A Distributed Object Based Design Technique*

Christie I. Ezeife and Ken Barker

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

{christie,barker}@cs.umanitoba.ca

Abstract

Distributed relational databases benefit greatly from fragmentation and allocation and these benefits should be realized in a distributed object environment. Fragmentation enhances application performance by reducing the amount of irrelevant data accessed and the amount of data transferred unnecessarily between distributed sites. This paper presents a review of early work on distributed database design, some existing object based systems including distributed object based systems. The paper finally proposes a technique for horizontally fragmenting some class models in a distributed object based system.

1 Introduction

The problem of distributed database design comprises first, the fragmentation of database entities and secondly, the allocation of these fragments to distributed sites. Distributed database design enables optimal application performance by minimizing the need for data transfer between distributed sites [15]. Two approaches are possible in a distributed database design – top-down and bottom-up. With the top-down approach, the input to the design process is the global conceptual schema (GCS) and the access pattern information, while the output from the design process is a set of local conceptual schemas [30]. The bottom-up approach constructs a GCS from pre-existing local schemas. In the relational database environment, the entity of distribution is a relation while in a distributed object based system (DOBS), our entity of distribution is a class.

A DOBS supports an object oriented data model including features of *encapsulation* and *inheritance*. The DOBS consists of a number of encapsulated objects. The data values (attribute values) are bundled together with the methods (procedures) for manipulating them to form an encapsulated object. Objects with common attributes and methods belong to the same class, and every class has a unique identifier. Inheritance allows reuse and incremental redefinition of new classes in terms of existing ones. Parent classes are called *superclasses* while classes that inherit attributes and methods from them are

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

subclasses. The *class lattice* captures the overall inheritance hierarchy of the database schema. Formally, we define a class as an ordered relation $\mathcal{C}$ = ($\mathbf{K}$,$\mathcal{A}$,$\mathcal{M}$,$\mathcal{I}$) where $\mathbf{K}$ is the class identifier, $\mathcal{A}$ the set of attributes, $\mathcal{M}$ the set of methods and $\mathcal{I}$ is the set of objects defined using $\mathcal{A}$ and $\mathcal{M}$[1]. A class can be fragmented in three different ways:

- Horizontal Fragmentation: Breaking up of a class into horizontal class fragments ($\mathcal{C}_h$) with only subsets of its instance objects ($\mathcal{I}' \subseteq \mathcal{I}$). Thus, $\mathcal{C}_h$ = (K,$\mathcal{A}$,$\mathcal{M}$,$\mathcal{I}'$).
- Vertical Fragmentation: Breaking up of a class into vertical class fragments ($\mathcal{C}^v$) with only subsets of its methods ($\mathcal{M}' \subseteq \mathcal{M}$) and attributes ($\mathcal{A}' \subseteq \mathcal{A}$). Thus, $\mathcal{C}^v$ = (K,$\mathcal{A}'$,$\mathcal{M}'$,$\mathcal{I}$).
- Hybrid Fragmentation: Breaking up of a class into hybrid class fragments ($\mathcal{C}_h^v$) with both subsets of its instance objects ($\mathcal{I}' \subseteq \mathcal{I}$) as well as subsets of its methods ($\mathcal{M}' \subseteq \mathcal{M}$) and its attributes ($\mathcal{A}' \subseteq \mathcal{A}$). Thus, $\mathcal{C}_h^v$ = (K,$\mathcal{A}'$,$\mathcal{M}'$,$\mathcal{I}'$).

Some of the benefits of fragmentation are [15, 30]:

- Since access pattern of different applications to classes differ, fragmenting classes to reflect these access patterns will reduce the amount of irrelevant data accessed by applications and thus improve on response time.
- It also reduces the amount of data that need to be transmitted between sites.
- Fragmentation allows greater concurrency because the "lock granularity" can accurately reflect the applications using the object base.
- Fragmentation reduces the amount of data transferred when migration is required.
- Fragment replication is more efficient than replicating the entire class because it reduces the update problem and saves storage.

This paper presents a review of early work on fragmentations of relations and DOBS, a review of some existing object based and distributed object based systems. It further contributes by proposing a technique for horizontally fragmenting some class models in a DOBS. Class models are classified based on the attributes and methods they contain.

The balance of the paper is organized as follows. Section 2 reviews previous work on fragmentation, relevant aspects of some existing object based and distributed object based systems. Section 3 briefly introduces our classification of class models and outlines a technique we propose for horizontal fragmentation of class models. Section 4 makes some concluding remarks and suggests future research directions.

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

2 Previous/Related Work

This section reviews other work on distributed database design, object-oriented database systems as well as work on distributed object systems and/or Client/Server object-oriented databases. Section 2.1 presents a review of the previous work on fragmentation in the relational database system, Section 2.2 reviews some existing object-oriented systems. Section 2.3 reviews previous work on distributed object systems as well as Client/Server object-oriented databases while Section 2.4 discusses previous work on fragmentation in object-oriented systems.

2.1 Distributed Design (Relational)

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

2.1.1 Horizontal Fragmentation

Several researchers have worked on fragmentation and allocation in the relational data model including Ceri, Negri and Pelagatti [5], Özsu and Valduriez [30], Navathe *et al.* [26], Navathe and Ra [28], and Shin and Irani [6].

Ceri, Negri and Pelagatti [5] show that the main optimization parameter needed for horizontal fragmentation is the number of accesses performed by the application programs to different portions of data (file of records). They characterize the horizontal partitioning problem in distributed database design, demonstrate file partitioning on primary and secondary memory, and distribute data to different devices. They define applications in terms of boolean *predicates* and use access pattern information to do the design. Predicates are collected into sets. A set of simple predicates is *complete* if and only if the probability of accessing any two records belonging to a minterm fragment is the same. A complete set of predicates is *minimal* if and only if all its elements are relevant. Further, implications exist between boolean expressions of simple predicates which eliminate meaningless minterms.

Navathe, Karlapalem and Ra [27] define a scheme for simultaneously applying the horizontal and vertical fragmentation algorithms on a relation to produce a grid. A technique similar to the vertical fragmentation schemes discussed in [26, 28] is used to produce horizontal fragments. They consider transactions with high access frequencies using simple or derived predicates to produce horizontal fragments. A predicate usage matrix is obtained from transaction information and leads to a predicate affinity matrix that clusters the predicate set into subsets. These are optimized using predicate inclusion and implications so the final horizontal fragments can be formed.

Özsu and Valduriez [30] define the database information needed for horizontal fragmentation of the universal relation and show how the database relations are reconstructible using joins. Ceri *et al* [7] model this relationship explicitly using directed links drawn between relations via equijoin operations. The relation at the tail of a link is called the *owner* of the link and the relation at the head is the *member* [7]. They use the same concepts and parameters as well as methodology identified in [5, 7] to define horizontal fragmentation algorithms. Primary horizontal fragmentation is performed on all owner relations of the database schema and the steps for primary horizontal fragmentation are: (1) obtain the set of complete and minimal predicates as in Ceri *et al.* [5] (generated by the COM-MIN algorithm), (2) define the set of minterm predicates which are the conjuncts of the predicates, either in their natural or negated forms, and (3) eliminate meaningless minterm fragments using the implication set [5]. They define derived horizontal fragmentation on all member relations of a link according to a selection operation specified on its owner relation. The link between the owner and member relations is an equijoin implemented as a semijoin.

Shin and Irani [33] partition relations horizontally based on estimated user reference clusters (URCs). URCs are estimated from user queries [5, 30] but are refined using semantic knowledge of the relations. There is a knowledge–based system containing specific knowledge about the data and an inference mechanism applied to the knowledge–base generates new equivalent queries from which the URCs are derived.

2.1.2 Vertical Fragmentation

Hoffer and Severance [12] define an algorithm that clusters attributes of a database entity based on their affinity. Attributes accessed together by applications have high affinity so the Bond Energy Algorithm developed by McCormick *et al.* [24] is used to form these attribute clusters. Navathe *et al.* [26] extends Hoffer's work by defining algorithms for grouping attributes into overlapping and nonoverlapping fragments. They apply vertical partitioning to three database types, namely, distributed databases, one memory level database and databases arranged in a memory hierarchy. The approach is to minimize the number of fragments visited by a transaction and to refine fragments using cost factors that reflect the physical environment where the fragments are stored. Cornell and Yu [8] optimized this work by developing an algorithm that obtains an optimal binary partitioning for relational databases. They use knowledge of physical factors[2] to decrease the number of disk accesses. Further refinement is accomplished by applying the binary vertical partition algorithm iteratively [26]. Navathe and Ra [28] developed an algorithm that uses a graphical technique where the attribute affinity matrix is represented as a graph from which a linearly connected spanning tree is generated and all cycles on the spanning tree form fragments of the relation.

Özsu and Valduriez [30] review these earlier work on vertical partitioning for distributed databases using the access frequency information and the Bond Energy Algorithm that groups attributes of a relation based on the attribute affinity values. Groups of attributes are clustered and cost equations are

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

used to define the best position along the diagonal of this clustered affinity matrix for splitting relations into fragments.

Muthuraj *et al.* [25] argue that earlier algorithms for vertical partitioning are *ad hoc*, so they propose an objective function called the Partition Evaluator to determine the "goodness" of the partitions generated by various algorthms. The Partition Evaluator has two terms; namely, irrelevant local attribute access cost and relevant remote attribute access cost. The irrelevant local attribute cost term measures the local processing cost of transactions due to irrelevant fragment attributes. The relevant remote attribute access term measures the remote processing cost due to remote transactions accessing fragment relevant attributes. The two components of the Partition Evaluator are responsive to partition sizes. The vertical partition algorithms whose results could be evaluated using this objective function include: the bond energy algorithm [24], binary vertical partition [26], Ra's graphical algorithm [28] and an exhaustive enumeration algorithm.

2.1.3 Hybrid

Pernul, Karlapalem and Navathe [31] argue the user view is an important semantic not captured in distributed design of the traditional data models. They argue that a view can encompass a set of transactions and different views may overlap to give rise to relationships between views. From these relationships fragments are derived by using the horizontal, vertical and derived fragmentation operators defined earlier [5, 26]. Their methodology involves decomposing relations into a set of disjoint fragments based on views where one or more fragments represents a view. To obtain mixed fragments, they find all overlapping views and for each pair of overlapping views perform a structured decomposition by applying vertical, primary and derived horizontal operators to acquire the smallest non-overlapping set.

2.2 Object-Oriented Database Systems

This section presents a review of some features relevant to the issue of encapsulation and inheritance supported by some existing object-oriented systems. The systems reviewed are: GEMSTONE [2], EXODUS [3], ORION [20], ENCORE [34], POSTGRES [35, 36], STARBURST [22], IRIS [10] and O_2 [9].

GEMSTONE is an object-based system developed at Servio [2, 23] whose basic architecture distinguishes two main subsystems; the GEM Server process for query evaluation and the Stone Monitor for allocating object identifiers in blocks and coordinating commit activity. Its object-oriented database language (OPAL) is used for data definition, data manipulation and general computation. GEMSTONE provides support for encapsulation, both single and multiple inheritance as well as attribute-value inheritance. User access to data is permitted through both declarative and procedural language. An object is stored as a chunk of memory. It uses class defining objects for management of its metadata and objects communicate through message passing. GEMSTONE does not provide distributed services.

ORION [18, 19, 20] is built at Microelectronics and Computer Technology Corporation (MCC) in

Austin, Texas. ORION has three different systems ORION-1, ORION-1SX, and ORION-2. ORION-1 is a single user, multitask system, ORION-1SX is a client/server system on LAN, while ORION-2 is a distributed object-management system in which all computers in the network participate in the management of the shared persistent database. The ORION-1 architecture consists of:

1. Object Subsystem for providing high-level functions such as schema evolution, version control, query optimization, and multimedia information management.

2. Storage Subsystem which provides access to objects on disk.

3. Transaction Subsystem which handles transaction management.

4. Message Handler which handles three types of messages - user-defined method, access message for accessing the value of an attribute of a class; and system-defined functions for schema definition, creation and deletion of instances; and transaction management.

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

ENCORE [13, 34] is an OODBS with origin at Brown university, department of Computer Science. The database has two subsystems, a typeless backend ObServer that manages the persistent object store and the other component that enforces the type system, ENCORE. ENCORE supports encapsulation, single and multiple inheritance. It stores an object as an aggregate of chunks of memory representing all superclasses up to the root. Metadata are managed by objects at a higher level and objects communicate through exported interface. The Client/Server aspect of ENCORE is discussed in Section 2.3.

EXODUS [3, 4] is an extensible database management system developed at the university of Wisconsin with the overall goal of implementing a storage system to support development of abstract data types, application-specific access methods, operations and version management, in addition to other features. Its storage objects are stored as uninterpreted byte sequences of unlimited size and a global set of operations are defined for manipulating every large storage object and another global set of operations is used for manipulating any file storage object. Thus, EXODUS provides weak support for encapsulation and there is no clear support for inheritance at this low level of representation. EXODUS uses hints for the management of metadata. There is no direct communication between objects at this level.

POSTGRES [32, 35, 36, 37, 38] is a database system built at the university of California, Berkeley. It supports a data model which is relational but extended with abstract data types, attribute and procedure inheritance. POSTGRES uses a 'no-overwrite' storage manager whereby old records remain in the database whenever an update occurs. There is a repository of procedures in the system which are bound to classes, attributes or tuples using the types of the arguments. Thus, there is no strong support for encapsulation. POSTGRES supports both single and multiple inheritance. POSTGRES uses a system catalog for managing metadata.

STARBUST [11, 22] is an extensible system built at IBM's Almaden Research Centre and is aimed at extending the relational database system to capture object-oriented and knowledge-based system features. Access to objects and their components is only through operations defined on the table. It provides support for both single, multiple and attribute value inheritance. User access to database objects is through a declarative language. Objects are stored as nested or ordinary relations and there is no direct communication between objects. STARBURST does not support distributed services and metadata are managed with tables.

IRIS [10, 39] DBMS is developed at Hewlett-Packard laboratories. The layered architecture of IRIS comprises the object manager on top of the storage manager. The IRIS storage manager is a conventional storage subsystem similar to system R's RSS. There is no explicit bundling of structure and behaviour of data as an object and thus IRIS provides weak support for encapsulation. Objects are accessed, using functions. It supports both single and mutiple inheritance but provides no support for attribute value inheritance. An object at a higher level is used for managing metadata and there is no direct communication between objects.

O_2 [9] is an OODBS with origin traced to ALTAIR, a project funded by 1N2 (a Siemens Subsidiary), INRIA (Institut National de Recherche en Informatique) and LRI (Laboratoire de Recherche en Informatique, University of Paris XI). The O_2 architecture consists of eight functional modules which include the object manager and the disk manager. O_2's disk manager is Wisconsin Storage System (WiSS). It provides both declarative and procedural user-interface and objects are stored as WiSS records. Metadata are managed by objects at a higher level and objects communicate through message passing. O_2 provides strong support for encapsulation and supports both single and multiple inheritance but no support for attribute value inheritance.

2.3 Early Work on Distributed & Client/Server OODBs

This section presents a review of existing object based systems that are either distributed or support client/server architecture. The OODB's reviewed are ITASCA [14], ENCORE [13, 34], GOBLIN [16], THOR [21], and EOS [29].

ITASCA [14] is a distributed active object database management system developed by Itasca Systems, a member of MCC. Itasca emerged from the ORION prototypes. ORION2 has a distributed, object-oriented architecture for multi-client, multi-server architecture and ITASCA is an extension of ORION-2. The distributed environment of ITASCA allows shared and private databases and the shared database is distributed accross workstations sites in the network. An ITASCA server controls the partition of the shared database at each site. Private databases allow private data not shared with other users of the database. The schema is stored redundantly at each site but each instance of data is stored at one site. There is no central data server nor central name server. The user or application does not need to know the location of a targeted object in the database. Location of data may be changed by the user or 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. Only data can move among sites at execution time. There is an object directory per site which maps unique identifications to physical locations on either the shared or private database partitions of the local site.

The ObServer component of ENCORE [13] reads and writes chunks of memory from secondary storage. The type level communicates with the Server through the UNIX remote procedure call (RPC) mechanism. The functions of ObServer include: (1) managing chunks of memory in secondary storage with a unique identifier (UID) attached to each chunk, and (2) maintaining correspondence between UIDs and chunks of memory. There is a network of workstations (nodes) running independent processes. A server and its data reside on a single node and processes on other nodes could access this server. To communicate with the server, the process binds a module called the client into its image; thus, the client and the server can reside on different machines. The ENCORE module uses the object server as a backend; and when two different processes on two different machines are using ENCORE database, separate copies of ENCORE must reside on each machine. A binder process provides a client with a connection to the desired database. When a client wants to access a database, it issues a request to the binder which returns information identifying the server attached to this database and to get the client connected to the server. This enables access to multiple databases. Clustering of groups of related objects as a segment stored on disk is used to improve performance. A segment is of variable size and is the unit of transfer for objects between client and server; and from secondary storage to main memory. Thus, the segments collectively provide a partition of the objects within a database; and every database object is contained in at least one segment.

GOBLIN [16] is a distributed OODBS developed at Kruislaan 413, Amster dam. GOBLIN runs in an environment consisting of a network of workstations, with memory-resident databases. The storage scheme used by GOBLIN is a three-level directory hierarchy for complex objects which uses attribute collections with multiple incarnations called Binary Association Tables (BAT). The Class Administration Table (CAT) describes the structural relationships between object components for a specific database schema. Finally, the mapping from schema to database instances is administered by a Redistribution Admnistration Table (RAT), including the partitioning and distribution information. The storage management functions are of three types – schema functions, path functions and storage functions. The schema functions describe the relationships to be maintained; e.g., functions that map a class identifier (clid) to the set of object ids (oid) representing instance objects of this class. The counterpart of the oid or clid at the physical level is called the Physical Identifier (PID). Schema functions have corresponding path functions; for example, the schema function P_person: CLID $\rightarrow$ SET(OID) that relates a class id to its set of extent has the corresponding path function P_{person}: PID $\rightarrow$ SET(PID) which maps a class to its instance objects. The mapping between the extensional and intensional layers is captured by the storage functions, e.g, functions that relate OIDs and values at the schema level to PIDs and values at the implementation level. This mapping between OIDs and PIDs is achieved using an object table; the OIDs and PIDs can also be made identical. The path function definitions are administered in Binary Association Tables; the schema functions are captured in a Class Administration Table while the storage functions are captured in a Redistribution Administration Table. Object persistence, stability and consistency are not addressed by the BAT, CAT or RAT but left for the distributed operating system or handled by the Class Manager, which resides on each processor.

THOR [21] is a distributed system developed at Massachusetts Institute of Technology (MIT), laboratory for Computer Science. THOR is a distributed system in which objects are stored at server nodes distinct from the machines where clients programs reside. Copies of THOR frontends run at client machines and perform the tasks of caching and prefetching objects and running operations on objects locally. Its implementation is fully distributed because clients are separated from servers which are themselves distributed. The system architecture comprises a set of computing nodes connected by a network. Some of these nodes are THOR servers that store objects in THOR universe, while others are client nodes where users of THOR run their programs. It is possible to have a single node act as both a server and a client. The THOR system runs frontends (FEs) at client nodes and backends (BEs) and object repositories (ORs) at the servers. While FEs and BEs handle type and operations on type, the ORs deal with managing the resilient storage for objects. Every resilient object resides at one of the ORs, usually the first OR selected when the object became permanent. However, objects can move from one OR to another under user control. An OR's persistent objects are stored on disk organized into variable-length segments. Each segment contains a group of related objects (a cluster). An object is identified by its *oref* which is local to the OR containing the object; and comprises of segment id (sid) and 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 [29] is an object-oriented programming environment for distributed systems developed at INRIA, France. Leos, the language of EOS is an object-oriented programming language that provides transparency for both distribution and persistence. EOS provides a flat, virtual and garbage collected object space which need to be paginated and distributed. Object grouping is used to minimize cost of distribution. Object grouping includes both object declustering onto nodes and object clustering on disks. Object declustering is the partitioning of an object into subobjects and a mapping of these subobjects onto nodes. Object clustering is the process of grouping objects on disks. Objects accessed together are grouped closely on disks, usually determined by access patterns of computations which are in turn estimated by object aggregation. EOS's object model permits the definition of arbitrarily complex objects as aggregation of objects of arbitrary depth. An object is an instance of a class which holds a system wide identifier. Uniform persistence is supported so any object persists as long as it is reachable from a root of persistence. Objects are grouped in *harbor* and *pier* containers. The harbors are used to improve on node locality and a harbor is local to a single node. A harbor is divided in pier containers each of which is a set of disk tracks. Within the containers, objects are grouped with their most relevant parent.

2.4 Distributed Design (Object-Oriented)

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

2.4.1 Horizontal Fragmentation

Karlapalem, Navathe and Morsi [15] identify some of the fragmentation issues in object bases including: How are subclasses of a fragment of a class handled? Which objects and attributes of the objects are being accessed by the methods? What type of methods are considered: simple methods that access a set of attribute values of an object or complex methods that access a set of objects and instance variables?[3] Further, they argue that a precise definition of the processing semantics of the applications is necessary. They do not present solutions for horizontally fragmenting class objects but argue that techniques used by Navathe *et al.* [27] for horizontal fragmentation could be applied.

2.4.2 Vertical Fragmentation

Karlapalem *et al.* [15] define issues involved in distribution design for an object oriented database system. They identify two types of methods – simple and complex methods. Their first model consists of simple methods. They argue a model using simple methods can be vertically partitioned using techniques described by Navathe *et al.* [26], while the second case, requires a method-based view (MBV). The MBV identifies the set of objects accessed by a method and the set of attributes or instance variables accessed by the method. The sets are further grouped into sets of objects and instance variables based on the classes to which they belong. This generates the set pairs of objects and instance variables (O_i,I_i) accessed from a class C_i by a method. This is called method m_j's view of class C_i. They further suggest the use of concepts developed by Pernul *et al.* [31] to fragment classes based on views.

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

2.4.3 Hybrid Fragmentation

Karlapalem et al [15] propose forming groups of objects ($\mathcal{O}_i$) and their attributes ($\mathcal{A}_i$) of class C accessed by each method ($\mathcal{M}_i$) in the database to obtain pair sets ($\mathcal{O}_i$,$\mathcal{A}_i$) for each ($\mathcal{M}_i$). And each ($\mathcal{O}_i$,$\mathcal{A}_i$) defines the mixed class-fragment of class C accessed by method ($\mathcal{M}_i$) and is method $\mathcal{M}_i$'s view of class C.

3 A Horizontal Fragmentation Scheme For Class Models

The classes making up the DOBS are classified based on the nature of the attributes and methods they contain. There are two types of attributes – simple and complex. Simple attributes are of primitive types [1], while complex attributes refer to other classes in the database and have those classes as their type. There are two basic types of methods – simple methods and complex methods. Simple methods are those methods that do not invoke other methods of other classes, while complex methods are those that invoke other methods of other classes. When a simple method of a class which a complex attribute of another class has as its type is referenced by the containing class, it is seen as contained simple method because it is a simple method of a class that is contained in another class. Thus, the variety of class models that could be defined in a DOBS are:

1. Class models consisting of simple attributes and simple methods.
2. Class models consisting of complex attributes and contained simple methods.
3. Class models consisting of simple attributes and complex methods.
4. Class models consisting of complex attributes and complex methods.

The balance of this section presents an outline of the scheme we propose for horizontally fragmenting the first two class models given above. Some definitions suitable for fragmentation in both models are first presented.

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

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

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

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

Definition 3.5 *The cardinality of a class* is the number of instance objects in the class (denoted $\mathbf{card}(C_i)$). ∎

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

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

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

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

The object base information needed by the fragmentation procedure are of two types:

1. **The class lattice** showing the superclass–subclass relationship between all classes.
2. **The dependency graph**[4] captures the method link, attribute link or object link between any two classes in the database. This means that, if an attribute of a class C_i is another class C_j, then there is a an attribute link between classes C_i and C_j and is represented by an arc in the dependency graph ($C_i \rightarrow C_j$).

We assume that the database and application information required for distributed design are predefined and application information definitions are given as required. Input to the fragmentation process consists of a set of applications or user queries, set of database classes and the inheritance hierarchy information. The output expected from this fragmentation process is the set of horizontal fragments for all classes in the database. Four steps are required.

[4] An example of a graph depicting dependency information is the class composition hierarchy shown in Kim [17].

3.1 Classes Consisting of Simple Attributes and Simple Methods

The four steps required to fragment this class model are. For purposes of easy reference, this algorithm is called *HorizontalFrag*.

Step 1: Generate Link Graph: With this class model, the link graph is restricted to class lattice information since all attributes and methods are simple. Thus, with the class lattice as input, the rule for generating the link graph is that every subclass depends on its superclass, because this rule preserves the implicit partitioning of the inheritance hierarchy. Thus, in the link graph if class C_i depends on class C_j, an arc is inserted as $(C_i \rightarrow C_j)$.

Step 2: Define Primary Horizontal Fragments of Classes: We define primary horizontal fragments on all owner classes of link, which constitute all classes in the object base. Simple predicates belonging to each class are generated from all user queries. Using the same technique as in [30], we generate a set of minterm predicates $M_i = \{m_{i1}, m_{i2}, \ldots, m_{iz}\}$ for each class using the set of simple predicates $Pr_i = \{P_{i1}, P_{i2}, \ldots, P_{im}\}$ for the class. Finally, the semantics of the class is used to eliminate the meaningless predicates.

Step 3: Derived Horizontal Fragmentation on Member Classes: A derived horizontal fragmentation is the partitioning of a member class according to the fragmentation of its owner classes. Thus, a derived horizontal fragmentation is defined on the member classes of links according to fragments of its owners.

Step 4: Combining Primary and Derived Fragments: Since derived fragments of each member class may have objects that overlap with two or more of its primary fragments, it is necessary to determine the most appropriate primary fragment for the member objects of every derived fragment. Several simple heuristics could be used such as selecting the smallest or largest primary fragment or the primary fragment that overlaps the most with the derived fragment. Although these heuristics are simple and intuitive they do not capture any quantitative information about the distributed object base. Therefore, a more precise approach is described that captures the environment and attempts to use it in merging derived fragments with primary fragments.

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

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

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

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

Irrelevant accesses are those in the derived fragment but not in the primary fragment.

Definition 3.10 *Irrelevant access* are those made to local objects of both fragments F^d and F^p as follows: $\sum_{I_j}$ acco(I_j), for all $I_j \in F^p - F^d$. ■

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

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

Two additional definitions are required. First, the affinity between two objects is the sum of each objects accesses (affinity), if they are both accessed by the same query.

Definition 3.12 *The Object Affinity* is the affinity between two objects I_i and I_j as follows:

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

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

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

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

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

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

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

The final step of this process places fragments when there is no suitable primary fragments by making the derived fragments primary.

3.2 Class Objects – Complex Attributes and Simple Methods

This section sketches an algorithm for horizontally fragmenting classes consisting of complex attributes that support class composition hierarchy using simple method invocations. The algorithm first uses the attribute links between classes using the class composition hierarchy [17]. The class composition hierarchy is used to define the link graph showing the dependencies between any two classes. If class C_i has class C_j as part–of it there is a link from class C_j to class C_i in the link graph. The first iteration invokes the Horizontal Fragmentation algorithm described above (*HorizontalFrag*) using the link graph from the composition hierarchy to produce an initial fragmentation. The second iteration of the algorithm executes *HorizontalFrag* with the link graph defined using the inheritance hierarchy to produce fragments based on applications that the preserve inheritance hierarchy too. This first iteration is summarized informally as follows:

1. Form a class link graph from the class composition hierarchy. If class C_i contains class C_j form a link from class C_j to C_i.

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

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

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

These steps describe the first iteration from which horizontal fragments of containing classes based on the class composition hierarchy are generated. These are taken as the primary horizontal fragments of the containing class in the class lattice. The final horizontal fragments of the containing classes are produced using the class inheritance lattice to obtain a "new" link graph. The *HorizontalFrag* algorithm uses this link graph and the fragments of containing class from the first iteration as its primary fragments. It is possible that the second application of the algorithm results in the original class. This means the application requirements make fragmentation unnecessary.

4 Conclusions

This paper presents a comprehensive review of early work on distributed database design in both the relational and object based systems. A review of some existing object and distributed object based systems is also presented. This paper further defines issues involved in class fragmentation in a distributed object based system. The issues considered include the inheritance hierarchy, the nature of attributes of a class, and the nature of methods in the classes. Finally, algorithms for horizontally fragmenting two class models are proposed, and the models considered are: (1) a model consisting of classes with simple attributes and methods, and (2) a model consisting of classes with attributes that support a class composition hierarchy using simple methods.

Ongoing research includes extending the model to support complex method invocations and discovering suitable theoretical models to allocate fragments. We are also working on vertical and hybrid fragmentation schemes for class models.

References

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

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

[3] M.J. Carey, D.J. DeWitt, J.E. Richardson, and E.J. Shekita. Object and file management in the exodus extensible database system. In *Proceedings of the 12th International Conference on VLDBs*, 1986.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[18] W. Kim, N. Ballou, H. Chou J.F. Garza, and D. Woelk. Features of the orion object-oriented database system. In Kim Won and Lochovsky F.H., editors, *Object-Oriented Concepts, Databases and Applications.* ACM Press, 1989.

[19] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the orion next-generation database system. *IEEE Transactions on knowledge and Data Engineering*, 2(1), March 1990.

[20] Won Kim and H. Chou. Versions of schema for object-oriented databases. In *Proceedings of the 14th Very Large Data.* Base Conference, Los Angeles, California, 1988. pp.148-159.

[21] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in thor. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

[22] G.M. Lohman, B. Lindsay, H. Pirahesh, and K.B.Schiefer. Extensions to starburst: Object, types, functions and rules. *Communications of the ACM*, 34(10), Oct 1991.

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

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

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

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

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

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

[29] Gruber Oliver and Amsaleg Laurent. Object grouping in eos. In M. Tamer Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann Publishers, 1994.

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

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

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

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

[34] A.H. Skarra and S.B. Zdonik. The management of changing types in an object-oriented database. In *Proceedings of the first ACM Conference on Object-Oriented Programming System, Languages and Applications*, volume 21(11), pages 483–495. SIGPLAN Notices, 1986. Vol.21, No.11, pp. 483-495.

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

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

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

[38] M. Sullivan and M. Olson. An index implementation supporting fast recovery for the postgres storage system. *IEEE Transactions on knowledge and Data Engineering*, 2(3), Sept. 1992.

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