STORAGE MANAGEMENT AND INDEXING IN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND INFORMATION SCIENCES
AND THE INSTITUTE OF ENGINEERING AND
SCIENCES OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

By
Reda AL-HAJJ
June 1990

STORAGE MANAGEMENT AND INDEXING IN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

A THESIS
SUBMITTED TO THE DEPARTMENT OF COMPUTER
ENGINEERING AND
INFORMATION SCIENCES
AND THE INSTITUTE OF ENGINEERING AND SCIENCES
OF BILKENT UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Reda Al-Hajj

By
Reda AL-HAJJ

June 1990

I certify that I have read this thesis and in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Masters of Science.

Prof.Dr. Erol Arkun (Principal Advisor)

I certify that I have read this thesis and in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Masters of Science.

Asuman

Prof.Dr. Asuman Doğaç

I certify that I have read this thesis and in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Masters of Science.

K. O

Asst.Prof.Dr. Kemal Oflazer

Approved for the Institute of Engineering and Sciences:

M. Baray

Prof.Dr. Mehmet Baray, Director of the Institute of Engineering and Sciences.

ABSTRACT

STORAGE MANAGEMENT AND INDEXING IN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEMS

Reda AL-HAJJ
M.S. in Computer Engineering and Information Sciences
Supervisor : Prof.Dr. Erol Arkun
June 1990

Storage management and indexing methods used in existing conventional database management systems are not appropriate for the object-oriented database management systems due to the distinctive features of the later systems. A model for storage management suitable for object-oriented database management systems is proposed in this thesis. It supports object identity, multiple inheritance, composite objects, a fine degree of granularity and schema evolution.

An index provides fast access to data stored in files at the price of using additional storage space and an overhead in update operations. Work has been carried out on indexing and an indexing method for the object-oriented database systems is proposed. Identity and equality indexes are treated. Object identity and information hiding are provided. Schema changes are handled without affecting existing indexes. It is general enough to be applicable to most existing object-oriented database systems. The mapping of the proposed storage and indexing approaches into a relational database scheme is also presented.

Keywords: object-oriented database management systems, storage management, inheritance, data encapsulation, identity, schema evolution, degree of granularity, composite objects, indexing, identity index, equality index.

ÖZET

NESNESEL VERİ TABANI SİSTEMLERİNDE VERİ SAKLAMA VE INDEKSLEME

Reda AL-HAJJ
Bilgisayar Mühendisliği ve Enformatik Bilimleri Yüksek Lisans
Tez Yöneticisi: Prof.Dr. Erol Arkun
Haziran 1990

Klasik veri tabanı sistemlerinde kullanılmakta olan veri saklama ve indeksleme metotları nesnesel veri tabanı sistemlerinde kullanılmaya uygun değildir. Bu tezde nesnesel veri tabanı sistemlerinde kullanılmaya uygun bir veri saklama modeli sunulmaktadır. Bu model nesne kimliği, çoklu sınıf sıradüzeni, bütünleşik nesneler, küçük granül olanağı ve çoklu sınıf sıradüzeni günlemesini içermektedir.

Fazladan bellek kullanma ve güncelleme işlemlerindeki dezavantajlarına ragmen indeksleme, kütüklerde saklanan verilere hızlı bir şekilde erişimi sağlar. Bu çalışmada nesnesel veri tabanları için bir indeksleme metodu da önerilmektedir. Bu indeksleme metodu hem nesneleri, hem de her nesnenin bileşenlerini ayrı ayrı indeksleme olanağı sağlar. Böylece nesne kimliği ve bilgi gizlenmesi sağlanır. Çoklu sınıf sıradüzeni üzerindeki değişiklikler oluşturulmuş indeksleri etkilemez. Bu metod, klasik veri tabanı yönetim sistemlerinde de kullanıma uygundur. Nesnesel veri tabanları için önerilen veri saklama ve indeksleme metotlarının bağıntısal veri tabanlarına dönüşümleri de sunulmaktadır.

Anahtar Kelimeler: nesnesel veri tabanı sistemleri, yardımcı bellek, bilgi gizlenmesi, küçük granül olanağı, indeksleme, bütünleşik nesneler, çoklu sınıf sıradüzeni.

ACKNOWLEDGEMENTS

I am very grateful to my supervisor Prof.Dr. Erol Arkun who gave his suggestions and comments throughout the period of thesis research. He contributed to the thesis in a fundamental way with his thoughtful comments and observations. Without his steadfast encouragement I would not have undertaken this work, nor finished it once started. I wish to thank the Bilkent University and the Department of Computer Engineering and Information Science for providing a pleasant climate in which this work could be undertaken. Thanks are also extended to Faruk Polat for translating the abstract into Turkish. My heart is filled with affectionate gratitude to the members of my family for the moral support they provided.

Table of Contents

1. Introduction 1

2. Problem Definition and Requirements Analysis 3
   2.1. Problem Definition 3
      2.1.1. Why Object_Oriented Systems? 3
      2.1.2. Object Identity 4
      2.1.3. Information Hiding 4
      2.1.4. Inheritance 5
      2.1.5. Clustering 5
      2.1.6. Composite Objects 6
      2.1.7. Persistence 6
      2.1.8. Schema Evolution 7
      2.1.9. Structural Dynamism: Extensible Object Size 7

   2.2. Goals and Requirements Specification 8
      2.2.1. Efficient Use of Memory 10

3. Existing Approaches to Storage Management 11
   3.1. Introduction 11
   3.2. ORION 12
   3.3. EXODUS 13
   3.4. GemStone 15
   3.5. ENCORE 16

4. Description of the Object Storage Model 23
   4.1. Rationalization 23
      4.1.1. A Step Toward the Proposed System 24
   4.2. The Proposed Model 24
      4.2.1. Tables and Mappings 24
      4.2.2. An Example 26
      4.2.3. Construction of the IT and NT 32
      4.2.4. Why to Separate Atomic from Nonatomic Values? 33
   4.3. Characteristics of the Proposed System 33
      4.3.1. Efficiency of Access and Memory Utilization 33
      4.3.2. Schema Evolution 33
      4.3.3. Composite Objects 33
      4.3.4. Clustering 34
      4.3.5. Locking 34
   4.4. Comparisons and Evaluations 34

5. Indexing 35
   5.1. What is Indexing 35
   5.2. Problem Definition 35
      5.2.1. What to Index? 35
      5.2.2. Identity Indexes and Equality Indexes 36
   5.3. Requirements of Indexing in Object_Oriented Systems 36
      5.3.1. Improving Performance 37
      5.3.2. Conserving Encapsulation 37
      5.3.3. What Should be Indexed? 37
   5.4. Existing Approaches to Indexing 38
      5.4.1. Indexing in GemStone 38
      5.4.2. Indexing in the CONTAINER 38
      5.4.3. Indexing in EXODUS 39
      5.4.4. Problems with the Described Approches 39

5.5. A Proposed Indexing Method 39
5.5.1. Identity Index 39
5.5.2. Equality Index 42
5.5.3. Index Creation 42
5.5.4. Schema Changes and Indexing 45
5.5.5. Query Processing 45
5.5.6. Application to Other Systems 46
5.5.7. Comparisons and Evaluations 46

6. Integrity Vs. Operations and Schema Changes 49
6.1. Integrity Preservation 49
6.2. Operations 49
6.2.1. Addition 50
6.2.2. Deletion 56
6.2.3. Fetching 60
6.2.4. Saving 63
6.2.5. Updating 63
6.3. Schema Evolution 63
6.4. Ease of Implementation 64
6.4.1. Data Structures 64
6.4.2. Function of the Storage System 64
6.4.3. How to Interact with the Storage system? 64

7. Mappings 65
7.1. Mapping Objects into Secondary Storage 65
7.1.1. Replacement Policy 65
7.1.2. Fetching Policy 68
7.2. Mapping the Storage System into a Relational Database System 69

8. Conclusions 73

References 74

LIST OF FIGURES

2.1. Related chunks 9

3.1. An example of a large storage object 15

3.2. The dereferencing process from an external UID to an object 19

3.3. The structure of the DBF and the segment 20

4.1. A three dimensional object model 25

4.2. Representation of a chunk inside the segment 25

4.3. Representation of a segment 25

4.4. Mapping dimensions of the object storage model into the IT, the NT and the segment 27

4.5. Format of the Disk Object Table (DOT) 27

4.6. A class hierarchy 28

4.7. An instance in the "student" class 29

4.8. The constructed DOT 30

4.9. The constructed IT 30

4.10. The constructed NT 31

4.11. The constructed segments 31

5.1. The constructed ROT for the example in Section 4.2.2. 40

5.2. The constructed SOT for the example in Section 4.2.2. 41

5.3. The augmented format of the Disk Object Table (DOT) 44

5.4. The constructed DOT for the example in Section 4.2.2. 44

LIST OF FIGURES

2.1. Related chunks 9

3.1. An example of a large storage object 15

3.2. The dereferencing process from an external UID to an object 19

3.3. The structure of the DBF and the segment 20

4.1. A three dimensional object model 25

4.2. Representation of a chunk inside the segment 25

4.3. Representation of a segment 25

4.4. Mapping dimensions of the object storage model into the IT, the NT and the segment 27

4.5. Format of the Disk Object Table (DOT) 27

4.6. A class hierarchy 28

4.7. An instance in the "student" class 29

4.8. The constructed DOT 30

4.9. The constructed IT 30

4.10. The constructed NT 31

4.11. The constructed segments 31

5.1. The constructed ROT for the example in Section 4.2.2. 40

5.2. The constructed SOT for the example in Section 4.2.2. 41

5.3. The augmented format of the Disk Object Table (DOT) 44

5.4. The constructed DOT for the example in Section 4.2.2. 44

LIST OF ALGORITHMS

4.1. Construct the IT, the NT for Objects in the database 32

5.1. Construct the IT, NT, SOT and ROT for Objects in the database 43

6.1. Add a class to the class hierarchy/lattice 51

6.2. Add a chunk to a class 53

6.3. Add an object to the database 54

6.4. Add an instance variable to a class 55

6.5. Delete a chunk from a class 57

6.6. Delete a class 58

6.7. Delete an Object 59

6.8. Delete an instance variable from a class 60

6.9. Retrieving instances of a class 61

6.10. Retrieving all the chunks of an object 62

6.11. Retrieving a chunk 62

6.12. Save a chunk 63

7.1. Find space in main memory 67

7.2. Evaluate an Expression 68

Chapter 1

INTRODUCTION

A database system deals with a huge amount of information that lives beyond the lifetime of the generating application. In addition, it is not possible to keep all the database information resident in main memory to serve a running application; only a small part of the information could be present in main memory and the rest should be kept on external storage on a permanent basis.

Traditional data models, such as the relational one, have achieved great efficiency in data storage and retrieval by restricting the modeling power; in particular, the database is assumed to be a complete and accurate model of a world where all the individual objects are restricted to be primitive values like numbers and strings and all their interrelationships are known and explicitly stated. The relational model of data has a flat view of the world, with all information expressed in the form of tables.

After conventional database management systems failed to satisfy the needs of new application areas -including Office Information systems (OIS), Artificial Intelligent (AI), Computer Aided Design/ Manufacturing (CAD/CAM), and others-one research direction on database systems is concerned withextending the object-oriented approach [16] to the database field and hence many object-oriented database systems have been designed [2, 7, 11, 22, 40, 45, 54, 56].

Although there is no clear definition of what object-orientation is; there are some basic concepts and properties of this approach. In an object-oriented system, all conceptual entities are modeled as objects which combine the properties of the procedures and data. In other words, an object has two parts: a private part and a public interface.

The private part specifies the internal implementation of the object and the public interface is used to communicate with other objects. These two parts capture both the state and the behaviour of the object. The state of an object is represented by instance variables, and the behaviour of the object is encapsulated in methods. In addition, both methods and instance variables are hidden from other objects.

Objects interact using the interface part to access the private part. The interface part of an object constitutes the messages understandable by the object and sent by other objects on the need to access the private part of the object. So, methods are invoked by messages.

Objects that have the same private part definition and interface part, may be collected into a class that includes the common definition of the private part and interface part. More than that, an object may have a part of its private part defined in an object in another class; so instead of duplicating the definition of the same part in the two classes, objects in the former class are said to inherit the common part from objects defined in the later class; leading to a class hierarchy if objects in the a class inherit from only one class, or a class lattice if objects in a class could inherit from more than one class.

Informally, an object-oriented system may be defined as a system which supports data encapsulation and inheritance. Another definition states that an object-oriented system is a system that supports data encapsulation and not necessarily inheritance [49].

The results reported here represent a continuation of the research work on the object-oriented database management system ODS. The earlier results of this research are: the design of an Object

Memory [32], a Message-Passing Model [44], and a Storage Manager [31, 45], the design of a Data Definition and Data Manipulation Language [53] and a Run-Time Environment [54, 57].

The emergence of object-oriented systems necessitates the development of new storage and indexing techniques due to the nature of object-oriented constructs [49] that make it not possible to use the existing conventional techniques, at least without some adjustments [24, 36].

In this thesis, we describe a model for storage management and indexing in object-oriented database management systems [4, 5]. In Chapter 2, problems of storage management in object-oriented database management systems are stated; goals and requirements to be achieved by the proposed storage system are identified. Problems arise due to the new constructs of object-orientation, such as encapsulation, inheritance, composite objects, complex objects, and schema evolution. The goal is to solve these problems as much as possible. In chapter 3, a study on related work on storage management is presented; the problems encountered with such approaches are identified. In chapter 4, the proposed object storage model is explained; the new structures employed are described and properties of the model are mentioned. Finally, a comparative study and evaluation of the proposed system is carried out.

Due to encapsulation and information hiding, indexing becomes a nontrivial problem to treat. However, a treatment of the indexing problem in object-oriented database management systems, consistent with the proposed storage system, is presented in Chapter 5. We tried to stay within the realm of object-orientation in the proposed indexing method. Problems of indexing in object-oriented database management systems are stated; goals and requirements to be achieved by the proposed indexing method are identified; related work is described; identity and equality indexes within the realm of the proposed system are discussed; index set up is described; application to the existing storage systems is described; comparisons and evaluations are presented.

One of the primary functions of a database system is to maintain the integrity of the database, i.e., to preserve the consistency and correctness of the database. Integrity preservation and how different operations are performed are discussed in Chapter 6. Also the algorithms for each operation are presented. In Chapter 7 the mapping of objects between main memory and secondary storage is explained and the mapping of the proposed storage model into a relational database system is presented. Chapter 8 includes the conclusions.

Chapter 2

PROBLEM DEFINITION AND REQUIREMENTS SPECIFICATION

2.1 Problem Definition

A database management system must be capable of handling large amounts of data. Dealing with large amount of data usually involves storing them on on-line, direct access secondary storage devices such as disks, and making information available to the application system by managing the transfer of data between main memory and secondary storage devices. Object-oriented database management systems have distinguishing characteristics that make it not possible to use the storage techniques of conventional database management systems in object-oriented database management systems. Due to these distinguishing features, that are to be discussed in the following sections, new storage techniques to be discussed in Chapter 3 are under research.

2.1.1 Why Object-Oriented Systems?

A database is normally used to maintain a model of reality. Traditional data models, such as the relational one, have achieved great efficiency in data storage and retrieval by restricting modeling power, in particular, the database is assumed to be a complete and accurate model of the world where all the individual objects are restricted to be primitive values like numbers and strings, and all their interrelationships are known and explicitly stated. The relational model of data has a flat view of the world, with all information expressed in the form of tables. While undeniably of extensive value, this makes traditional data models unsuitable for a number of situations [9], for example:

- when complex objects are the natural way of describing the domain,
- when information about the domain is incomplete or becomes available incrementally, and
- when the database should be taking a more active role in deducing relationships rather that being a passive repository of data.

Object-oriented database systems is a new approach that tries to model the real world by representing each item by an object. Because many items have common properties, behaviors, and structure they said to fall in the same class. A class keeps the definitions common to its objects and all the functions that are applicable to the objects that the class acquires.

Object-oriented database systems have evolved after it became a fact that existing conventional database systems are not able to model well enough the new application areas, like Office Automation, Computer Aided Design and Manufacturing, and Artificial Intelligence. Object-Oriented database management systems can be distinguished from their more conventional counterparts by their ability to deal with arbitrary object types in an environment that is constantly changing. It should be noted that an object-oriented database management system is a natural evolution of conventional database technology.

With conventional database systems, an item undergoes some normalizations before it comes to the state which can be represented in the database. These normalizations are required due to the restricted data types defined in conventional database systems. Such restrictions lead to a semantic gap between items in the real world and their representations in conventional database systems.

Instead, the new research area, titled Object-oriented database systems, tries to overcome the semantic gap by dealing with each item as a stand alone object which has its own behaviour and structure. An object can been looked at as a closed box. Nothing is known related to what is found inside the box, except that it is possible to communicate with the box using some messages, understandable by the object. By message passing [1], it is possible to extract what is needed from the closed box; even if it is not known how the messages are to be executed inside the box. The object receives the message and replies by giving the result of the message interpretation, without allowing the message sender to know how the result was obtained.

In object-oriented database systems, an entity is no longer represented by using tuples (records) with atomic attributes (fields) of restricted types. Instead, an object forms the basis in object-oriented systems, to replace tuples and records used with conventional systems. An object models the real world in a better and more powerful way than all the preceding representations [49]. Objects are more powerful than records in that, they do not have only atomic fields as the values for their attributes (instance variables), but an attribute may have another object as its value. This may go on by nesting to have an object's instance variable as another object, up to the level that all the attributes of the final object in the link are atomic.

Storage management in an object-oriented database management system can be computationally expensive for the functions of storage allocation, object identity maintenance, garbage collection, and variable size object management. Large number of small objects, and small number of very large objects, must both be handled efficiently in both storage space and access time.

2.1.2 Object-Identity

The mapping of main memory objects [32] to their secondary storage counterparts must preserve the identity [33] of objects. Identity is that property of an object that distinguishes it from other objects. One powerful technique for supporting identity is through surrogates which are system generated globally unique identifiers, completely independent of any physical location or object values, called Object Oriented Pointers (OOPs). An OOP is the identity of an instance object in a class. The private memory of an instance object is a contiguous series of words which is called a chunk [32]. Objects interact by message passing [44] using object identity, not contents.

It is important that the identity of an object remains unchanged regardless of changes in its state, both in its internal main memory representation and external secondary storage representation. The concept of object-oriented pointers (OOPs) in main memory should be further extended to cover secondary storage. The mapping of main memory objects to their secondary storage counterparts must preserve the identity of the object. This implies that operations like retrieving or storing an object must be idempotent, i.e., if the same object is stored multiple times consecutively, its final effect should be the same as if the operation has been performed only once. The storage manager may employ different techniques to improve performance of retrieval, yet, the preservation of object identity must always be ensured.

2.1.3 Information Hiding

Information hiding [42] provides reliability and modifiability by reducing interdependencies among software components. The state of a software module is contained in private variables (the state of an object is contained in its private part), visible only from within the scope of the module. Only a localized set of procedures directly manipulates the data found in the private part. In addition, since the internal state variables of a module are not directly accessed, a carefully designed module interface may permit the internal data structures and procedures to be changed without affecting the implementation of other system modules. Object-orientation provides information hiding since an object captures both the state and the behaviour of an entity.

2.1.4 Inheritance

Objects that are defined to be in class B may have some properties that are also inherited from objects of some other class A. In this case instead of listing again all the properties with their accessing functions in the definition of class B, we let class B to inherit those properties from class A. Then with every object defined in class B we link an object that is defined in class A. Class A is called the superclass of class B, and class B is the subclass of class A. Class A contains two kinds of objects. The first kind contains objects defined in conjunction with its subclasses. The second kind contains objects that are rooted in class A without any external reference to them from the subclasses.

An instance X in class A, which is defined in conjunction with an instance Y in class B, cannot be found, because within class A nothing is included with the instances to refer to which instances in class B they are related to. Therefore, to find the instance in class A that is defined in conjunction with an instance in class B, communicate with the instance in class B, by sending a message to it, asking for the instance in class A. In other words, instances in class B contain references to instances in class A as supers. These references are considered unidirectional because no references are found from instances in class A to those in class B. Having in hand an instance from class A it is not possible to find the instance that references it as the super from class B.

Objects in class B are composed of two parts. The first part is an instance in class B, while the second part is an instance in class A. Each part is called a chunk. A chunk is that part of an object which is restricted to hold the properties and behavior as imposed by being in a particular class.

By this representation of classes, an object may inherit properties from one, called simple inheritance, or more classes, called multiple inheritance. In simple inheritance, a class can have at most one immediate superclass. While in multiple inheritance a class may have one or more immediate superclasses, forming a superclass list [49]. However, in both cases, one or more classes may form the subclass list of a certain class. In other words, more than one class may inherit properties from the same class.

Inheritance introduces name conflicts, i.e. the problem of two or more classes having variables or methods with the same name. The conflict may be between a class and one of its superclasses or between the superclasses of a class. The conflict problem between a class and one of its superclasses may also be seen in simple inheritance, and is solved by giving priority to the class. To solve the conflict problems in multiple inheritance, either all variable or method names of the superclasses must be distinct or the priority order for the superclasses should be specified [3].

2.1.5 Clustering

Forcing objects that have some properties in common to occupy adjacent locations on disk is known as clustering. Clustering is essentially an efficient and performance related usage of secondary storage which is not unique to object-oriented database management systems. In relational systems, clustering may be seen as taking rows from separate relations and storing them together on the same disk page. This means that clustering will improve the performance of join queries because the rows that are to be joined together are stored together.

The aim of any clustering scheme is to organize semantically related data together, which results in reduced diskhead movement and reduced physical I/O. With object- oriented database management systems, clustering is not as easy as it is with the conventional database management systems. This is because objects are multi-dimensional instead of being flat. One dimension results from the fact that objects can have other objects as the values for their instance variables. Another dimension can be seen along the hierarchy/lattice due to inheritance.

Algorithms used for manipulating multi-dimensional data in main memory are highly inappropriate for secondary storage, since they are usually implemented using linked structures and pointers; and such indirections are very expensive in secondary storage as they involve many disk lookups and transfers. Being disk-based in this sense does not simply mean paging main memory to disk as it overflows. The database should be intelligent about staging objects between main memory and disk. It should try to group objects accessed together onto the same disk pages, and try to anticipate which objects in main memory are likely to be used again soon, and organize its query processing to minimize disk traffic [43].

2.1.6 Composite Objects

Many applications require the ability to define and manipulate a set of objects as a single logical entity. A composite object is an object with a hierarchy/lattice of exclusive components considered as a unit of storage, retrieval, and integrity. The hierarchy/lattice of classes to which the object belongs forms a composite object hierarchy/lattice [7].

The basic object-oriented data model does not support composite objects; an object references but does not own other objects. A composite object captures the IS-PART-OF between a parent class and its component classes, while a class hierarchy/lattice represents the IS-A relationship between superclasses and their subclasses.

Composite objects introduce the concept of dependent objects [7, 34] which adds to the integrity features of an object-oriented data model. A dependent object is one whose existence depends on the existence of other objects and is owned by a single object. Since a dependent object can not be created before its owner exists, the composite object hierarchy/lattice must be developed in a topdown fashion, i.e., the root object of the hierarchy/lattice must be created first and then the children. When an object of a composite object is deleted, all its dependent objects must also be deleted.

An object may contain references to both dependent and independent objects, or to only dependent or independent objects. Such a general collection of objects is called an aggregate object. A composite object is, in fact, a special kind of an aggregate object.

When a composite object is instantiated all its parts are also instantiated. The instantiation process is recursive so composite objects can be used as parts. The automatic instantiation of all parts brings the restriction that a composite object can not be a part of itself An alternative is to instantiate parts on demand [49].

The composite object concept supports performance improvement through the clustering of related objects on disk. All components of a composite object can be clustered together, since whenever the root is accessed, most probably the other parts will also be accessed.

Composite objects increase information hiding and data encapsulation through the property of value propagation [7] which refers to the sharing of the value of an instance variable between instance objects.

2.1.7 Persistence

Information stored in the database should stay alive after the termination of the application that generates it. In other words, objects are expected to live beyond the user sessions in which they are created. The information managed must be persistent. The persistence of data should be transparent to the user and as a consequence, there should not be any specific operators to make an object persistent.

The assumption should be made that every kind of data should be potentially persistent, so that a procedure written to implement an algorithm on temporary data can work also on persistent data, and vice versa. Using different data types for persistent and temporary data is only an inconvenience for the programmer, while a complete homogeneity between persistent and temporary data allows him to focus on the algorithmic aspects of the problem [3].

Persistent objects that have regular structure, i.e., objects that form classes of homogeneous records, can be stored in the persistent storage in some sophisticated way, grouping and splitting records optimize the access time for some critical operation. But, persistent objects that have complex structures can not be represented efficiently using simple schema on persistent storage.

2.1.8 Schema Evolution

One of the important requirements of object-oriented database systems is schema evolution, i.e., the ability to dynamically make a wide variety of changes to the database schema.

Conventional database systems allow only a few types of schema changes. This is because the applications they support, i.e., conventional record-oriented business applications do not require more than a few types of schema changes; and also the data models they support are not as rich as object-oriented data models. In addition, traditional database models, including the relational one, separate the static aspects of databases from the dynamic aspects, primarily by defining an essentially static database schema, and separately defining queries and transaction languages. On the other hand, a central aspect of the object-oriented paradigm in the context of databases can be incorporated directly into the database schema, in the form of methods.

Most object-oriented systems support only a few changes to the schema and to the class definitions without requiring system shutdown. The operations that should be supported by an object-oriented system can be listed as follows [6, 7]:

1. Changes to the contents of a class.
   - (a) changes to an instance variable.
     - i. Add a new instance variable to a class
     - ii. Drop an existing instance variable from a class
     - iii. Change the name of an instance variable of a class
     - iv. Change the domain of an instance variable of a class
     - v. Change the default value of an instance variable
   - (b) changes to a method
2. Changes to an edge in the class hierarchy/lattice.
   - (a) Make a class a superclass of another class
   - (b) Remove a class from the superclass list of a class
   - (c) Change the order of superclasses of a class
3. Class changes
   - (a) Add a new class
   - (b) Delete an existing class
   - (c) Change the name of a class

An important problem related to schema evolution is seen when the structure of a class having some instances is modified. One approach is to modify all instances to reflect these changes immediately after the change is made in the class definition. A second approach is just to modify the class definition and modify the instances whenever they are referenced. The first approach is cumbersome and presents an overhead. However, the second approach is very difficult to implement and may cause inconsistencies. It also requires a way of keeping track of which instances have been modified and which have not [41].

2.1.9 Structural Dynamism: Extensible Object Size

Almost all of the conventional database management systems impose restrictions on the underlying data model, such as the size of a field. The effect of these restrictions may not be noticed in conventional data processing. However, in an environment -CAD/CAM, OIS, and AI- where there are objects of arbitrary size and structure, such restrictions can pose serious limitations.

Objects can be attributes in, and can inherit properties of, other objects. Due to that, the existing storage techniques used for conventional database systems are no more applicable to the Object-oriented database systems. This is because the object length is subject to change dynamically in object-oriented systems. There is no guarantee that two objects of the same class will have the same length. Add to this, that an object length may change dynamically due to what follows. A class may be added to the superclass chain of a certain class, between a class and one of its superclasses; as a result, a new chunk is added to the instances of all classes along the hierarchy/lattice affected by the change. A class may be deleted from the superclass chain/list that an existing class is inheriting from; resulting in diminishing the size of instances of all classes affected by the change. In addition, in the same class, new instance variables may be added or existing instance variables may be deleted. Therefore, the class structure evolution, and dynamic updating facilities are added to the database. Hence, record-field storage techniques are not useful. They can not be used even to model the object-oriented system. Because with the object-oriented system, there is no restriction on the value of an instance variable, we have objects with extensible, dynamically changeable, length.

2.2 Goals and Requirements Specification

Existing record-oriented database management systems fulfill many of the requirements of traditional database application domains, but they fall short of providing facilities well-suited to applications in OIS, CAD/CAM, and AI. The major goal, therefore, is to build a storage system that can meet the storage needs of the new application areas.

The proposed storage system should be flexible enough, so that it can be extended further in future research to include those database features that will be left out for the current time to keep the work within the scope of the current thesis work. Among the features that will be left out are some issues that are associated with multi-user database systems -such as authorization and concurrency control.

Bounds on the number and size of data objects should be determined only by the amount of secondary storage, not main memory limitations or artificial restrictions on data definitions. Thus, fields in a record can be of variable- length, with no fixed upper bound. Collections of objects such as arrays and sets, should not have a bound on the number of elements. Similarly, the total number of objects in a database system should not be arbitrarily limited. The system should handle both small and large objects with reasonable efficiency. The many small objects and the small number of large objects must be handled efficiently in both storage space and access time.

A goal that is also to be satisfied is that persistence of objects should be transparent to the users; since any object that a user has access to is implicitly persistent. The user does not need to specify direct operations on the persistent store of objects, it is rather the storage system's responsibility to do address mappings and all the associated database activities.

Another goal is that the storage system should be responsible for managing the transfer of objects between main memory and secondary storage, while making sure that the object identity is preserved throughout its internal and external representation.

Another goal can be seen as the need to cluster objects that are likely to be used together onto the same storage area by taking into consideration the relation between objects due to inheritance or an object (or a collection) being the value of an instance variable in another object.

Another goal is to satisfy the storage of composite objects by considering the root object together with all the component objects as a unit of storage and retrieval from the secondary storage. Components of composite objects should be treated as dependent objects.

Another goal is to satisfy the schema evolution functions. A class may be added or deleted from the class hierarchy/lattice; an instance variable may be added or deleted from a class definition; in addition to other schema updates described in the previous section, all are intended to be satisfied by the proposed system.

An important goal is that the storage system should be designed in a way so that indexing can be provided for fast and alternative access paths to the persistent object store. Indexing should not go out the realm of object-oriented concepts.

Finally, stable storage of data objects on disk should be supported, while location transparency to the application programmers on the movement of objects between main memory and secondary storage should be provided.

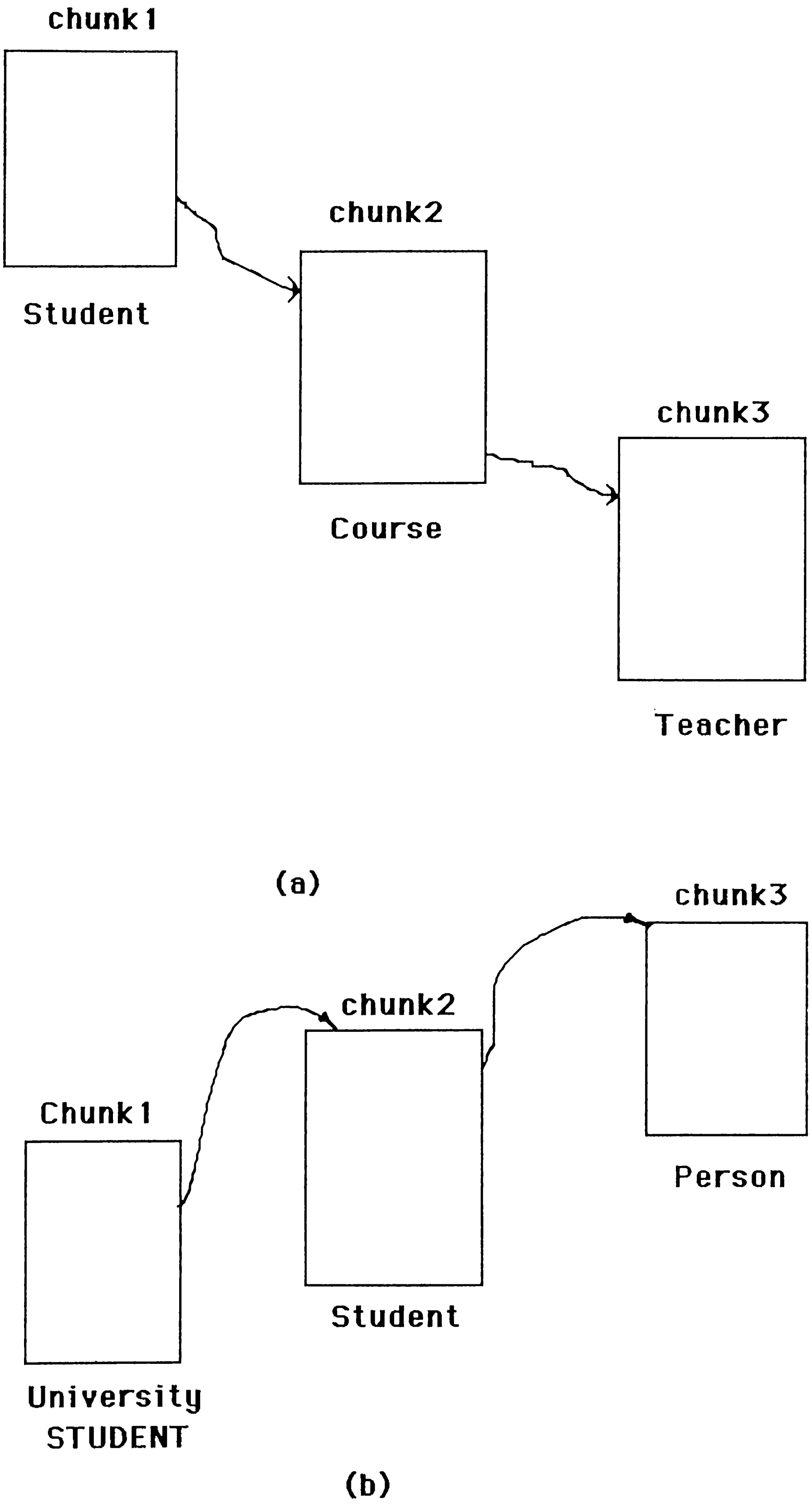


Figure 2.1: Related chunks

(a) Nested chunks
(b) Super chunks

2.2.1 Efficient Use of Memory

It is obvious that a database in any system is treated differently by different applications. All the information may not be interesting to a user that is only accessing the database for only a small piece of information. So why to let the information to be transferred entirely to the main memory. Instead, if only the needed information can be accessed then less space will be used in the main memory. The free space can be used to hold other useful information.

In addition, sometimes the whole information about an object, from which we need only a chunk, may be large enough so that it can not fit in main memory all at once. In this case, in addition to main memory loss think of time loss due to more accesses to get to the required piece. More than that, the problem will be more complicated and loss will be more and more if we need pieces from a set of objects that cannot fit each alone in main memory or even all in main memory at once!.

For example, on accessing a university database system as shown in Figure 2.1, for getting information about a student only chunkl is needed. So why to have chunk2 and chunk3 in main memory! Instead, if only chunkl can be brought to main memory, then three chunks, which are instances in the student class, may be present in main memory at once. These three chunks will occupy nearly the same space which was to be occupied by chunkl, chunk2 and chunk3 altogether. Another application may need to have chunk2 and chunk3 and may be the three chunks at once. So why not to serve the application with the needed chunks only.

Chapter 3

EXISTING APPROACHES TO STORAGE MANAGEMENT

3.1 Introduction

Object-oriented database management systems arose after existing data models, including the relational model, failed to satisfy the requirements of the new application areas. Each new application area has a specialized set of operations that must be efficiently supported by the database system. Efficient support for the specialized operations of each new application area is likely to require new types of storage structures and access methods as well.

Although the relational model has a flat view of the world, with all information expressed in the form of tables, some models have been proposed with extensions to the relational model to accommodate object-oriented needs. Those systems were derived by enforcing some object-oriented concepts into a relational model.

In [29] an attempt to fold the concept of hierarchy into a relational model of data storage is done by permitting classes to be used as attribute values in a relation. Other data models like POSTGRES [51] makes several extensions to relational algebra to be able to support object-oriented databases.

The database management system IRIS [20, 24, 36] is a research prototype of a next generation database management system, designed at the Hewlett-Packard Laboratories. The IRIS database management system has a relational storage subsystem that supports the dynamic creation and deletion of relations, concurrency, recovery, indexing, and buffer management.

ODDESSY [22] is implemented using Smalltalk-80 by incorporating the major features of the Semantic Data Model (SDM), the Structural Model and the entity-relationship model and aims at transforming the conceptual model into normalized relations using rules to generate functional dependencies which in turn produce third normal form relations, and finally mapping the logical design onto a specific Relational Database Management System.

On the other hand, other systems deal with the concept of storage management away from the existing systems; without any need to transform an object-oriented problem into, say a relational one. ODE [2] is a database system and environment based on the object paradigm. In ODE all persistent objects of the same type are grouped together into a cluster; the name of the cluster is the same as the name of the corresponding type; i.e., one cluster is allocated per class. Search is done by simply iterating over the contents of a cluster.

Gordion [23] is a server developed at the Microelectronics and Computer Technology Corporation to provide permanence and sharing of objects within an object- oriented environment. Gordion has the ability to communicate with multiple languages; it supports concurrency control; it has the ability to manipulate objects of arbitrary size. The storage system of Gordion uses a hashing scheme and UNIX files to store objects. Among the major functional components of Gordion are history and inquiry and maintenance.

The Complex Record Manager [19] is a storage manager to manipulate complex objects, and further supports set-oriented data structuring capabilities that can be made use of by a relational database system for supporting non-first-normal- form relations.

The CONTAINER [31, 45] is a storage system, developed at Bilkent University to support the interaction of the Object- Oriented Database System (ODS) [54] with the external storage. In the CONTAINER all components of an object are clustered together according to the philosophy that all components need to be brought into main memory as the root object is accessed.

The following few sections include detailed descriptions of some other object-oriented systems.

3.2 ORION

ORION [6, 7, 8, 25, 35] is an object-oriented database system being designed and implemented in the Advanced Computer Architecture Program at Microelectronics and Computer Technology Corporation, MCC. ORION serves many applications from the CAD/CAM, AI, and OIS domains, with multimedia documents. ORION supports the basic concepts in object-oriented systems, namely, objects, classes, inheritance and methods. Concerning inheritance, the system supports multiple inheritance, leading to class lattices. ORION has been implemented using CommonLisp.

ORION manages secondary storage by placing all instances of a class in the same storage segment. Thus, a class is associated with a single storage segment, and all its instances reside in that storage segment. Storage segment allocation for classes is done automatically. All storage functions are transparent to the user. The storage subsystem provides access to objects on disk. It manages the allocation and deallocation of segments of pages on disk, places objects in the database, searches the database for objects, moves pages of data to and from the disk.

However, ORION takes care of the fact that, some objects' existence is dependent on the existence of other objects in the system. For example, a vehicle is an object which contains a body object, the body object has a set of door objects, and each door has a position object and a color object. A body object is a part of a vehicle instance, a set of doors is a part of a body, and position is a part of a door, and so on. The existence of the position object depends on the existence of the door object, whose existence depends on the existence of the body, whose existence depends on the existence of the vehicle itself. A door and a body are examples of dependent objects, whose existence depends on the existence of other objects. A dependent object can be owned by exactly one object. The body of a vehicle is owned by one specific vehicle and cannot be generated without the existence of that vehicle. The vehicle is a composite object, because it is composed of subobjects which are dependent on it. A composite object consists of a root object connected to multiple dependent objects.

In the secondary storage, composite objects violate the rule that one storage segment is assigned per class. It is so because, composite objects are likely to be accessed together. Therefore, it will be advantageous if multiple classes, more than one class, are stored in the same storage segment. This leads to composite objects being treated as units of storage. ORION considers a composite object as a unit for clustering related objects on disk. The root object as well as dependent objects that constitute a composite object, may usually be considered a single unit for the purpose of retrieval from the database. If the root object is referenced, it is often the case that all, or most dependent objects will be referenced as well. Thus, it is considered advantageous to store all constituents of a composite object as close to one another as possible. A composite object can be stored in a sequence of linked pages. A new page is added if the object increases in size, and pages may be released or compacted if the size of the object decreases. The only problem occurs when two composite objects exchange parts. The two objects should also exchange storage locations. However, ORION does not perform this reclustering. Moreover, ORION is not intelligent enough to identify those classes that share or are stored in the same segment. It is the responsibility of the user to specify which classes are to be stored in the same storage segment.

ORION supports dynamic schema evolution [6]. It is one of the distinguishing characteristics of the system. A detailed study of schema evolution requirements has been carried out by the developing team of ORION. Some of the major function handles in schema evolution are to add a new class, add a new instance variable to a class, delete an existing class, and delete an existing instance variable from a class.

A new class may be defined as a specialization of an existing class or classes, which form the superclasses of the new class. The new class may redefine some of the instance variables and methods. Conflicts are resolved following the rules discussed in Section 2.1.4.

Addition of a new instance variable to a class is treated differently by the class and its subclasses. If there is a conflict with an inherited instance variable, the new instance variable will override the old definition. All instances of the class will be modified to include the new instance variable. Subclasses of the class, to which a new instance variable is added, will inherit the new instance variable, but if there is a conflict the new variable will be ignored.

On deleting an existing class, all its instances are deleted automatically, but its subclasses are not deleted. The deleted class is removed from the superclass list of its subclasses. The superclasses of the deleted class will replace it in the superclass list of its subclasses. Instance variables and methods of the deleted class will cease to exist. So, the subclasses of the deleted class will lose the instance variables and methods they inherit from the deleted class. If the definitions of the instance variables and methods in the deleted class have overridden some other definitions, these definitions will be inherited. If the class to be deleted is the domain of a variable in a class, the superclass of the deleted class will be taken as the domain of the variable unless another domain is specified. When an instance of a class is dropped, all objects that reference it will be referencing a non-existing object. ORION does not automatically identify references to non-existing objects, because of the performance overhead.

On deleting an instance variable from a class, the class may inherit the same instance variable from a superclass if there was a conflict involving the deleted instance variable. All subclasses that inherit the deleted instance variable will be affected by the change. Methods which involve the deleted instance variable will become invalid, they may be deleted or else redefined.

Another schema evolution operation could be the change of the domain of an instance variable of a class. The domain of an instance variable is always a class and the domain of an instance variable can only be changed to a superclass of the old domain. Thus, the instances of the class undergoing the change are not affected.

In addition, ORION supports versions [14].

3.3 EXODUS

EXODUS [11, 12], Extensible Object-oriented Database System, is being designed in the Computer Science Department at the University of Wisconsin, as a modular and modifiable system rather than as a complete database system intended to handle new application areas. EXODUS is intended more as a toolbox that can be easily adapted to satisfy the needs for new application areas. Later, a data model named EXTRA and a query language named EXCESS [12] were developed for the EXODUS extensible database system. EXTRA and EXCESS are intended to serve as a test vehicle for tools developed under the EXODUS extensible database system project.

In some sense, EXODUS is a software engineering project-the goal is to provide a collection of kernel DBMS facilities plus software tools to facilitate the semi- automatic generation of high performance, application specific DBMSs for new applications. EXODUS makes use of a new programming language, E; E is an extension of C that includes support for persistent objects via the Storage Object Manager of EXODUS, which is at the lowest level of the system. E is the implementation language for all components of the EXODUS system.

The Storage Object Manager provides support for concurrent and recoverable operations on arbitrary size storage objects. The basic abstraction at the bottom level of the EXODUS is the storage object; an untyped uninterpreted variable length byte sequence of arbitrary size. Class instances are mapped into storage objects in a one-to-one manner. The storage object is the basic unit of data in the Storage Object Manager.

The Storage Object Manager provides capability for reading, writing, and updating storage objects, or pieces of them, without regard for their size. Buffer management, concurrency control, and recovery mechanisms for operations on shared storage objects are also provided. A versioning mechanism is supported. Whenever persistent objects are referenced, the E translator is responsible for adding the appropriate calls to fix/unfix buffers, read/write the appropriate piece of the underlying storage object, lock/unlock objects, log images and events.

Layered above the Storage Object Manager is a collection of access methods that provides associative access to files of storage objects.

The Storage Object Manager provides a procedural interface, including procedures to generate and destroy files that contain storage objects, to generate and destroy storage objects within the file, and to open and close these files for certain scans. A file of storage objects is known as a file object. The Storage Object Manager provides a call to get the object identifier (ID) of the next object within a file object. It also provides a call to get a pointer to a range of bytes within a given storage object that helps in reading a part of a storage object. For writing storage objects, a call is provided to tell EXODUS that a subrange of the bytes that were read have been modified. For shrinking/growing storage objects, calls to insert bytes into and delete bytes from a specific offset in a storage object are provided, as is a call to append to the end of an object. In addition, the Storage Object Manager is desired to accept a variety of performance related hints about where to place a new object and how large the object is expected to be.

The storage objects can either be small or large, a distinction that is known only within the Storage Object Manager. Small storage objects reside on a single disk page, whereas large storage objects occupy potentially many disk pages. In either case, the object identifier (OID) of a storage object is an address of the form (page#, slot#). The OID of a small storage object points to the object on disk; for a large storage object, the OID points to its large object header. A large object header can reside on a slotted page with other large object headers and small storage objects, and it contain pointers to other pages involved in the representation of the large object. Other pages in large storage objects are private rather than being shared with other objects. When a small storage object grows to the point where it can no longer be accommodated on a single page, the Storage Object Manager will automatically convert it into a large storage object, leaving its object header in place of the original small object. Storage objects are accessed with a dense surrogate index.

Conceptually, a large storage object is an uninterpreted byte sequence; physically it is represented as a B+ tree like index on byte position within the object plus a collection of leaf blocks, with all data bytes residing in the leaves. The large object header contains a number of (count, page#) pairs one for each child of the root. The count value associated with each child pointer gives the maximum byte number stored in the subtree rooted at that child, and the rightmost child pointer's count is therefore, also the size of the object. Internal nodes are similar, being recursively defined as the root of another object contained within its parent node, so an absolute byte offset within a child translates to a relative offset within its parent node. The left child of the root in Figure 3.1 contains bytes 1-421, and the right child contains the rest of the objects, bytes 422-786. The rightmost leaf node in Figure 3.1 contains 173 bytes of data. Byte 100 within this leaf node is byte 192+100=292 within the right child of the root, and it is byte 421+292=713 within the object as a whole.

The storage object manager provides primitive support for versions of storage objects. One version of each storage object is retained as the current version, and all the preceding versions are simply marked as being old versions. The Storage Object Manager provides concurrency control and recovery services for storage objects.

In EXODUS, buffer space is allocated in variable length buffer blocks, which are integral numbers of contiguous pages rather than in single page units. When an EXODUS client requests that a sequence of N bytes be read from an object X, the non-empty portions of the leaf blocks of X containing the desired byte range will be read into one contiguous buffer block, in byte sequence order, placing the first data byte from a leaf page in the position immediately following the last data byte from the previous page. A scan descriptor will be maintained for the current region of X being scanned, including such information as the OID of X, a pointer to its buffer block, the length of the actual portion of the buffer block containing the bytes requested by the client, a pointer to the first such byte, and information about where the contents of the buffer block came from. The client will receive a pointer to the scan descriptor through which the buffer contents may be accessed.

Concerning the file object, related storage objects can be placed in the same storage file for sequential scanning purposes on them. File objects provide support for objects that need to be co-located on disk. Like large storage objects, a file object is identified by an OID which points to

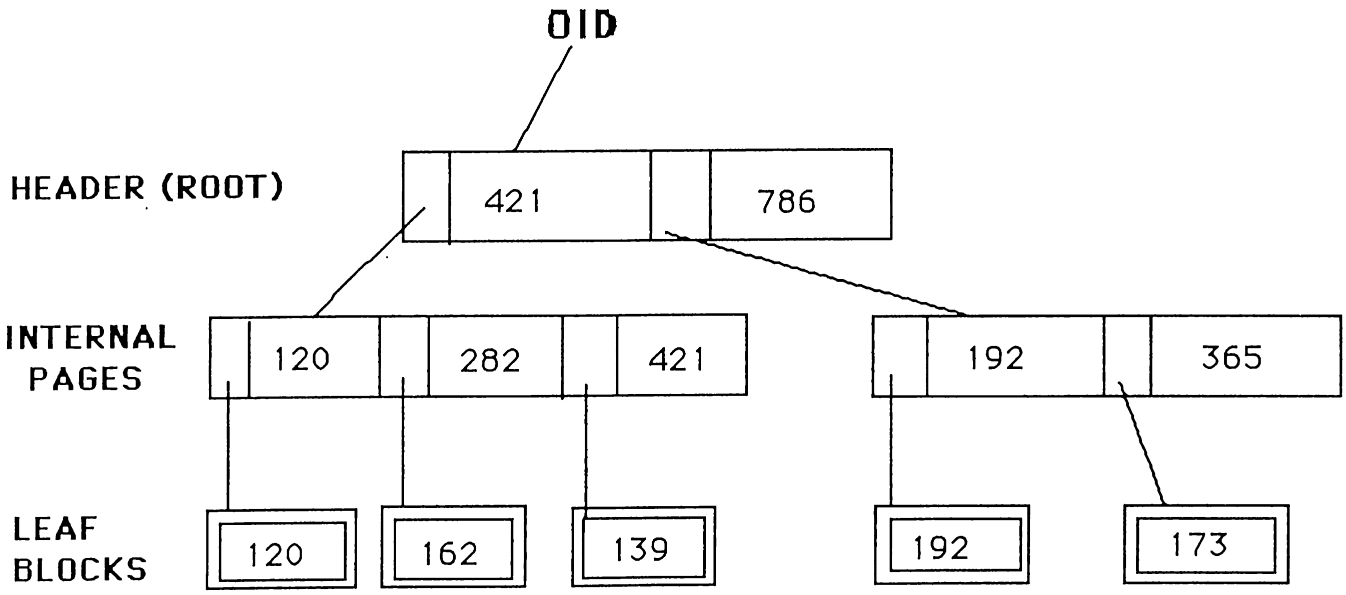


Figure 3.1: An example of a large storage object

its root, an object header; storage objects and file objects are distinguished by a header bit. When a file is created, it is constrained to contain objects of only one class. This is not restrictive as it first sounds, as all objects are transitively considered of every class from which they inherit. Thus a file of objects of class Object may contain objects of every subclass in the lattice.

Finally replication has been introduced into the storage system of EXODUS to speed up query processing. For this purpose three replication strategies are in use [13].

3.4 GemStone

GemStone [37, 38, 39, 40, 46] is an object-oriented database system developed at Servio Logic Corporation. It combines the data type definition and code inheritance of Smalltalk-80 [15, 21, 26, 30], i.e., object-oriented programming features with permanent data storage, concurrency control, transactions and secondary indexing features of database technology. GemStone has overcome the impedance mismatch problem, found in conventional database systems, by providing an object oriented database language, OPAL. OPAL is used for data definition, data manipulation, and general computations. OPAL is a computationally complete language and can express various associative searches on a collection.

The GemStone system has two major pieces, Gem or the object manager, and Stone or the executor, corresponding to virtual machine and object memory of the standard Smalltalk implementation. Stone provides secondary storage management, concurrency control, authorization, transactions and recovery, in addition to its job of managing the workspaces for active sessions. Objects are referenced in Stone using unique surrogates called Object Oriented Pointers. GemStone organizes its memory around an object table, which supports the mapping between an object's OOP and a chunk of memory holding the state of the object. Stone is built upon the underlying VMS file system. The data model provided by Stone is simpler than the full GemStone model, and only provides operators for structural update and accesses. The usage of OOPs to reference objects means that objects can be moved easily in secondary storage. It is not necessary to store an object together with all the objects that it references, they can be stored separately, but the OOPs for the values of an object's instance variables are grouped together.

The object manager performs operations related to the storage and access of objects. It handles operations related to concurrency control and secondary storage management. These operations are transaction control, authorization, data replication, recovery, and directory management. In addition, it provides access to different versions of the data. Each user session has its own object manager with a private object space. Sessions have shared access to the permanent database through transactions. GemStone supports simple inheritance in addition to other basic concepts of object-oriented systems, namely class, method, and object.

All objects in the system reside in a disk based object space which is divided into repositories. A repository represents a dismountable partition of the object space and is implemented as a direct access disk file. Repositories are divided into disjoint regions, called segments, for purposes of authorization and concurrency control. In other words, the objects in a GemStone database are partitioned into logical units called segments. A segment is a chunk of object storage which is owned by a particular user, who can store objects in it, and grant access to other users. The database administrator, or a savvy application programmer should be able to hint to GemStone that certain objects are often used together, and so should be clustered together on the disk. If an application has a group of private objects, all those and no others can be placed together in one segment. Segments expand to accommodate the objects stored in them.

Repositories may be replicated on disk against media failures. Replication is used instead of transaction log files. Because repositories of objects are dismounted, a mechanism must be provided to preserve consistent object identity when information is taken off-line and later brought back on-line. GemStone hides from application designers the paging of objects between secondary and primary memory, and supports objects larger than the size of the server's primary memory. GemStone supports auxiliary storage structures that provide alternative paths to data, and should give users some control over physical grouping of objects, to improve efficiency of specific access paths

Stone supports five storage formats for objects [37]. In addition, Stone has several subcomponents. The transaction manager is shared by all invocations of the Stone and handles concurrent use to the database in an optimistic manner. For each session, it records accesses to the database and validates them for consistency when a transaction commits. Read-only transactions are given priority over read-write transactions, when they require a commit. The approach is based on the assumption that read-only transactions are more frequent than read-write transactions. The directory manager generates and maintains directories which handle object histories. The linker incorporates updates made by a transaction in the permanent database at commit time, calling for restructuring of directories as needed. The Linker is called by the Boxer whose job is to fit objects into tracks after the database changes. The track manager schedules reads and writes of tracks. The commit manager provides save writing for groups of tracks since versions are kept. No garbage collection is needed; garbage collection for temporary data can be done by discarding the workspace at the end of a session.

3.5 ENCORE

ENCORE [27, 47, 48, 58, 59, 60] is a two level storage system for object oriented database systems developed at Brown University. In ENCORE objects are mapped through two levels of abstraction. The first level is responsible for managing the use of persistent object store. It is a typeless backend, results in a stream of bytes for each object. The second level, takes the responsibility of the type system.

For the first level, an OBject SERVER, known as ObServer, reads and writes chunks of memory from secondary storage. The first level brings to the second level a stream of bytes and the second level has to form the objects by using the type system. ObServer is used at Brown University not only as the backend of an object-oriented database system, but also as the storage system for an object-oriented interactive programming environment. ObServer is a general purpose system that can serve many applications like the mail or blackboard systems.

The server is a resource which manages chunks of memory allocated in a shared memory space. A chunk is a contiguous string of bytes. The server must allocate space and a Unique IDentifier (UID), for each chunk that it stores. So, to maintain the correspondence between UIDs and the chunks of memory is one of the principal functions of the server.

The second level, the type level, is normally referred to as ENCORE, Extensible and Natural Common Object REsource. The type level deals with the semantics of objects through type definitions.

The system supports concurrent access. Concurrent access to the shared memory is accomplished by means of UNIX Remote Procedure Calls (RPC) mechanism. By means of RPC, the type level communicates with the ObServer in asynchronous fashion. Having more than one process being able to run concurrently, each process that wants to communicate with the server must bind a model called the client into its image. Therefore, it is possible for the client and the server to reside on different machines. When a process needs to request service from the server, it makes a call on the client code that hides the details of the RPC interface. The ENCORE model uses the server as a backend. It makes calls directly on its main copy of the client model. If there are two different processes on two different machines using the ENCORE database system, separate copies of ENCORE must reside on each machine, using the common server.

The chunks of memory that are managed by the server can be used to implement type objects as presented by the ENCORE interface. In an object-oriented database system, the type lattice introduces the problem of an object being an instance of more than one type. Consider the type Toyota as a subtype of the type Car. An instance X of the type Toyota is also an instance of the type Car. There will be a chunk of storage that represents the part of X that is an instance of Toyota, and a chunk of storage that represents the part of X that is an instance of Car. The term instance is used to refer to each chunk and the term object to refer to the aggregate of all instances that make up X.

The reading and writing of objects is done on block basis. On object creation, UID allocation is separated from storage allocation. This allows an application to request UIDs in anticipation of their use without reserving space for them in the file. Space is not allocated until the objects are actually written. The UIDs of deleted objects are not reused, however, since references to the objects may remain in the database.

ENCORE deals with abstract objects that are instances of types. These types participate in inheritance relationships and allow for the implementation of an object to be distributed across several type definitions. At the type level, every object might consist of several instances, one instance for each type in which it participates. For example, if Toyota is a subtype of Car, Car is a subtype of Vehicle, and Vehicle is a subtype of Object, then a given Toyota will be an instance of all the four types. Since each type has its own private representation as required by the abstract data type scheme, the Toyota object would need four chunks of storage for its representation. Each of these chunks would be accessible through the operations of the corresponding type.

A single UID is associated with each object. When a UID is dereferenced, it leads to a header block for that object. Conceptually, the header is a part of the chunk for the instance of the type object that every object must have. The header for object X contains some general bookkeeping information, as well as a set of pairs of the form (t,p), where t is a pointer to a type object, and p is a pointer to the beginning of the chunk that holds the representation of the instance of t which is a part of X. Most often, those chunks are allocated contiguously such that the pointer p is the offset into that contiguous storage at which the chunk for t begins. In this case, there would be a single UID for the large chunk that contains the instance chunks. This UID is the one that is used by ENCORE to represent the object identity. It is also possible for the chunks to be noncontiguous. Since p can be a UID, the chunks can be stored in any physical location. The decision, whether to allow instances of different types for the same object to be stored in different storage areas, would depend on the access patterns for objects of the given type. Internal UID are only used by the ObServer, they are not available to the application programs.

To achieve efficiency, the segment is used to cluster groups of related objects on the disk. A segment contains objects that the object-oriented database system expects a client to access during a transaction, thus eliminating frequent disk head motion and single object transfers. So, a segment clusters a logically related set of objects into a variable sized single package. The transferred segment is expected to contain other objects that are to be accessed by the client, leading to preloading of required objects. A segment is the unit of transfer for objects between a client and the server, and from secondary storage to main memory. Objects may migrate from one segment to another by being deleted from the first segment and inserted into the second.

Once a client receives a segment, the objects are individually placed in an object hash table and the segment is freed. The client has no further use for the segment structure once it has acquired its objects. The server, then, receives a set of object changes from the client containing a client's operation, and other information necessary for the change to take place in the server's copy of the segment. By returning only the final changes to the server in one package, the amount of network traffic is minimized and the server processing is reduced. Sending every change on isolated fashion may lead to the access of the communication network for every change.

A client may have a unique name for the segment group that it uses. Different clients may have common segments in their segment groups. When a client requests an object, the server returns the segments in which the object resides.

The ObServer maintains master segments containing the current versions of all objects resulting from committed object changes. A client obtains from the server copy segments the client accesses locally. Clients may share the same copy segments by each having a copy at their location.

Whereas segments provide accesses to objects in groups, the UID provides individual object accesses. External and internal UIDs are employed in the ENCORE system. An external UID provides a user with a constant reference to a database object. When the server dereferences a valid external UID, there results an internal UID, manipulated by the system to locate an object physically. Both internal and external UID have the same length of 32-bits quantities that are allocated sequentially from a free-list by the server upon the request from a client creating objects. The internal structure of the internal and external UIDs is different. Each external UID maps either directly or indirectly into one or more internal UIDs. A mapping to multiple internal UIDs results from replicated objects.

Object replication requires an object to appear in more than one segment, everywhere it is referenced. This scheme, of course, incurs a penalty for update, but is extremely useful for objects that are either seldom updated or read only.

The server sequentially allocates external UIDs that are not recycled when objects are deleted. Deleted objects have external UIDs that map to a tombstone internal UID. This makes it possible to detect a reference to an object that no longer exists. The dereferencing process from an external UID to an object is shown in Figure-3.2. The various mappings are maintained in two files called the Object Location Table (OLT) and Duplicate Object Table (DOT). In Figure-3.2, the code field in the UID structure indicates the UID type, either external or internal. This information is used in both the client and the server processes. The OLT maintains the external to internal UID mapping.

Due to replication of objects, an external UID may have more than one corresponding internal UID. So, an external UID maps to an index in the DOT that is maintained by the server and provides the internal UIDs with all copies of a replicated object. When dereferencing an external UID that maps to a replicated object, the system checks whether a client already has a segment containing the object. If so, the corresponding internal UID is returned; otherwise, the object is fetched using an internal UID.

Updating a replicated object is a more costly operation, because every copy of the object in each segment has to be updated. The system guarantees that the update of all copies of a replicated object occurs automatically. Thus, a client cannot obtain a segment that contains a duplicate copy of X until all segments containing X have been updated.

All database objects are contained in at least one segment. A DataBase File (DBF) represents a separate and independent set of objects and type specifications. A segment contains a pointer table and a set of objects. Each segment object is referenced by exactly one entry in the pointer table. Segments are stored in the DBF. The DBF structure is similar to that of the segment, a pointer table and a set of segments. The pointer table comprises one or more pointer table blocks and additional fixed sized blocks are inserted as segments acquire more objects. This feature reduces the frequency of segment expansion each time an object is installed. The structure of the DBF and the segment are show in Figure 3.3.

A DBF contains the number of Segment Pointer Table Entries (SPTEs), the Segment Pointer Table (SPT), and segments. The number of SPTEs represents the next available segment number to allocate. Each SPTE is composed of an offset to specify the segment location within the file, and a size to specify the number of bytes occupied by the segment.

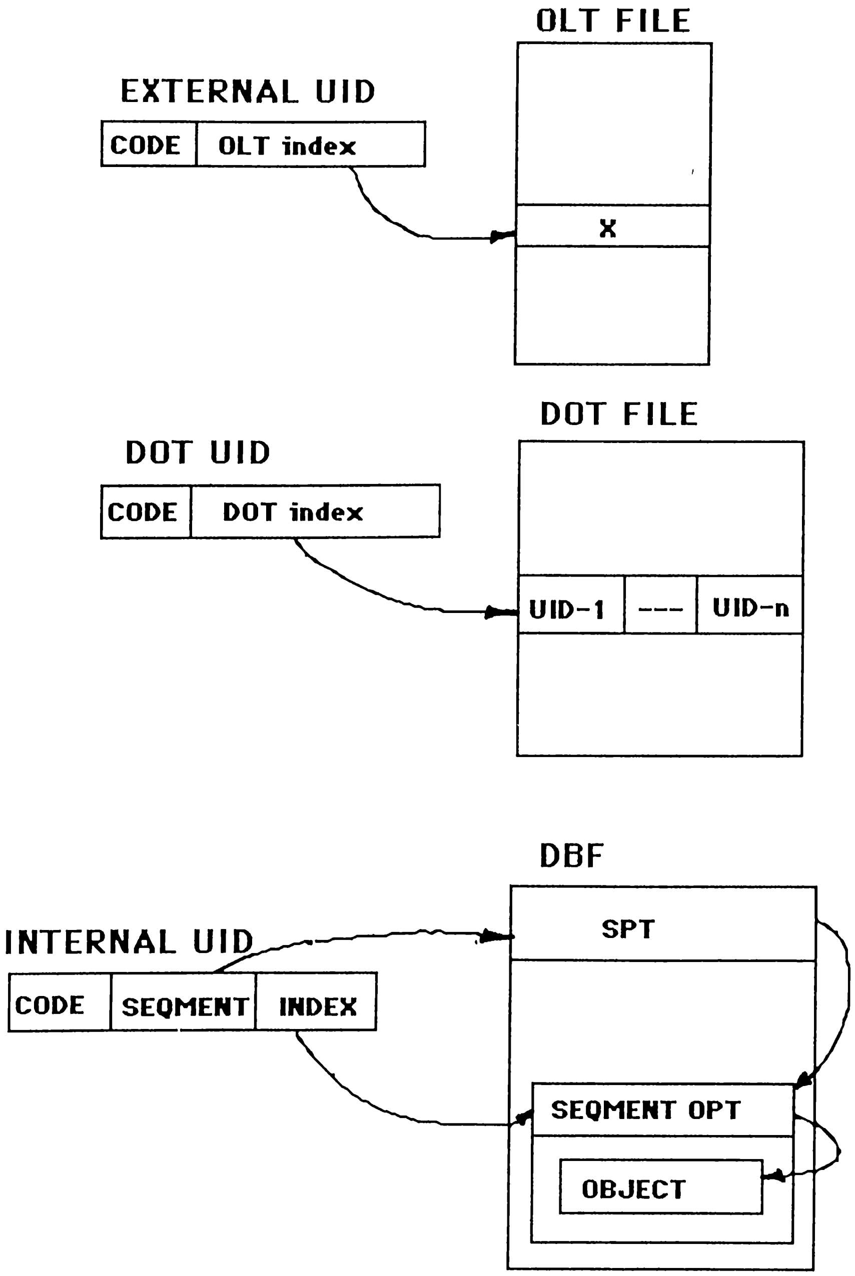


Figure 3.2: The dereferencing process from an external UID to an object

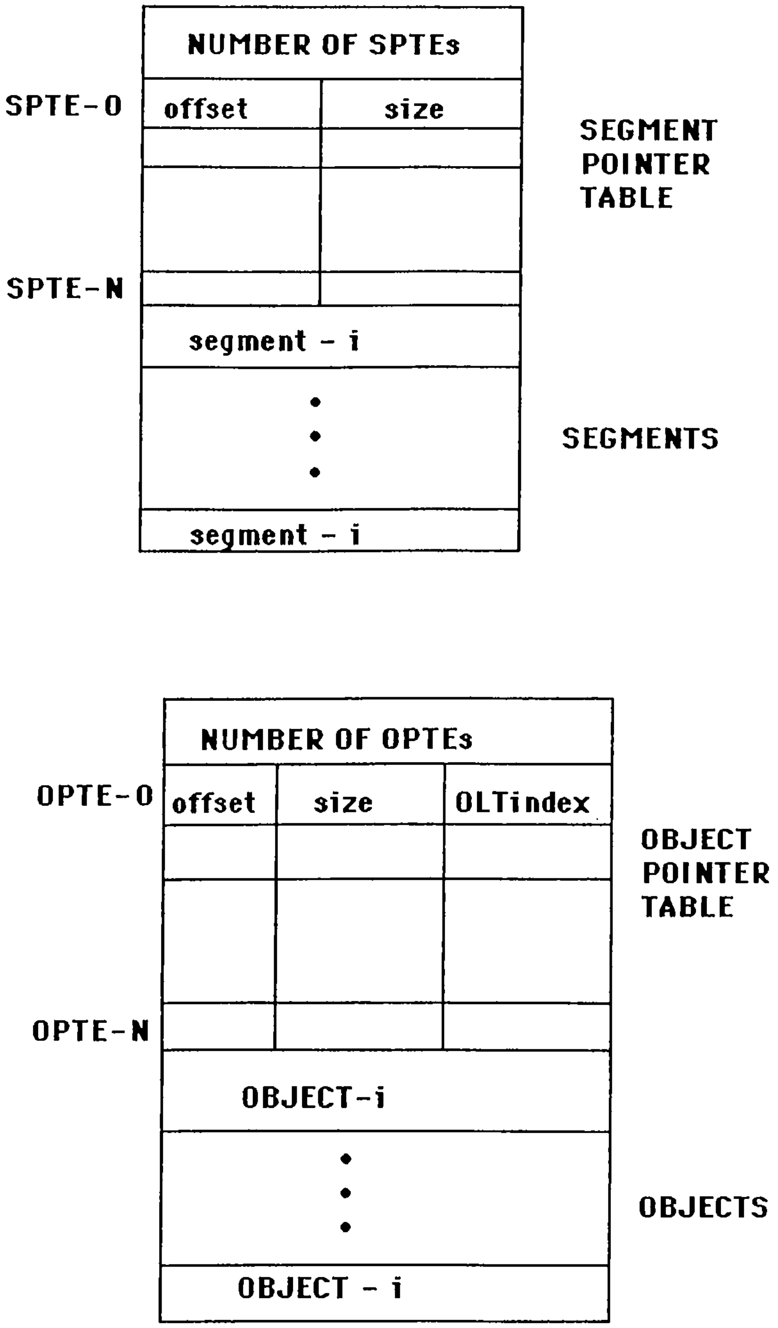


Figure 3.3: The structure of the DBF and the segment

A segment in secondary storage, likewise, contains three sections: the number of Object Pointer Table Entries (OPTEs), an Object Pointer Table (OPT), and objects, corresponding to the SPTEs, SPT, and segments of the DBF. An Object Location Table Index (OLTindex) is introduced in the OLT to provide a back pointer to the OLT that facilitates object migration.

Overflow blocks are added to the segments as the sizes of the objects expand or new objects are added to the segment. Objects in the overflow blocks are accessed as though, the segment and overflow blocks were contiguous in main memory.

Object structure depends on the user defined type specification, but this does not affect the ObServer, since it handles an object as a string of bytes when installing and retrieving objects.

Chapter 4

DESCRIPTION OF THE OBJECT STORAGE MODEL

4.1 Rationalization

In general, regardless of the underlying data model, database management systems take care of secondary storage management. Secondary storage management is considered necessary because persistence is an important aspect of any database management system. Items generated by a database application must have the ability to stay alive after the application terminates. They should be persistent to be referenced when they are needed, or else deleted. External storage is, therefore, important to keep persistent items of the database.

Concerning object-oriented database management systems, persistent storage should have the capability to keep all constituent chunks of every object in the database; but the efficiency of performing different operations on stored chunks should also be considered. Different approaches exist in each system trying to overcome some disadvantages found in other systems, but having some disadvantages that are considered minor with respect to its application domains. It can be said that one storage system is preferred to other systems because it has more advantages or fewer disadvantages than others.

The CONTAINER [31] tries to cluster chunks of an object into the same container. In the CONTAINER system it is supposed that on accessing a chunk all related chunks will be accessed sooner or later. So all the chunks are clustered together to be fetched into main memory by one disk access if possible. But if a chunk is found in more that one container, then it is physically included in one container with its OOP replacing it in the rest. It is clear that this clustering strategy is inefficient in trying to iterate over chunks that are instances in the same class, because to get a single chunk all the contents of the container in which it is found are brought to main memory. This results in more disk accesses and loss of time and space. In addition, not all the chunks that form an instance of a class may be needed so all the contents of the container should not be needed in the main memory at the same time. The storage philosophy that the CONTAINER depends is best applicable for composite objects where component chunks are dependent and likely to be accessed together.

A second approach is that used by ENCORE [27], where chunks of an object are stored as near to each other as possible and mostly in contiguous storage locations. In ENCORE, chunks are replicated wherever it is necessary to improve the performance of retrieval operations at the expense of costly updates.

A third approach is that used by ORION [7]; ORION assigns a separate storage area for each class to include its instances. This approach has the advantage of being efficient in iterating over all instances of the same class. It is inefficient in trying to get some of the chunks of an object or even a chunk that falls deep within the object as a super or a value for an instance variable.

4.1.1 A Key Step Towards the Proposed System

The Object Memory Module [32] provides the primitive functions that are necessary in the development of the whole system. When a new instance of a class is generated, a chunk of memory is allocated. This instance will also be an instance of the superclass in the class hierarchy/lattice. Since every class has its own private representation, a separate chunk is allocated for each class in the superclass chain up to the OBJECT class. The value for a non-atomic valued instance variable within an object can also be an instance in a class.

The key step in the proposed storage system is to assign a separate storage area per class to hold from each instance in the class only instance variables with atomic values, and keep track of the relations between objects in other storage areas. For each instance in a class information related to its immediate supers and values of non-atomic instance variables are kept in these two other areas using only the OOPs as the information representing object relationships. This approach overcomes the inefficiency in fetching a chunk related to another chunk of an object. The areas separated for the non-atomic values are accessed until the OOP of the target chunk is obtained. Then the storage area isolated for the chunk's class is accessed and the target chunk itself is fetched.

4.2 The Proposed Model

The object model can be treated as a three dimensional system as shown in Figure 4.1. The first dimension represents instances in the same class. The second dimension represents nesting of objects, i.e., objects that are values of instance variables. The third dimension represents class inheritance chains.

Some instances in a class may be root chunks, nothing referencing them either as a super or as a nested subobject. Others may be chunks referred to by some chunks from subordinate classes or from nesting chunks in other classes. The former reference is along the inheritance dimension and the later is along the nesting dimension.

A chunk X which is referenced by some other chunk Y is called Y's immediate super chunk if the class in which chunk X is an instance is a superclass of the class in which chunk Y is an instance.

On the other hand, a chunk X which is referenced by some other chunk(s) Y is called Y's immediate nested chunk if the class in which chunk X is an instance is the range for an instance variable defined in the class in which chunk Y is an instance. In some cases the value of an instance variable may be a collection in another class. In other words, more than one chunk which are instances in the same class as chunk X may form the value for an instance variable of chunk Y. This group of chunks in which X falls, forms a collection which is referred to as the value for the instance variable in chunk Y.

4.2.1 Tables and Mappings

Taking into consideration class references along the nesting and inheritance dimensions, a given class C may have say R>0 immediate superclasses and at the same time C may have S>0 classes as the ranges for its S non-atomic valued instance variables. An instance X in class C is inheriting an instance from each of its R superclasses and the values for the S non-atomic valued instance variables of C are found in the S range classes.

Atomic valued properties of each class are collected together in one place called a segment. So a segment may be defined as the storage area which keeps track of all instances of a class. A chunk's representation is shown in Figure 4.2; while a segment's representation is shown in Figure 4.3.

A chunk Y as an instance in C, may have S non-atomic instance variables each of which may have a single chunk or a collection as its value; each of the value chunks whether single or in a collection is said to be an immediate nested chunk of Y. The chunk Y is called the nesting chunk. To keep track of all immediate nested chunks of chunk Y, a variable length record can be built to include:

$$(Chunk\_Y - OOP(Class\_OOP, Chunk\_OOP, Flag)^*)$$

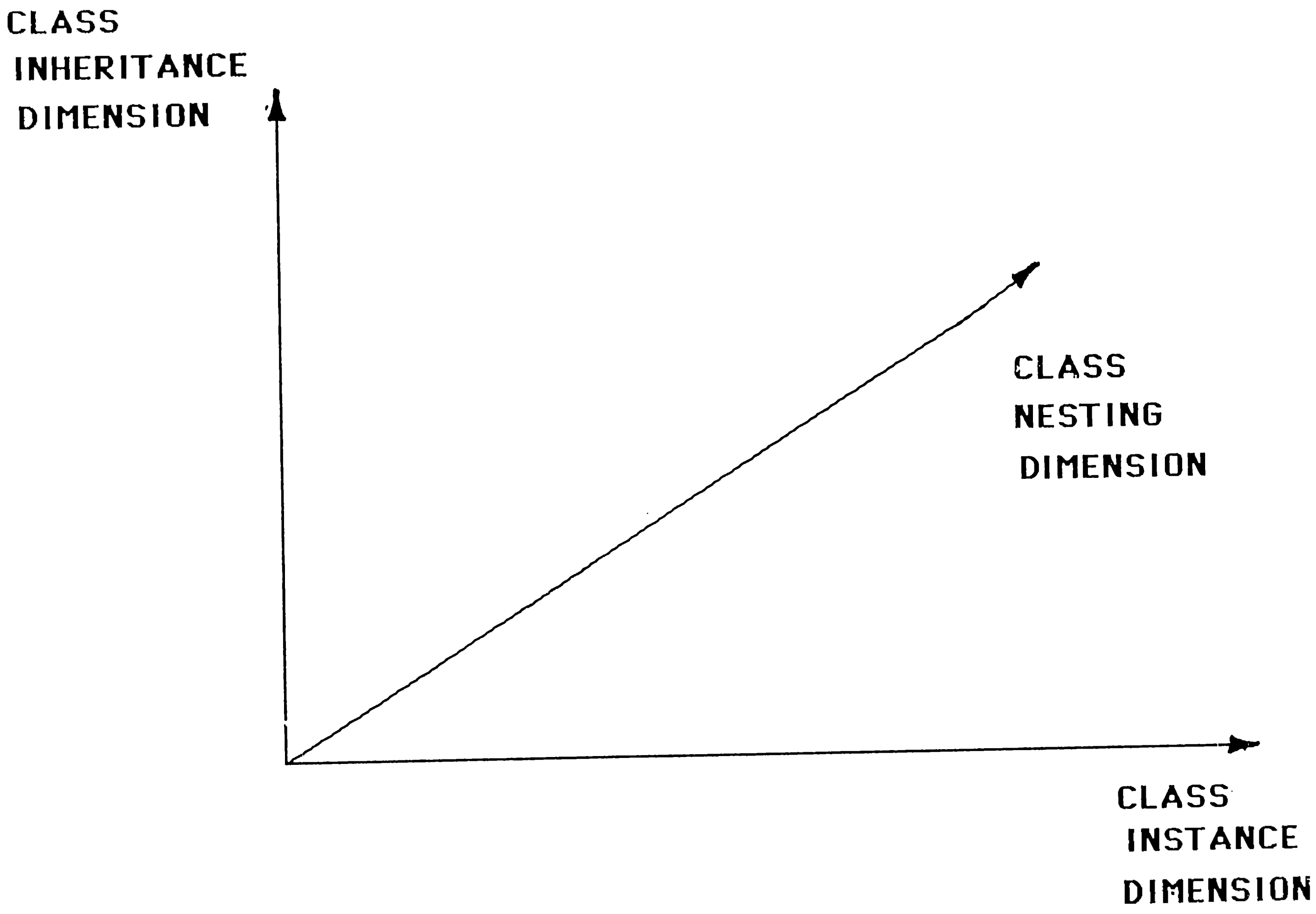


Figure 4.1: A Three Dimensional Object Model

CHUNK_OOP	INSTANCE_VARIABLE$_1$	INSTANCE_VARIABLE$_2$	• • •	INSTANCE_VARIABLE$_N$

Figure 4.2: Representation of a chunk inside the segment

CHUNK$_1$	CHUNK$_2$	• • •	CHUNK$_N$

Figure 4.3: Representation of a segment

where chunk_OOP and class_OOP are representing an immediate nested chunk and its class, respectively. The Flag is set to 1 for all chunks but the last which has a value of 0 for the Flag, that as a collection form the value for an instance variable.

However, the immediate nested chunks of Y have their own respective immediate nested chunks; thus each one should have a variable length record similar to that formed for chunk Y. All such records may be accumulated together to form a table, call it a Nesting Table (NT). Each class may have its own NT.

On the other hand, a chunk Y may be referencing $0<R<1$ immediate super chunks in case of class hierarchy or $R>0$ immediate super chunks in case of class lattice. Here also, variable length records may constitute the following structure:

$$(Chunk\_Y - OOP(Class\_OOP, Chunk\_OOP)^*)$$

to show the immediate super chunks of chunk Y, if any. A separate record is to be built for each chunk to show its immediate super chunks along the class inheritance dimension. All these records may be collected in a table, call it an Inheritance Table (IT). Each class may have its own IT.

Concerning composite objects, the relationships between the constituent chunks of a composite object are not reflected into the IT and NT but the constituent chunks of a composite object are collected together in the same segment because they are likely to be accessed as a single unit.

Here a direct mapping is seen between the three dimensions, namely instances of a class, nesting of objects and class inheritance on one side and the segment, the NT, and IT on the other side as shown in Figure 4.4.

In this way, three relations have been defined. The first relation is between atomic valued parts of chunks in the same class, the second relation is between nesting chunks and their immediate nested chunks, and the third relation is between chunks and their immediate super chunks.

X	R1	Y	where X and Y are instances in the same class;
X	R2	Y	where Y is an immediate nested chunk of the nesting chunk X;
X	R3	Y	where Y is an immediate super chunk of X.

It is important to notice that for a chunk to be stored on the disk, the chunk has to be in one of the segments. So, to access a chunk in its segment information which relates a chunk's OOP with the chunk's address in the segment of its class, the addresses of the records that represent the chunk in the NT and IT have to be kept in a table, call it a Disk Object Table (DOT). The DOT contains an entry for each chunk. As shown in Figure 4.5 an entry in the DOT includes the chunk's OOP, the chunk's class OOP, the chunk's address in the segment of its class, the address of the record that contains the OOPs of the chunk's immediate nested chunks in the NT, and the address of the record that contains the OOPs of the chunk's immediate super chunks in the IT.

By having an entry for every chunk in the DOT and using the chunk's OOP, its class and the three addresses associated with the given chunk can be obtained. It is obvious that all accesses to the segments, IT, and NT to get the information related to a given chunk are to pass through the DOT, using the chunk's OOP. In other words, the DOT is the first table to be accessed on any trial to get information related to a chunk.

4.2.2 An Example

The following example is given to illustrate the described structures. Consider the class hierarchy given in Figure 4.6. The class"course" is the range for the instance variable "course" defined in the class "student" and the class "teacher" is the range for the instance variable "teacher" defined in the class "course".

Suppose that there is one instance in the class "student" as shown in Figure 4.7. On constructing the DOT one entry is included for each class to show the address of its segment, and one entry per chunk as shown in Figure 4.8. In Figure 4.8, PA, SA, CA and TA stand for the addresses of the segments for the person, student, course and teacher classes, respectively. SN and CN stand for the starting address of the NT for the student and course classes respectively. A zero is written for the NT addresses of the person and teacher classes because they have no NT entries. SI and TI are the addresses of the IT for the student and teacher classes respectively. A zero address is

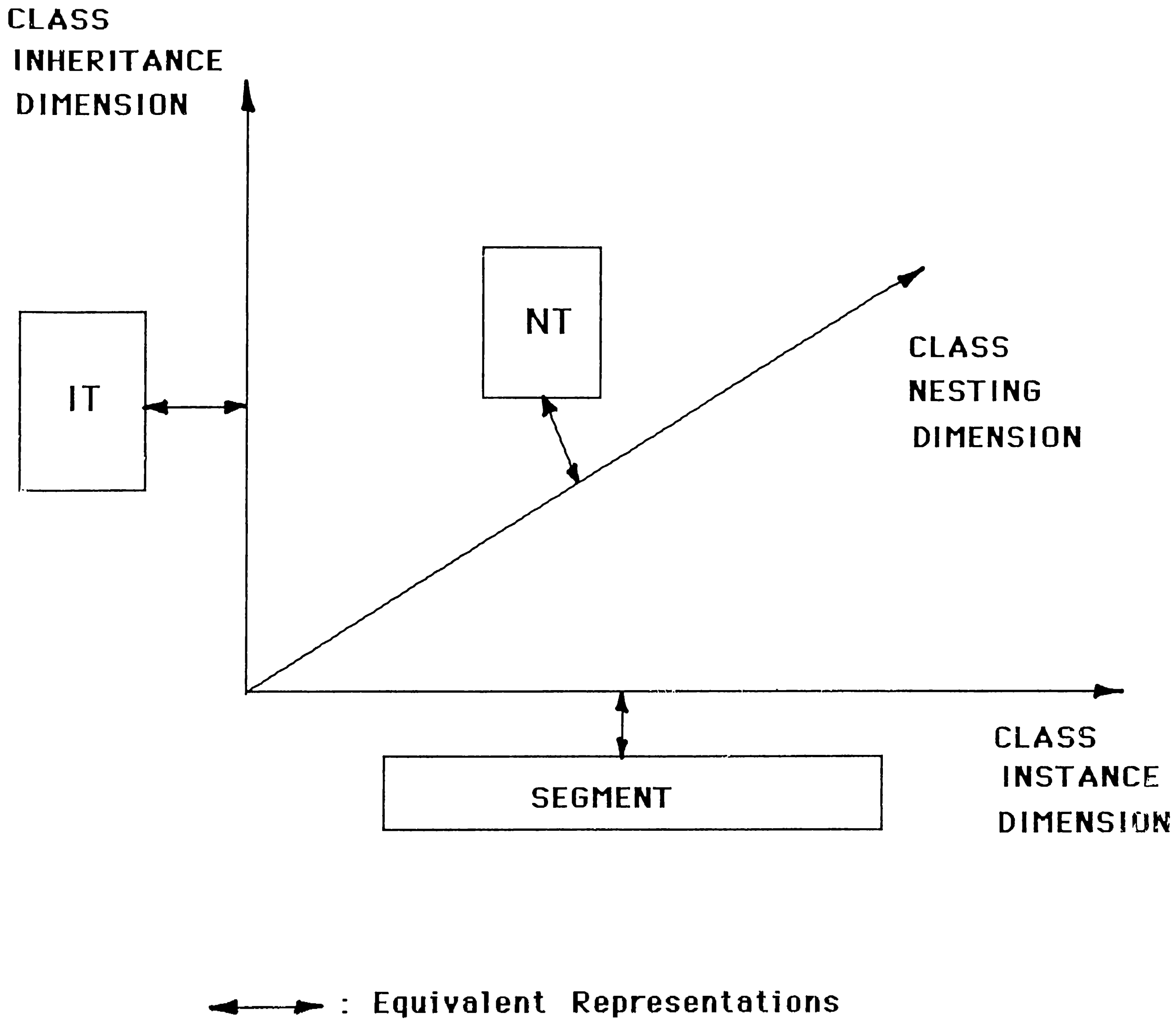


Figure 4.4: Mapping dimensions of the object model into the IT, the NT and the segment

CHUNK_OOP	CLASS_OOP	ADDRESS IN SEGMENT	ADDRESS IN NT	ADDRESS IN IT

Figure 4.5: Format of the Disk Object Table (DOT)

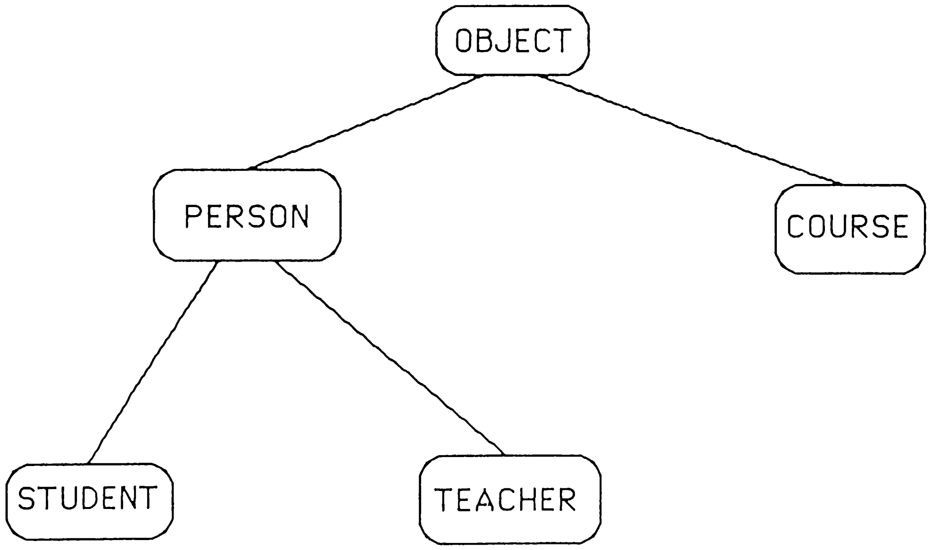


Figure 4.6: A Class sub-hierarchy

written for the IT addresses of the the person and course classes because they have no IT entries. PA1 is the address of OOP6 relative to the starting address of the segment of the person class. A zero is entered for the segment address of the other chunks because each is the first chunk in the segment of its class. The NT and IT addresses for all the chunks are 0, either to indicate that no entries corresponding to the chunk is found or to show that the entry for the chunk is that found at the starting address of the table. Notice that there is no entry for OOP3 in the DOT because the information related to OOP3 is included in the NT.

Remember that every class may have its own IT and NT. In this example all the IT entries for all the chunks are shown in one table and also the NT entries of all the chunks are presented in one table. As shown in Figure 4.9 the IT includes two variable length records to show the immediate super chunks in the "person" class of each of the "teacher" and the "student" chunks. As shown in Figure 4.10 the NT contains two variable length records to show the immediate nested chunks of each of the "student" and the "course" chunks in the "course" and "teacher" classes respectively. As shown in Figure 4.11 there are four segments one per class instances. A segment contains only the atomic valued instance variables.

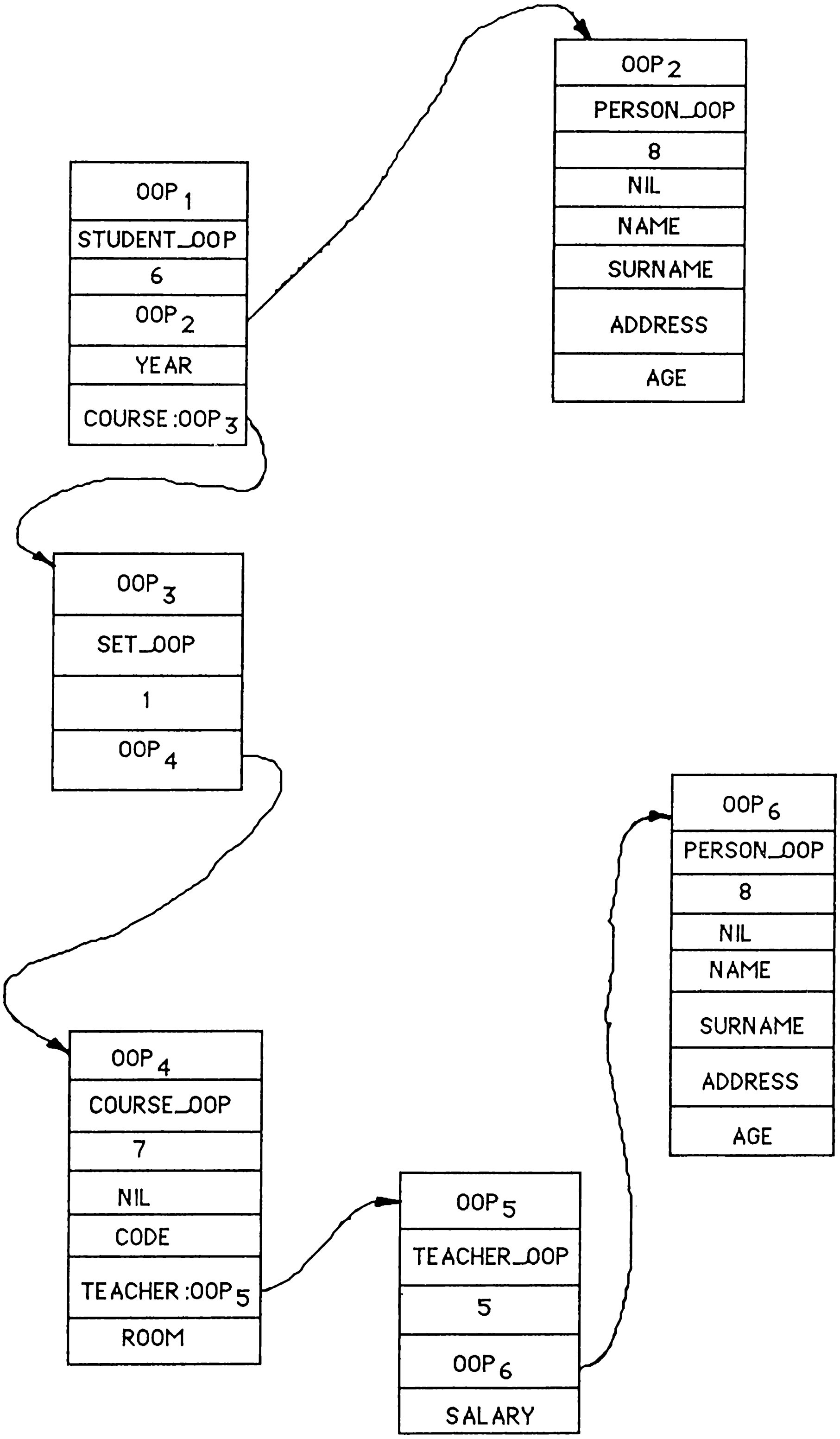


Figure 4.7: An instance in the "student" class

CHUNK_OOP	CLASS_OOP	ADDRESS IN SEGMENT	ADDRESS IN NT	ADDRESS IN IT
OOP1	STUDENTOOP	0	0	0
OOP2	PERSON.OOP	0	0	0
OOP4	COURSE.OOP	0	0	0
OOP5	TEACHEROOP	0	0	0
OOP6	PERSON OOP	PA1	0	0
PERSON.OOP	PERSON.OOP	PA	0	0
STUDENT OOP	STUDENTOOP	SA	SN	SI
COURSE_OOP	COURSE_OOP	CA	CN	0
TEACHER.OOP	TEACHEROOP	TA	0	TI

Figure 4.8: The constructed DOT

(OOP_1,(PERSON_OOP, OOP_2))
(OOP_5, (PERSON_OOP, OOP_6))

Figure 4.9: The constructed IT

(OOP_1, (SET_OOP, OOP_3, 1), (COURSE_OOP, OOP4, 0))

(OOP_4, (TEACHER_OOP, OOP_5, 0))

Figure 4.10: The constructed NT

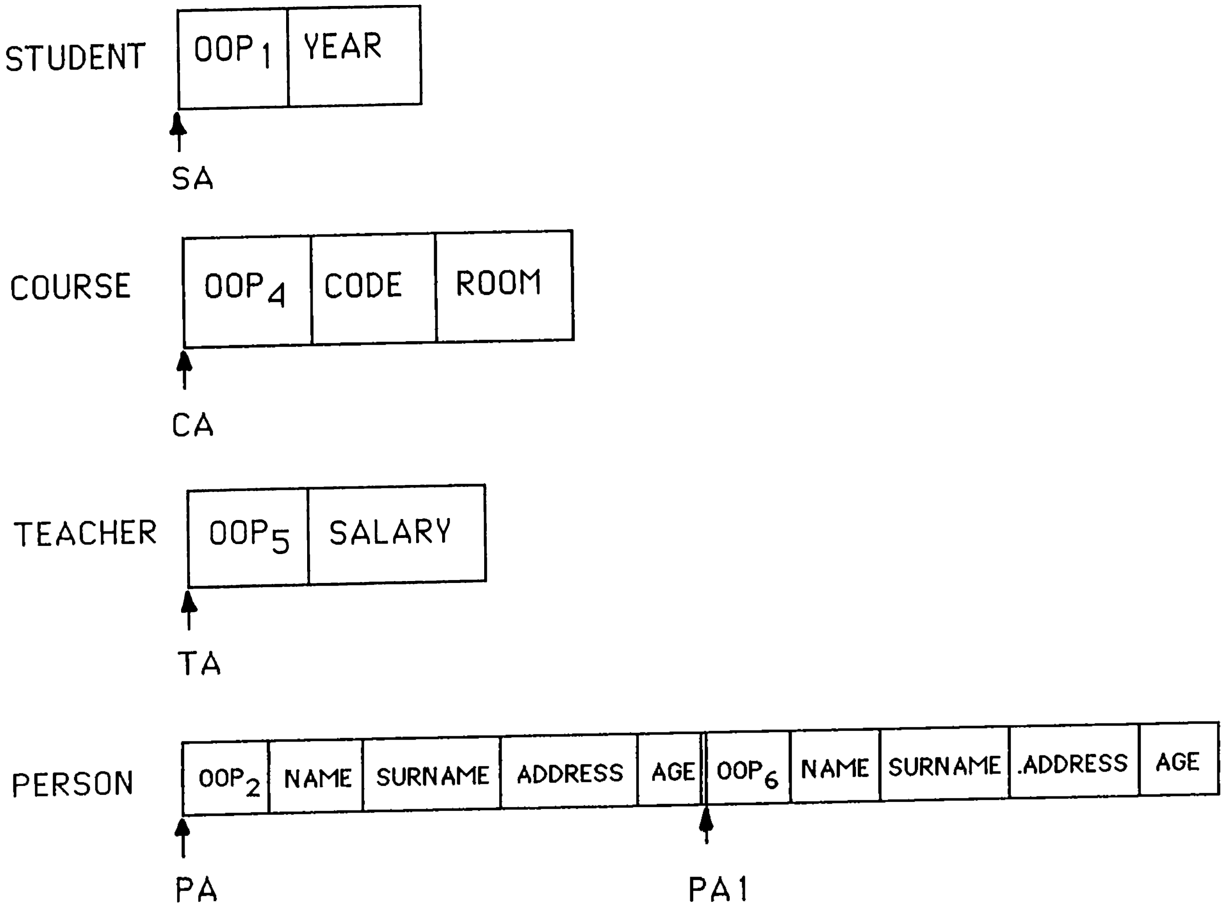


Figure 4.11: The constructed segments

```
Procedure Build_IT_NT;
  begin
    For every class (C) in the class hierarchy/lattice do
      begin
        If (supers(C)<>OBJECT) then { in the OBJECT class, there are no instances }
          begin
            For every instance (i) in C do
              begin
                vlr= OOPi; { vlr is the variable length record that shows the immediate supers of chunk
                                        i in C }
                For j=1 to R do { R is the number of supers of C }
                  begin
                    vlr=vlr||(class_OOPj, super(i,j)); { super(i,j) returns the OOP of the super chunk
                                                                within class j of i}
                  end;
                  If (R>0) then Append_IT(vlr);
              end;
          end;
If (there are S>0 non-atomic instance variables in C) then
{ S is the number of range classes for instance variables of C }
    begin
      For every instance (i) in C do
        begin
          vlr1=OOPi; { vlr1 is the variable length record that shows the immediate nested chunks
                            of i }
          For j=1 to S do
            begin
              If (value(j) is a collection) then
                begin
                  m=size(collection);
                  For k=1 to m-1 do
                    begin
                      vlr1=vlr1||(class_OOPj, OOPk, 1); { 1 is the value of the Flag }
                    end;
                end
              else
                    m=value(j);
            vlr1=vlr1||(class_OOPj,OOPm,0);
            end;
            Append_NT(vlr1);
        end;
      end;
    end;
  end;
```

Algorithm 4.1 Construct the IT and NT for objects in the database

4.2.3 Construction of the IT and NT

According to Algorithm 4.1, take chunks from class C which are related by R1; as a consequence, atomic valued parts of these chunks can be stored in one segment. By applying R2 to those chunks, the variable length records to represent the nested chunks of chunks in class C in the NT are obtained. By applying R3 to the chunks in class C, the variable length records to represent them in the IT are obtained. Therefore, by the application of relations R2 and R3 to instances in class C, the variable length records to represent them in the NT and IT are obtained. Apply relations R2 and R3 for chunks in each class in the hierarchy/lattice to get more variable length records to represents instances of the classes in the NT and IT. The application of relation R2 stops when all the chunks of the remaining classes have all instance variables with atomic values. Relation R3 stops to be applicable when the chunks of the remaining classes have no immediate super chunks. This condition holds when a class is the immediate subclass of the OBJECT class.

The repeated application of relations R2 and R3, as stated above, leads to the construction of the NT and IT of the class hierarchy/lattice.

4.2.4 Why Separate Atomic from Nonatomic Values?

A segment contains chunks from the same class. Remember that the OOPs of the immediate super chunk(s) of any chunk were included in the chunk's entry in the IT. Also, the OOPs of the immediate nested chunks of the same chunk were included in the chunk's entry in the NT. Therefore, there is no need to duplicate these OOPs inside the chunks within the segment. Another reason for not including these OOPs inside the chunks within a segment is that the usage of these OOPs is only to access new chunks and this can be done from the IT and the NT. Hence, as shown before the chunks inside a segment include the values of the atomic valued instance variables only.

The IT and the NT can be used to find the OOP of the target chunk starting at the root chunk of the object. By following from the IT and NT the records which show the immediate super chunks and the immediate nested chunks, any chunk between the root chunk and the target chunk can be obtained. After getting the OOP of the target chunk the values of its atomic valued instance variables can be obtained from the related segment.

4.3 Characteristics of the Proposed System

The following are some of the characteristics of the proposed storage model.

4.3.1 Efficiency of Access and Memory Utilization

To get to a chunk of an object, which is neither an immediate super chunk, nor an immediate nested chunk of the object's root chunk, all chunks between the root chunk and this target chunk must be accessed either along the inheritance/ or nesting dimension. On accessing a chunk that is found between the root chunk and the target chunk, from each chunk either an immediate super chunk or an immediate nested chunk is accessed. Therefore, the instance variables needed in each chunk, except those of the target chunk, are only non-atomic valued instance variables. Hence, the instance variables which have atomic values, need not be accessed except in the target chunk. Due to this, only the IT and the NT being indexed can be efficiently used to find the OOP of the target chunk starting at the root chunk of the object.

4.3.2 Schema Evolution

Handling schema changes properly and efficiently are among the important aspects of any proposed storage system. Clearly, it is highly desirable for a storage system to satisfy as many schema operations as possible. Schema changes are presented in Chapter 6 after the description of the operations.

4.3.3 Composite Objects

Being composed of dependent chunks a composite object is thought to be a unit of storage and retrieval. In the proposed system, component chunks of a composite object are treated altogether as a single chunk in storage and retrieval operations depending on the well known philosophy that components of a composite object are likely to be accessed together. However, the user should indicate which objects are composite or which classes have composite instances.

Remember that, the IT and NT are introduced to facilitate access to target chunks within objects without any need to access the intermediate chunks in their entirety. With composite objects the situation is different, all component chunks are to be accessed together. Therefore, the fact that component chunks of a composite object are related by R2 or R3 need not to be reflected onto the NT and IT respectively. As a whole, components of a composite object are all aggregated together to form a single chunk that can be stored into a segment. The OOP of any of the component chunks of a composite object maps onto the single chunk as a whole, to facilitate the retrieval of all component chunks of a composite object depending on a one component chunk retrieval.

4.3.4 Clustering

Clustering of chunks is governed by data semantics and frequency of expected operations. Different chunks are related by R1, R2 or R3. Being related by R1, chunks from a class are candidates to be clustered together into one segment. For composite objects, all constituent chunks of a composite object are clustered together.

Other clustering possibilities exist. For instance, the supers of an instance in IT can be clustered, or nested chunks of an instance in NT can be clustered.

4.3.5 Locking

Information related to an instance of a class is found in three locations, IT, NT, and the segment. Accesses to one of the nested, inherited, or atomic parts of an object do not affect the remaining two parts; so, the accessed part can be locked leaving other parts still accessible. This results in the achievement of finest degree of granularity for more concurrency.

4.3.6 Comparisons and Evaluations

The proposed storage system uses a new approach in storage management for object-oriented systems. While existing storage systems store together the whole chunk, i.e., instance variables with atomic values together with non-atomic values; the proposed storage system separates external references from atomic valued instance variables due to the fact that external references are not needed until related chunks are required.

It became impossible with existing systems to access a nested chunk without passing all the way through chunks leading to it without violating the encapsulation principle. For example, to get the name of the manager of the department for an employee, it is necessary to fetch the two chunks which lead to the manager's chunk; in other words, employee's chunk and department's chunk are to be accessed before getting to the manager's chunk. To access a nested or inherited chunk, therefore, all chunks leading to that chunk are to be fetched into main memory with the attendant memory overhead in addition to a loss in time.

Comparatively large main memory allocations are needed in systems like the CONTAINER where chunks of an objects are clustered into contiguous storage locations and fetched together. On the other hand, in systems like GemStone and EXODUS where chunks are spread over the disk space, or like ORION and ODE where instances of a class are clustered into one storage area, it is possible to have one of the chunks leading to the target chunk in main memory at a time; because each fetched chunk, except the target, can be overlaid by the next chunk leading to the target chunk.

While this is the situation with existing systems, in the proposed system a target chunk is accessed by having only the small amount of required information in main memory at any time; the required information comes from the IT and the NT since these tables include only chunk references but not the atomic values in the chunks. For instance, to get the manager of an employee, no chunk except the manager's chunk is fetched into main memory as chunk references are followed in the NT.

Concerning schema evolution, most schema changes include adjustments to chunk references that can be done with more flexibility in the IT and NT.

Another important distinguishing feature of the proposed system is the finest granularity achieved due to the separation of chunk contents between a segment, the IT, and the NT.

Finally, with systems that cluster all chunks of an object together, it is cumbersome to iterate over instances of a class; but with the proposed system, a segment clusters atomic valued parts of the instances of a class facilitating such operations.

Chapter 5

INDEXING

5.1 Indexing Basics

In general, an index is a collection of <key_value, address> pairs used to facilitate access to a collection of records. For instance, an index is found at the end of most books including an alphabetically ordered list of keywords each followed by page numbers of those pages in which the keyword can be found. This is done to facilitate the usage of the book by employing keywords as the search arguments.

In database systems, an index consists of a collection of entries one for each data record or group of data records, and a pointer or disk address which allows immediate access to those records; could also be a symbolic value, i.e., primary key of the referenced record (OOP of the object chunk). An entry per record is called a dense index [55], while an entry per more than one record is called a sparse index [55].

Hashing and B-trees [28] are well known indexing methods. Although hashing is fast, it does not facilitate an easy access to records in the order of sorted key values. B-trees are quite fast and permit access to records in the order of sorted key values; a common need in database systems, at the expense of extra storage.

The index provides easy access to data stored in files at the price of using more storage space. But decreasing the required access time is more important than decreasing the amount of storage used; as the price of storage continuously decreases in parallel with developments in technology.

5.2 Problem Definition

The additional features of object-oriented systems [49], such as information hiding and encapsulation, inheritance, complex and composite objects, object identity, and schema changes make it more nontrivial to handle the indexing problem with such systems. These features are what conventional database systems lack. The basic problem in indexing in a database system is to efficiently select from a collection those records meeting a selection criteria. We may want to find all objects that either contain a given object, or an object equal to a given object as the value of a particular instance variable. But, the nonnormalized nature of objects introduce some difficulties and also accessing an object by its value is somewhat contradictory to the notion of object-oriented identity and the data encapsulation principle.

5.2.1 Indexed Objects

Indexing approaches used in conventional database systems are vague no more applicable in object-oriented systems, because the new features are to be taken into consideration on deriving an indexing model for an object- oriented database system. On indexing objects on their internal structure, one limitation is how deeply to index.

We can index on either immediate atomic valued instance variables known as single level indexing, or on instance variables found in inherited or nested objects known as multilevel indexing.

Indexing on the atomic valued instance variables of an object is straightforward. Any change to the value of an atomic valued instance variable can be reflected into the index without any problems.

With multilevel indexing, problems arise because an object's position in an index can be invalidated by a change in a subobject that is not manifested in the object itself, i.e., any chunk within a complex object can be accessed directly inside its class as a root chunk, and changes due to such accesses can not be reflected to other chunks that reference the changed chunk either along the nesting dimension or along the inheritance dimension. In other words, the difficulty arises on trying to index on instance variables of those objects found along the nesting and inheritance dimensions, because the relationships along these two dimensions are unidirectional and many to one in the forward direction, i.e., directed away from the root chunk of an object, not towards it. It is necessary to keep the relationships in the backward direction, i.e., from the referenced chunks towards the root. The relationship is "many to one" in the forward direction and "one to many" in the backward direction. Keeping both the forward and backward references along the nesting and inheritance dimensions will facilitate indexing at any depth within an object without any problems.

5.2.2 Identity Indexes and Equality Indexes

Identity is a distinguishing property of objects found in a database [33]; each object has its own identity. Regardless of the change in the contents of an object, its identity remains unchanged; i.e., object identity is preserved. One powerful technique for supporting identity is through surrogates. An OOP is the identity of an instance object in a class. The private memory of an instance object is a contiguous series of words which is called a chunk [32].

As discussed in Chapter 4, an object may reference other objects along the nesting and inheritance dimensions as its nested and super chunks, respectively. Therefore, indexes may be built on the identities of objects found at a certain level along the nesting or inheritance dimension within the indexed objects. Such an index is called an identity index, because it depends on the identity of the index argument; regardless of the contents. An important thing to point out here is that, identity index is applicable for non-atomic valued instance variables.

An identity index facilitates accesses to objects in a class depending either on their identities or on the identity of objects found in another class and referenced by objects in the former class. The reference may be either in the forward or in the backward direction along the nesting or inheritance dimension. Depending on the given identities, access to the objects in the former class can be facilitated.

Additionally, it is also possible to build an index based on the value of an atomic valued instance variable, called an equality index. So, an equality index may be built based on the internal state of objects; such an index may support range searches on values of the key instance variable.

An equality index facilitates accesses to objects in a class depending on the value of an atomic valued instance variable. Given a possible value for an atomic valued instance variable, an equality index establishes the identities of objects having that value in their instance variables. Depending on the result of the comparison of a given search argument with the value of an atomic valued instance variable in the class, objects from the class are located.

While an identity index deals with object identities, an equality index deals with the values of some instance variable included within objects.

5.3 Requirements of Indexing in Object-Oriented Systems

Indexing is an important issue in the storage management of database systems. The presence of an index in a database system facilitates associative access to objects found within the database, instead of searching the whole database to find the same objects in the absence of indexes. Indexes are especially useful when the user wishes to select a small subset of a large collection based on the value of a specific instance attribute; the search argument. Indexes are, therefore, helpful for operations iterated over members of a large collection; but it is necessary to get the same answer on evaluating a query with or without indexing.

The concept of object-orientation has introduced some new features such as information hiding and encapsulation, inheritance, complex and composite objects, and object identity that make it not possible for conventional indexing methods to be still applicable with object-oriented database systems. Current relational database systems use indexes for fast access to records in a single relation or to achieve efficiency in natural join operations. However, it is misleading to equate relations with classes. A relation serves both to provide the scheme for its component tuples and to collect all those tuples. In an object-oriented database system a class defines the structure of its instances, but rarely keeps track of all those instances. Instead, collection objects -bags and sets- serve to group those instances. One of the differences between object- oriented database systems and relational database systems is that objects are not flat. One should be able to index on the instance variables that are found several levels deep in an object, along the nesting or inheritance dimension; i.e., given a value for an instance variable (atomic value or object identity), one should be able to find efficiently the objects that reference this value and are several levels away in the inheritance or nesting dimension.

5.3.1 Improving Performance

Searching a large collection by a sequential scan will give unacceptable performance with disk-based objects. Searching for elements should be at least logarithmic in the size of the collection, rather than linear. Thus, object- oriented database systems should support associative access on elements of large collections to avoid sequential iteration over all the elements on every access. They should supply storage representations and associative access methods in support of locating an element by its internal state. They should support equality and identity indexes.

5.3.2 Conserving Encapsulation

Information hiding and complex objects [35] are two of the important distinguishing features of object-oriented systems. Complex objects are those objects that are related to some other objects along the nesting and inheritance dimensions. An important contribution of object-orientation in database systems is that what is found inside an object is invisible to the outside environment of the object, even the relations along the nesting and inheritance dimensions are so. The only way by which values can be extracted from within an object is through message sending [44]. It is not necessary for the sender to know how the message is interpreted by the receiving object; the interpretation is done internally inside the object, transparent to the surrounding environment, by executing the method that corresponds to the received message. The object, therefore, receives a message and gives an answer, if required, without letting the sender to know how the answer is obtained or even the internal structure of the object.

5.3.3 What Should be Indexed?

Accesses to objects depending on the value of some instance variable require a search of all objects in the same class in the absence of an index. In addition, accesses to referencing chunks located in the backward direction along the nesting or inheritance dimension with respect to the referenced chunk some of whose characteristics are given, require index employment in order not to spend a considerable amount of time in sequentially performing a forward access to extract the referencing chunk(s). Finally, the presence of equality indexes on atomic valued instance variables is necessary for improved performance.

5.4 Existing Approaches to Indexing

5.4.1 Indexing in GemStone

Indexing in GemStone [37, 40] is based on the structure-instance variables- of objects and it is supported at the Stone level. In order to facilitate associative access, both paths and instance variable typing have been introduced into OPAL; the database language supported by GemStone. A path is a variable name, a path prefix, followed by a sequence of one or more instance variable names called links to form a path suffix. Both equality and identity indexes are supported in GemStone. To build an index on a collection using a particular path, the path expression must be defined for every object in the collection. Finally, OPAL differentiates between selection expressions and associative queries by the use of brackets and braces. Braces indicate an associative query, whereas brackets indicate a selection expression.

5.4.2 Indexing in the CONTAINER

An indexing model was added to the CONTAINER [31, 45] in order to provide alternative access paths to objects, based on the values of their instance variables, i.e. to provide associative access to objects.

Indexing can be provided on the immediate instance variables of an object or on the inherited instance variables or on the instance variables that belong to the objects referenced by the indexed object. Indexing is performed on classes, which means that all instances of that class are indexed; thus the methods updating the value of an instance variable in an indexed class can provide easier index handling services. An index is created by specifying a pair of the form:

$$< class\_index\_path, instance\_variable\_index\_path >$$

Where the first component, class_index_path, specifies the class on which the index is to be built, and the second component, instance_variable_index_path, specifies the actual instance variable providing the key for the index set. Both components use the path expression described by GemStone. The formal definitions of these path expressions, as they are used in the CONTAINER, are presented below, yet, informally the class_index_path contains in its first component the target class, whose objects will be returned by indexed access, followed by zero or more classes separated by dots, and the last component being the class that contains the instance variable being indexed by the instance_variable_index_path. The instance_variable_index_path has in its last component the instance variable being indexed and the whole path shows the way to access that instance variable from the object which is an instance of the class being indexed, and whose OOP will be associated with the value of this instance variable.

A class_index_path is a string of the form:
A1.A2...An where Ai belongs to {user defined classes} and Ai is a subclass of Ai+1 for i=1...n-1 and there does not exist any i such that Ai=Class class, which has all the classes as its instances [32], and the indexed instance variable is among the instance variables of An.

An instance_variable_index_path is a string of the form:
V1.V2...Vn where Vi belongs to {instance variables of class Vi-1} for i=2...n.
If n=1 then V is called a simple index path.

Indexing a path A1.A2...An on the instance variable V will associate the OOPs of the objects found in class A1 with the value of V in the corresponding object, i.e. given a value for V, all OOPs of objects in class A1 associated with that value of V are returned.

If V is a simple index path, i.e. it is an immediate instance variable of class An, then this is a one_level_index. Multi_level_indexing is performed by indexing each link along the variable path rather than maintaining a single index for the whole path.

Each index specification is independently specified by a B_tree. A one_level index is straightforward, but multilevel indexes have been designed by a sequence of index components, one for each link along the path, as described for Gemstone in the previous section.

5.4.3 Indexing in EXODUS

In EXODUS [11, 13], the indexing problem is treated by using the path concept introduced by GemStone, but by adding data replication concept. The designing group argue that there is basically no reason why an index can not be built on replicated data, and by allowing indexes to be built on replicated data, new indexing opportunities are created.

It is necessary to emphasize that in EXODUS replication was added to the path dependent indexing method to gain efficiency. To illustrate how replication is used by EXODUS indexes, consider the following example [13]:

Replicate Emp.dept.org.name
Build index on Emp.dept.org.name

Here Emp, dept, and org stand respectively for employee, department, and organization; name is an instance variable defined within organization.

Because of replication, an index for the path Emp.dept.org.name can be built on the replicated values that are stored in Emp. The index would map organization names directly to objects in Emp, and could support queries that require an associative lookup on the path Emp.dept.org.name. A direct object-to-object mapping is provided by EXODUS indexing model that uses replication; this is the result of adding replication to the path concept introduced by GemStone where the mapping is indirect via B-tree components.

5.4.4 Problems with the Described Approaches

The described approaches treat the indexing problem by violating some object-oriented features, namely accessing information inside objects supposed to be hidden and the interaction with objects by means other than message passing. The three mentioned systems use the path concept initially introduced by GemStone.

A path describes the relation between objects by making their internal structure visible to the outside, a violation to the information hiding principle of object-oriented systems. While the introduction of the path concept has many advantages in forming the basis for some indexing models for object-oriented systems, it is not suitable for object- oriented systems if their distinguishing features are to be preserved.

Finally, the usage of two different query formats for associative and selection accesses by GemStone introduces an impedance mismatch problem in OPAL.

5.5 A Proposed Indexing Method

Although the path dependent indexing approach introduced by GemStone may also be used in the proposed storage system [4], it is important to treat the indexing problem within this system from a different point of view by taking advantage of its particular structure. It is also important to preserve the distinguishing features of object-oriented systems.

5.5.1 Identity Index

An important requirement in indexing is to be able to access certain chunks depending on the identity of a given chunk which is related to the former in either of the directions along the inheritance or nesting dimension. A second one is to be able to access certain chunks depending on the comparison of a search argument to the value of some instance variable within a collection of chunks; as discussed in section 5.5.2.

Recall from Chapter 4 that in the storage model, we defined three relations, R1, R2, and R3. Here, the discussion will start with the inverse of relations R1, R2, and R3, i.e., $R1^{-1}$, $R2^{-1}$, and $R3^{-1}$. Consider x and y to be any two chunks in say classes Cx and Cy respectively.

$R1^{-1}$=R1 because (x R1 y) iff x and y are chunks in the same class, and (x R1 y) gives (y R1 x), i.e., (y $R1^{-1}$ x) is satisfied.

(x R2 y) iff y is the value for a non-atomic valued instance variable in x, then (y $R2^{-1}$ x) because x is an instance with an instance variable having the value y.

(x R3 y) iff y is a super chunk of x, then (y $R3^{-1}$ x) because x is a subobject of y or x inherits from y.

Therefore, if two objects in different classes are related by R2 or R3 then the same two objects are related by $R2^{-1}$ or $R3^{-1}$ respectively.

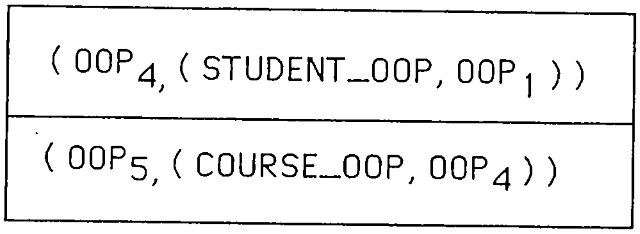
Via the relation $R2^{-1}$, for each object y we can find all objects, say x, such that (x $R2^{-1}$ y); the group of objects obtained are referencing y along the forward nesting dimension as their immediate nested chunk. Because the number of objects related to y by $R2^{-1}$ is arbitrary, a variable length record with the following format is built to show the immediate referencing chunks of y.

$$(OOPy\,(OOPC,\ OOPx)^*)$$

where OOPy and OOPx are the OOPs of the chunks and OOPc is the OOP of the class C in which x is an instance.

For each object having a nested object in the database, a variable length record can be formed using $R2^{-1}$. All those variable length records form what we call a Referencing Object Table (ROT).

The ROT serves to establish all objects that are referencing a given object as their immediate nested chunk along the nesting dimension. Returning back to the example given in Section 4.2.2, the corresponding ROT is shown in Figure 5.1.



(OOP$_4$, (STUDENT_OOP, OOP$_1$))

(OOP$_5$, (COURSE_OOP, OOP$_4$))

Figure 5.1: The constructed ROT for the example in Section 4.2.2

By the same way, using relation $R3^{-1}$, for each object y as a super chunk we can find all subobjects, say x, such that (x $R3^{-1}$ y). The obtained group of objects have y as an immediate super chunk along the inheritance dimension. An object y can be the immediate super chunk of more than one object, so a variable length record with the following format:

$$(OOPy\,(OOPC,\ OOPx)^*)$$

where OOPy and OOPx are OOPs of the chunks and OOPC is the OOP of the class C in which x is an instance, is formed to show all the objects that are related to y by $R3^{-1}$. For each object acting as a super chunk of some object in the database, a variable length record can be formed using $R3^{-1}$. All variable length records obtained using $R3^{-1}$ form what we call SubObject Table (SOT).

The SOT serves to locate all the immediate subobjects of a given object, i.e., all the objects with which the given object is related by R3. The SOT for the example in Section 4.2.2 is shown in Figure 5.2.

(OOP$_2$, (STUDENT_OOP, OOP$_1$))
(OOP$_6$, (TEACHER_OOP, OOP$_6$))

Figure 5.2: The SOT for the example in Section 4.2.2

Given an expression of messages and as described in Section 5.5.5, it is possible to establish the class of an object receiving a message and push it into a stack. The OOPs of the objects in the class at the top of the stack are used to get the OOPs of the objects in the class to become the top of the class after popping the stack.

Therefore, the SOT and ROT facilitate locating of any chunk x that references a given chunk y, regardless of how deep y is with respect to x. Knowing the OOP of y, the SOT or the ROT is accessed to get the subobjects of y or the objects that are referencing y as their immediate nested chunk. By repeating this look up using the obtained chunks instead of y, any object x can be reached. In other words, given the identity of an object, all objects that are referencing the given object in the backward direction along the nesting or inheritance dimensions can be obtained from the ROT and SOT.

On the other hand, to find all chunks found in the forward direction along the nesting or inheritance directions with respect to a given chunk y, the IT and NT are used. The identity of y is used to get from the IT or NT the immediate supers or nested chunks of y. By repeating this look up with the obtained chunks instead of y, the target chunks are reached.

The SOT and ROT give objects' identities not their locations. Using its established identity and accessing the DOT, the location of the object in the segment of its class can be reached.

The IT, NT, SOT and ROT are structured as B-trees with chunk_OOP as the index argument. Identity indexes are, therefore, based on the ROT and SOT if the target object falls in the backward direction along the nesting or inheritance dimension with respect to the given object and on the NT and IT if the target object falls in the forward direction along the nesting or inheritance direction with respect to the given object.

It may be required to find in a class objects that are referencing objects in another class; given the identities of the later objects. Here, the target objects are found in the reverse direction along the nesting or inheritance dimension with respect to the given objects. In the absence of an identity index, one should iterate over all the instances in the former class. For each such instance, the IT and NT are traced to find out objects referencing that instance. On the other hand, using an identity index, the SOT and ROT contain indirect references from the given object to the target objects. The SOT and ROT are used to extract the target objects from the former class. This is done transparently to the user and even without any hints from the user. Therefore, given a chunk's OOP and using the SOT and the ROT, all the instances of a certain class that reference the given chunk as a super or nested subobject can be found.

Depending, therefore, on the identity of a chunk, all related chunks in a target class may be located.

5.5.2 Equality Index

An identity index helps in locating objects depending on the identity of some objects they reference; but what if access is to be done based on a predicate on the value of some atomic valued instance variable within the referenced objects? An equality index is introduced for such situations. An equality index look up is actually a two step process; in the first step the object identities of objects that satisfy the predicate are formed and in the second step identity index is applied based on the resultant identities in the first step.

An indexing technique given in [55] is used in constructing equality indexes. It relates different possible values of the search argument to the object identities. This relationship is based on the value of the instance variable defined in the class whose instances are to be indexed. Such an instance variable should be defined locally inside the class; neither inherited, nor nested. For different values of the instance variable, the identities of the objects that have the same value are collected together. Such collections facilitate the retrieval of object identities for objects possessing the same value of the indexed instance variable.

To satisfy the requirements of range queries B-trees are used to implement the equality indexes. A B-tree is built to index instances of a certain class on the values of some instance variable defined locally within the class. Hence, given a possible value for an instance variable, the B-tree related to that instance variable can be accessed to get the OOPs of chunks that contain the search argument as the value of their indexed instance variable or to satisfy the search predicate. Then the identity index is used to get to the chunks themselves.

5.5.3 Index Creation

An index cannot be used until it is created. So the first step in index manipulation is index creation.

Two index creation methods may be thought of; in the first method, an index is set up automatically, i.e., for identity indexes SOT and ROT are set up and for each specified atomic valued instance variable defined in a class an equality index is set up. In the second method, the user is given means to set up an index. The means required in the second method are nothing more than provision of messages for staying within the realm of object-orientation.

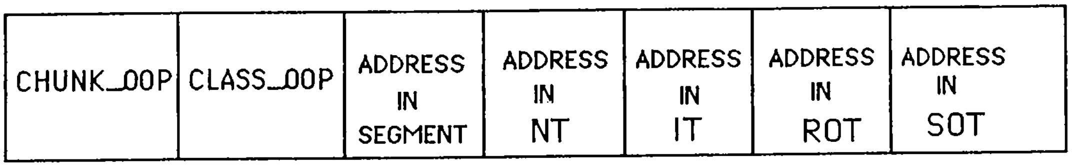
The philosophy to be followed in index creation is to automatically set up SOT and ROT to serve for identity indexes; but to use the second method for equality indexes.

```
Procedure Build_IT_NT_SOT_ROT;
  begin
    For every class (C) in the class hierarchy/lattice do
      begin
        If (supers(C)<>OBJECT) then
          begin
            For every instance (i) in C do
              begin
                vlr= OOPi; { vlr is the variable length record that will show the immediate supers of
                                       chunk i in C }
                For j=1 to R do { R is the number of supers of C }
                  begin
                    vlr=vlr||(class_OOPj, super(i,j)); { super(i,j) finds the super chunk of i in class j }
                    a=Locate_SOT(super(i,j));
                    If (a is found) then
                      Append_record((class_OOPc,OOPi)) { Add to the record found the node that shows
                                                                              i as a subobject }
                    else
                      Append_SOT( super(i,j), (class_OOPc,OOPi)); { Add a record to represent the
                                                                              super of i in the SOT }
                  end;
                  If (R>0) then Append_IT(vlr);
              end;
          end;
      If (there are S>0 non-atomic instance variables in C) then { S is the number of range classes
                                                                              for instance variables of C }
        begin
          For every instance (i) in C do
            begin
              vlr1=OOPi; { vlr1 is the variable length record that shows the immediate nested chunks
                                       of i }
              For j=1 to S do
                begin
                  If (value(j) is a collection) then
                    begin
                      m=size(collection);
                      For k=1 to m-1 do
                        begin
                          vlr1=vlr1||(class_OOPj, OOPk, 1); { 1 is the value of the Flag }
                          b=Locate_ROT(OOPk);
                          if (b is found) then
                            Append_record( (class_OOPc, OOPi))
                          else
                            Append_ROT(OOPk, (class_OOPc, OOPi));
                        end;
                    end
                  else
                            m=value(j);
                  vlr1=vlr1||(class_OOPj,OOPm,0);
                  Locate_ROT(OOPk);
                  if (found) then
                            Append_record( (class_OOPc, OOPi))
                  else
                            Append_ROT(OOPk, (class_OOPc, OOPi));
                end;
                Append_NT(vlr1);
            end;
        end;
      end;
  end;
```

Algorithm 5.1 Construct the IT, NT, SOT and ROT for the objects in the database

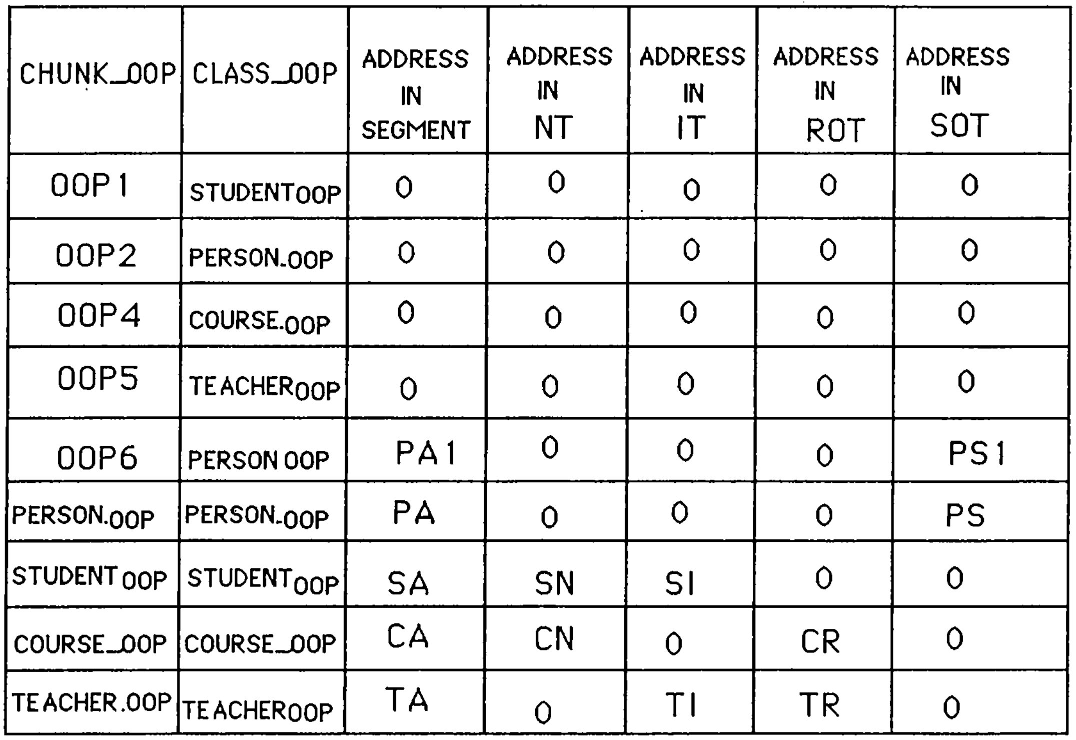
According to Algorithm 5.1, the SOT and ROT are built in parallel with the IT and NT. For every object x in a class, R3 is applied to find the supers of x in the range classes, i.e., the superclasses of x's class. The result of the application of R3 using x is a variable length record that shows the immediate supers of x. However, the application of $R3^{-1}$ is done in parallel with R3 with the range classes of R3 as the domain classes for $R3^{-1}$ and the domain class of R3, i.e., x's class, as the range for $R3^{-1}$. The result of $R3^{-1}$ is the addition of a node to the variable length record in SOT of each of the immediate supers of x, to show x as an immediate subobject. By the same way, the application of $R2^{-1}$ is done in parallel with the application of R2 to result in the addition of a node to the variable length record in ROT of each of the immediate nested chunks of x to show x as an immediate referencing chunk.

As shown in Figure 5.3, two columns are added to the DOT to keep the address of the record that represents a chunk in the ROT and SOT. The DOT for the example of Section 4.2.2 is given in Figure 5.4. In Figure 5.4 CR and TR stand for the starting address of the ROT of the "course" and "teacher" classes respectively. PS is the address of the SOT of the "person" class while PS1 is the address of the record of OOP6 relative to PS.



CHUNK_OOP	CLASS_OOP	ADDRESS IN SEGMENT	ADDRESS IN NT	ADDRESS IN IT	ADDRESS IN ROT	ADDRESS IN SOT

Figure 5.3: The augmented format of the Disk Object Table (DOT)



CHUNK_OOP	CLASS_OOP	ADDRESS IN SEGMENT	ADDRESS IN NT	ADDRESS IN IT	ADDRESS IN ROT	ADDRESS IN SOT
OOP1	STUDENTOOP	0	0	0	0	0
OOP2	PERSON.OOP	0	0	0	0	0
OOP4	COURSE.OOP	0	0	0	0	0
OOP5	TEACHEROOP	0	0	0	0	0
OOP6	PERSON OOP	PA1	0	0	0	PS1
PERSON.OOP	PERSON.OOP	PA	0	0	0	PS
STUDENT OOP	STUDENTOOP	SA	SN	SI	0	0
COURSE_OOP	COURSE_OOP	CA	CN	0	CR	0
TEACHER.OOP	TEACHEROOP	TA	0	TI	TR	0

Figure 5.4: The DOT for the example in Section 4.2.2

For the creation of an equality index, the following is proposed. In ODS [54] two instance methods are automatically generated with the definition of an instance variable. For the manipulation of each defined instance variable two messages, get<variable_name> and set<variable_name> are provided. A third method can be generated to index instances of a class on a defined instance variable. The message index<variable_name> can be sent to the class to set up an index on that instance variable. The third method, however, when executed builds up a B-tree out of the instances of the class where the index argument instance variable is defined. For example, the instance variable 'salary' is defined in the class 'teacher'. In ODS two methods are generated for this instance variable and executed on sending the messages setsalary() and getsalary(). A third method can be generated to set up an index on 'salary' within the class 'teacher' and executed on sending the message indexsalary() to the class 'teacher'. The third method builds an equality index based on the 'salary' instance variable. The result is a B-tree that relates different values of the 'salary' instance variable to the identities of the instances of the class 'teacher'.

It is possible to create an equality index for more than one class in the hierarchy/lattice. An instance variable defined in the class serves as the index argument. It is also possible for more than one instance variable in a class to be the index argument for an equality index. For such cases, variables may be added to the class definition to represent certain permutations of its instance variables. On creating an equality index on more than one instance variable, the message index<variable> may then be sent to the class. An Index Directory (ID) is used to keep information related to all equality indexes set up in a database. The ID includes the pair (instance variable OOP and the address of B_tree root) for each instance variable that serves as the index argument for an equality index. The ID is constructed as a B-tree with instance variable OOP as the index argument.

5.5.4 Schema Changes and Indexing

It is important to consider the relationship between schema changes [6] and the proposed indexing approach. We claim that schema changes do not affect existing indexes. How and to what extent this claim is achieved is discussed next.

The proposed indexing approach defines two kinds of indexes, identity index and equality index. Identity index introduces SOT and ROT that show the immediate nesting and the immediate referencing objects for all objects found in the database.

On the other hand, an equality index involves a preliminary look up step after which an identity index is applied. This step is required to retrieve identities of the chunks that satisfy the comparison condition between a search argument and an atomic valued instance variable in a class. The ID is accessed based on the instance variable OOP to get the address of the root of the B-tree serving the equality index on the instance variable. Then the B-tree is accessed using the search argument to obtained object identities.

It is obvious that schema changes will not affect an equality index, unless the change is to a class whose instances are indexed by the preliminary step in an equality index. Even identity indexes will not be affected by schema changes, except for the addition of new instances to a class or the deletion of existing instances from a class where entries need to be added to or deleted from the SOT, ROT, IT and NT.

For instance, suppose that instances in class 'employee' reference instances in class 'person' and instances in class 'manager' along the inheritance and nesting dimensions respectively. An equality index may be set up on an instance variable of class 'manager'. An identity index is automatically set up for instances of classes 'employee', 'manager', and 'person'. The addition of the class 'department' between the classes 'employee' and 'manager' will affect neither the equality nor the identity indexes. Because instances from class 'manager' will keep on satisfying the equality index which is local to class 'manager'. The identity index built before will be automatically extended to include instances of 'department' class.

5.5.5 Query processing

In a message based object-oriented language, on instantiating an expression of the form:

$$< object > < message1 > < message2 > \cdots < messageN >$$

until no message is left, every object receiving the succeeding message produces a new object, that replaces the receiving object-message pair. The result of the whole expression will be the final object which has no messages left to receive.

In processing an expression, it is possible to establish the class of an object receiving a message and push it into a stack. The class of the object that receives the last message, i.e., top element of the stack, is used in the case of an equality index to look up the ID for the address of the root of the B-tree that indexes instances of this class on the value of the instance variable given by the last message. The B-tree is accessed to extract the identities of instances that meet the comparison condition following the last message. But if there is no equality index set up on the instance variable, all instances of the class at the top of the stack receives the last message to get iteratively the OOPs of the objects that meet the comparison condition.

Depending on the OOPs of the resultant objects from the previous step, the SOT and ROT are accessed to get the OOPs of the objects in the class popped from the stack; the same thing is repeated with the objects obtained from the previous step, until the stack becomes empty.

5.5.6 Application to Other Systems

The proposed indexing approach meets the index requirements for the existing object-oriented storage systems.

Inheritance and nested objects are common in all object-oriented systems, they are among the basic constructs of object-orientation. Due to that, with any object-oriented database management system R2, R3, $R2^{-1}$, and $R3^{-1}$ can be applied to objects in the database to build the NT, IT, ROT, and SOT.

Equality index is applicable as it is without any change, since an equality index indexes instances of a class depending on different possible values of an index argument.

5.5.7 Comparisons and Evaluations

The indexing problem in object-oriented database systems is not a trivial issue, due to the mentioned distinguishing features of object-oriented systems. To index an object-oriented database system in such a way that all the objects that satisfy a given search argument can be located without any violations of the object-oriented concepts and principles is the goal.

The indexing methods used by the described existing methods are all using the path concept which is not an object-oriented construct; on the contrary it violates some of the object-oriented principles. A path is nothing more a trace of the internal structure of objects. A path includes a detailed description of the way, derived from the internal structure of objects, to be followed in searching for the target objects. How the nesting and inheritance dimensions are to be followed, on going from the root chunk of an object to the target chunk, is found within the path. Such a description violates the fact that the internal structure of an object must be hidden from the outside and is accessible only via message passing. From this point of view, therefore, an indexing method that depends on message passing is preferable for not to go out of the realm of object-orientation.

In addition, any change to the class hierarchy/lattice will be reflected by changing those paths and rebuilding the indexes that include something related to the changed part of the class hierarchy/lattice. It is, therefore, necessary to have a system that stands unchanged regardless of minor changes in the class hierarchy/lattice. By minor changes we mean changes that do not affect the instance variable on which indexing is done.

The proposed indexing method is dependent on message passing. Minor changes in class hierarchy/lattice do not affect the existing indexes in the proposed method. Simply, the proposed indexing method tries not to violate the object- oriented concepts.

Notice that the IT, NT, SOT, ROT are automatically built transparent to the user. Given the identity of an object these tables serve in locating objects in some other class related to the former object. An equality index gives the identities of instances in a class that meet a predicate. In the absence of an equality index a sequential iteration is done over all the instances of the class to receive the last message in an expression. Having the identities in hand, identity index is manipulated. This is not facilitated by any of the described approaches where iteration is done over the instances of the class to receive the first message in the absence of an index. Each instance is instantiated using the expression; a time consuming process.

Moreover, to satisfy the indexing requirements in GemStone OPAL was adjusted to differentiate between associative accesses and sequential selection where braces and brackets were used respectively, resulting in an impedance mismatch problem in the language.

Chapter 6

INTEGRITY VS. OPERATIONS AND SCHEMA CHANGES

6.1 Integrity Preservation

The preservation of integrity [17, 28] of a database system is concerned with the maintenance of the correctness and consistency of data. The database implementor is responsible for the problem of maintaining the correctness and consistency of the database. Clearly in any database environment it is desirable that maintaining the correctness and consistency of relationships in the database is not a user responsibility.

Concerning object-oriented database systems, maintaining the integrity of the database after a deletion operation is the crucial point. On deleting an instance from a class it is necessary to consider the references to the deleted chunk along the nesting and inheritance dimensions. Unless informed about the deletion other objects will keep the reference to the deleted chunk and this may result in more accesses to recognize the absence of the chunk.

In existing object-oriented database management systems a chunk captures the atomic and non-atomic valued instance variables. Dropping all the references to a deleted chunk is a time consuming operation if not impossible. Nothing related to the referencing chunks is known when deleting a chunk. To drop all the references it is necessary to iterate over all the instances in the classes that define some instance variable which has the class of the deleted chunk as its range. It seems to be an inefficient operation. Due to that, the existing object-oriented database systems are satisfied by doing more accesses to recognize the absence of a chunk over dropping all the references to a deleted chunk. For instance, Encore marks the OOPs of the deleted chunks unreachable to serve the accesses to a deleted chunk. It is a short run solution. Inefficiency appears in the long run considering that the more accesses are done on every trial to access a deleted chunk.

The introduction of the IT, NT, ROT and SOT adds much to the maintenance of the database integrity. These tables help in dropping all the references to a deleted chunk. The SOT and the ROT show the references in the backward direction along the inheritance and nesting dimensions. Via these two tables references to a deleted chunk can be efficiently dropped without any troubles. The detailed description of integrity preservation in the proposed system is presented next in this Chapter.

6.2 Operations

The key consideration in any storage management system is how different operations of addition, deletion, fetching, saving, and updating of entries can be handled efficiently. Efficiency is an important issue in storage systems. In this section a description of the operations is presented and the related algorithms are included. It is important to emphasize that all the operations are message based. The operations are manipulated by message-passing for not to go out of the realm of object-orientation. Messages are sent to a class, an object or a chunk. A received is interpreted by executing the method that implements the algorithm of the corresponding operation.

The policy to be followed for the deletion operation is that a chunk can not be deleted unless it is a root chunk. A root chunk has no entry in the SOT. An exception is that a non-root chunk can be deleted as being an instance in a deleted class; although it has an entry in the SOT. The immediate super chunks of a deleted non-root chunk replaces it as immediate super chunks of its immediate subobjects. References along the nesting dimension cease to exist after the deletion, i.e., the references to a deleted chunk as immediate nested chunk will disappear.

6.2.1 Addition

There are many candidates for the addition operation: a class may be added to the class hierarchy/lattice, an object with all its constituting chunks may be added; a chunk may be added to a class, or even an instance variable can be added to the definition of a class.

According to Algorithm 6.1 the addition of a class may be done either at the leaves or between a class and one of its superclasses. Chunks in the new class, being related by R1, are put in a new segment. Chunks inside the new segment are related to other chunks in other segments either by the relation R2 or by the relation R3. These relations are reflected by adding entries to the IT, NT, ROT and SOT to represent the chunks in the new segment. One entry per new chunk, is added to the DOT.

The addition of a new class results in the insertion of a new entry in the DOT to hold the class OOP together with the starting address of the segment's class.

According to Algorithm 6.2 the addition of a new chunk to a class is handled by adding the chunk to the segment of its class. An entry is added to the IT and another to the NT to reflect the relations of the new chunk to other chunks. An entry representing the new chunk is added to the DOT.

According To Algorithm 6.3 the addition of a new object is recursively done by starting with the root chunk and adding each reachable chunk until no more chunks are left.

```
Procedure Add_Class(OOP1,OOP,OOP2); { add class OOP as superclass of OOP1 and subclass
                                  of OOP2}
 begin
  Locate_DOT(OOP1); { find the entry for the subclass OOP1 }
  Generate_Segment(OOP); { to initialize a segment for the new class }
  For each instance (i) in OOP1 do
   begin
    j=generate_super(i,OOP); { to generate the super chunk in class OOP for chunk i }
    Append_Segment(OOP,j); { to put the atomic valued instance variables of the generated
  chunk in the segment }
    Locate_IT(OOPi); { find the record of i in the IT }
    k=Locate_record(OOP2); { find the node that represents the super chunk of i in
                        class OOP2 }
    Append_IT(OOPj,k); { to show the previous immediate super of i in OOP2 as the immediate
                      super chunk of j in class OOP2 }
    Replace_record(OOPi,k,(OOP,OOPj)); { to replace node k in the record of i in the IT by the
                                  node that shows j as a new immediate super chunk
}
    Locate_SOT(OOPk);
    m=Locate_record(OOP1); { find the node that represents the subobject in class OOP1 }
    Append_SOT(OOPj, m);
    Replace_record(OOPk,m,(OOP,OOPj));
    If (j has S non-atomic valued instance variables) then
     begin
      vlr=OOPj;
      For l=1 to S do
       begin
        If (value(l) is a collection) then
         begin
          n=size(collection);
          For p=1 to n-1 do
           begin
            vlr=vlr||(Class_OOPl, OOPp,1);
            Locate_ROT(OOPp);
            If (found) then
             Append_record(OOP,OOPj)
            else
             Append_ROT(OOPp, (OOP,OOPj));
           end;
         end
        else
            n=value(l);
         vlr=vlr||(class_OOPl, OOPn,0);
         Locate_ROT(OOPn);
         If (found) then
             Append_record(OOP,OOPj)
         else
             Append_ROT(OOPn, (OOP,OOPj));
       end;
       Append_NT(vlr);
     end;
       Append_DOT(OOPj, OOP, address_in_segment, address_in_NT, address_in_IT,
                             address_in_ROT, address_in_SOT);
  end;
  If (there are some instances rooted in class OOP) then
      begin
       For each instance (j) rooted in OOP do
        begin
         Append_segment(OOP,j);
         k=super(j,OOP2); {the super chunk in class OOP2 of chunk j }
         Append_IT(OOPj, (OOP2,OOPk));
          Locate_SOT(OOPk);
          If (found) then
            Append_record(OOP,OOPj)
          else
            Append_SOT(OOPk, (OOP, OOPj));
           If (j has S non-atomic valued instance variables) then
            begin
             vlr=OOPj;
             For l=1 to S do
              begin
               If (value(l) is a collection) then
                begin
                 n=size(collection);
                 For p=1 to n-1 do
                  begin
                   vlr=vlr||(Class_OOPl, OOPp,1);
                   Locate_ROT(OOPp);
                   If (found) then
                    Append_record(OOP,OOPj)
                   else
                    Append_ROT(OOPp, (OOP,OOPj));
                  end;
                end
               else
                n=value(l);
              vlr=vlr||(class_OOPl, OOPn,0);
              Locate_ROT(OOPn);
              If (found) then
                  Append_record(OOP,OOPj)
              else
                  Append_ROT(OOPn, (OOP,OOPj));
              end;
             Append_NT(vlr);
            end;
           Append_DOT(OOPj, OOP, address_in_segment, address_in_NT, address_in_IT,
                            address_in_ROT, address_in_SOT);
          end;
       end;
 end;
```

Algorithm 6.1. Add a class to the class hierarchy/lattice

```
Procedure Add_chunk(OOP, class_OOP); { adding the chunk given its OOP and the class_OOP }
 begin
  Locate_DOT(class_OOP);
  Append_segment(class_OOP, OOP); { to add the atomic valued instance variables to the
                                    segment of the class}
  vlr=OOP;
  For i=1 to R do { suppose the class class_OOP has R supers}
   begin
    k=super(OOP, i); { the super of the chunk OOP in class i }
    vlr=vlr||(OOPi, OOPk);
    Locate_SOT(OOPk);
    If (found) then
          Append_record(class_OOP, OOP)
    else
          Append_SOT(OOPk, (class_OOP, OOP);
   end;
  If (R>0) then Append_IT(vlr);
  If (OOP has S non-atomic valued instance variables) then
    begin
     vlr=OOP;
     For l=1 to S do
      begin
       If (value(l) is a collection) then
        begin
         n=size(collection);
         For p=1 to n-1 do
          begin
           vlr=vlr||(Class_OOPl, OOPp,1);
           Locate_ROT(OOPp);
           If (found) then
                Append_record(class_OOP,OOP)
           else
                Append_ROT(OOPp, (class_OOP,OOP));
          end;
        end
       else
           n=value(l);
       vlr=vlr||(class_OOPl, OOPn,0);
       Locate_ROT(OOPn);
       If (found) then
             Append_record(class_OOP,OOP)
       else
             Append_ROT(OOPn, (class_OOP,OOP));
      end;
      Append_NT(vlr);
    end;
   Append_DOT(OOP, class_OOP, address_in_segment, address_in_NT, address_in_IT,
                             address_in_ROT, address_in_SOT);
 end;
```

Algorithm 6.2 Add a chunk to a class

```
Procedure Add_object(OOP,class_OOP); { add the object with the root OOP in class_OOP }
 begin
  Locate_DOT(class_OOP);
  Append_segment(class_OOP,OOP);
  If (supers(class_OOP)<>'OBJECT') then
   begin
    vlr=OOP;
    For i=1 to R do { suppose the class has R superclasses}
     begin
      k=super(OOP,i); { the super of chunk OOP in the class i }
      vlr=vlr||(OOPi,OOPk);
      Locate_SOT(OOPk);
      If (found) then
          Append_record(class_OOP, OOP)
      else
          Append_SOT(OOPk, (class_OOP, OOP);
       Add_object(OOPk, OOPi); { add the object OOPk to the class OOPi }
     end;
    If (R>0) then Append_IT(vlr);
   end;
   If (there exist S nonatomic valued instance variables) then
       begin
        vlr=OOP;
        For i=1 to S do
         begin
          If (value(i) is a collection) then
           begin
            n=size(collection);
            For p=1 to n-1 do
             begin
              vlr=vlr||(Class_OOPi, OOPp,1);
              Locate_ROT(OOPp);
              If (found) then
                  Append_record(class_OOP,OOP)
              else
                  Append_ROT(OOPp, (class_OOP,OOP));
              Add_object(OOPp, class_OOPi);
             end;
           end
          else
             n=value(l);
          vlr=vlr||(class_OOPl, OOPn,0);
          Locate_ROT(OOPn);
          If (found) then
              Append_record(class_OOP,OOP)
          else
              Append_ROT(OOPn, (class_OOP,OOP));
          Add_object(OOPn, class_OOPi);
         end;
         Append_NT(vlr);
       end;
   Append_DOT(OOP, class_OOP, address_in_segment, address_in_NT, address_in_IT,
                   address_in_ROT, address_in_SOT);
 end;
```

Algorithm 6.3 Add an object to the database

```
Procedure Add_Instance_Variable(IV,OOP); { to add the instance variable IV to the class OOP }
  begin
    Locate_DOT(OOP);
    If (atomic_valued(IV)) then
      begin
        get_segment(OOP);
        For each instance (i) in OOP do
          begin
            Add_value(IV);
            Adjust_DOT(OOPi); { to update the address of the chunk i in the segment }
          end
      end
    else
      begin
        For each instance (i) in OOP do
          begin
            Locate_NT(OOPi);
            j=value(IV);
            If (j is a collection) then
              begin
                vlr='';
                n=size(collection);
                For p=1 to n-1 do
                  begin
                    vlr=vlr||(Class_OOPj, OOPp,1);
                    Locate_ROT(OOPp);
                    If (found) then
                            Append_record(OOP,OOPi)
                    else
                            Append_ROT(OOPp, (OOP,OOPi));
                  end;
              end
            else
                n=value(j);
            vlr=vlr||(class_OOPj, OOPn,0);
            Locate_ROT(OOPn);
            If (found) then
                  Append_record(OOP,OOPi)
            else
                  Append_ROT(OOPn, (OOP,OOPi));
          end;
      end;
  end;
```

Algorithm 6.4 Add an instance variable to a class

According to Algorithm 6.4, the addition of a new instance variable to a class can be done in two different ways each depending on whether the instance variable is atomic or non-atomic. In the first method, applicable when the new instance variable has an atomic value, the segment of the class to which the new instance variable is to be added, is accessed using its address as obtained via the DOT using the class_OOP. The segment's contents are changed by adding the value of the new instance variable to each chunk. In the second method, applicable when the new instance variable has a non-atomic value, the NT is adjusted to reflect the change.

6.2.2 Deletion

It is obvious that, anything which can be added, can also be deleted. Therefore, the candidates for the deletion operation are the same as those for addition, namely, a class, a chunk, an object with all its chunks, and an instance variable.

The OOP of a chunk can be found within the segment, in which the chunk is put, and in five tables, the DOT, the IT, ROT, SOT and the NT.

According to Algorithm 6.5 a chunk is deleted by deleting the entry of the chunk from the DOT; deleting the records of the chunk from the IT, ROT, SOT and NT; deleting the chunk from the segment in which it is found.

According to Algorithm 6.6 the deletion of an immediate super chunk of a certain chunk can be done in the IT. By being able to delete individual super chunks, a superclass may be deleted from the superclass list of a class by doing N super chunk deletions, one for each chunk in the class. The super(s) of the deleted super will replace it in the superclass(list) of its subclasses as immediate super(s).

According to Algorithm 6.7 the deletion of an object is done recursively starting by deleting the root chunk after specifying chunks reachable from it. The process is repeated until all chunks of the object are deleted.

According to Algorithm 6.8 the deletion of an instance variable depends on whether the instance variable has a chunk or a collection as its value or just an atomic value. An instance variable which has a chunk (or a collection) as its value can be deleted inside the NT by deleting the node(s) that represent the value of the instance variable from the record of the chunk. The deletion of atomic valued instance variables can be done within the related segment.

```
Procedure Delete_chunk(OOP); { the chunk should be a root chunk with no reference to it from
                                elsewhere as a super chunk }
 begin
  Locate_DOT(OOP);
  Locate_SOT(OOP); { to check if OOP is a root or not }
  If (not found) then
   begin
    Drop_chunk(OOP); { to delete the atomic valued instance variables }
    j=Locate_IT(OOP);
    Drop_IT(OOP);
    For every node (i) in j do
     begin
      Locate_SOT(OOPi);
      Drop_record(OOPi,OOP); { delete the node that represents the chunk OOP in the record
                              of i }
     end;
      p=Locate_NT(OOP);
      Drop_NT(OOP);
      For every node (k) in p do
       begin
        Locate_ROT(OOPk);
        Drop_record(OOPk,OOP);
       end;
      p=Locate_ROT(OOP);
      Drop_ROT(OOP);
      If (p is found) then
       begin
        For every node k in p do
         begin
          Locate_NT(OOPk);
          Drop_record(OOPk,OOP);
         end;
       end;
      Drop_DOT(OOP); { delete the entry of the chunk from the DOT }
   end;
 end;
```

Algorithm 6.5 Delete a chunk from a class

```
Procedure Delete_class(OOP); { to drop a class from the class hierarchy/lattice given its OOP }
 begin
  Locate_DOT(OOP);
  Drop_segment(OOP);
  For every instance (i) in OOP do
   begin
    j=Locate_IT(OOPi);
    Drop_IT(OOPi);
    k=Locate_SOT(OOPi);
    Drop_SOT(OOPi);
    If (k is found) then
     For every node (m) in k do
      begin
       n=Locate_IT(OOPm);
       Drop_record(OOPm,OOPi);
       Append_record(OOPm, j);
      end;
    If (j is found) then
     For each node (o) in j do
      begin
       p=Locate_SOT(OOPo);
       Drop_record(OOPo,OOPi);
       Append_record(OOPo, k);
      end;
    j1=Locate_NT(OOPi);
    Drop_NT(OOPi);
    k1=Locate_ROT(OOPi);
    Drop_ROT(OOPi);
    If (k1 is found) then
       For every node (m1) in k1 do
        begin
         n1=Locate_NT(OOPm1);
         Drop_record(OOPm1, OOPi);
        end;
    If (j1 is found) then
       For every node (o1) in j1 do
        begin
         p1=Locate_ROT(OOPo1);
         Drop_record(OOPo1, OOPi);
        end;
    Drop_DOT(OOPi);
   end;
  Drop_DOT(OOP);
 end;
```

Algorithm 6.6 Delete a class

```
Procedure Delete_object(OOP); { OOP of the root chunk }
  begin
    Locate_DOT(OOP);
    k=Locate_SOT(OOP);
    If (k is found) then
              write ('can not delete', OOP)
    else
      begin
        Drop_SOT(OOP);
        k1=Locate_ROT(OOP);
        For every node n in k1 do
          begin
            p=Locate_NT(OOPn);
            Drop_record(OOPn,OOP);
          end;
        Drop_ROT(OOP);
        Locate_segment(class_OOP(OOP));
        Drop_chunk(OOP); { to delete the atomic valued instance variables }
        j=Locate_IT(OOP);
        If (j is found) then
          begin
            Drop_IT(OOP);
            For every node m in j do
              begin
                t=Locate_SOT(OOPm);
                If (size(t)>1) then { m is the super chunk of more than one chunk }
                  begin
                    write('can not delete', OOPm);
                    Drop_record(OOPm,OOP);
                  end
                else
                    begin
                      Drop_SOT(OOPm);
                      Delete_object(OOPm);
                    end;
              end;
         end;
           u=Locate_NT(OOP);
           If (u is found) then
             begin
               Drop_NT(OOP);
               For every node v in u do
                 begin
                   w=Locate_ROT(OOPv);
                   Drop_record(OOPv, OOP);
                   Delete_object(OOPv);
                 end;
             end;
           Drop_DOT(OOP);
     end;
 end;
```

Algorithm 6.7 Delete an object

```
Procedure Delete_instance_variable(IV,OOP); { delete the instance variable IV from the class
                                                        OOP }
 begin
  Locate_DOT(OOP);
  If (atomic_valued(IV)) then
   begin
    get_segment(OOP);
    For each instance (i) in OOP do
     begin
      Drop_instance_variable(i,IV); { delete the instance variable IV from the chunk i }
      Adjust_DOT(OOPi); { to reflect the new address }
     end;
   end
  else
   begin
    For every instance (i) in OOP do
     begin
      Locate_NT(OOPi);
      r=range(IV); { the range class of IV }
      j=Locate_record(OOPi, OOPr); { the first node from the range class r in the record of
                                                  OOPi }
      If (Flag=1) then { the value of IV is a collection }
       begin
        repeat
         m=Locate_ROT(OOPj);
         Drop_record(OOPj, OOPi); { delete from the record of OOPj in ROT the node of
                                                  OOPi }
         Drop_record(OOPi, OOPr); { delete the located node }
         j=Locate_record(OOPi, OOPr); { the next node from the range class r in the record
                                                     of OOPi }
        until (Flag=0);
       end;
      Drop_record(OOPi, OOPr);
     end;
   end;
 end;
```

Algorithm 6.8 Delete an instance variable from a class

6.2.3 Fetching

The candidates for the fetching operation may be chunks from the same class; chunks which together form an object; or a single chunk. Actually all of the candidates involve chunk fetching.

According to algorithm 6.9 chunks from the same class are fetched in the segment for the class. The DOT is accessed only once to fetch the starting address of the segment.

```
Procedure Fetch_class(OOP); { to fetch instances of a class given its OOP }
 begin
  Locate_DOT(OOP);
  get_segment(OOP);
   end;
```

Algorithm 6.9 Retrieving instances of a class

According to Algorithm 6.10 an object is fetched recursively by fetching the root chunk. The process is repeated for chunks reachable from a fetched chunk until no more chunks are left unfetched.

According to Algorithm 6.11 to fetch a single chunk, it is necessary to have its OOP in hand. The DOT is accessed using the OOP to get the class_OOP of the chunk along with its address within the segment.

```
Procedure Fetch_object(OOP); { OOP of the root chunk}
 begin
  Locate_DOT(OOP);
  Locate_segment(class_OOP(OOP));
  j=Locate_IT(OOP);
  If (j is found) then
   begin
    For every node (n) in j do
     begin
      Fetch_object(OOPn);
     end;
   end;
    k=Locate_NT(OOP);
    If (k is found) then
     begin
      For every node (m) in k do
       begin
        Fetch_object(OOPm);
       end;
     end;
 end;
```

Algorithm 6.10 retrieving all the chunks of an object

```
Procedure Fetch_chunk(OOP);
 begin
  Locate_DOT(OOP);
  Locate_DOT(class_OOP(OOP)); { locate in the DOT the entry of the class of the given
                                chunk }
  get_chunk(OOP);
 end;
```

Algorithm 6.11 Retrieving a chunk

6.2.4 Saving

The candidates for the saving operation are the same as those for fetching. Actually all of the candidates are treated as chunk saving.

According to Algorithm 6.12 on saving a chunk, the DOT is checked, using the chunk OOP, to find if the DOT includes an entry for the chunk. If the check succeeds then the address of the chunk in the segment of its class is extracted from the DOT and the new version of the chunk replaces the old version in the segment with the related records of the chunk in the IT and NT being adjusted to reflect the change in the non-atomic valued instance variables, otherwise the chunk is added as per the procedure of Section 6.1 above.

```
Procedure Save_chunk(OOP);   begin
  Locate_DOT(OOP);
  If (found) then
    begin
     If (changed(atomic_values)) then
       begin
         Locate_segment(class_OOP(OOP));
         replace_chunk(OOP);{reflect the change in the atomic values}
       end;
     If (changed(nonatomic_values)) then
       begin
        j=Locate_NT(OOP);
        For each node (m) in j do
          If (changed(m)) then
              Adjust_ROT(OOPm); { reflect the presence or absence of a node in j }
        Adjust_NT(OOP); { reflect the change in the non-atomic valued instance variables }
       end
     else
        Add_chunk(OOP, class_OOP(OOP));
    end;
    end;
```

Algorithm 6.12 Save a chunk

6.2.5 Updating

An object and a chunk are all candidates for an update operation. The update operation can be seen as executing a fetch operation followed by a save operation, or a deletion operation followed by an addition operation.

6.3 Schema Evolution

Handling schema changes properly and efficiently is among the important aspects of any proposed storage system. Clearly, it is highly desirable for a storage system to support as many schema operations as possible.

An important advantage due to the separation of non- atomic references through IT, ROT, SOT and NT is realized here, whereby changes to an edge in the class hierarchy/lattice can be performed solely via the IT without any need to access individual chunks. Making a class a superclass of another class can be done solely via the IT that contain information related to the change. Removing a class from the superclass list of a class can again be done solely via the IT that contains information related to the change by dropping references to instances of the deleted class. Changing the order of the superclasses of a class can also be done inside the IT.

The IT, ROT, SOT and the NT contain all the associations between chunks which are instances in different classes.

A new atomic valued instance variable may be added to a class definition resulting in the value of the new instance variable being added to the existing chunks found in the segment representing the class.

6.4 Ease of Implementation

6.4.1 Data Structures

It is necessary to keep in consideration the possibility of implementing what is being proposed on the SUN workstations running under the Unix operating system.

The DOT can be seen as a file; a record in the file contains one field per column of the DOT. Entries in the ROT, SOT, IT and the NT, are represented using variable length records. The entries of each class in each of the four tables are kept in a separate file, i.e., four files per class.

A chunk in a segment is represented by a record whose fields correspond to instance variables with atomic values. Each segment is represented by a separate file.

For every method definition in a class, there is a method file in the secondary storage where the code implementing the method is kept.

The general structure of the class hierarchy/lattice, including instance variables defined in each class and relations between classes are kept in a separate file to be loaded at the beginning of each session.

6.4.2 Functions of the Storage System

One of the principal functions of the storage system is to maintain the correspondence between objects' OOPs and chunks of memory. It is responsible for the storage, retrieval, and update of chunks that reside in persistent store. It also guarantees that a chunk is stored in exactly one segment by checking the DOT before the addition of any chunk.

An important function of the storage system is to load and maintain the tables used by the Object Memory [32]; namely the instance variable definition table (IVDT), method definition table (MDT), instance access table (IAT), and the class hierarchy/lattice. Information inside the file that contains class hierarchy/lattice definition, should be kept up-to-date to reflect the final situation of the class hierarchy/lattice.

6.4.3 How to Interact with the Storage System?

Being a stand alone model in a database management system, the storage system has to provide some means by which other modules can access it. The interaction with the storage system can be done by using the interface of the algorithms described in Section 6.2. Dealing with those algorithms satisfies all the necessary operations of addition, storage, deletion, retrieval, and update of chunks in persistent store. The interaction should be message based for not to violate the object-oriented concepts. the messages should be sent to classes, objects or chunks based on the desired operation. The methods that implement the discussed algorithms are accumulated under the OBJECT class to facilitate their usage by all the classes in the class hierarchy/lattice.

Chapter 7

MAPPINGS

7.1 Mapping Objects into Secondary Storage

Programs and data grow larger than the size of main memory. So, part of the program is kept in main memory at a time and the rest in secondary storage. The operating system is responsible for managing the transfer of program parts between main memory and secondary storage. This method is known as virtual memory [52]. In computers without a virtual memory system, a virtual address [50] is put directly into the memory bus [50] and causes the physical memory word with the same address to be read or written. On using virtual memory, a virtual address does not go directly to the memory bus. Instead, a virtual address goes to the Memory Management Unit (MMU) [52]. The MMU is a chip or a collection of chips that maps a virtual address into the corresponding physical memory address. Most virtual memory systems use a technique called paging [10, 52] where the virtual address space is divided into units of equal sizes called pages. Another technique that does not necessitate that the sizes of the units be equal is called segmentation [10].

The attempt to access a missing page or segment is called a page fault or a segment fault . respectively [50]. Following the same terminology, we call the attempt to access a missing object an object fault.

The correspondence between the objects of a database system and the main memory counterparts that store them is specified in an Object Table (OT) [32]. The OT has one entry per object. The OT keeps track of objects that are in main memory; not all objects. So it is enough to check the OT to know whether the object is in main memory or in secondary storage. For each class, a number field is assigned. Classes are numbered in ascending order of accessing their instances, i.e., on accessing instances from a class it is assigned a number larger than the numbers assigned to the classes whose instances are accessed before. The usage of this field is discussed next in conjunction with the replacement policy.

On accessing an object its flag is checked. If the object is not in main memory, an object fault is said to happen. Each time an object is loaded, an entry is added to the OT. More object faults result in more I/O traffic and slow down the system performance. Therefore, it is important to decide on the policy to be used for deciding what object to fetch into main memory and what object already in main memory the newly fetched object is to replace.

7.1.1 Replacement Policy

To replace an object the following can be done. If there is a vacant location, a newly fetched object can be stored in that location. If no location is vacant, room must be made by replacing an object currently in main memory. The key point is the choice of which object to replace. It is necessary for the replacement policy to minimize the I/O traffic, and hence object faults. If a read-only object can be recognized, the traffic represented by saving it back to secondary storage can be eliminated.

The following policies may be thought of following the page replacement policies [10, 52]. Random selection of the object to be replaced and replacing the object that has been in main memory for the longest time or First-In-First-Out (FIFO) are two easy to implement policies. Replacing the object to which no reference has been made for the longest time or Least Recently Used (LRU) and replacing the object to which the fewest references have been made in a given interval or Least Frequently Used (LFU) are two other replacement policies.

The replacement policy to be used is given in Algorithm 7.1. Numbering the classes in ascending order of accessing their instances is the criteria to be followed in the replacement policy; known as aging in operating systems concept [10]]. Object from the class with the minimum assigned number are the candidates to be replaced on fetching an object into main memory because numbers are assigned in increasing order of accessing. However, replacing unchanged objects has priority over replacing changed objects because an unchanged object is a read-only object. Therefore, to find an unchanged chunk instances from all the numbered classes but the last are checked in ascending order of numbering. If an unchanged chunk is not found, the process is repeated to find a changed chunk. On the other hand, if all the chunks in the main memory are from the class of the object to be fetched, a check is done for an unchanged instance else for a changed instance of the class.

```
Procedure Find_Space(class_OOP); { find space for an object in the class with class_OOP }
 begin
  If (max_number>min_number) then { there are objects from more than one class in
    main memory, min_number is the number assigned to the class whose instances are
    fetched before instances of all other classes and max_number is the number assigned
    to the class whose instances are fetched after instances of all other classes }
   begin
    If (number(class_OOP)=min_number) then
    min_number=min_number+1;
    max_number=max_number+1;
    number(class_OOP)=max_number;
    i=min_number;
    check='not changed';
    repeat
     j=Locate_OT(class_OOP(i), check); { find an unchanged chunk from the class with the
                                         number i }
     If (found) then
      begin
       If (check='not changed') then overwrite
       else
        begin
         save_chunk(chunk_OOP(j)); { using Algorithm_6.12 }
         overwrite;
        end;
      end
     else
      begin
       i=i+1;
       If (i=max_number) then
        begin
         If (check='not changed') then { look for a changed chunk }
          begin
           i=min_number;
           check='changed';
          end
         else
         If (check='changed') then { look for a non-changed chunk in the class with the
                                     maximum number }
               check='not changed';
     end
       else
       If (i>max_number) then
        begin      { look for a changed chunk in the class with the maximum number }
         i=max_number;
         check='changed'
        end;
      end;
     until (found);
   end
  else { there are chunks from only the class with the maximum number in main memory }
   begin
    check='not changed';
    i=max_number;
    repeat { look for a non-changed chunk, else a changed chunk from the class whose instances
             are accessed }
     j=Locate_OT(class_OOP(i), check);
     If (found) then
      begin
       If (check='not changed') then overwrite
       else
        begin
```

```
            save_chunk(chunk_OOP(j));
            overwrite;
          end;
        end
      else
        check='changed';
    until (found);
  end;
end;
```

Algorithm 7.1 Find space in main memory

7.1.2 Fetching Policy

The simplest policy for choosing which objects to have in main memory is to load all objects used by the database at start. Having all the objects in main memory will eliminate all object faults. But having more objects than the main memory can keep at once necessitate the presence of part of the objects in main memory and the rest in secondary storage. An object is fetched only on demand as a result of an object fault. It is guaranteed that every object fetched in indeed needed and will be accessed.

According to Algorithm 7.2 on evaluating an expression an object is fetched when it is needed, otherwise accesses passing through the object are followed inside the IT and NT.

```
Procedure Find_Object(OOP, expression); { finds the resultant object resulting from the expression
                                           being received by the object OOP}
begin
  OOP(0)=OOP;
  For i=1 to (N-1) do { assume N messages in the expression}
    begin
      If (OOP(i-1) is not in main memory) then
        begin
          Locate_DOT(OOP(i-1));
          Find_Space(class_OOP(OOP(i-1)); { using Algorithm_7.1 }
          If (nested) then { OOP(i) is a nested chunk of OOP(i-1) }
                Locate_NT(OOP(i-1))
          else
            If (super) then { OOP(i) is a super chunk of OOP(i-1) }
                Locate_IT(OOP(i-1));
        end;
          OOP(i)=receive(OOP(i-1), message(i)); { the object OOP(i-1) receives the message(i) to
                                                   give OOP(i) }
    end;
      If (OOP(N-1) is not in main memory) then
        begin
          Find_Space(class_OOP(OOP(N-1))); { using Algorithm 7.1 }
          Fetch_chunk(OOP(N-1)); { using Algorithm 6.11 }
        end;
end;
```

Algorithm 7.2 Evaluate an Expression

7.2 Mapping the Storage System into a Relation Database System

Object-oriented database systems and relational database systems are two major database research areas in recent years. A relational database is a collection of relations [17, 18, 28, 55]. Taking all the constructs of the relational database system into consideration, the proposed storage and indexing approaches are to be mapped into a relational scheme.

In the proposed system six structures were defined, namely the IT, NT, SOT, ROT, DOT and segment. The IT keeps for each chunk in the database a variable length record that includes the chunk_OOP and the class_OOP -chunk_OOP pair to show the immediate supers of the chunk. For each variable length record in the IT, each class_OOP -chunk_OOP pair is preceded by the chunk_OOP of the chunk that owns the record to get triples. All the triples resulting from the transformation of the variable length records in the IT can be the tuples in a relation with three attributes, called IT_Relation. The three attributes are chunk_OOP, class_OOP, and chunk_OOP. The first attribute shows the OOP of a chunk while the second and third attributes show the class_OOP and chunk_OOP of an immediate super of the former chunk.

IT_Relation(OOPy, OOPC, OOPx)

where OOPy and OOPx are chunk OOPs and OOPC is the class OOP of OOPx.

To get all the immediate supers of a chunk, a selection is done from the IT_Relation for all the tuples that have the given chunk_OOP as the value for the first attribute. To get the immediate super in a given class of a given chunk, a selection is done from the IT_Relation based on equating the value of the given chunk_OOP and class_OOP to the first and second attributes respectively.

Concerning the SOT that shows for each chunk its immediate sub_objects, it contains variable length records of the same format with the records found in the IT. So following the same construction steps described for the IT_Relation, the SOT_Relation can be constructed. It has three attributes, namely the chunk_OOP and the class_OOP and the chunk_OOP. The first attribute shows the OOP of a chunk while the second and third attributes show the class_OOP and chunk_OOP of an immediate subobject of the former chunk.

SOT_Relation(OOPy, OOPC, OOPx)

where OOPy and OOPx are chunk OOPs and OOPC is the class OOP of OOPx.

To get all the immediate subobjects of a chunk, a selection is done from the SOT_Relation for all the tuples that have the given chunk_OOP as the value for the first attribute. To get the immediate subobject in a given class of a given chunk, a selection is done from the SOT_Relation based on equating the value of the given chunk_OOP and class_OOP to the first and second attributes respectively.

However, adding a column to the IT_Relation to keep the class_OOP for the chunk of the first column will render the SOT_Relation unnecessary.

IT_Relation_Adjusted(OOPy, OOPCy OOPCx, OOPx)

where OOPy and OOPx are chunk OOPs and OOPCx and OOPCy are the class OOPs of OOPx and OOPy.

Selection from IT_Relation_Adjusted can be done based on the first column for a forward reference along the inheritance dimension, and the third column for backward references.

The ROT shows for each chunk its immediate referencing chunks, it contains variable length records of the same format with the records found in the IT. So following the same construction steps described for the IT_Relation, the ROT_Relation can be constructed. It has three attributes, namely the chunk_OOP and the class_OOP and the chunk_OOP. The first attribute shows the OOP of a chunk while the second and third attributes show the class_OOP and chunk_OOP of an immediate referencing chunk of the former chunk.

ROT_Relation(OOPy, OOPC, OOPx)

where OOPy and OOPx are chunk OOPs and OOPC is the class OOP of OOPx.

To get all the immediate referencing chunks of a chunk, a selection is done from the ROT_Relation for all the tuples that have the given chunk_OOP as the value for the first attribute. To get the immediate referencing chunk in a given class of a given chunk, a selection is done from the ROT_Relation based on equating the value of the given chunk_OOP and class_OOP to the first and second attributes respectively.

The NT shows for each chunk its immediate nested chunks, it contains variable length records of the same format with the records found in the IT except that pairs are replaced by triples with the Flag as the third argument. So following the same construction steps described for the IT_Relation, the NT_Relation can be constructed. It has four attributes, namely the chunk_OOP and the class_OOP, the chunk_OOP and the Flag. The first attribute shows the OOP of a chunk while the second and third attributes show the class_OOP and chunk_OOP of an immediate nested chunk of the former chunk and the Flag is set to 1 for all the nested chunks that are found in the same collection.

NT_Relation(OOPy, OOPC, OOPx, Flag)

where OOPy and OOPx are chunk OOPs and OOPC is the class OOP of OOPx.

To get all the immediate nested chunks of a chunk, a selection is done from the NT_Relation for all the tuples that have the given chunk_OOP as the value for the first attribute. To get the immediate nested chunk in a given class of a given chunk, a selection is done from the NT_Relation based on equating the value of the given chunk_OOP and class_OOP to the first and second attributes respectively.

However, adding a column to the NT_Relation to keep the class_OOP for the chunk of the first column will render the ROT_Relation unnecessary.

NT_Relation_Adjusted(OOPy, OOPCy OOPCx, OOPx, Flag)

where OOPy and OOPx are chunk OOPs and OOPCx and OOPCy are the class OOPs of OOPx and OOPy.

Selection from NT_Relation_Adjusted can be done based on the first column for a forward reference along the nesting dimension, and the third column for backward references.

The segment contains from each chunk its atomic valued instance variables. For each segment a relation is defined with one attribute per atomic valued instance variable. The first attribute shows the OOP of the chunk and the rest contain the values of the atomic valued instance variables. To get all the atomic valued instance variables of a chunk, a selection is done from the relation defined for the segment of the class for all the tuples that have the given chunk_OOP as the value for the first attribute. To get the value for a single atomic valued instance variable of a given chunk, a selection is done from the relation for all the tuples that have the given chunk_OOP in the first attribute. A projection on the column that represent the desired instance variable is done.

The DOT includes for each chunk, its OOP, the class_OOP, and five addresses one for the information related to the chunk in each of the segment, the IT, the NT, the ROT and the SOT. It serves in giving the address of the information related to a given chunk in the five mentioned locations. But as it is shown above, chunk_OOP is enough to get the corresponding information from the IT_Relation, NT_Relation, ROT_Relation, SOT_Relation and the relation that represent the segments. So, this table need not to be represented by a separate relation. Indeed, the information found in the DOT has been distributed among the above defined relations.

To summarize, the IT, NT, SOT and ROT can be mapped into a relational system by two different representations. In the first representation three attributes are defined for each of the IT, ROT and SOT and an additional fourth attribute is defined for the NT. The three attributes are a chunk_OOP and the pair (class_OOP, chunk_OOP) for an immediate related chunk. The fourth attribute in the NT is the Flag that keeps track of collections. In the second representation the ROT_Relation and SOT_Relation are removed after the addition of a column to each of the IT_Relation and NT_Relation. The added attribute keeps the class_OOP of the chunk in the first column.

Chapter 8

CONCLUSIONS

The proposed storage system is a new approach to storage management in object-oriented database management systems. The proposed storage system supports multiple inheritance of which simple inheritance is a special case.

The introduction of IT, NT, and segments facilitates much schema changes, results in the finest degree of granularity, and improves the number of disk accesses.

It is not a problem to access instances of a class, because they are clustered into segments. Segments for related classes are grouped into sets to improve access to all chunks of an object.

The user need not worry about persistence of objects, it is carried out automatically by the system.

Version control is not supported by the storage system. In order to be able to represent the temporal aspects of the data, the basic storage scheme used for object instance variables has to be modified. Instead of a value, a value and a time pair must be stored for each instance variable. However, a future work can be carried out on version control and the proposed storage system can be extended to satisfy version control.

Performance appears to be the fundamental problem with most object-oriented programming languages and database management systems. One performance problem is that, the instances of a class are accessed by searching them all in the absence of indexing.

The presented indexing methods satisfy the indexing requirements of the proposed storage system. It also satisfies the indexing requirements of existing object- oriented database systems. We claim this because the proposition is based on the inheritance and complex objects features of object-oriented systems; and these two features are common to all object-oriented database systems. Multilevel indexing is achieved. Identity and equality indexes are considered.

Being message based, the discussed indexing method does not violate the encapsulation concept of object-oriented systems. Schema changes do not affect an existing index, unless the change is to the index argument instance variable.

The user does not need to worry about identity index set up which is done automatically by the system. The user has the opportunity to set up an equality index where it is considered necessary.

Finally, the introduction of the IT, NT, ROT and SOT adds much to the integrity of the database. Using these tables, the deletion of a chunk can be handled because these tables keep relations between chunks in the forward and reverse directions along the nesting and inheritance dimensions.

At the end, the mapping of the proposed storage and indexing approaches into a relational scheme is presented. So the proposed storage system and indexing approaches can be either implemented from scratch or put on top of an existing relational database system.

REFERENCES

1– Agha, G., A Message Passing Paradigm for Object Management, Database Engineering, Vol.8, no.4, December 1985, pp. 311-318.

2– Agrawal R., Fehani N.H., ODE (Object Data Model): The Language and Its Implementation, Proceeding of the ACM SIGMOD International Conference on Management of Data, 1989, pp.36-45.

3– Albano A., et al., A Strongly Typed, Interactive Object- Oriented Database Programming Language, Proc. of the Workshop on Object-Oriented Database Systems, September 1986, pp.94-103.

4– Al-Hajj R., E. Arkun, A Model for Storage Management in Object-oriented Database Management Systems, Proceeding of the Fifth International Symposium on Computer and Information Sciences, October 1990, Cappadocia , Turkey.

5– Al-Hajj R., E. Arkun, A Model for Indexing in Object- oriented Database Management Systems, Proceeding of the Fifth International Symposium on Computer and Information Sciences, October 1990, Cappadocia , Turkey.

6– Banerjee, J.H.J. Kim, W. Kim, and H.F. Korth, Schema Evolution in Object-Oriented Persistent Databases, Proc. of the 6th Advanced Database Symposium (Tokyo, Japan, Aug.) Information Processing Society of Japan's Special Interest Group on Database Systems, 1986, pp.23-31.

7– Banerjee, J. et al, Data Model Issues for Object-Oriented Applications, ACM Transactions on Office Information Systems, vol.5, no.1, Jan.1987, pp.3-26.

8– Banerjee, J., W. Kim,, and, K. C. Kim, Queries in Object- Oriented Databases, MCC Technical Report, 1987.

9– Borgida A., et al., Classic: A structural data model for objects, Proceeding of ACM SIGMOD International Conference on Management of Data, 1989, pp.58-67.

10– Calingaert P., Operating system Elements: A User Perspective, Prentice Hall, Inc., 1982.

11– Carey, M.J. and Dewitt D.J., The Architecture of the EXODUS Extensible DBMS, Proceedings of the International Workshop on Object-Oriented Database Systems, Sept 23- 26, 1986 Pacific Grove, pp.52-65.

12– Carey M.J., et. al., A Data Model and Query Language for EXODUS, Proceeding of ACM SIGMOD International Conference on Management of Data, 1988, pp.413-422.

13– Carey M.J. and Shekita E.J., Performance Enhancement Through Replication in an Object-Oriented Database Management System, Proceeding of ACM SIGMOD International Conference on Management of Data, 1989, pp.325-336.

14– Chou, H.T. and W. Kim, A unifying Framework for Version Control in a CAD Environment, Proc. International Conference on Very Large Databases, Kyoto, Japan.

15– Copeland, G., and D. Maier, Making Smalltalk a Database System, Proc.ACM SIGACT / SIGMOD International Conference on the Management of Data, 1985.

16– Cox, Brad J., Object-Oriented Programming An Evolutionary Approach, Addison-Wesley, 1986.

17– Date, C.J., An Introduction to Database Systems, Fourth Edition, vol.1 and vol.2, Addison-Wesley, 1986.

18– Date, C.J., An Introduction to Database Systems, Addison-Wesley,Vol.2, 1983.

19– Deppish, U., et al., A Storage System for Complex Objects, Proceedings of the International Workshop on Object-Oriented Database Systems, Sept 23-26, 1986 Pacific Grove, pp.52-65.

20– Derrett N. et al., Some Aspects of Operations in an Object-Oriented Database, Data Engineering, 1985, pp.302-310.

21- Diederich, J., and J. Milton, Experimental Prototyping in Smalltalk, IEEE Software, May 1987, pp.50-64.

22- Diederich, J., ODDESSY: An Object-Oriented Database Design System, Proc. of the Third International Conference on Data Engineering, Feb 3-5 1987 L.A., U.S.A., pp.235-245.

23- Ege, A., Ellis, L.A., Design and Implementation of GORDION, an Object Base Management System, Proc. of the Third International Conference on Database Engineering, Feb 3-5 1987 L.A., U.S.A. pp.226-234.

24- Fishman, D.H., et al., IRIS: An Object-Oriented Database Management System, ACM Transactions on Office Information Systems, vol.5, no.1, January 1987, pp.48- 69.

25- Garza J.F. and Kim W., Transaction Management in Object- Oriented Database System, Proceeding of ACM SIGMOD International Conference on the Management of Data, 1988, pp.37-46.

26- Goldberg, A., and D. Robson, Smalltalk-80: The Language and Its Implementation, Addison Wesley, 1983.

27- Hornick, M.F., and S.B. Zdonik, A Shared, Segmented Memory System for an Object-Oriented Database, ACM Transactions on Office Information Systems, vol.5, no.1, January 1987, pp.70-95.

28- Hughes J.G., Database Technology: A software engineering approach, Prentice Hall, 1988.

29- Jagadish H.V., Incorporating Hierarchy in a Relational Model of Data, Proceeding of ACM SIGMOD International Conference on Management of Data, 1989, pp.78-87.

30- Kaehler, T., and D. Patterson, A Small Taste of Smalltalk, Byte, August 1986, pp.145-159.

31- Karaorman, M., Secondary Storage Management in an Object-Oriented Database Management System, M.S. Thesis, Bilkent University, Ankara, July 1988.

32- Kesim, N., An Object-Oriented Database Management System, M.S. Thesis, Bilkent University, Ankara, July 1988.

33- Khoshafian, S.N., and G.P. Copeland, Object Identity, ACM OOPSLA'86 Proceedings, Sept. 1986.

34- Kim W, Bertino E., Garza J.F., Composite Objects Revised, Proceeding of ACM SIGMOD International Conference on Management of Data, 1989, pp.337-347.

35- Kim , W., H. Chou, and, J. Banerjee, Operations and Implementations of Complex Objects, IEEE Transactions on Software Engineering, Vol. 14, No. 7, July 1988.

36- LyngbaeK, P., and V. Vianu, Mapping a Semantic Database Model to the Relational Model, ACM SIGMOD International Conference on Management of Data, 1987, pp.132-142.

37- Maier, D., and J. Stein, Indexing in an Object-Oriented DBMS, Proc. of the Workshop on Object-Oriented Database Systems, September 1986.

38- Maier, D., A. Otis, and A. Purdy, Object-Oriented Database Development at Servio Logic, Database Engineering, IEEE, vol.8, no.4, December 1985.

39- Maier, D., J. Stein, A. Otis, and A. Purdy, Development of an Object-Oriented DBMS, ACM Conference on Object- Oriented Programming Systems, Languages and Applications, 1986.

40- Maier, D., and, J. Stein, Development and Implementation of an Object-Oriented DBMS, Research Directions in Object-Oriented Programming, Shriver, B., and, P. Wegner Eds, 1987.

41- Nierstrasz O.M., Active Objects in Hybrid, OOPSA'87 Proceedings.

42- Nierstrasz O.M., What is Object in Object-Oriented Programming? Objects and Things, ed. D.Tsichritzis, Centre Universitaire D'Informatique, Universite de Geneve, March 1987, pp.1-13.

43- Ossher, H., A Mechanism for Specifying the Structure of Large, Layered Systems, Research Directions in Object- Oriented Programming, ed. B. Shiver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp.218-251.

44- Ozelci, M.S., Message Passing in an Object-Oriented Database Management System, M.S. Thesis, Bilkent University, Ankara, July 1988.

45- Ozelci, S.M., N. Kesim, M. Karaorman, E. Arkun, An Experimental Object-oriented Database Management System Prototype, Proc.of the Third International Symposium on Computer and Information Sciences, October 1988, Cesme, Turkey.

46- Purdy A., et al., Integrating an Object Server with other Worlds, ACM Transactions on Office Information Systems, Vol.5, No.1, January 1987, pp.27-47.

47- Skarra, A.H., and S.B. Zdonik, Type Evolution in an Object-Oriented Database, Research Directions in Object- Oriented Programming, ed. B. Shiver, and P. Wenger, MIT Press Series in computer Systems, 1987, pp.393-415.

48- Skarra, A.H., and S.B. Zdonik, An Object Server for an Object-Oriented DBMS, Proceedings of the International Workshop on Object-Oriented Database Systems, Sept.23- 26, 1986 Pacific Grove, pp.196-204.

49- Stefik, M., and D.G. Bobrow, Object-Oriented Programming: Themes and Variations, AI Magazine, January 1986, pp.40-62.

50- Stone H.S., High-Performance Computer Architecture, Addison-Wesley Publishing Company, 1987.

51- Stonebraker M., Object Management in POSTGRES Using Procedures, Proc. of the Workshop on Object-Oriented Database Systems, September 1986, pp.66-72.

52- Tanenbaum A.S., Operating Systems: Design and Implementation, Prentice Hall, Inc., 1987.

53- Turkmen, S., Data Definition and Manipulation Languages for an Object-Oriented Database Management System (ODS), M.S. Thesis, Bilkent University, Ankara, July 1989.

54- Turkmen , S., C. Yengul, E. Arkun, An Object-Oriented Database System Prototype, Proceeding of the Fourth International Symposium on Computer and Information Sciences, October 1989, Cesme, Turkey.

55- Ullman, J.D., Principles of Database Systems, Computer Science Press, 1982.

56- Woelk D., et al., Enhancing the Object-Oriented Concepts for Database Support, Proceeding of the Third International Conference on Database Engineering, Feb 3- 5 1987 L.A.; U.S.A.,pp.291-292.

57- Yengul C., A run-Time Environment for an Object-Oriented Database Management System, (ODS), M.S. Thesis, Bilkent University, Ankara, July 1989.

58- Zdonik, S.B., Why Properties are Objects or Some Refinements of 'is-a', ACM/IEEE Joint Computer Conference, 1986, pp.41-47.

59- Zdonik, S.B., Maintaining Consistency in a Database with Changing Types, ACM SIGPLAN Notices 21:10, Oct. 1986, pp.120-127.

60- Zdonik S.B., Object Management Systems for Design Environments, Data Engineering, 1985, pp.259-266.