
              begin
                if k≠i then
                  begin
                    $R_{itk} = ([AI]_{tc} - [Fragment]_i) \cap [Fragment]_k$
                    $n_{itk}^{rem}$ = number of element in $[Fragment]_k$
                    if $AF_{tc} \neq 0$ then $E_{Rtemp}^2 = E_{Rtemp}^2 + (AF_{tc})^2 * |R_{itk}| * |R_{itk}| / |n_{itk}^{rem}|$
                    else $E_{Rtemp}^2$ =maxint   // maxint stands for not needed here //
                  end //if//
                if i=1then $E_{R\min}^2 = E_{Rtemp}^2$
                  else if $E_{Rtemp}^2 < E_{R\min}^2$ then $E_{R\min}^2 = E_{Rtemp}^2$
              end  // for k//
            $E_R^2 = E_R^2 + E_{R\min}^2$       end //fori//
      end  // for t//
  PE= $E_M^2 + E_R^2$
End //OHPE//
```

Figure 8: The Object Horizontal Partition Evaluator Algorithm

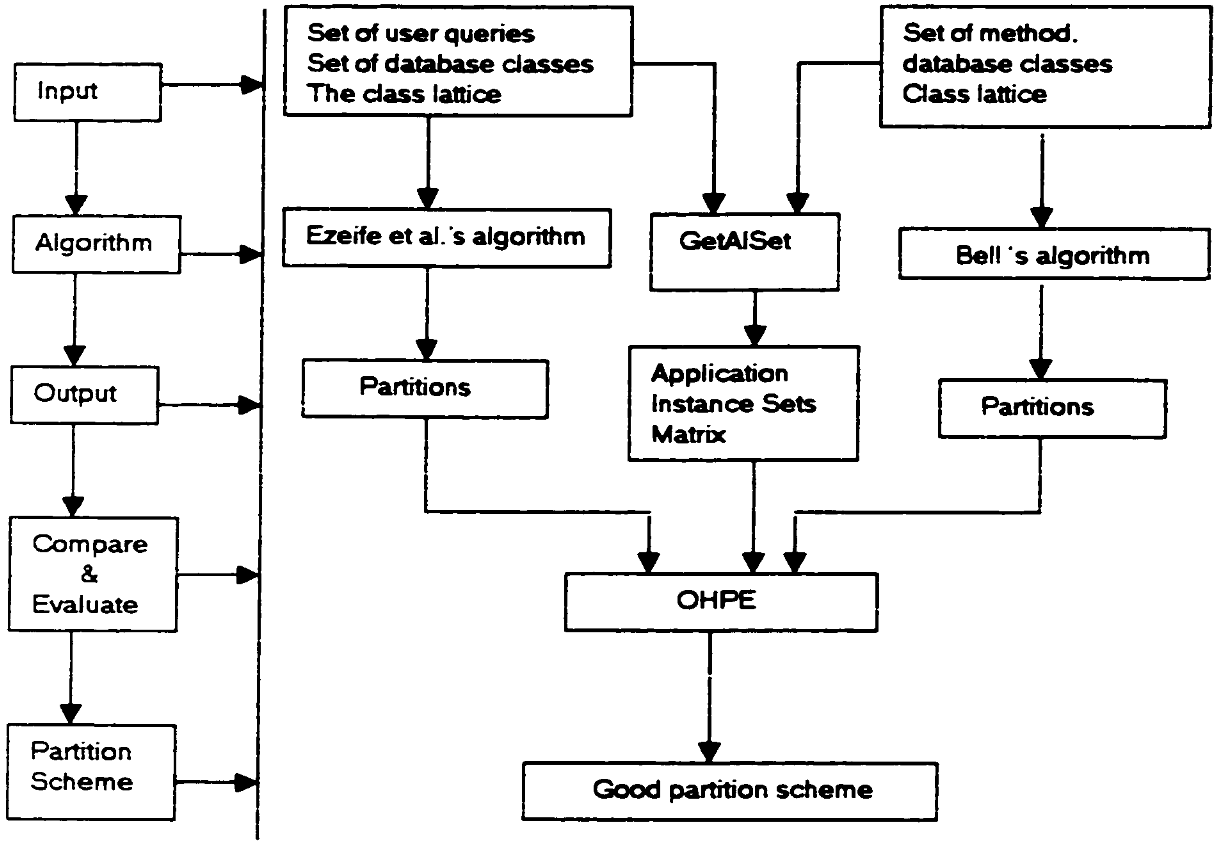


Figure 9: Comparing performance of object horizontal fragmentation schemes

In [EB95a], Ezeife and Barker present an example of horizontal fragmentation of simple attributes and simple methods using OOHF. Class *Prof* has two fragments, namely $F_1^h = \{I_2, I_3, I_4\}$ and $F_2^h = \{I_1\}$. With the same example, we get three fragments, $F_1^h = \{I_2, I_3\}$, $F_2^h = \{I_1\}$, $F_3^h = \{I_4\}$ using Bell's algorithm. Applying OHPE to the fragments got from these two different object fragmentation schemes results in the table shown as Table 4. It can be seen that OOHF results in a PE value of 0 meaning that for this class no processing cost is incurred both locally and remotely by applications. Bell's distribution gave a remote process cost of 3475. Thus, the better scheme in this case is OOHF.

Prof

Partition Scheme	E_M^2	E_R^2	PE
Ezeife *et al.*'s	0	0	0
Bell's	0	3475	3475

Table 4: PE Comparison results of OOHF and Bell's algorithm

4 The OO Monitor and Compute Threshold Algorithm

This chapter presents two algorithms that are part of the dynamic distributed object horizontal database design architecture. Section 4.1 presents the OO Monitor algorithm and section 4.2 presents the Compute Threshold algorithm.

4.1 The OO Monitor algorithm

In Figure 6 (Dynamic distributed object horizontal database design architecture), determining how frequently the current PE value of the system is measured is very important. Continually checking the current PE value is not realistic. It increases the system cost and then decreases the system performance. Figure 10 shows a monitor module used to monitor changes on the object oriented database in order to trigger the event of measuring current PE value. For most applications, monitoring changes twice a day is suitable. However, the time delay can be adjusted to suit application needs. The object-oriented monitor algorithm serves to compute the input changes used to determine the system change variable S by the Compute Threshold algorithm. The monitor also determines whether the system performance should be checked or not. The overhead cost incurred doing the checks is reduced through determining the times it is necessary to re-compute the system current PE value. The monitor recommends that the system PE values be re-computed only if during any system input change it encounters an input change that is higher than the average system input change. It re-computes the average system input only after a re-fragmentation.

The input to the monitor algorithm comprises the previous AQ, AF, CH, CCH, MNH, and IS fragments and their site information; the current AQ, AF, CH, CCH, MNH, and IS fragments; periodic wait time; the previous changes in AQ, AF, CH, CCH, MNH, and IS. The output of the monitor algorithm is a boolean variable where "true" means the system PE value should be re-computed. The other output data are the average changes in the input data of AQ, AF, CH, CCH, MNH, and IS fragments. These average input changes are used to compute the system change variable S.

```
Algorithm OO Monitor

Input:  Previous AQ, AF, CH, CCH, MNH, IS
        Current AQ, AF, CH, CCH, MNH, IS
        Wait_time(length of time in seconds)
        Previous system change averages (ΔAF, ΔAQ, ΔCH, ΔCCH, ΔMNH, ΔIS)
Output: Check_current_PE (boolean)
        New system change average(ΔAF, ΔAQ, ΔCH, ΔCCH, ΔMNH, ΔIS)
Begin
  Check_current_PE=0 //initialize//
  Time = systemtime -lastruntime
  Read all input data
  While not (Check_current_PE ) and (Time >=Wait_time)
      Beign
          Current change AQ = |Current AQ- Previous AQ|
          Current change AF = |Current AF- Previous AF|
          Current change CH = |Current CH- Previous CH|
          Current change CCH = |Current CCH- Previous CCH|
          Current change MNH = |Current MNH- Previous MNH|
          Current change IS = |Current IS- Previous IS|
          // Monitor first determines if the system PE should be recomputed//
          // Before re-computing, generates the new average system changes//
          if Current change AQ > Previous change AQ then Check_current_PE = 1
          else if Current change AF > Previous change AF then Check_current_PE = 1
          else if Current change CH > Previous change CH then Check_current_PE = 1
          else if Current change CCH > Previous change CCH then Check_current_PE = 1
          else if Current change MNH > Previous change MNH then Check_current_PE = 1
          else if Current change IS > Previous change IS then Check_current_PE = 1
          average ΔAQ = (current ΔAQ+ previous ΔAQ)/2
          average ΔAF = (current ΔAF+ previous ΔAF)/2
          average ΔCH = (current ΔCH+ previous ΔCH)/2
          average ΔCCH = (current ΔCCH+ previous ΔCCH)/2
          average ΔMNH = (current ΔMNH+ previous ΔMNH)/2
          average ΔIS = (current ΔIS+ previous ΔIS)/2
          lastruntime = systemtime
  end //while//
End
```

Figure 10: Monitor in object dynamic horizontal fragmentation system

4.2 The Compute Threshold Algorithm

This algorithm computes the threshold in the object dynamic horizontal fragmentation system. The value of S is determined as the inverse of the average change in the system input data plus 1. A change in any input data of AF, AQ, CH, CCH, MNH, IS fragments is computed as the percentage of the original data. This means that in a class hierarchy with five classes, if a class gets added or dropped or moves to another location in the hierarchy, the change is (1/5)*100=20%. This definition applies to all hierarchies. In the case of object instance fragments, if a new object gets created in any site, it stays in that site until a re-fragmentation. To compute the change in object instance fragments, we count any new object that is created at any site as 1, any destruction or relocation of object also counts as 1. The total change in the object instance fragments of a class is calculated as the sum of all changes at all sites divided by the number of instance objects in the class *100. The change in application queries is defined as the percentage of new queries that access a class, no longer accessing a class or change the site of access to a class. Thus, the change in application queries accessing a class is obtained as the sum of the number newly accessing it, or no longer accessing it divided by the total number of queries originally accessing the class multiplied by 100. The change in access frequency of queries is defined as the maximum of the percentage change in any one query at all sites. Thus, the changes in access frequency of queries is computed as maximum of the total change at all sites for each query divided by total access at all sites for that query *100. The 50 in the compute threshold algorithm is a compensate factor to adjust the threshold defined according to S to a reasonable value.

```
Algorithm OO Threshold
Input: ΔAF, ΔAQ, ΔCH, ΔCCH, ΔMNH, ΔIS , best PE
Output: Threshold
Begin
    S = 1+50/(avg(ΔAF, ΔAQ, ΔCH, ΔCCH, ΔMNH, ΔIS))
    Threshold = S * best PE
End{Begin}
```

Figure 11: Compute Threshold Algorithm

5 Implementations and Examples

The OHPE algorithm in the dynamic distributed object horizontal database design architecture is fully implemented in C/UNIX. The name of program is OHPE.c and its size is 15kb (approximately 500 lines of code). OHPE receives three inputs, namely, access frequency of queries at distributed site, allocation of fragments to distributed site and application instance set on each site of each class. There are three input datafiles corresponding to these three inputs, namely, AFMATRIX, FRAGSITE and AISET. The program uses a dynamic two-dimensional array whose element is an integer to store input datafile AFMATRIX. The program uses a dynamic two-dimensional array whose element is a link list to store the application instance sets (AISET) from the input. It uses a complex link list whose element is a link list to store the allocation information (FRAGSITE) from input because each site may have several fragments. Also it defines functions used to get the number of element of a linklist, the intersection of two linklists and difference of two linklists. Having all these structures and functions, it is easy to follow OHPE algorithm to compute the irrelevant local instance access cost and relevant remote instance access cost, the sum of which produces the PE value. The OHPE program can be run with the existence of the three required datafiles.

This chapter illustrates how to compute the PE value of object horizontal fragmentation schemes with the proposed object horizontal partition evaluator (OHPE) and when access frequency of the example changes, compute the PE value to determine whether a re-fragmentation is triggered. The working of the scheme is presented using the object database example horizontally fragmented by an object oriented horizontal partition scheme (OOHF) in [EB95b]. We also include a comparison with a case where data is not fragmented at all but replicated at different sites and the case where data is neither fragmented nor replicated but shipped to remote sites on request.

The sample object database represents a university database as in Figure 12. The inheritance hierarchy is in Figure 13 and the class composition hierarchy is in Figure 14. The five queries running on the classes of database and their predicates are given:

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

Figure 12: The complex Sample Object Database Schema

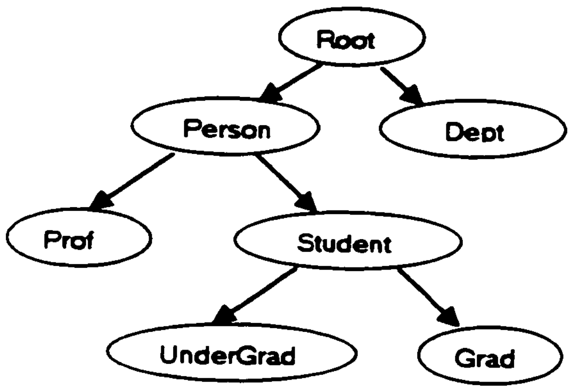


Figure 13: Complex Class Hierarchy

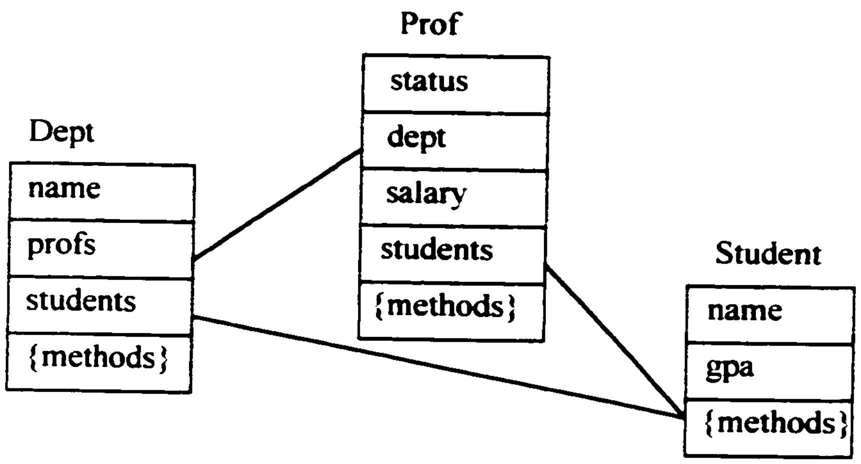


Figure 14: Class Composition Hierarchy

Q_1: This application groups grads according to their area of specialization that is determined by the name of their supervisor. The methods used are defined on class *Grad* and the predicates are, { P_1: supervisor = " Prof John West", P_2: supervisor = " Prof Mary Smith"}.

Q_2: This application groups profs by their addresses. The methods used are defined on class *Prof* with the following predicates, { P_1: address = " Winnipeg", P_2: address = "Vancouver", P_3: 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 >=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:"Stats"}.

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

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

Using the given query access frequency and other input data, the object horizontal fragmentation scheme (OOHF) produces the following fragments for the classes in this database.

Class Person

$F_1^h = \{I_1, I_7\}$, $F_2^h = \{I_3, I_4\}$, $F_3^h = \{I_2, I_8\}$, $F_4^h = \{I_{10}\}$, $F_5^h = \{I_6, I_9\}$, $F_6^h = \{I_5\}$

Class Student

$F_1^h = \{I_1, I_5\}$, $F_2^h = \{I_2, I_4\}$, $F_3^h = \{I_3, I_6\}$

Class Prof

$F_1^h = \{I_2, I_3, I_4\}$, $F_2^h = \{I_1\}$,

Class Dept

$F_1^h = \{I_2, I_3\}$, $F_2^h = \{I_1\}$, $F_3^h = \{I_4\}$

Class Grad

$F_1^h = \{I_1\}$, $F_2^h = \{I_2, I_3\}$

The fragments above are allocated to three distributed sites using the simple allocation scheme that places fragments to sites where they are most needed by queries, and Table 5 shows the placement of above fragments to sites while Figure 15 shows the access frequencies of queries at the three sites.

From the application information gathered at distributed sites, the algorithm GetAISet computes the Application / Predicate matrix and the Application / Instance sets as follows:

	S1	S2	S3
Q1	10	20	0
Q2	20	30	15
Q3	25	10	0
Q4	15	25	10
Q5	20	20	20

Figure 15: Access Frequencies of queries at distributed sites

Class	Site 1	Site 2	Site3
Person	$F_1^h = \{I_1, I_7\}$ $F_5^h = \{I_6, I_9\}$ $F_4^h = \{I_{10}\}$	$F_6^h = \{I_5\}$ $F_2^h = \{I_3, I_4\}$	$F_3^h = \{I_2, I_8\}$
Student	$F_1^h = \{I_1, I_5\}$	$F_2^h = \{I_2, I_4\}$	$F_3^h = \{I_3, I_6\}$
Prof	$F_1^h = \{I_2, I_3, I_4\}$	$F_2^h = \{I_1\}$	
Dept	$F_1^h = \{I_2, I_3\}$	$F_2^h = \{I_1\}$	$F_3^h = \{I_4\}$
Grad	$F_1^h = \{I_1\}$	$F_2^h = \{I_2, I_3\}$	

Table 5: Allocation of Fragments to Distributed Sites

Class *Person*

Set of predicates $P^{person} = P^{person} \cup P^{prof} \cup P^{student} \cup P^{grad}$

$P^{person} = \{ P_1^{prof}, P_2^{prof}, P_3^{prof}, P_4^{prof}, P_5^{prof}, P_1^{student}, P_2^{student}, P_3^{student}, P_1^{Grad}, P_2^{Grad} \}$

Set of queries $Q^{person} = Q^{person} \cup Q^{prof} \cup Q^{student} \cup Q^{grad}$

$Q^{person} = \{q_1, q_2, q_3, q_4\}$

The computed application instance set for all classes and for each query accessing the class at the three different sites is given as Table 6.

The local irrelevant access costs, remote relevant access costs and the PE values obtained after executing the OHPE algorithm are shown in the Table 7 below:

Class	Application	Site 1	Site 2	Site3
Person	Q1	$\{I_1\}$	$\{I_4, I_7\}$	
	Q2	$\{I_6, I_9, I_{10}\}$	$\{I_5\}$	
	Q3	$\{I_6, I_9, I_{10}\}$	$\{I_5\}$	
	Q4	$\{I_1, I_7\}$	$\{I_3, I_4\}$	$\{I_2, I_8\}$
Student	Q1	$\{I_1\}$	$\{I_2, I_5\}$	
	Q4	$\{I_1, I_5\}$	$\{I_2, I_4\}$	$\{I_3, I_6\}$
Prof	Q2, Q3	$\{I_2, I_3, I_4\}$	$\{I_1\}$	
Grad	Q1	$\{I_1\}$	$\{I_2, I_3\}$	
Dept	Q5	$\{I_2, I_3\}$	$\{I_1\}$	$\{I_4\}$

Table 6: Application / Instance Sets at all Sites

Case1 (With fragmentation)

Class	E_M^2	E_R^2	PE
Person	900	1600	2500
Student	900	800	1700
Prof	0	0	0
Grad	0	0	0
Dept	0	0	0

Case2 (No fragmentation, full copy resides on every site)

Class	E_M^2	E_R^2	PE
Person	20625	0	20625
Student	8400	0	8400
Prof	12225	0	12225
Grad	2700	0	2700
Dept	4800	0	4800

Case3 (No fragmentation, no replication)

Class	E_M^2	E_R^2	PE
Person	6875	140	7015
Student	2800	83	2883
Prof	4075	250	4325
Grad	900	33	933
Dept	1600	100	1700

Table 7: Evaluation Results

The PE value comparison of three cases is illustrated in Table 8.

Class	OOHF	Replicated data	No Frag No Replicate
Person	2500	20625	7015
Student	1700	8400	2883
Prof	0	12225	4325
Grad	0	2700	933
Dept	0	4800	1700

Table 8: The comparison of PE value

Figure 16 graphically shows the comparative performance of the three approaches. Lower PE value means better performance. This shows that OOHF produces best performance, followed by No Frag, No replication approach. The worst approach from the figure is the Replicated data approach.

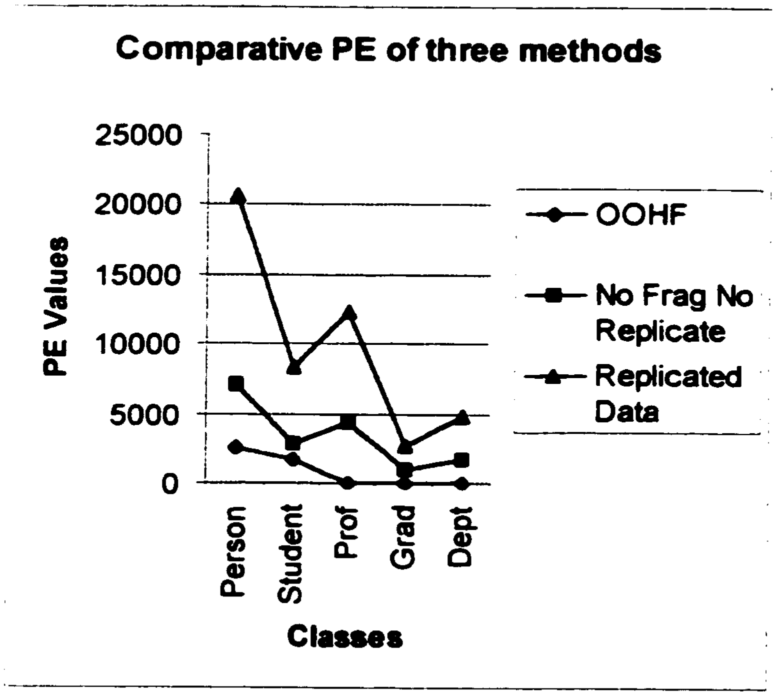


Figure 16: Comparative PE of three methods

Showing How to determine Performance Threshold

Based on previous example, we have Table 9 that shows the PE value of classes fragmented using OOHF scheme.

Class	E_M^2	E_R^2	PE
Person	900	1600	2500
Student	900	800	1700
Prof	0	0	0
Grad	0	0	0
Dept	0	0	0

Table 9: PE value of OOHF

In Figure 6, given previous system change averages ΔAF=20, ΔAQ=40, ΔCH=60, ΔCCH= 80, ΔMNH= 100, ΔIS=60. Using the algorithm OO Threshold of Figure 11, we compute S as S= 1+ 50/((20+40+60+80+100+60)/6)=1.83, so the performance threshold PT=1.83*bestPE. Table 10 shows the performance threshold for every class.

Class	PE	PT
Person	2500	4583
Student	1700	3117
Prof	0	0
Grad	0	0
Dept	0	0

Table 10: Performance Threshold of each class

For the case of class *Student*, we change the access frequency from Figure 15 to Figure 17. The monitor module finds changes in access frequency ΔAF=(10/30)*100 =33, which is greater than previous system changes $\Delta AF=20$. So, it re-computes current PE. Applying algorithm OHPE, we get PE=2723, which is less than performance threshold of 3117. This means that re-fragmentation is not need right now because the change in Figure 17 is not big enough to trigger a system re-fragmentation. We change Figure 15 again to Figure 18, this time the AF change is bigger than in Figure 17. In Figure 18, $\Delta AF=66$, we apply algorithm OHPE again and get PE=4000, which is greater than the performance threshold of 3117. Then, the system triggers a re-fragmentation of class *Student* to improve the system performance.

This example gives a general idea of how the dynamic distributed object horizontal database design architecture works. It includes getting the object horizontal fragments, getting application instance set through using GetAISet, computing PE value, building system threshold, monitoring system changes, determining whether to invoke a re-computation of PE value, comparing current PE value with system threshold to determine whether to trigger a re-fragmentation. In the mean time, it also includes the comparison of three different object horizontal partition schemes.

	S1	S2	S3
Q1	15	25	0
Q2	20	30	15
Q3	25	10	0
Q4	25	35	20
Q5	20	20	20

Figure 17: Access Frequencies of queries at distributed sites

	S1	S2	S3
Q1	20	30	0
Q2	20	30	15
Q3	25	10	0
Q4	25	35	20
Q5	20	20	20

Figure 18: Access Frequencies of queries at distributed sites

6 Conclusions And Future Work

This chapter presents conclusions in section 6.1 and future research directions in section 6.2.

6.1 Conclusions

In this thesis, a dynamic distributed object horizontal database design architecture is first presented, which is used to determine the time for triggering a dynamic re-fragmentation. Then it proposes a set of algorithms used by the architecture, which includes measuring the performance of object database horizontal fragmentation schemes, determining system performance threshold and monitoring changes in system inputs. The system cost is reduced by using the dynamic distributed object horizontal database design architecture because the unacceptable low performance is promptly detected by the monitor and corrected earlier. In addition, it eliminates the need and cost of full requirements analysis in order to evaluate system performance. This approach has the resultant effect of increasing user confidence in distributed object database techniques. An added advantage of this thesis is that the OHPE algorithm can be used to compare different object horizontal fragmentation and select the one most suitable for an application. This thesis presents significant contributions to object dynamic horizontal fragmentation and provides an avenue for improving performance and use of distributed database design.

The main contributions of this thesis are:

(1) Proposing system architecture for dynamic object horizontal fragmentation.

(2) Defining an algorithm *GetAISet* that is responsible for collecting the application instance sets. The set of user queries accessing class C_i and its descendant classes, all of the classes containing this class in the class composition hierarchies and all of the classes using their complex methods to access this class method from method nesting hierarchies are given as input.

(3) Modifying the vertical object partition evaluator [EB98] to measure object horizontal fragmentation schemes using the input of Application Instance Sets created from step (2) above and writing an algorithm to implement it.

(4) Proposing a method for defining a PE value for a given system that serves as the performance threshold for determination when a re-fragmentation is necessary.

(5) Providing a monitor module that monitors changes in access frequency of application, application queries, class lattice, class composition, method nesting hierarchies and numbers of instance object. Any of these change greater than their predefined change (average change) will cause a current PE value check.

(6) Implementing the OHPE algorithm component of the architecture.

6.2 Future Work

This section discusses several problems related to the broad topic of distributed design in object oriented system appropriate for future work.

6.2.1 Horizontal Re-fragmentation of Objects

This section defines problems of estimating a threshold value in dynamic horizontal fragmentation and the impact of dynamic fragmentation on the system cost. Different object-oriented distributed systems have different requirements to the response time of the transactions. Getting a good threshold value is non trivial. Consider the following situation: With a small threshold value, the efficiency of the object oriented distributed system is increased but the distributed system needs to be re-fragmented very often. However, it increases the overhead cost of the system due to frequent re-fragmentation. A technique for measuring and analyzing system performances (overall processing costs and response times) at varying system thresholds may be needed for determining the optimal threshold. A measure of the impact of dynamic fragmentation on the overall system cost is also needed. This architecture supports dynamic re-fragmentation by fragmenting the entire database data again (old and new). To cut down on processing cost, future work may consider incremental re-fragmentation using only new data additions to the database.

6.2.2 Extensions to support Hybrid Fragmentation

Several vertical fragmentation schemes have been evaluated to find the best scheme in [EB98]. It is very easy to modify that part so that we can deal with vertical fragmentation dynamically.

Since we use an application instance usage matrix and a method usage matrix to get the corresponding mean vector in horizontal and vertical fragmentation, we think a grid usage matrix should be used in the case of dealing with hybrid fragmentation because it has both the instance usage matrix and the method usage matrix. The grid usage matrix comes from the instance usage matrix and the method usage matrix.

6.2.3 Analysis of Algorithms

This section addresses the problems of analyzing research results. The dynamic horizontal fragmentation scheme we defined must be tested to show that:

- System automatic re-fragmentation improves the performance of the system. This could be done through showing that system performance is more efficient by applying the dynamic algorithm rather than the static algorithm. Through simulation or experimentation, using the same application, we can see which one produces lower overall processing cost over a period of time.
- A technique for experimentally verifying the correctness of results produced by OHPE is needed. This can be done through recording the number of remote accesses and irrelevant local accesses made by applications after fragmentation and allocation using a test object base.
- A lot of external factors outside the scope of this research may decrease system performance. However, if the previous two points are proven, it can be argued that system performance is reasonable. A more formal approach for defining what

constitutes a reasonable system performance in a distributed object base environment may be needed.

References

[Ape88] Apers, P.M.G. Data Allocation in Distributed Database Systems. *ACM Transaction on Database Systems 1988.*

[BB95] Bhar, S., Barker, K. Static allocation in distributed object-base systems: a graphical approach. *Information Systems and Data Management. 6th International Conference*, CISMOD '95. Proceedings.

[BKS97] Bellatreche, L., Karlapalem, K. and Simonet, A. Horizontal Class Partitioning Design in Object-Oriented databases. In *Lecture Notes in Computer Science, volume 1308, p.58-67*, Toulouse-France, September 1997. DEXA.

[BM98] Baião, F., Mattoso, M. A MIXED FRAGMENTATION ALGORITHM FOR DISTRIBUTED OBJECT ORIENTED DATABASES. Proc. of the 9th International Conference of Computing *and Information*, ICCI'98 University of Manitoba, Wennipeg.

[BSS96] Bellatreche, L., Simonet, A. and Simonet, M. Vertical fragmentation in Distributed Object Database Systems with Complex Attributes and Methods. In *Proc. of 7th Intl. Conf. and Workshop on Database and Expert Systems and applications, DEX96, Sep. 1996*

[CDN+94] Carey, M., DeWitt, D., Naughton, J., Solomon, M. Shoring Up Persistent Applications. *Proc. of the 1994 ACM SIGMOD Conference*, Minneapolis, MN, May 1994.

[CMV+93] Charkravarthy, S., Muthuraj, J., Varadarajan, R. and Navathe, S.B. An Objective Function for Vertically Partitioning Relations in Distributed Databases and Its Analysis. *Distributed and Parallel Database 2(1) 1993.*

[CNP82] Ceri, S., Negri, M., Pelagatti, G. Horizontal data partitioning in database design. In *proceedings of ACM SIGMOD International Conference on Management of Data '82.*

[CNW83] Ceri, S., Navathe, S.B., Wiederhold, G. Distributed design of logical database schema. *IEEE Trans. on Software Engineering '83.*

[CY87] Cornell, D. and Yu, P.S. A vertical partitioning algorithm for relational databases. In *Proceedings of the Third International Conference on data Engineering*, 1987.

[Deu91] Deux, O. *et al.* The O2 system. *Communication of the ACM '91.*

[EB94] Ezeife, C.I. and Barker, K. Vertical Class Fragmentation in a Distributed Object Based System. *Technical Report* - Department of CS, University of Manitoba, TR 9402.

[EB95a] Ezeife, C.I. and Barker, K. A Comprehensive Approach to Horizontal Class Fragmentation in a Distributed Object Based System. *International Journal of Distributed and Parallel Database*, Kluwer Academic Publishers, V1, 1995.

[EB95b] Ezeife, C.I. and Barker, K. Class Fragmentation in a Distributed Object Based System. *A Thesis Presented to the University of Manitoba in partial fulfilment of requirements for the degree of PhD in CS.*

[EB98] Ezeife, C.I. and Barker, K. Distributed Object Based Design: Vertical Fragmentation of Classes. *International Journal of Distributed and Parallel Databases*, Vol. 6, No. 4, pp. 327-360, Kluwer Academic Publishers, October 1998.

[EZ99a] Ezeife, C.I. and Zheng, J. Measuring the Performance of Database Object Horizontal Fragmentation Schemes. Submitted to the 3rd International Database Engineering and Application Symposium (IDEAS99), 1999.

[EZ99b] Ezeife, C.I. and Zheng, J. Dynamic Database Object Horizontal Fragmentation. Submitted to the 8th international conference on Information System Development (ISD99): Methods and Tools, Theory and Practice, Boise, Idaho, USA, August 1999.

[HS75] Hoffer, J.A. and Severance, D.G. 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. Vol1, No.1.

[HZ87] Hornick, M.F. and Zdonik, S.B. A Shared, segmented memory system for an object-oriented database. *ACM Trans. on Office Information Systems '87.*

[Ibe95] IBEX Corporation. Itasca distributed object database management system. *Technique Report Technique Summary Release 2.3.5*, IBEX Corporation, SA 1995.

[KLV96] Karlapalem, K., Li, Q. and Vieweg, S. Method Induced Partitioning Schemes in Object Oriented Databases. In *Proc. of 16th Intl. Conf. On Distributed Computing Systems, Hongkong, May 1996*

[KNM92] Karlapalem, K., Navathe, S.B. and Morsi, M.M.A. Issues in distribution design of object-oriented database. In *Pre-Proceedings of the International Workshop on Distributed Object Management*. SIGPLAN Notices, 1992, Vol.1, No.1.

[NKR95] Karlapalem, K., Navathe, S.B., Ra, M. A Mixed Fragmentation Methodology for Initial Distributed Database Design. Journal *of Computer & Software Engineering* 3(4), 1995.

[KPV94] Kersten, M.L., Plomp, S., Van Den Berg, C.A. Object storage management in goblin. In M.Tamer Ozsu, U.Dayal and Valduriez editors, *Distributed Object Management*. Morgan Kaufmann Publisher, 1994.

[LAD+96] Liskov, B., Adya, A., Day, M., Castro, M., Ghemawat, S., Gruber, R., Shrira, L., Myers, A.C., Masheshwari, U. Safe and efficient sharing of persistent objects in THOR. In *proceedings of ACM SIGMOD International Conference on Management of Data '96.*

[LHC+90] Lohman, G.M., Haas, L.M, Chang, W., McPherson, J. Starburst Mid-Flight: As the dust clears. *IEEE Trans. on Knowledge and data engineering '90.*

[LLP+91] Lohman, G.M., Lindsay, B., Pirahesh, H., Schiefer, K.B. Extensions to starburst: Object, types, function and rules. *Communication of the ACM '91.*

[MS72] McCormick, W.T., Schweiter, P.J. and White, T.W. Problem decomposition and data reorganization by a clustering technique. *Operation Research*, 20(5),1972.

[NCW84] Navathe, S.B., Ceri, S., Wiederhold, G., Dou, J. Vertical partitioning algorithms for database design. *ACM Trans. on Database Systems '84.*

[NR89] Navathe, S.B., Ra, M. Vertical partitioning for database design: A graphical algorithm. In *proceedings of ACM SIGMOD International Conference on Management of Data '89.*

[OL94] Oliver, G. and Laurent, A. Object grouping in EOS. In M.Tamer Ozsu, U. Dayal and Valduriez editors, *Distributed Object Management.* Morgan Kaufmann Publisher, 1994.

[OV91] Ozsu, M.T., Valduriez, P. Principle of Distributed Database Systems. Prentice Hall, 1991.

[PKN91] Pernul, G., Karlapalem, K. and Navathe, S.B. Relational database organization based on views and fragments. In *Proceedings of the second International conference on Data And Expert Systems Applications*, 1991. DEXA'91, Berlin.

[RZ95] Ravat, F. and Zurfluh, G. Issues in the Fragmentation of object oriented database. Proceedings, *Basque International Workshop on Information Technology*, 1995.

[SI91] Shin, D.G., Irani, K.B. Fragmenting relations horizontally using a knowledge-based approach. *IEEE Trans. on software engineering V17 Sep '91.*

[SK91] Stonebraker, M. and Kemnitz, G. The POSTGRES next generation database management system. *Communications of the ACM '91.* Vol 34, No. 10.

[SRH90] Stonebraker, M., Rowe, L., Hirohama, M. The Implementation of the POSTGRES. *IEEE Trans. on Knowledge and data engineering '90.*

[STY98] Savonnet, M., Terrasse, M., Yetongnon, K. Fragtique: A Methodology for Distributing Object Oriented Databases. Proc. of the 9th International Conference of Computing and Information, ICCI'98 University of Manitoba, Wennipeg.

[SY95] Savonnet, M., Yetongnon K. A qualitative approach for class fragmentation in distributed object oriented databases. *Proceedings of the 5th Annual Workshop on Info. Technologies and systems.* WIT'S 95.

VITA AUCTORIS

Jian Zheng was born in 1969 in **Shanghai, P.R.China**. He graduated from **Zhujiajiao** High School in 1987. From there he went on to **Tongji University** where he obtained a B.C.S. in Computer Science in 1993. He is currently a candidate for the Master's degree in Computer Science at University of Windsor and will graduate in the spring of 1999.