

**SECONDARY STORAGE MANAGEMENT IN AN
OBJECT-ORIENTED DATABASE
MANAGEMENT SYSTEM**

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

Murat Karaorman

July 1988

QA
76-9
-D3
K143
1988

SECONDARY STORAGE MANAGEMENT IN AN
OBJECT-ORIENTED DATABASE
MANAGEMENT SYSTEM

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

Murat Karaorman

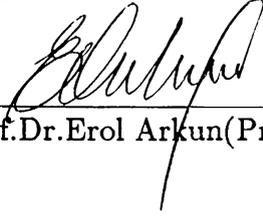
July 1988

Murat Karaorman
tarafından bağlanmıştır.

QA
76.9
.D3
X143
1988

B 1866

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



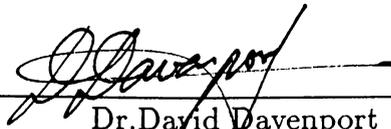
Prof. Dr. Erol Arkan (Principal Advisor)

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



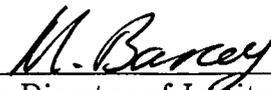
Dr. Altay Güvenir

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



Dr. David Davenport

Approved for the Institute of Engineering and Sciences:



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

ABSTRACT

SECONDARY STORAGE MANAGEMENT IN AN OBJECT-ORIENTED DATABASE MANAGEMENT SYSTEM

Murat Karaorman

M.S. in Computer Engineering and
Information Sciences

Supervisor: Prof.Dr.Erol Arkun

July 1988

In this thesis, a survey on object-orientation and object-oriented database management systems has been carried out and a secondary storage management and indexing module is implemented for an object-oriented database management system prototype developed at Bilkent University.

First, basic concepts, characteristics, and application areas of object-oriented approach are introduced, then, the designed prototype system is presented, the secondary storage management module is explained in detail and the functions of the other modules are summarized. Finally, the current research issues in the object-oriented database systems are introduced.

Keywords: object, class, object-oriented databases, secondary storage, indexing

ÖZET

NESNESEL BİR VERİ TABANI SİSTEMİNDE YARDIMCI BELLEK

Murat Karaorman

Bilgisayar Mühendisliği ve Enformatik Bilimleri Yüksek Lisans

Tez Yöneticisi: Prof.Dr.Erol Arkun

Temmuz 1988

Bu tezde nesnel yaklaşım ve nesnel veri tabanı işletim sistemleri üzerinde araştırma yapılmış ve Bilkent Üniversitesinde geliştirilen bir nesnel veri tabanı sistemi prototipi için yardımcı bellek tasarlanmıştır.

Tezin birinci kısmı yapılan araştırmanın sonuçlarını özetlemektedir. Nesnel yaklaşımın başlıca kavramları, özellikleri ve uygulama alanları anlatılmaktadır. İkinci kısımda, tasarlanan prototip tanıtılmaktadır. Sistemin yardımcı belleği ayrıntılı olarak anlatılmakta, diğer bölümleri özetlenmektedir. Son olarak nesnel veri tabanı sistemlerindeki en son araştırma konuları sunulmaktadır.

Anahtar kelimeler : nesnel veri tabanı sistemleri, nesne, sınıf, yardımcı bellek

ACKNOWLEDGEMENT

I would like to acknowledge first the help and cooperation of my supervisor Professor Erol Arkun without whom this work could not have been completed. I would also like to thank Nihan Kesim and Sibel Özelçi with whom we worked together on the project of developing an object-oriented database management system prototype, which forms the basis of this thesis, for their patient suggestions and comments. I also acknowledge the help of Attila Gürsoy, Özgür Ulusoy, Mesut Göktepe, and Ahmet Coşar in the preparation of this thesis. Dr. Nierstrazs has also been very helpful by his remarks and suggestions.

TABLE OF CONTENTS

1	INTRODUCTION	1
2	SURVEY OF OBJECT-ORIENTED SYSTEMS	4
2.1	Background	4
2.2	Basic Concepts of Object Orientation	5
2.3	Basic Properties of Object-Oriented Systems	7
2.3.1	Data Abstraction	7
2.3.2	Independence and Object Identity	8
2.3.3	Message Passing Paradigm	11
2.3.4	Inheritance	12
2.3.5	Reusability	15
2.3.6	Overloading and Polymorphism	15
2.3.7	Concurrency	16
2.3.8	Homogeneity	17
2.3.9	Dynamic Binding	17
2.3.10	Interactive Interfaces	17
2.4	Object-Oriented Programming Languages	18
2.4.1	Historical Perspective of Object-Oriented Languages	19

2.4.2	Examples of Some Object-Oriented Languages	20
2.5	Object Oriented Databases	20
2.5.1	Object Oriented Databases versus Object Oriented Programming Languages	21
2.5.2	Object Oriented Databases versus Traditional Databases	22
2.5.3	Making Object Oriented Database Systems	24
2.5.4	Existing Object-Oriented Database Management Systems	25
2.5.5	Language Issues on O-O DBMSs	25
2.5.6	Performance Issues in O-O DBMSs	26
2.5.7	Schema Evolution	27
2.5.8	Indexing	28
3	AN EXPERIMENTAL OBJECT-ORIENTED DBMS PROTOTYPE	31
3.1	An Overview of the Prototype	31
3.2	Main Subsystems of the Prototype	32
3.2.1	Object Memory and Schema Evolution	32
3.2.2	Message Passing	35
3.2.3	Secondary Storage Management and Indexing	38
3.2.4	The User Interface	38
4	SECONDARY STORAGE MANAGEMENT AND INDEXING	39
4.1	Statement of the Problem	39
4.1.1	Main Issues of Secondary Storage Management	40

4.2	EXISTING APPROACHES TO SECONDARY STORAGE MANAGEMENT	47
4.2.1	Gemstone	49
4.2.2	IRIS	52
4.2.3	ORION	53
4.2.4	ENCORE	57
4.3	SECONDARY STORAGE MANAGEMENT OF THE PRO- TOTYPE	62
4.3.1	The Goals and Requirements	62
4.3.2	The Secondary Storage Architecture	64
4.3.3	Implementation of the Storage Manager	77
4.4	INDEXING	79
4.4.1	Design Considerations	82
4.4.2	Implementation	85
4.5	Problem Areas and Directions for Future Research	85
5	CONCLUSION	87
A	APPENDIX	89

LIST OF FIGURES

2.1	Inheritance graph with multiple inheritance	14
3.1	The four main modules of the prototype.	32
3.2	The format of an allocated object	33
3.3	The format of a class object	34
3.4	The initial class hierarchy and the system defined classes	34
4.1	Major Pieces of GemStone	50
4.2	Dereferencing process in ENCORE	60
4.3	DBF and Segment Structures	61
4.4	Allocated chunks for a memory object	66
4.5	The abstract view of a variable sized container	68
4.6	The abstract view of a variable sized container with external super-part	68
4.7	Container for TitledNameWithLetters object	70
4.8	Class definitions and allocated chunks for a memory object .	73
4.9	Secondary Storage representation of a memory object	74
4.10	Allocated chunks for a memory object	75
4.11	Secondary storage representation of an Employee object	76

4.12 Save Algorithm	80
4.13 Retrieve Algorithm	81

1. INTRODUCTION

As computers became more available and powerful, the demand and sophistication of the users of these systems has increased with influences to various areas of computing, and computer applications. The demand for more sophistication has in many ways rendered conventional problem solving approaches inefficient and impractical. The areas of database systems, programming languages, and artificial intelligence already had overlaps in many ways. Then, newer applications like Computer Aided Design/Computer Aided Manufacturing (CAD/CAM) and office information systems (OIS) have evolved with demands that can not be handled efficiently by existing approaches. At this point, Object Orientation represents a most successful unifying paradigm in various areas of computing, including Programming Languages, Databases, Knowledge Representation, Computer Aided Design, and Office Information Systems. However, being one of the most fashionable, and overused terms of recent years, there is no clear definition of what Object-Oriented means. In the survey part of this thesis, a focused survey of different approaches will be presented and properties of object-orientation and especially object-oriented database systems will be elaborated.

Informally, an object-oriented database management system can be defined as follows: a system which is based on a data model that allows the representation of an entity, whatever its complexity and structure, by exactly one object of the database. No decomposition into simpler concepts is necessary. As entities may be composed of subentities which are entities themselves, an object-oriented data model must allow recursively composed objects.

Conventional record-oriented database management systems reduce application development time and improve data sharing among applications. However they are subject to the limitations of a finite set of data types and the need to normalize data. In contrast, object-oriented systems offer flexible abstract data-typing facilities and the ability to encapsulate data and

operations with the message metaphor. In addition, they reduce application development efforts. Object-oriented database management systems support more direct modeling and require less encoding compared to other data models and they capture more information semantics [1]. Also, one can easily represent models which can not be represented using normalized relations, thus keeping the semantic gap as small as possible and representing most of the problem semantics in the database itself. Another point is that, object-oriented systems aim at solving the impedance mismatch problem seen in conventional database systems in which there are separate languages for data definition and data manipulation by providing a unified language supporting both functions. Lastly, object-oriented database systems allow nested (non-first normal form or N1NF) relations, can capture the temporal aspect of the data and can handle multiple versions [16].

The object-oriented database management system prototype designed and implemented at Bilkent University consists of four major modules which are object memory and schema evolution; message passing; secondary storage management, indexing and the user interface [18]. Object memory handles the representation, access and manipulation of the objects in the system [31]. The schema evolution module supports some basic modifications to the class hierarchy. The message passing module is built on top of the object memory and schema evolution module and forms the basis for the user interface module [29]. It includes the definition and support of the designed command language and error handling in addition to message passing. It consists of five submodules which are the lexical analyzer, parser, code generator, executor module and query processor. The designed language aims at solving the impedance mismatch problem. The secondary storage management and indexing module handles persistent objects by storing and retrieving them from secondary storage files and the indexing facility provides B-tree structures for efficient execution of value-based queries. The user interface module is object-oriented and supports three types of users, namely, the developer/maintainer, the domain specialist and the end-user.

The prototype has been implemented on Sun workstations running under Berkeley Unix¹ and the C programming language. The system is single-user and all objects are persistent and passive. Simple inheritance is supported resulting in a class lattice in the form of a tree. Authorization, concurrent access to data and versions are not supported.

¹Unix is a trademark of AT&T Bell Laboratories

The thesis has two parts, the first part discusses various aspects of object-orientation and a survey of object-oriented systems and concepts. The second part will give information on the prototype developed at Bilkent University with emphasis on Secondary Storage Management Issues. Some open problems and future extensions to the system are also presented.

2. SURVEY OF OBJECT-ORIENTED SYSTEMS

The term object-oriented has gained tremendous popularity and is used widely for diverse areas from operating systems to user interfaces, from programming languages to databases. However, there is no agreement in literature on what the minimum specifications are to make a system object-oriented. The survey aims at introducing the general concepts and the properties and discussing various approaches to object-orientation.

2.1 Background

The aim of this section is to introduce general terminology and concepts that are used in the rest of the thesis within the context of programming.

Objects represent the entities and concepts from the application domain being modeled. They are unique entities in the environment, with their own identity and existence, and they can be referred to regardless of their attribute values.

All of the action in object-oriented programming comes from sending messages between objects. Message sending is a form of indirect procedure call. Instead of naming a procedure to perform an operation on the object, one sends the object a message.

Objects with similar implementations and interfaces constitute a *class*; and the members of a class constitute its *instances*. Each class of objects is associated with a set of procedure-like operations called *methods*; and methods are performed when objects are sent *messages*. A message is a request for an object to access, modify, or return part of its private part. Objects provide

methods as a part of their definition. Methods describe how to carry out the necessary operations and a message specifies which method is desired but not how that operation is performed. The set of messages to which an object can respond is called its *interface*. When a message is sent to an instance, the method that implements that message is found in the class definition. Methods are not visible from outside the object. Objects communicate with one another through messages. A crucial property of an object is that its private memory can be manipulated only by its own operations and the messages are the only way to invoke an object's operations [17].

2.2 Basic Concepts of Object Orientation

It is generally agreed [32] that object-orientation is an approach, or style rather than a specific set of language constructs, and object-oriented programming is primarily characterized as a "code-packaging" technique rather than a coding technique. In fact one can use an arbitrary programming language and still write in object-oriented style.

One thing common to all object-oriented systems is the concept of object which brings about the related concepts such as classes, hierarchies, message passing, etc., which will be elaborated later in detail.

Yet, the meaning of object also varies. To some, object is merely a new name for abstract data type where data and operations are encapsulated into objects. To others, objects and classes are a concrete form of type theory. To still others, object-oriented systems are a way of organizing and sharing code in a large system [8].

The object concept originally belongs within the paradigm of imperative programming. It is an offspring of the block concept as introduced in Algol 60 and exploited more extensively in Simula [47], a language for programming computer simulations. Algol features procedures and in-line blocks whereas Simula adds the concept of classes. Within the Algol context, a block is a collection of declared 'quantities', typically variables and procedures operating on these variables but possibly also entities of other kinds. Some kind of blocks also contain a behavior pattern describing own actions in an imperative style. An object then is a dynamic instance of a block, possibly a class body block. The variables of the object have values representing its current state and the object behaves through time according to its given capabilities.

The state of an object can change as the result of its own actions, if any, or as the result of local updating procedures invoked from outside the object. It is useful to distinguish between the concepts "object" and "class". The latter is the common description of the potentially unlimited set of objects that might be generated which are said to belong to that same class.

The association of data structures and algorithms inherent in the class and object concepts makes it possible to construct entities meaningful on more abstract levels. In its most basic form we have a module of a program consisting of a number of static variables together with the set of procedures that are used to manipulate the variables. This data abstraction is by far the most important concept in the object-oriented approach [47],[17].

Another important aspect of object orientation follows from the locality of identifiers declared in an object: there is no name conflict with those of other, disjoint objects, even for objects which belong to the same class. Thus if x is an object and f is a function local to it, $x.f$ identifies that function independently of functions that are named elsewhere in the system. Since x in $x.f(..)$ is at the same time an argument to f , the locality principle provides a simple and natural rule of function overloading [47].

Since all objects belonging to the same class contain textually similar declarations, it is sometimes convenient to think of a function as being local to the class rather than the object. With this perspective $x.f(...)$ means $C.f(x,...)$, where C is the class which x belongs to.

The idea of subclasses introduced in Simula provides a convenient means of formulating general concepts which are easy to reuse and to specialize in different directions. It is important that objects belonging to a subclass at the same time belong to its superclasses and, via inheritance, can play roles defined in them. The concept of subtypes is equally useful, in particular to the extent that theorems valid for a type remain valid for its subtypes.

Object orientation implies a technique of system (de)composition: a system is viewed as the collection of objects it contains together with their interrelations and interactions. One often wants better, i.e. more complete, decomposition than that usually obtained with older languages of the Algol Pascal type. Programs written in these languages are essentially structured as textually nested blocks, and the unrestricted access to nonlocal quantities, especially the write access to variables, makes such programs resistant to decomposition. Consequently object-oriented languages prohibits direct

access to nonlocals and textual enclosure in the Algol sense. As an extreme case distributed systems consist of objects which can interact only through communication lines by sending and receiving messages of globally known types [47].

2.3 Basic Properties of Object-Oriented Systems

The main properties of the object-oriented approach can be listed as follows:

1. Data abstraction and encapsulation.
2. Independence (object identity).
3. Message-passing paradigm.
4. Inheritance.
5. Reusability.
6. Overloading and Polymorphism.
7. Concurrency (some systems).
8. Homogeneity.
9. Dynamic Binding
10. Interactive interfaces (with menus, windows and mouse).

2.3.1 Data Abstraction

Abstraction is perhaps the most powerful human tool for managing complexity. It allows one to deal with high-level concepts and understand them, before proceeding to consider details of instances; in certain contexts, it might never be necessary to consider the instances at all. Equally, it allows one to classify instances one has examined according to the perceived similarities.

By far the most important concept in object-oriented approach is data abstraction. Data abstraction in this context means that we are interested in the behavior of an object rather than its representation, which also means that an object packages an entity and the operations that apply to it. A

language has data abstraction when it has a mechanism for bundling together all of the procedures for a data type [8].

Object-oriented languages support abstraction through classes and messages. Classes support data abstraction and concept classification. Messages support procedural abstraction. Classes also support hierarchical classification, which is extremely useful for managing complexity. Classes are arranged in a hierarchy such that each is an abstraction of all its descendants.

Every object has a clearly defined interface which is independent of the object's internal representation. The interface is a collection of operations or "methods" which may be invoked by another object. Furthermore, one may have many instances of an object type (class), and new types can be added without restrictions. A type definition is very much like a module from our understanding of software engineering. In the type definition there are a collection of permanent variables encoding the state of the object, and a set of methods that use and change the state. All that one should know to create a new instance of a type (class) is the interface, that is, the names of the methods and the types of the input and output parameters [27].

One benefit of this approach is the fact that the programmer is free to use higher levels of abstraction as appropriate. (that is, at each level of abstraction one concentrates on that level's functionality, while hiding the lower level details of implementation.) This can be compared with the layering concept of OSI in computer networking where each layer is a level of abstraction. In object oriented design one is encouraged to decompose a programming problem into a collection of cooperating objects of various levels of complexity.

The separation of interface and implementation of a new class renders the classes representation-independent to some extent. This enables the programmer to experiment with different implementations, and increase maintainability of the software due to the global structural visibility [46] induced by this inherent decomposition.

2.3.2 Independence and Object Identity

Identity is that property of an object which distinguishes it from other objects. Most programming languages use variable names to distinguish temporary objects, mixing addressability and identity. Most database systems use identifier keys (i.e. attributes which uniquely identify a tuple) to distinguish

persistent objects, mixing data value and identity. Both of these approaches compromise identity. Object-oriented languages use separate mechanisms to handle these concepts, so that each object maintains a separate and consistent notion of identity regardless of how it is accessed or how it is modeled with descriptive data.

There are two important dimensions involved in the support of identity, namely the representation dimension and the temporal dimension. The representational dimension distinguishes languages based on whether they represent the identity of an object by its value, by a user defined name, or built into the language. The temporal dimension distinguishes languages based on whether they preserve their representation of identity within a single program or transaction, between transactions or between structural reorganizations [11].

Most general-purpose programming languages are designed without the notion of persistent data in mind. For this reason they provide weak support of identity in the temporal dimension. As far as the language is concerned, the data lives only during the execution of the program. Persistent data is handled by the file system which is not part of the language. The structures supported in the virtual address space of the program are not usually supported in the file system.

Database languages are designed to support large and persistent data that models large and persistent real-world systems. These characteristics require strong support of identity in both the representation and temporal dimensions.

The way computational languages and database languages approach to support of identity thus induces different concepts and structures as far as programming is concerned (e.g. lists, arrays, atomic variables versus sets, records) and it could be generalized that the notion of identity in programming languages is typically weaker than that of database languages. This diversion brings about the problem of "impedance mismatch" [CopelandMaier "Making Smalltalk a DB"] because much of the meta information (e.g. structures and operations) in either system is reflected back at the interface rather passing through it. This meta information must be redundantly defined in both languages, and also transformations might be needed when data or operations need to pass through the interface. The impedance mismatch problem has led to the evolution of hybrid languages and ultimately to object-oriented languages which tend to bring solutions with their support of identity.

There are different implementation techniques to provide object identity in programming languages and database languages and some of these techniques are briefly given below

- Identity through physical address.
Achieved by assigning an object a real or virtual address. An example could be a Pascal record implemented through a virtual heap address. They provide minimal location independence, as single objects can not easily be moved within the address space.
- Identity Through Indirection.
The object-oriented-pointer (oop) concept introduced by Smalltalk-80 is an example for this kind of support. An oop is an entry in an object-table and therefore identities are implemented through a level of indirection. This mechanism provides full data independence and stronger location independence.
- Identity Through Structured Identifier.
This mechanism is used in implementing file systems for distributed environments and the identifiers of files (the objects of the system) are structured, where part of the structure contains information related to the location of the object. They provide full data and location independence.
- Identity Through Identifier Keys.
This is the main approach for supporting identity in database management systems by direct implementation of user-supplied identifier keys. Identifier key implementations provide full location independence, but no value independence. They also do not provide structure independence because they are unique only within a single relation and applied only to tuples and not to attributes.
- Identity Through Tuple Identifiers.
They are system generated identifiers which are unique for all tuples within a single relation and have no relationship to physical location, but they are typically used in internal implementation of relational databases (such as System R, INGRES) and do not directly correspond to any conceptual notion of identity. They provide full location independence and value independence but not full structure independence since they are unique only within a single relation and they are applied only to tuples and not to attributes.

- Identity Through Surrogates.

The most powerful technique for supporting identity is through surrogates. Surrogates are system generated globally unique identifiers, completely independent of any physical location. They provide full location independence and value independence, but not full structure independence. If surrogates are associated with every object as in OPAL [15], then they provide full data independence.

Object-Oriented systems have an inherent notion of unique object identifiers for object identity and thus have the capabilities to use surrogates, the most powerful technique for supporting identity. The fact that objects can be referenced regardless of their attribute values is also the basis of "referential integrity" in object-oriented databases.

2.3.3 Message Passing Paradigm

Independence of objects is supported conceptually by using message-passing as a model for object communication. The object-oriented model disallows an object to operate on another object. The only way an object can interact with the outside world is by sending and receiving messages. Consequently, object A invokes a method of object B by sending B the message "please execute this method". How object B interprets the method and what subsequent actions it assumes are the responsibility of object B; it may choose to delay responding, or that it does not wish to handle the request at all and return an exception report. The results or acknowledgments are also sent back using message-passing [32].

The term message-passing has several meanings. The first object-oriented language Simula, had coroutines, which is an asynchronous form of message passing in which the sender saves its state and must be explicitly reawakened by a resume call rather than by an automatic reply from the receiver. Smalltalk and Loops equate message-passing with remote procedure calls, a synchronous form of message-passing in which the sender must wait for a reply from the receiver before continuing. Modules in distributed systems may communicate by rendezvous, which combines remote procedure call with synchronization between the calling and called processes, by asynchronous message-passing, or both.

It should be realized that message-passing is a model for object communication rather than an implementation requirement. During implementation the message-passing can be accomplished by straightforward procedure calls, especially in non-concurrent environments. In concurrent environments with active objects, real message-passing seems to be the natural implementation technique, though. There are also some hybrid approaches that combine procedure calls with message sending.

2.3.4 Inheritance

In object oriented languages inheritance is the concept that is used to define objects that are almost like other objects. Inheritance is, in this sense, the mechanism providing the ability to specialize object types. It allows new classes to be built on top of older, less specialized classes instead of being rewritten from scratch. A specialized type (subclass) inherits the properties of its parent class and then possibly adds more properties, this helps to keep programs shorter and more tightly organized [8].

Grouping objects into classes helps avoid the specification and storage of much redundant information. The concept of a class hierarchy extends this information hiding further. A class hierarchy is a hierarchy of classes in which an edge between a node and a child node represents the IS-A relationship; that is the child node is a specialization of the parent node (and conversely the parent node is a generalization of the child node). For a parent-child pair node on a class hierarchy, the parent is called the superclass of the child, and the child is called the subclass of the parent. The instance variables and methods (collectively called properties) specified for a class are inherited (shared) by all its subclasses, and additional properties may be specified for each of the subclasses. A class needs to inherit properties only from its immediate superclass. Since the latter inherits properties from its own superclass, it follows by induction that a class inherits properties from every class in its superclass chain. The concept of inheritance, like the concept of object-orientation has different connotations in literature, and a way of classifying inheritance mechanism is found in [8]:

- **Type Theory Inheritance.** This is related to the similarity of the data structure between a subclass and a superclass. The structure of a subclass contains all the instance variables of its superclass and may include its own instance variables. For example,

```
labeled-point = ( x-coord : integer;
                  y-coord : integer;
                  label : string )
```

is a subclass of

```
point = ( x-coord : integer ;
          y-coord : integer )
```

because labeled-point has two instance variables of point plus one additional instance variable.

- External Interface Inheritance. This refers to the similarity of the externally visible interface provided by a class and its superclass. The class is able to provide all the external interface of its superclass and may specialize its superclass by providing its own interface as well [23]

```
deque = ( push-right, pop-right, push-left, pop-left )
```

is a subclass of

```
queue = ( push-right, pop-left )
```

even if deque is implemented with an array and queue is implemented with a linked list.

- Code Sharing and Reuse. Here a subclass can use the functions provided by its superclass as if they were defined in the subclass itself. Hence redundancy of some code is eliminated. As a result, more complex programs can be built out of simpler ones. In the previous example with queues and deques, a queue is a subclass of a deque, because queue can be implemented by deque, that is, the queue exports two of the deque's functions and hides the other two. This interpretation of subclass is opposite to the deque example given above.

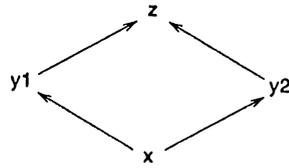


Figure 2.1: Inheritance graph with multiple inheritance

- Polymorphism [23]. In the context of object-oriented languages, association of generic names with behaviours is called overloading of operator names or polymorphism. For example, many objects may respond to Delete messages, each with a method specific to that object but each fulfilling the same role for the object with which it is associated. The advantage of this encapsulation is that the programmer needs keep track of the names of only a (relatively) few behaviors that are exhibited by a set of objects; the names of the larger set of specific procedures that implement the behaviors need not be remembered.

Object-Oriented systems combine some or all of the above kinds of inheritance into one structure, which is usually a tree. According to these structural aspects, inheritance can be viewed as either simple inheritance or multiple inheritance. In simple inheritance, a class may have only one superclass forming a tree structured class hierarchy, while in multiple inheritance, a class may have more than one superclass inheriting the definition and properties of all of its superclasses and forming a lattice structure as the class hierarchy. (Note that the term lattice in this context is used to mean a directed acyclic graph structure, rather than the lattice in lattice algebra). Fig.2.1 shows an inheritance graph with multiple inheritance. In this example, class x is the root class. Class x inherits from classes y1 and y2, and classes y1 and y2 both inherit from class z.

Multiple inheritance simplifies data modeling and often requires fewer classes to be specified than with simple inheritance. However it introduces name conflicts, that is, the problem of two or more classes having instance variables or methods with the same name. The conflict may be between a class and its superclass or between the superclasses of a class. The name conflict problem between a class and its superclass may also be seen in simple inheritance and is solved by giving priorities to the classes. To solve the conflict problems in multiple inheritance, either all instance variables or method names of superclasses must be distinct, or a priority order for the superclasses should be specified.

2.3.5 Reusability

Traditionally encapsulation of procedures, macros and libraries was used to enhance the reusability of software. Currently, object-oriented techniques provide further capabilities for reusability through the encapsulation of programs and data. In this way objects refine the idea of a library or a package.

Reusability can be enhanced in many ways:

1. Instantiation . Multiple objects can be statically or dynamically created from either an object class description or from a prototypical object.
2. Class inheritance. The key idea of class inheritance is to provide a simple powerful mechanism for defining new classes that inherit properties from existing classes. The internal structure (instance variables) and the implementation of operations (methods) may be shared between object classes in this way.
3. Overloading and polymorphism. The realizations of outwardly similar object classes may be transparently altered, thus permitting greater software independence. Polymorphism enhances software reusability by making it possible to implement generic software that will work not only for a range of existing objects but also for objects to be added later.
4. Parameterization. Whereas the mechanism of class inheritance achieves software reusability by factoring out common properties of classes in parent classes, generic classes do so by partially describing a class and parameterizing the unknowns. These parameters are typically the classes of objects that instances of generic classes will manipulate.

For an elaborate discussion of these reusability concepts, see [32].

2.3.6 Overloading and Polymorphism

Another important feature of object orientation is operator overloading. Operator overloading describes the notion of using the same operator symbol to denote distinct operations on different data types (e.g. using minus sign

for both arithmetic subtraction and set difference). The meaning of the operator in this way is overloaded and can be resolved only on the basis of its operand type(s). In interpreting a message, an object-oriented language first binds the message head to an object class, then binds the rest of the message to a method for that class. Overloading follows from the fact that distinct methods can be given the same name in two different classes [32].

The advantages of overloading become apparent if we take, for instance, an application where the printout of different objects, each with their own format, is requested via a print message. Then, new objects, each with their own print method, can simply be appended on with no further need for program modification. Polymorphism may or may not impose run-time overhead depending on whether dynamic binding is permitted by the programming language. If all objects are statically bound to variables, we can determine the methods to be executed at compile-time. In this case polymorphism is just syntactic sugar. On the other hand if variables can be dynamically bound to instances of different object classes, some form of run-time method lookup is necessary.

2.3.7 Concurrency

Programming languages had attacked the concurrency problem using:

- Active entities (processes) communicate indirectly through shared passive objects.
- Active entries communicate directly with one another by message passing.

When the first approach is adopted, shared memory could be structured as a collection of passive objects. Then the process itself can be viewed as an active object also. This approach needs synchronized access to shared objects. One problem with this approach is that it cannot be extended to a distributed environment without employing some form of hidden message passing [32].

With the second approach, any two objects can communicate, and objects become active in response to a communication. Explicit synchronization is not required because message passing packages both communication and synchronization [32].

2.3.8 Homogeneity

Homogeneity in this context comes from the fact that everything is an object. Classes and even messages can be objects themselves and this notion of homogeneity makes for a consistent view of the environment.

2.3.9 Dynamic Binding

Generally, conventional languages perform early binding. For example code is bound to a name at compilation and a name to an address at link time. Late binding provides flexibility at the expense of efficiency in contrast to early binding. Early binding should be applied in a stable environment where the bindings will not change. Late binding is applied in unstable environments.

Operator overloading and generic functions are only suitable if the data is homogeneous and thus the types of the operations can be determined at compile time. Dynamic binding is necessary when dealing with heterogeneous data. The basic approach used in dynamic binding is polymorphism which is similar to operator overloading where the procedure invoked is fixed at compile time. In polymorphism, the same operator performs different operations depending on its operands and the operation is determined at run-time. In object-oriented systems messages support polymorphism and dynamic binding. The same message may elicit a different response depending on the receiver.

2.3.10 Interactive Interfaces

In the most general sense, objects are pieces of compiled code that are manipulated by a particular application to perform a task. It follows that each object has a view for the user to see and if necessary, interact with the object through it. This leads to various window, menu, icon, etc. configurations on the screen that are formatted with respect to user specifications and object requirements [42]. These windows then act as communication media between the user and the application, controlling the object. This input-output technique is indeed independent from object oriented programming and can also be used for multiple tasks running concurrently on a particular machine, or, under window managers that support multiple window environments and detect events for generating standard inputs to applications.

2.4 Object-Oriented Programming Languages

Object-oriented programming is a programming style in which operations are grouped together with structured objects. Descriptions of operations and structure are collected together in classes which share operations and structural descriptions with their superclasses. Object-oriented programming supports the object-oriented paradigm by providing linguistic, semantic, execution, and environmental support. However, clear definitions of these supports have not been made yet and object languages differ even in fundamentals. A classification based on inheritance has been proposed in [48].

A language is called object-based if it provides linguistic support for objects having the following properties:

Object: An object has a set of operations and a state that remembers the effect of the operations. Objects communicate by sending each other messages to perform operations.

Object-oriented programming is sometimes defined broadly so that any language or style of programming in which objects have a state and applicable operations is said to be object-oriented.

An alternative way of defining the notion "object-oriented" which more directly emphasizes software methodology is by the form of their modules and module management mechanisms [48] :

1. The modular building blocks include:
 - objects with operations and a state that persists between calls on operations;
 - classes which specify the interface of collections of objects with common behavior.
2. Module management is facilitated by the fact that:
 - objects are first-class values that can be managed by computation within the language.
 - Inheritance allows classes to be specified in a modular, incremental fashion.

There is yet another issue of whether message-passing or class inheritance characterizes object-oriented languages. Since object-oriented programming

models computing at the level of message exchanging among a collection of objects, rather than at the level of execution of expressions and statements, message-passing appears to be the characterizing feature. Object-oriented systems emphasize communication among objects rather than sequential statement execution, and messages are the basic mechanism for communication. However, any form of message-passing appears to be compatible with object-oriented programming and the precise nature of the communication mechanism is not central to the definition of object-oriented programming. On the other hand, the requirement that objects have classes with inheritance is explicit and definitive. Consequently, object-oriented programming is prescriptive in its methodology for classifying objects but is permissive in its methodology for communication [48].

2.4.1 Historical Perspective of Object-Oriented Languages

SIMULA has been the language which brought about most of the ideas of object-oriented programming. Then, the first substantial interactive, display-based implementation was the Smalltalk language [7], which is responsible for the visibility of the object-oriented paradigm in programming languages. Although Smalltalk has found limited commercial use due to its lack of speed, it has inspired the emergence of other object-oriented languages each of which potentially introduced new concepts and approaches to object-orientation. There are lisp-based extensions to Smalltalk such as Flavors, or Loops, which have gained acceptance. Similar extensions proposed for languages such as Prolog, or functional languages, reemphasize the flexibility and portability of the approach. Other systems, such as Actors, or Concurrent Prolog are based on the concept of processes communicating through messages. Some languages strive to add object-oriented tools to existing programming languages, such as, Objective C, C++.

Object-oriented programming can be considered either revolutionary or evolutionary, depending on the degree to which access to conventional programming techniques is retained [27]. Pure object-oriented languages such as Smalltalk-80 represent the revolutionary approach and provide the advantage of conceptual simplicity; the break between the past is clean and crisp. The evolutionary approach adds object-oriented concepts on top of conventional languages. Languages such as Objective-C, C++, Flavors and the like do not offer the conceptual consistency of Smalltalk-80 but their advantage

is the fact that they can often be used for production programming, where pure languages like Smalltalk are usually unacceptable.

2.4.2 Examples of Some Object-Oriented Languages

There are many object-oriented programming languages but they are not very distributed mainly due to performance reasons. Some are based on the existing languages such as Loops, Flavors which are based on Lisp, Objective-C and C++, while some are designed as a completely new language such as Smalltalk and Hybrid. Among these Smalltalk is the most well known and has influenced the prototype a lot.

2.5 Object Oriented Databases

A database system is a collection of stored data together with their description (the database) and a hardware/software system for reliable and secure management, modification and retrieval. In conventional approaches it is usually impossible to represent all interesting semantics within a database. The remainder has to be captured by the application programs using the database and this is referred to as the semantic gap within the database management system [5].

Object oriented databases are based on a data model that allows an entity in the environment to be modeled by exactly one object of the database. The objects are unique entities in the database, with their own identity and existence, and can be referred to regardless of their attribute values. This concept of object identity inherently supports the referential integrity [5]. This is a major advantage over record-based data models in which objects, represented as records can be referred to only in terms of their attribute values.

Objects are described by their behavior and can only be accessed and manipulated in terms of predefined operations relevant to the class that the object belongs to. As long as the semantics of the operations remains the same the database can be both physically and logically reorganized without affecting the existing application programs. This provides a very high degree of data abstraction and data independence [6].

2.5.1 Object Oriented Databases versus Object Oriented Programming Languages

Object-oriented systems first evolved as programming language systems, and as such, their data models completely ignore many important database issues, such as deletions of persistent objects, dynamic changes to the database schema, and predicate-based query capabilities [2]. Although they enforce the object-oriented paradigm on live computational objects, they neglect to enforce it on the long-term storage representation of those objects. The way they treat persistence is by storing a program that consists of logically distinct objects, is by merging the representations of those objects into single string for long-term file storage. When retrieving the file's contents they parse the string and reconstruct the objects it describes. This means that they typically do file input and output during the start and end of a session and the intermediate states of the database are transient.

Object oriented programming language systems also lack concepts that are important to applications, such as composite objects and aggregate objects for defining and manipulating complex collections of related objects. Further, They do not include version control, which most application systems in the CAD/CAM and OIS domains require. Consequently, it may be said that object oriented databases differ from their programming language counterparts in the following fundamental ways [34].

- persistence.
- unique naming.
- sharing.
- transactions.
- versions.

Objects that are created by a process persist beyond the lifetime of that process. The database system assigns all objects a unique identifier that is guaranteed to remain unique even across multiple processes. Any number of applications can share the objects that reside in the persistent memory space. In the process of using these objects a given process can define the boundaries of transactions that are guaranteed to be atomic and resilient and that preserve some set of correctness criteria.

2.5.2 Object Oriented Databases versus Traditional Databases

Object-oriented database management systems , extending the concepts of their underlying object-oriented programming environments, are powerful and semantically rich tools when compared with their counterparts from existing commercial systems. This is mainly due to the fact that object-orientation provides many important concepts such as data abstraction and encapsulation, inheritance and in general conceptual simplicity in approaching and realizing a complex software project. Some of the shortcomings of commercial database systems could be given as in the following paragraphs.

Most existing database systems supply only a fixed set of data types-integer, real, string, etc., and maybe a few specialized data types such as date or money. However, they do not provide any facilities to define new types or define operations on existing types. The abstract data types can only be virtually implemented by going outside the database system to an application programming language [16].

Database systems often impose artificial limitations on the modeled environment, which are not easily evolvable without substantial program and structure modification. Some examples are setting limits on field lengths, number of fields in a record, etc. which are due to the implementation artifacts creeping into the data model [6].

Data structuring capabilities of current database systems have been optimized to support flat structures, and the possible complexities and variations that occur in real data can not be supported efficiently. Records of a given type must be identical in structure, and changing the structure often requires the reorganization of existing database.

Whenever data structures in a database system can not support the actual structure of information in the real-world, then the form of the real-world information gets over-simplified in the database scheme, or it must be encoded into available data structures. If the structure of the real-world is over-simplified, the utility and reliability of the data is compromised. When information is encoded, such as flattening a set valued field into several tuples, application programs must deal with the encoding.

An important point where object-oriented approach and traditional approaches differ is *Data Dictionary* concept. In a conventional database management system, the data dictionary/directory is used to control access to the

database, ensure data integrity and supervise the distribution of data. In the past, the data dictionary was a collection of static record structures designed and built after a study of the problem to be modeled. It was fixed throughout the life of database applications. Dictionaries were viewed as static tools for the control of data and information resources [24].

Especially for CAD/CAM and knowledge representation applications, dictionaries are required to be dynamic and active in the design and management of databases. Database design, dictionary definition and data acquisition must be integrated. This brings two features for the dictionary:

- the need for more dynamic structures capable of evolving over time and with changing requirements
- a closer integration between data and metadata

Traditional database management systems make a clear distinction between data (the database) and the metadata (the data dictionary/directory). To make full use of the knowledge, database and data dictionary functions must be integrated. This idea will be developed into expert database systems or knowledge base systems. Expert database systems support data, knowledge and application programming within one integrated framework [16].

The purpose of the data dictionary is to enforce the structure of new data instances and keep track of existing ones. There are some problems with current dictionary organizations. One deficiency is the lack of an active or dynamic schema, that is, a data dictionary that can be referenced, accessed and modified during database processing. The need for a dynamic schema is motivated by the following characteristics of a domain:

- the structure of the data is defined as the data is generated,
- the structure of the data is not uniform across data objects,
- there exist many differences of data with many different formats,

The desired functionality includes schema viewing, schema modification and consistency checking among schema items. For these reasons existing data dictionary facilities are not sufficient.

An object-oriented dictionary facility uses an object-oriented organization to represent and describe a data dictionary schema. Objects are used to

represent classes and instances of schema structures. All schema related operations are implemented as methods [24].

Dictionary facilities have been static since building and managing a database schema requires an enormous bookkeeping effort to maintain consistency. By building a schema description as an object-oriented hierarchy, a data structure management facility to serve as an assistant for automatically describing data representations and transparently maintaining them is provided. Schema descriptions are maintained as object properties and procedures for adding, modifying or deleting dictionary objects are represented as methods associated with the schema object. These procedures maintain the consistency of the schema and database objects when schema modifications are made.

Conventional systems pose problems when working in the temporal dimension. Although historical access is common in manual systems, it is usually not provided in automated database systems. Temporal extensions of data models have been researched and are still being researched, but no elegant solutions have apparently come into commercial use yet [24].

Another major problem in the database world is that, data manipulation languages are not computationally complete which in turn necessitates an interface to a general purpose programming language. Thus, one language must be embedded in the other. This problem is referred to as the impedance mismatch problem [4]. Impedance mismatch implies redundancy in data modeling issues and an implementation dependent interface between the languages, which might in the most extreme case even destroy identity.

Finally, object-oriented database management approach strives to bring solutions to these problems by the way they facilitate extensible typing mechanisms, the way they model the world, by providing entity identity using surrogates, easily incorporating version mechanisms, and by removing the impedance mismatch problem implicitly.

2.5.3 Making Object Oriented Database Systems

It has been proposed that [16] a combination of object oriented language capabilities with the storage management functions of a traditional data management system would result in a system that offers reductions in application development efforts. Also the extensible data-typing facility of the system

would facilitate storing information not suited to normalized relations, and that an object-oriented language can be complete enough to handle database design, database access, and applications. There have been many approaches to building object-oriented database management systems in the past few years [12], [4], [6], [15], [38] that realize the fundamental aspects of object-orientation and brought up many interesting questions and research directions which will be further elaborated and discussed later in this thesis, as well as their impact on the design of our own prototype database management system.

The power of an object-oriented DBMS lies in the data modeling concepts realized in the implementation. The data model should support the actual structure of information in the real world in an easily comprehensible and efficient manner.

2.5.4 Existing Object-Oriented Database Management Systems

A lot of research has been done on object-oriented database management systems and currently, there are several prototypes of which GemStone, Iris and Orion are the most well known. Iris is implemented on top a relational database system and maps object-oriented concepts to relations and tuples. Orion is designed to support multiple inheritance, composite objects, schema evolution and version management. GemStone has recently become commercial. It is implemented on top of Smalltalk. It supports simple inheritance and provides an indexing mechanism which they are currently trying to improve. The prototype developed as part of this thesis has been greatly influenced by GemStone.

2.5.5 Language Issues on O-O DBMSs

It is an important issue to define the language for handling the database management tasks. Some of the approaches are presented below:

The first approach is to use a special purpose database language for specifying operations. TAXIS system [37] uses this approach. An operation written in the database language is compiled into some internal form, stored in the database and later interpreted. It has the advantage that the language

can be tailored to the DBMS, but has the disadvantage that a new language should be designed and implemented and there will be two separate languages with their own constructs.

The second approach is to use an existing language and its implementation for defining and implementing database operations. Advantages are obvious in that no effort is needed to learn or implement a brand new language. However the difficulty lies in the fact that it may not be possible to find such a language that provides constructs to reference, and manipulate sets of data in a database. This leads to the actual design of ones own language having the necessary computational and database constructs. Indeed this has been our own preference in the design of the prototype database management system at Bilkent.

A third approach is to use a subset of an existing programming language, but to write a compiler which compiles operation bodies written in this subset, into a form which can be interpreted by a database management system. Indeed this is like the first approach, except that instead of designing a new language an existing one is used.

2.5.6 Performance Issues in O-O DBMSs

Performance of an object-oriented database system is a complex issue, since it is pretty much dependent on the nature of the environment being modeled. Typical business applications where structures and processes are clear and well defined may respond to conventional database approaches better than object-oriented ones. Yet, this is indeed not very surprising, because the unrivaled research efforts, and technology investments that typically address this type of applications ever since the emergence of database management systems have resulted in almost ideal performance results. Thus we must actually consider non-standard applications, that are not easy to model with traditional approaches, as our target domain, to be able to speak of high performance object-oriented DBMSs. It may be hoped too that object-oriented DBMSs will achieve better performance ratings as the studies on better secondary storage management techniques, query processing, and associative access techniques continue [24].

In many non-standard applications (e.g., CAD systems) conventional DBMSs fall short of providing satisfactory results. This is due to :

- In conventional DBMSs, accessing arbitrary, single fields induces a lot of overhead.
- Use of direct pointers are not supported.(indirections by key values)
- Classical query optimization techniques do not necessarily fit into these environments.

The reasons object-oriented DBMSs have better performance in such applications are:

- Arbitrary connectivities between objects are supported and the database has an execution model which models behavior.
- Objects can be accessed directly (by identity), and local address mappings and caching can be used to achieve high performance.
- Complex entities can be represented more directly, with less encoding.

Object oriented database management systems, thus, not only meet performance needs, but also increase functionality. Better version and configuration management can be provided and more behavioral semantics of design entities can be incorporated into the database.

2.5.7 Schema Evolution

In order for object-oriented systems to become vehicles for rapid prototyping, ease of maintenance, and ease of modification, a well-defined and consistent methodology must be developed. Another consideration in designing a class modification methodology is how to bring existing objects in line with the new definition. One approach suggests screening, to defer modifying the persistent store; filter or correct values before they are used. Another approach does a reorganization after a schema update.

1. changes to the contents of a node (a class)
 - (a) changes to an instance variable
 - (b) changes to a method
2. changes to an edge

- (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. changes to a node

- (a) Add a new class
- (b) Delete a class
- (c) Change the name of a class

2.5.8 Indexing

Indexing is a technique used in database management systems to provide alternate access paths to objects, when the existing access strategies would involve a search over a large volume of data. Object oriented database management systems, too, need indexing when they are used in certain data intensive application domains. The nonnormalized nature of objects introduce some difficulties and also accessing an object by its value is somewhat contradictory to the philosophy of object-oriented identity notion, for this is the reason why indexing is of little published research area and all existing publication comes from the design of commercial products like GemStone [14].

2.5.8.1 Language issues

There are two basic considerations: when to invoke auxiliary access paths for associative searching and whether to index on an object's structure or protocol. One approach is to provide a special class for handling indexes. This approach reduces physical data independence and the user has to perform index maintenance. Another approach is to consider every expression as a candidate for indexed access. A better approach is to denote certain statements as candidates for indexed access or to have a sublanguage to make use of indexes. Adding an index handling sublanguage to an existing language may cause an impedance mismatch problem and will complicate the compiler. The sublanguage may be procedural or declarative. The other major issue regarding languages is whether indexes are based on the instance variables, that is the structure of the objects or the responses to messages, that

is the protocol. Indexing on structure violates the privacy of an object while indexing on protocol introduces problems when the protocol changes.

2.5.8.2 Index structure

Indexing could be provided only on the immediate instance variables of an object or on the instance variables and their instance variables. If an index is provided on paths with multiple links that is multiple instance variables, a single index could be provided for the whole path or several indexes could be provided, one for each link. The sequence of links is called a path expression. With a single index for each path, there are fewer indexes to maintain and fewer indirections to be made during associative access. Indexing by links allows sharing of indexes. Some other considerations are

- The type of the objects to be indexed. Indexing is generally applied to collection or set objects. The objects constituting the elements of the collection or set to be indexed should be of a certain type. They could be required to be an instance of a class. An alternative to using a class as a type is the use of kinds. A kind is a class and all its subclasses.
- Manipulation of undefined values along the index path
- Supporting identity indexes or equality indexes. An identity index supports searching a collection on the identity of some subobject without reference to an object's internal state. It does not support range queries. An equality index supports look-up on the basis of the value or internal state of objects and range queries. In a path expression, all links except the last one must be identity indexes and the last one could be an identity or equality index.
- The comparison operators supported during range indexes
- Indexing on classes or collections. Indexing on classes presents some authorization problems and also applications which do not use the index are subject to the index-related overhead for indexed instances they use. However, it is easier to trace changes to an object which affect the index on that class. Each subclass may maintain its own index or the index on a class may include its subclasses. As an object may be a member of several collections, if class indexes are supported and queries against collections are made, there will be a test for collection membership in addition to the index access. Indexing on collections allows the

possibility that instances of subclasses be included in a collection that is indexed. A collection of all instances of a class may be created and indexed to implement indexing on classes. A third approach which is the combination of the other two approaches , maintains a single index per class but only adds members of a certain collection to that class.

3. AN EXPERIMENTAL OBJECT-ORIENTED DBMS PROTOTYPE

3.1 An Overview of the Prototype

There has been on-going research on object-oriented databases at Bilkent University during the past year, and an experimental prototype has been developed to gain a real feeling for the object-oriented paradigm with the special connotations it makes to databases and to gain an insight on the implementation issues. One of the design goals was to build the core of a full object-oriented database management system that future researchers can build upon their own enhancements and extensions, as well as to be able to set forth research directions for future study at Bilkent University [18].

A computationally complete language has been designed and implemented. This language covers the data definition, data manipulation, and computation aspects of the prototype. One of the design goals was to provide a unified language performing all the database and programming tasks and solving the impedance mismatch problem [18] [31] [29].

The designed object-oriented database management system prototype consists of four major modules which are object memory and schema evolution; message passing ; secondary storage management, indexing and the user interface. The user interface is the highest level module. It is built on top of the message passing module which is in turn built on the object memory and schema evolution module. At the lowest level is the secondary storage management module. The way the modules interact is given in figure 3.1.

The developed prototype is a single-user system and thus does not support concurrent access to objects and authorization control. It supports basic

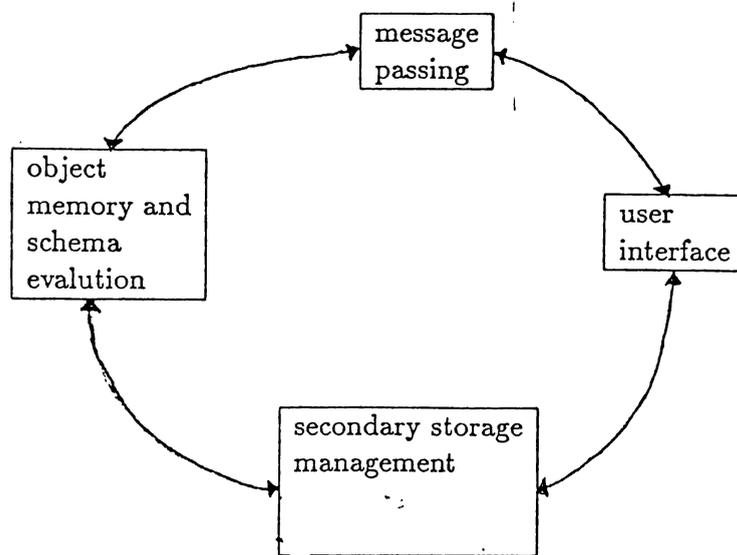


Figure 3.1: The four main modules of the prototype.

object-oriented concepts such as classes, inheritance, message passing and class hierarchy and object identity. The system provides simple inheritance in which each class may have a single superclass and the class hierarchy is in the form of a tree and has its own command language which includes both data definition and data manipulation statements. Type theory inheritance, external interface inheritance, code sharing and reusability are supported but polymorphism is not supported since generic operations are not allowed.

This chapter explains the four modules of the prototype in their most general aspects, and the next chapter presents the secondary storage management module.

3.2 Main Subsystems of the Prototype

3.2.1 Object Memory and Schema Evolution

Object memory handles the representation, access and manipulation of the objects in the system [31]. Each object is associated with a unique surrogate called an *object-oriented pointer (oop)*. Object-oriented pointers are used to

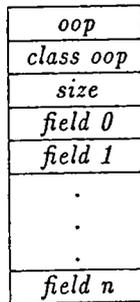


Figure 3.2: The format of an allocated object

identify objects independently of their values. The message passing module and the object memory communicate about objects using object-oriented pointers. An oop is a 32 bit positive even number allowing approximately 2^{30} objects to be referenced. Object memory supports primitive type objects, string objects, class objects, collection objects and instance objects. The primitive type objects are integers and characters. To provide efficiency, the values of the primitive type objects are encoded in their oops.

Object memory uses an object table which maps the oops of the objects to their physical locations in the memory. All references to an object are made through the object table. Thus, the oops of the objects are in fact indices into the object table. This indirection provides the benefit of moving the objects easily in the memory. Object memory is implemented as a hash table in which oops are used to provide direct access.

Objects are represented as contiguous series of words. Each word is used to store the value of an instance variable. The actual data of the object are preceded by a header information which includes the oop of the object, the oop of the class to which the object belongs and the size of the allocated space for the object. The format of an allocated object is shown in Figure 3.2. The fields of an object are accessed by zero-relative integer indices.

Classes are themselves objects. The representation of a class object is different from the representation of an instance object. It contains information necessary to construct and use its instances. This information includes the name and oop of the class, oop of its superclass, the number of its instances, the names and definitions of its instance variables, the names of its

<i>class oop</i>
<i>class name</i>
<i>super oop</i>
<i>instance count</i>
<i>instance variable count</i>
<i>ptr to variable definitions</i>
<i>ptr to method definitions</i>
<i>ptr to instance variable domains</i>
<i>ptr to the first instance</i>
<i>ptr to the place in the hierarchy tree</i>

Figure 3.3: The format of a class object

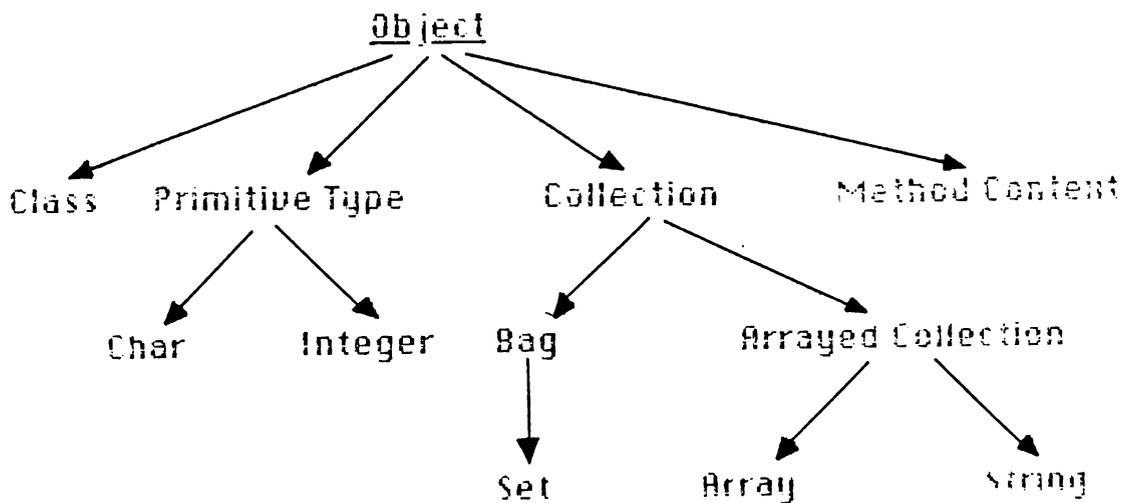


Figure 3.4: The initial class hierarchy and the system defined classes

messages and methods, the domains of the instance variables, and pointers to its instances. The format of a class object is shown in Figure 3.3.

Classes form a hierarchy, that is, each class has only one superclass except for the root class which is the *class Object*. The hierarchy is implemented as a tree. There are five basic system defined classes as shown in Figure 3.4. These are the class *Object*, the class *CLASS*, the *Collection* class, the class of *Primitive Type* and *Method Context* class. The class *Object* is the root of the hierarchy. The user defined classes are instances of the class *Class* and they are inserted into the hierarchy when they are created. The information stored in the nodes of the tree includes the oop and name of the class, a pointer to its superclass, a pointer to its subclass list and a pointer to the

next sibling in the subclass list of its superclass.

When a new instance of a class is created, a chunk of memory is allocated. This new instance will also be the instance of the superclasses in the hierarchy. Since every class has its own private representation, a separate chunk is allocated for each class in the superclass chain.

The object memory provides the following fundamental functions:

- Determine an object's size, class and implementation
- Access and change the value of an object's instance variable
- Access a class object
- Create a new object

One of the important requirements of database applications is the schema evolution, that is the ability to change the database schema dynamically. In object-oriented databases, there can be changes to the class definitions or to the structure of the class hierarchy. The types of changes include creation and deletion of a class, alteration of inheritance between classes, addition and deletion of instance variables and methods. In the proposed system, only a few of these changes are supported such as adding or deleting a class which is a leaf node in the class hierarchy, adding or deleting instances of a class and adding or deleting an instance variable.

3.2.2 Message Passing

The message passing module is built on top of the object memory and schema evolution module and forms the basis for the user interface module. It includes the definition and support of the designed command language and error handling in addition to message passing. It consists of five submodules which are the lexical analyzer, parser, code generator, executor module and the query processor [29].

3.2.2.1 The Command Language

The command language of the object-oriented database management system prototype is designed to provide unification for both the data definition

and data manipulation language aspects to solve the impedance mismatch problem. The language can be used both interactively, that is, command by command or in the batch mode, that is, in the form of methods.

The commands can be classified into two major groups: interactive mode statements and batch mode statements. The interactive mode statements can be further classified as follows:

1. Definition statements- These are for defining a new class, method or instance
2. Schema evolution statements- These are for modifying the class hierarchy, and class definitions.
3. Query statements- These are for accessing and manipulating objects. They include statements for retrieving instances and class information, index manipulation, object duplication, equality checks and method manipulation.

The batch mode statements may only be used in methods and provide iteration, conditional execution, declarations, assignments and message calls. There are two types of message calls. These are the system calls which are implemented as C function calls and actual message calls which are executed by the executor module.

3.2.2.2 Method Handling and Message Passing

A method is used to access and manipulate objects and is invoked using the corresponding message. A method is created using a method definition statement and is formed of a header and a body. The header contains the method name, the corresponding message name, the name of the class to which the method belongs and a list of optional or mandatory arguments of any system defined type or of any class. The method body is formed of a group of batch mode or interactive mode statements. The method and message name may be the same. All methods are persistent and the code for a method and its compiled form are kept in separate data files.

Methods are accessed through a method definition table. Each class object has its own method definition table. Each entry of the table corresponds to a method defined for the class and contains the following information:

- the method name
- the message name corresponding to the method
- the number of arguments
- a pointer to the list of arguments
- the name of the file that contains the method

The lexical analyzer, parser and code generator form the compiler for the command language. Every time a new method is created or a method is modified and a compile method statement is executed or each time a message is invoked and the compiled form of the corresponding method is not available, these subroutines are invoked. At the end of the code generation phase, the interactive statement or the method is converted into a set of integer codes and stored in a file. The executor module takes the generated integer codes as input and performs the corresponding operations using a structure called an *activation record*. During the execution phase, the interactive statements are considered as methods with no arguments for the class Object.

Each message returns a fixed size and fixed structure block. This block contains an error flag, a flag indicating whether a value is returned or not, returned value type, the address of the memory location containing the returned value and for indexed return values the maximum length and the element type.

Each occurrence of a literal in a method is converted into an index for the reference or symbol table. Each activation record has its own program counter, accumulator, condition register, symbol table and reference table. There is a global expression evaluation stack used by all methods.

Activation records are created whenever a message call is executed. The previous activation record is pushed on to the *activation stack*. Whenever a return from a message invocation is performed, an entry is popped from the stack and it becomes the current activation record. This solves the parameter passing and the return address handling problems.

The query processor handles various associative retrieval queries using the routines provided by the object memory and the indexing modules.

Error handling is performed at all stages. Each time an error occurs, an error code is generated and the corresponding error message is retrieved from the system error file and displayed or written to a file.

- the method name
- the message name corresponding to the method
- the number of arguments
- a pointer to the list of arguments
- the name of the file that contains the method

The lexical analyzer, parser and code generator form the compiler for the command language. Every time a new method is created or a method is modified and a compile method statement is executed or each time a message is invoked and the compiled form of the corresponding method is not available, these subroutines are invoked. At the end of the code generation phase, the interactive statement or the method is converted into a set of integer codes and stored in a file. The executor module takes the generated integer codes as input and performs the corresponding operations using a structure called an *activation record*. During the execution phase, the interactive statements are considered as methods with no arguments for the class Object.

Each message returns a fixed size and fixed structure block. This block contains an error flag, a flag indicating whether a value is returned or not, returned value type, the address of the memory location containing the returned value and for indexed return values the maximum length and the element type.

Each occurrence of a literal in a method is converted into an index for the reference or symbol table. Each activation record has its own program counter, accumulator, condition register, symbol table and reference table. There is a global expression evaluation stack used by all methods.

Activation records are created whenever a message call is executed. The previous activation record is pushed on to the *activation stack*. Whenever a return from a message invocation is performed, an entry is popped from the stack and it becomes the current activation record. This solves the parameter passing and the return address handling problems.

The query processor handles various associative retrieval queries using the routines provided by the object memory and the indexing modules.

Error handling is performed at all stages. Each time an error occurs, an error code is generated and the corresponding error message is retrieved from the system error file and displayed or written to a file.

3.2.3 Secondary Storage Management and Indexing

Secondary storage management and indexing module is responsible for the efficient storage and retrieval of objects in the secondary storage. This module was implemented as part of this thesis is covered in detail in the next chapter.

3.2.4 The User Interface

The User Interface of the designed prototype is also object-oriented and the user is navigated by a pop-up menu driven system to the operations he/she desires to perform. The User Interface provides three different environments corresponding to three groups of users: (i) developer/maintainer , (ii) domain specialist, (iii) end-user .

The first environment contains the tools for doing schema changes such as defining new classes, instance variables, updating existing ones, editing methods and customized applications in the prototype's command language.

The second environment contains tools for creating, updating new instances of classes , invoking methods of objects, and doing operational maintenance.

The third environment is for running only customized applications and thus interacting with the database in a controlled manner.

4. SECONDARY STORAGE MANAGEMENT AND INDEXING

Efficient storage and retrieval of objects in secondary storage constitutes an important and integral part of the prototype implementation. This chapter is organized as follows. The first section describes the major issues of secondary storage management in object-oriented databases and states the associated problems. The second section describes existing approaches to secondary storage management and to the stated problems. The third section describes the actual design of the secondary storage management module of the prototype and its relationship with the other modules. Implementation details and some examples are also given in this section. The fourth section describes the *indexing* problem in object-oriented databases, existing approaches to its solution and the design and implementation of the indexing subsystem of the prototype.

4.1 Statement of the Problem

A database management system must be capable of handling large amounts of data that is desired to be processed by the application systems using that environment. Dealing with large amounts of data usually involves storing the information on on-line, direct access secondary storage devices such as discs, and making the information available to the application system by managing the transfer of data between main memory and secondary storage devices. There are several issues involved in the process of accessing and updating the permanent store of data that resides on disk to reflect the most recent state of the database . This section will describe the main issues of secondary storage management in object-oriented databases, their relationship with conventional data models, and finally state the functional requirements of the secondary storage management module of the prototype.

4.1.1 Main Issues of Secondary Storage Management

Providing support for secondary storage of data is a key issue for any database management system and the recently emerging commercial object-oriented database management system products, have introduced new problems and issues related with secondary storage and permanence. Prior to elaborating on the main issues of secondary storage management, some characteristics of objects to be stored and manipulated are presented below, to form a basis for their problem implications that will be treated in the following subsections.

- Objects do not have uniform structures. For example, different instances of a class may refer to an arbitrary number of other objects, or the instances may override the inherited instance variable structure.
- Objects can be of varying sizes, which is closely related with the non-uniformity of structure.
- Objects are organized in an inheritance tree or lattice structure through classes and subclasses.
- Objects can be grouped together to form composite objects, so that they can be referred to collectively rather than on an individual basis.
- Instance variables of objects can themselves be objects or collection of objects.

4.1.1.1 Data Modeling Related Issues

Conventional database management systems are based on several data models each with a separate mechanism for organization and manipulation and these database management systems are limited by the structural limitations and the modeling power of the underlying data model as a major drawback. Object-oriented data model has received appraisal by presenting better modeling power and relatively few structural limitations, yet the secondary storage implications of this data model involve many problems especially because of the need to support abundant pointer usage, extensible typing and schema evolution.

The specific database influences on an object model may be classified into three main groups [3] :

1. Requirements arising from the long term persistence and sharing of objects, e.g., evolutionary change to the definitions of objects, types, and type graphs; version management; concurrency and access control; and transaction management and recovery;
2. The need for general treatment, not merely of isolated objects, but of large collections of objects and relationships between them, with powerful and efficient retrieval and update facilities, including view mappings;
3. The desirability of increasing the semantic content of databases, so that more information can reside in usable form in databases rather than being scattered unintelligibly among programs that access them, and so that databases can grow into knowledge bases and support inferential retrieval without requiring another revolution.

The first two of these groups are for transferring of existing database ideas to the object world, together with some enhancements. The third group however stresses the need to add semantics to databases, which was long before realized but not easily incorporated with conventional database approaches and was associated more with semantic data models [35] and semantic networks [36].

A great amount of research effort combined with the investments in technology has established high standards for access, maintenance, sharing, privacy, security and other related database activities for commercial database management systems. However, activities being modeled have been assumed to have uniform structure, such as the tuples of a relation, instances of a segment, or record types. This uniformity is maintained to some extent in the implementations where variable length records are supported (through repeating groups or arbitrary number of segment instances). Neither the class-objects nor the instances of these classes show this structural uniformity because of occasional overriding of the inherited properties and structures. Therefore the conventional data models are not suitable for representing objects [39], and this means that all the classical database problems needs a retreatment with object-oriented database management systems and additionally, general solutions to those problems can not be found before negotiations on different approaches within object-oriented data model are made. Since the secondary storage management is directly related with the data model, every object-oriented database management system should build its own storage manager.

4.1.1.2 Clustering

Clustering is essentially an efficiency and performance related manner of secondary storage use which is not unique to object-oriented database management systems. Clustering involves grouping entities that have some important common properties into a cluster, associating those properties with the cluster itself, then regarding the cluster as an atomic entity wherever possible [46]. The aim of any clustering scheme is to organize semantically related data together, which results in reduced diskhead movement and reduced physical I/O. For object-oriented database management systems clustering is especially important because objects are *multi-dimensional* instead of being flat. One dimension is the immediate simple type instance variables of an object instance, another dimension is the instance variables and properties inherited along the inheritance hierarchy and yet another dimension is the fact that any instance variable, private or inherited, can be another object (or a reference to it). Algorithms used for manipulating multi-dimensional data in the 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 must be intelligent about staging objects between disk and memory. 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.

Some possible ways that clustering can be made are the following:

- *One chunk¹ per container* can be used for very large objects, since they tend to be accessed individually and are costly to transfer.
- *Storing the chunks of an object together* for objects that are accessed together will enhance performance since objects will get preloaded.
- *Storing individual chunks of a class together* can be useful for aggregate queries on all instances of a class when iterations on class instances are frequent.
- *Storing objects of a collection together* can significantly enhance speed because of the contiguity of these semantically related entities.

¹The private memory of an instance object is a contiguous series of words which is called a chunk.

To illustrate some of the conceptual issues that it is not an easy problem to find an efficient clustering scheme, consider the following example [24]:

```
Class : Employee
      instance variables : Number : integer
                          Name   : object;
                          Dept   : object;
                          Salary : integer;
```

Note that Name (with fields First, Last) and Dept are compound objects. There are two immediate ways of storing employee objects. One is to decompose them into their fields, and represent each field as a binary relation. Thus we would have one relation storing Employees and their numbers, another storing Employees and their names and so forth. Actually, since Name is a complex object, a surrogate is stored for each Name in the Employee-Name relation, and that surrogate is related to the First field and Last field in two further binary relations. The other way to store objects is to group all fields of one object together on disk.

When we compare the two representations, the binary representation is better for associative access, since all the tuples are likely to be stored in few disk blocks, and thus can be accessed quite efficiently . However, looking at all fields of a particular employee is expensive.

In the case of object-based storage, only one block is read to access the instance variables of a single Employee (if the object size fits into it). However trying to access other variables through surrogates will require many blocks to be read. Another problem is having to guarantee that objects that are referenced by others are not replicated throughout the database. As is shown by these examples, it is difficult to say which approach is better than the other, without the specification the application domain.

4.1.1.3 Dynamic Schema Evolution

In conventional database management systems, the data dictionary/directory concept is used to control access to the database, ensure data integrity and supervise the distribution of data. Dictionaries, generally built after the design

of database schema has been finished, were viewed as static tools for the control of data and information resources. However, object-oriented database applications, especially, CAD/CAM and knowledge representation applications, require capabilities to dynamically define and modify the database schema, that is, the class definitions and the inheritance structure of the class lattice. The need for a dynamic schema is motivated by the following characteristics of a domain [24]: the structure of the data is defined as the data is generated; the structure of the data is not uniform across data objects; there exist many differences of data with many different formats.

The possible ways of changing the schema can be summarized as: addition or deletion of a new class; updating the position of a class in the class lattice; modifying a class by adding, deleting or modifying properties, operations, or constraints defined by that class. These changes have direct effects on instances and users of the instances in the database which must be handled by the system. The instances of the class that exist in the database are affected because they are defined by the class. Additionally, instances of the class's subclasses may also be affected because of inheritance. Some of the class definition modifications may render existing instances illegal representations of the class. Information stored in the instances may be missing, garbled, or undefined according to the updated definition. For example, if the implementation defined by the class is changed by rearranging properties, objects created before the change will be in a non useful form. If interproperty constraints are changed, some objects in the database may contain illegal values. Repositioning a class in the class lattice has the same effect as removing some of the properties (i.e. those of the former superclass(es)) while adding others (i.e. those of the new superclasses) [1] [2]. A change in the class definition may affect the programs that use the objects of that class since the programs manipulate objects via their interfaces, supplying and receiving values according to constraints defined by the object's class. When the interface defined by the class is changed, errors may occur when a program uses the object, for example, when a property or operation is no longer defined or if a value is outside the constraints defined by the class or expected by the program or when a new name conflict takes place.

The approaches in solving the schema evolution problem can be classified into two groups: one approach, screening, delays the modification of the database indefinitely (screening); the other, conversion, changes all instances of the class to the new class definitions [20]. The screening approach requires a more intelligent interface, each time a value is to be accessed it is either filtered or corrected. The screening concept can be seen as a late binding on

the representation of objects. The two approaches offer the choice of ,”pay me now or pay me later” [20].In the screening approach, execution speed is compromised by screening, and in the conversion approach, much time can be consumed at the time a class is modified. These approaches can be combined as well , so that when the running system cannot tolerate the long term degradation of performance a conversion may be requested, however, the conversion process may result in some information loss, and historical data that can be kept by versions (versions will be treated in the following subsection).

4.1.1.4 Version Management

As a direct consequence of schema evolution comes the version management problem, which is an important issue especially in design databases. Since schema is defined only once and it remains the same throughout the life the database in conventional static databases, there is no issue of dealing with different versions of the same database at the same time, however, in object-oriented database management systems, a new version of the database is created each time a schema change is made , if conversion is not applied to the existing database to restructure the existing object instances to conform with the new definition.

An alternative approach has been presented in the previous subsection, that is, all the instances be converted to the new type version. This approach could be reasonable when the change applies uniformly to all instances of the class, both old and new, and/or has a value that must be stored with each instance [21], [38]. However, as a general approach it might be objected due to the following reasons:

- It might not be practical. If there are a large number of objects, the conversion could be very expensive.
- It might not be possible. If the information held in instances of one class is significantly different from that held in another class, conversion may require making guesses for values or discarding values that might be useful later.
- It might not be desirable. If there are old programs that must operate with instances of old type versions, those programs would be inoperative if the instances they use are converted.

To illustrate some of the conceptual difficulties consider the case where the class definitions of a class is changed (e.g. type of an instance variable). This change may be caused by the changing requirements in the environment and it may as well be the case that only future object instances of that class will be affected; therefore there would have to be two different representations for the same object type (this case can be applied to situations where specialization of the existing class may not be possible or desirable). Therefore, the storage manager should be intelligent enough to deal with objects with different versions and storage representations consistently, where late binding is in effect. In this example, another observation is the fact that a *history of class definitions* corresponding to each version must be kept and manipulated. Another conceptual problem with versions is due to the fact that versions may have versions, and if an older version can be used to create another version then a non-linear, tree-like set of versions would develop since any version may have more than one distinct successor version, and it is not conceptually clear which version is the latest one representing the final state of the database .

4.1.1.5 Extensible Data Typing

It is important to realize that, there are user implemented types that make use of existing types and primitive types (like integer, char, etc.,) to implement their own methods and protocols. When it comes to its secondary storage implications, however, operations should be transparent to the user as if every type is a primitive type. Storage manager should be able to interact directly with the type definitions and method implementations to be able to fulfill this requirement. For example, consider an object with one of its instance variables being date type; no further encoding or decoding should be needed by the user as far as the storage operations are concerned; this means that, a general typeless backend must be supported by the storage manager in order to provide uniform operations to manipulate user defined types.

4.1.1.6 Maintaining Object-Identity

It is important that identity of an object remains unchanged regardless of the changes in its state both in its internal (main memory) and external (secondary storage) representation. The concept of object-oriented pointers 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 an object or storing an object must be idempotent, that is, if you store the same object 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, like replicating objects, in order to improve performance, yet, the preservation of object-identity must always be insured.

4.1.1.7 Indexing

Providing associative access is a key issue in any database management systems and there are many interesting problems that exist with indexing. Indexing problem will be treated in detail at the end of this chapter as a special section .

4.1.1.8 Secondary Storage Management

Within this historical perspective, object-oriented database management systems has been a revolutionary approach as it appeared with brand new demands from technology which cannot be easily adopted from existing techniques and which is indeed in conflict with the old. Object-oriented database management systems have to deal with all the classical problems that exists in conventional database management systems such as transaction management, recovery, concurrency, access, maintenance; and in addition to these they introduced new problems such as schema evolution and version management. Due to these stated differences between the approaches, in designing object-oriented databases it is necessary to build secondary storage managers from scratch with their own file management, access structures and services, which has been the case for the prototype implementation.

4.2 EXISTING APPROACHES TO SECONDARY STORAGE MANAGEMENT

Object-oriented approach to database management has been investigated and many prototypes and few commercial applications have been developed.

Among these, two general classifications can be made, one is those database management systems which strive to augment existing object-oriented programming systems with notions of persistence and some database concepts like transaction, concurrency, etc. The other approach has been to design a database management system with a stronger support for database concepts while using the object-oriented programming constructs.

The concept of persistent objects has its roots from object-oriented programming languages, as some notion of permanence is required in order to develop a serious application. LOOM [43], for example is an object-swapping virtual memory system designed to assist main memory resident Smalltalk-80 in managing large number of objects. Gemstone [4], [15], [16] adds to Smalltalk-80 permanent data storage, multiple concurrent users, transactions and secondary indexes. Flavors [45] and Object-Lisp [44] are other examples of virtual memory based systems where object permanence is achieved through copying the entire world to a file. Flavors* and BiggerTalk, are systems under development at MCC, which attempt to remove objects from the local environment in which they are created and make them permanent and sharable. Both BiggerTalk* and Flavors* translate their objects to an external form.

There have also been efforts to extend existing database management systems to store objects and their relations. IRIS [6] for example is implemented on top of a relational database system and maps object-oriented concepts to relations and tuples. ODDESSY [49] is implemented using Smalltalk-80 incorporating the major features of the 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.

There are also efforts to provide storage management tools to be used as general object servers for the design of object-oriented database management systems. One such system is ENCORE [38],[10] designed to be used as a backend for an object-oriented database system and which is responsible for managing objects on secondary storage, managing transactions, and providing a persistent and sharable storage. GORDION [39] is a server which provides permanence and sharing of objects within an object-oriented environment. It supports concurrency control, manipulation of objects of arbitrary sizes, history and inquiry, and maintenance. CRM-Complex Record Manager [40]

is a storage manager to manipulate complex objects, and further support set-oriented data structuring capabilities that can be made use of by a relational database system for supporting non-first-normal-form relations. EXODUS Storage Object Manager [41] provides support for concurrent and recoverable operations on arbitrary size storage objects. It also provides primitive support for versions of storage objects, buffer management, and indexing.

Lastly, ORION [2] one of the most widely known object-oriented database management systems is implemented in Common LISP providing general object-oriented concepts with support for version management, storage and presentation of unstructured multi-media data and dynamic changes to the database schema.

This section is intended to present some implementation details of existing object-oriented database management systems with focus on the secondary storage issues. The design and implementation of the prototype has been greatly influenced by the systems that are presented in the following subsections.

4.2.1 Gemstone

Gemstone [15], [16] is an object-oriented database management system which combines the powerful data type definition and code inheritance properties of Smalltalk-80 with permanent data storage, multiple concurrent users, transactions and secondary indexes. GemStone provides an object-oriented database language called OPAL, which is used for data definition, data manipulation and general computation.

Figure 4.2.1 shows the major pieces of the GemStone system. The major pieces of the GemStone system, Stone (the executor) and Gem (the object manager), correspond to the object memory and the virtual machine of the standard Smalltalk implementation. Stone provides secondary storage management, concurrency control, authorization, transactions and recovery. Stone also manages workspaces for active sessions. Stone uses unique surrogates called object-oriented pointers (OOPs) to refer to objects, and an object table to map an OOP to a physical location. This indirection means that objects can easily be moved in secondary memory. Object table can potentially have 2^{31} entries. Stone is built upon the underlying VMS file system. The data model that Stone provides is somewhat simpler than the full GemStone model, and only provides operators for structural update and access. An

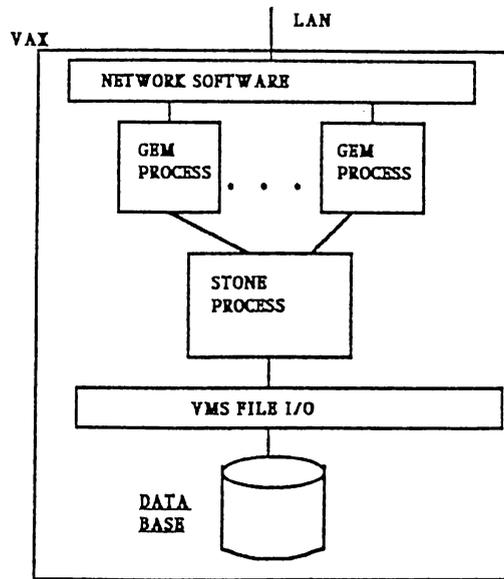


Figure 4.1: Major Pieces of GemStone

object may be stored separately from the objects it references, but the OOPs for the values of an object's instance variables are grouped together.

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. A segment is a chunk of object storage that is owned by a particular user, who can store objects in it and grant access to other users. 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 online.

GemStone's transaction control uses an optimistic approach that gives read-only transactions 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.

Stone supports five basic storage formats for objects, self identifying (e.g.

small integer, character, boolean), byte (e.g. string, date, float), named, indexed and nonsequenceable collections. The byte format is used for classes whose instances may be considered atomic. The named format supports access to the components of an object by unique identifiers, instance variable names. The indexed format supports access to the components of an object by number, as in instances of class Array. This format supports insertions of components into the middle of an object and can grow to accommodate more components. The non-sequenceable collection (NSC) format is used for collection classes in which instance variables are anonymous. Members of such collections are not identified by name or by index, but a collection can be queried for membership, and have members added, removed or enumerated. Both the indexed and NSC format support dynamic growth of objects, and are bound in size only by the total number of objects in the system and the physical limits of secondary storage. When objects in these formats grow large, their representation changes from a contiguous one to a B-tree which maintains the members by OOP for NSCs and by offset for indexed object. The byte format also supports dynamic growth in a manner similar to that for the indexed format.

Stone has several subcomponents. The transaction manager is shared by all invocations of the Stone and handles concurrent use of the permanent database in an optimistic manner. It records accesses to the database for each session and validates them for consistency when a transaction commits. The directory manager creates 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 it is to fit objects into tracks after database changes. The track manager schedules reads and writes of tracks. The commit manager provides safe 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 work space at the end of a session.

Gem sits atop Stone, and elaborates Stone's storage model into the full GemStone model. Gem also adds the capabilities of compiling OPAL methods into bytecodes and executing that code, user authentication, and session control. The Gem layer contains the virtual image, that is the collection of OPAL classes, methods and objects that are supplied with every GemStone system. OPAL is a computationally complete language and can express various associative searches on a collection.

As far as *dynamic schema evolution* is concerned, GemStone takes on the conversion approach; when a class is modified, GemStone attempts to coerce the underlying database to conform to the new definition and thus maintain a consistent database .

As a feature of GemStone, there is no file handling related language construct in the language, because all objects that the user manipulates are persistent inherently. 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.

Finally, GemStone is unique among other existing object-oriented database management systems in that it has an elegant *indexing* mechanism which will be elaborated in the Indexing Section in this chapter.

4.2.2 IRIS

IRIS is an object-oriented database management system which is intended to meet the needs of emerging database applications, including office information, and knowledgebased systems, engineering test and measurement, and hardware and software design [6].

Iris database management system consists of a query processor that implements the Iris object-oriented data model, a storage manager that provides access paths and concurrency control, backup and recovery, and a collection of programmatic and interactive interfaces.

The query processor translates Iris queries and operations to an internal relational algebra format which is then interpreted. Instead of inventing a totally new formalism, the system relies on the relational algebra. Storage manager is like a relational storage subsystem. It supports the dynamic creation and deletion of relations, concurrency control, logging and recovery, archiving, indexing, and buffer management. Every IRIS schema is mapped to a relational schema with appropriate constraints, and every IRIS instance is implemented as a corresponding relational instance. IRIS queries are translated into relational select-project-join queries, and IRIS updates become relational transactions.

Iris data model distinguishes *literal objects*, such as character strings and numbers, and *nonliteral objects*, such as persons and departments. Literal

objects are directly representable, whereas nonliteral objects are represented internally in the database by surrogates. The Object Manager provides operations for explicitly creating and deleting nonliteral objects, and for assigning values to their properties. Referential integrity is supported in the current prototype by allowing objects to be deleted only if they are not being referenced.

Objects are classified by type. Types are named collections of objects. Types may overlap; for example a person object may be an instance of the types Employee, Taxpayer and Manager. Properties of objects are expressed in terms of functions which are defined over types. They are applicable to the instances of the types. Therefore types are constraints. Types are organized in a type structure that supports generalization and specialization. The Iris type structure is a directed acyclic graph. A given type may have multiple subtypes and multiple supertypes. The subtypes may be overlapping and they do not necessarily partition the supertype. Each object of the subtype must belong to all the supertypes.

Properties may be generic; that is, properties defined on different types may have identical names even though their definitions may differ. The rules for property selection are not yet finalized. The type Object is the supertype of all other types. Types are themselves objects and their relationships to subtypes, supertypes and instances are expressed as functions in the system.

Object Manager allows the type graph to be changed dynamically, however, there are some limitations on the schema update operations. An *object versioning mechanism* has been proposed for IRIS, which will also form the basis for the implementation of concurrency control.

IRIS strives to support *extensible typing* by providing filters to translate between character strings and the new type's internal representation. Also, to enable defining new operations on the new types it provides a mechanism to define a syntax, context-sensitive rules (e.g. precedence rules), and a procedure to execute the operation.

4.2.3 ORION

ORION [1] [2] is an object-oriented database system which is operational at MCC - Microelectronics and Computer Technology Corporation. It adds persistence and sharability of objects created and manipulated in object-oriented

applications. The system supports the basic object-oriented concepts such as objects, classes, inheritance and methods. The system is being developed especially for CAD/CAM, artificial intelligence applications and office information systems with multimedia documents. It supports *version control*, *storage and presentation of unstructured multimedia data*, and *dynamic schema changes*. It also supports appropriate access paths and techniques for *query processing*, *buffer management* and *concurrency control*.

ORION supports the primitive types integer, float, string and boolean as the class Ptype. These can be used as primitive domains of instance variables. Collection and set objects are also supported. All user defined classes are instances of the system defined class Class and it is sufficient to send a message to the class Class to create a new class. The root class is Object. The class structure is a lattice structure, so multiple inheritance with default conflict resolution rules have been defined.

ORION provides automatic access to the set of all instances of a class and its subclasses by implicitly generating an instance of a special meta class, namely Set-of class, for each user defined classes. The notion of the Set-of class is especially important for persistent objects. While a program is executing, objects created by the program can be referenced through symbols that point to them. A program's symbol table provides handles for the objects. However, a newly started program will have no direct references to instances of classes through its symbol table. Instead, the program can refer to the special instances of the Set-of class of the required class. Predicate-based queries are messages to these set objects and return subsets of these sets. Another motivation for the automatic generation of Set-of classes for user defined classes is that instance variables often require values that are sets of objects. Set objects must belong to some class. Without these Set-of classes, the user would have to either explicitly create a class to capture the structure and semantics of these objects or treat them as instances of class Object, losing their semantics.

In the secondary storage, all instances of a class are placed in the same storage segment. Thus a class is associated with a single segment, and all its instances reside in that segment. All of this is transparent to the user; a separate segment for each class is allocated automatically. For *clustering* composite objects, however, it could be more advantageous if multiple classes may be stored in the same segment. The user is required to specify which classes are to be stored in the same segment.

One of the main contributions of ORION has been the elaboration of many *dynamic schema evolution* concepts [1] . A complete taxonomy of schema evolution has been investigated during the development of ORION, however, a full treatment of this study is outside the scope of this thesis and only the most important functions are mentioned below. The most important functions are to add a new class, add an instance variable to a class, delete a class and delete an instance variable from a class.

A new class may be defined as a specialization of an existing class or classes which may be specified as the superclasses of the class. It may redefine some of the instance variables and methods. If there is a conflict the conflict resolution rules are applied.

When an instance variable is added to a class, if there is a conflict with an inherited instance variable, the new variable will override the older definition. All instances of the class will be modified to include the new variable. If the class has any subclasses, they will inherit the new instance variable and if there is a conflict the new variable will be ignored.

Whenever a class is deleted, all of its instances are deleted automatically but subclasses of the class are not deleted. The deleted class will be removed from the superclass lists of its subclasses and the subclasses will be assigned the superclasses of the deleted class as superclasses. Also, the subclasses will lose the instance variables and methods they inherited from the class. If these definitions had 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-existent object. ORION does not automatically identify references to non-existent objects, because of the performance overhead.

When an instance variable is deleted from a class, the class may inherit the instance variable from another superclass if there had been a conflict involving the variable. All subclasses of the class will be affected if they had inherited the variable. Methods involving that variable will become invalid. These methods may be deleted or redefined.

Another schema evolution operation could be the changing of the domain of an instance variable of a class. The domain of an instance variable is always a class and the domain of a variable can only be changed to a superclass of

the old domain. Thus, the instances of the class undergoing the change are not affected.

Version Management is also a very important contribution of ORION. In ORION, there are two types of versions [1]. A *transient version* can be updated or deleted by the user who created it and a new transient version may be created from an existing transient version. The previous transient version then becomes a *working version*. A working version is stable and cannot be updated, it can be deleted by its owner and a transient version can be derived from a working version. A transient version can be promoted to a working version either explicitly or implicitly.

Since more than one transient version can be derived from a working version, version history is represented in a hierarchy called the *version derivation hierarchy*. Dynamic binding of an object with a versioned object is supported. The user may specify a particular version in the hierarchy as the default version. If a default value is not specified, the system selects the version with the most recent timestamp as the default.

Version handling is quite a performance overhead so versions are only kept on classes which are specified to be *versionable*. A version derivation hierarchy is kept for each instance of a versionable class. A *generic object* is used as the data structure for the version derivation hierarchy.

One of the enhancement goals of ORION is to support composite objects. A composite object is a complex object formed of a set of subobjects that are treated as units of storage, retrieval and integrity checking. For example, a vehicle is an object that contains a body object, which 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, and a set of doors in turn is a part of a body, and so on. Composite objects add to the integrity features of an object-oriented data model through the notion of dependent objects. A dependent object is one whose existence depends on the existence of other objects and that is owned by exactly one object. For example, the body of a vehicle is owned by one specific vehicle and cannot be created without that vehicle. ORION considers a composite object as a unit for clustering related objects on disk, because it is often likely to access all or most dependent objects when the root object is accessed.

The components of a composite object should be clustered. A composite object can be stored in a sequence of linked pages. If the object increases in

size, a new page can be added and if the object decreases in size, pages may be released or compacted. The only problem occurs when two composite objects exchange parts. They should also exchange storage locations. However, ORION does not perform this reclustered.

4.2.4 ENCORE

ENCORE is a shared, segmented memory system for an object-oriented database developed at Brown University [10] [38]. The main focus of this section will be on the storage management aspects of ENCORE. The database system is decomposed into two distinct subsystems. One subsystem is a typeless backend that is responsible for managing the use of the persistent object store, and the other piece is responsible for managing the enforcement of the type system.

The Object SERVER, known as ObServer, reads and writes chunks of memory from secondary storage. These chunks are used by the higher level module to store the state of objects. It also has a primitive notion of transactions which makes it possible to support a variety of shared memory applications.

The type level is referred to as ENCORE(Extensible and Natural Common Object REsource), and it is this level that deals with the semantics of objects through type definitions. The type level communicates with the server through the UNIX remote procedure call (RPC) mechanism in an asynchronous fashion.

The server is a resource that manages chunks of memory allocated in a shared memory space. Here, a chunk is any contiguous string of bytes. The server allocates space and a UNique IDentifier (UID) for each chunk that it stores. One of the principal functions of the object server is to maintain a correspondence between UIDs and chunks of memory.

Each process that wants to communicate with the server must bind a module called *client* into its image. It is, therefore, possible for the client and the server to reside on different machines. When a process needs to request a service from the server, it makes a call on the client code that hides the details of the RPC interface. The ENCORE module uses the object server as a backend. It makes calls directly on its own copy of the client module.

The chunks of memory that are managed by the server can be used to implement class objects as presented by the ENCORE interface. To set some terminology, consider the case where we have the class *Toyota* as a subclass of the class *Car*, then, an instance x of the class *Toyota* is also an instance of the class *Car*, and 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 referred to each chunk and the term *object* is referred to the aggregate of all instances that make up x . Upon 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.

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

As to how these chunks are held together, 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 part is part of the chunk for the instance of the class *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 for the instance of t that is a part of x .

Most often, these 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 *object identity*.

However, 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. This allows

for a partitioning scheme in which instances of different classes for the same object can be stored in different storage areas.

In ENCORE the *segment* provides the *clustering* mechanism. A segment contains objects that the object-oriented database management system expects a client to access during a transaction, thus eliminating frequent diskhead motions and single object transfers. Thus a segment clusters a logically related set of objects into a variable sized single package. Since a client is expected to access other objects in a transferred segment, greater system performance results from preloading required objects. A segment is thus the unit of transfer between client and server and from secondary storage to main memory. When a client requests an object, the server returns the segment in which the object resides.

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 the objects. The server, then receives a set of object changes from the client containing a client's operations and other necessary information to install the changes 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 and server processing is reduced.

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

Whereas segments provide access to objects in groups the unique identifier (UID) provides individual object access. The segmentation scheme employs two type of UIDs: *external* and *internal*. 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. Each external UID maps either directly or indirectly onto one or more internal UIDs. A mapping to multiple internal UIDs results from replicating objects (discussed below). 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. Figure 4.2.4 shows the dereferencing process from an external UID to an object. The various mappings are maintained in files called the *Object Location Table* (OLT) and *Duplicate Object Table* (DOT). In figure 4.2.4, the

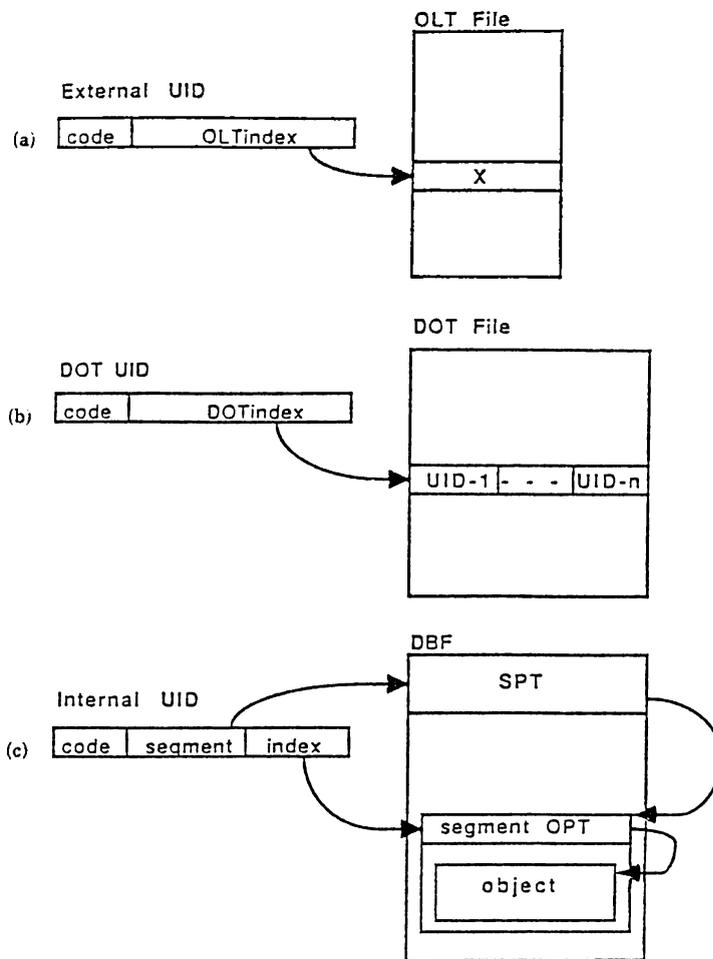


Figure 4.2: Dereferencing process in ENCORE

code field in the UID structure indicates the UID type, either external or internal. This information is used both by the client and sever processes. The OLT maintains external-to-internal UID mapping.

Object replication render it possible to cluster an object in more than one way where it might be reasonable. This incurs a penalty for update but is extremely useful for objects that are either seldom updated or read only.

The implementation of replicated objects require the introduction of a level of indirection between the external UID and the internal UID. Here, an external UID maps to an index in the *Duplicate Object Table* (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 UID is returned. Also the system guarantees that the update of all copies of replicated object occurs automatically.

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 a *Database File* (DBF). The DBF structure is similar to that of a segment: a pointer table and a set of segments. The pointer table allows a

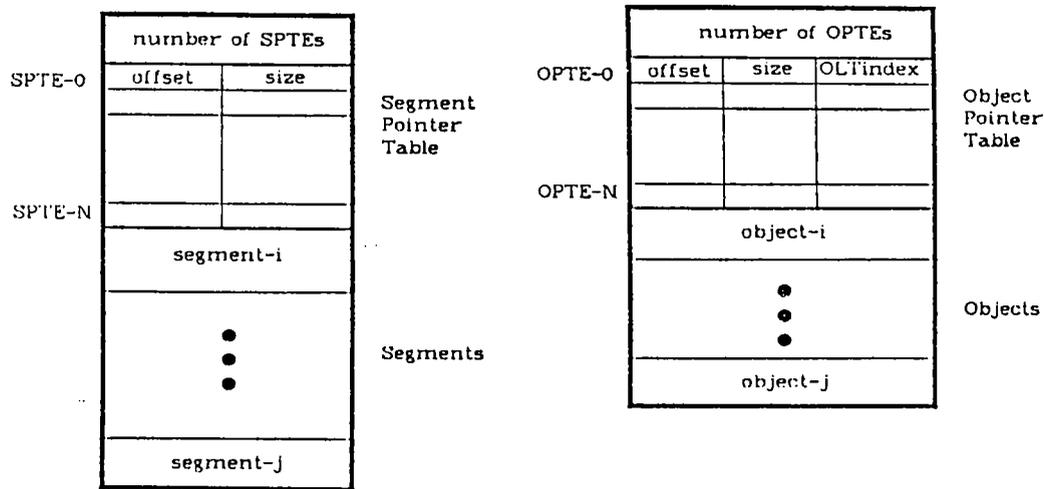


Figure 4.3: DBF and Segment Structures

reference to an object (or segment) without knowing its exact position. This makes it possible to move objects (or segments) within a segment (or DBF). The pointer table comprises one or more *pointer table blocks*, and additional fixed-size blocks are inserted as a segment acquires more objects. This feature reduces the frequency of segment expansion each time an object is installed. Figure 4.2.4 shows the DBF and segment structures.

A DBF contains the number of *Segment Pointer Table Entries* (SPTes), the *Segment Pointer Table* (SPT), and segments. Each SPTe is composed of an *offset* and *size*. The offset specifies the segment location within a file, and the size specifies the number of bytes occupied by the segment.

A segment in the secondary storage contains three sections: the number of *Object Pointer Table Entries* (OPTes), an *Object Pointer Table* (OPT), and objects. Each OPTe contains an *offset, size*, and *OLTindex*, (Object Location Table Index). The offset and size are the same as for the DBF. The OLT index provides a back pointer to the OLT that facilitates object relocation.

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

4.3 SECONDARY STORAGE MANAGEMENT OF THE PROTOTYPE

An overview of the object-oriented database management system prototype has been presented in the previous chapter. The aim of this section is to describe the secondary storage management subsystem of the prototype which was implemented as part of the thesis. The implemented version runs at the Sun Workstations [26] , under the UNIX 4.2.BSD [28] and the programs have been written in C Language [30].

4.3.1 The Goals and Requirements

The main initiative in designing and implementing the secondary storage manager has been to obtain a hands-on experience on the database issues of object-oriented database management approach, while developing an experimental test bed which will allow future researchers to further extend it to cover other aspects of database management that have not been included within the current implementation. It was not one of the design objectives to build a full fledged object-oriented database management system that would treat and provide solutions to all of the secondary storage management issues mentioned previously in this chapter, because of the complexity of designing such a system from scratch. Therefore, in order to render it a manageable task, some issues that are associated with a multi-user database management system - such as concurrency, authorization, locking; and other database issues such as version management, transaction management and recovery have been deliberately left out. However, future extendability of the current implementation to cover these topics has been taken into consideration.

The secondary storage module is responsible for managing the transfer of objects between main memory and disk storage while making sure that the object identity is preserved throughout its internal and external representation .

The secondary storage subsystem should provide a data management function compatible with the data model of the main memory. The techniques should allow uniform and efficient performance when dealing with the storage and retrieval of very small and very large single objects. The many small objects and the small number of large objects must be handled efficiently in both storage space and access time. Although these issues are

easily handled in main memory because of the inherent random access via address pointers, secondary storage possesses practical limitations on the use of random access ,therefore, the database must be intelligent about staging objects between disk and memory. It should try to group objects accessed together onto the same disk pages (that is, *clustering*),in order to reduce the number of indirections and pointer dereferencing, 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.

Persistence of objects should be transparent to the user since any object that the 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 Manager's responsibility to do address mappings and all the associated database activities.

Indexing should be provided to provide fast and alternative access paths to the persistent object store.

The requirements of secondary storage module can be summarized as:

1. Access- Fast random access to objects (and to their chunks) via their oops should be provided; clustering and preloading of objects to attain better performance and providing associative access to an object via value (indexing on value) should be available; the system should also allow noncontiguous storage of chunks to provide a vertical partitioning scheme [10].
2. Updates and reorganization- Updates that may change the size of objects must be tolerated and stability against relocation should be guaranteed without having to reorganize the whole database for avoiding unsatisfactory performance.
3. Extensible typing- Schema updates such as class definition updates, addition and deletion of instance variables and class evolutions must be supported in the secondary storage.

4.3.2 The Secondary Storage Architecture

4.3.2.1 The Module Structure

The secondary storage module is divided into two distinct subsystems. One subsystem is the lowest level *object server*, and the other is the *storage manager*.

The object server is responsible for providing the operating system like primitives to read and write byte streams in the secondary storage, without the notion of any types. It is an essentially typeless backend, implemented by using low level UNIX file handling primitives [25] [28]. This module provides all the essential primitive functions to interact with the physical storage, so future transportability of the prototype to other machines, or integration of the prototype with other file servers, or operating systems require the modification of the object server only.

The storage manager is responsible for interacting with the other components of the prototype and from the transfer of objects between main memory and secondary storage. Since the object server has no notion of types, it is the responsibility of the storage manager to enforce the type system, and interpret and manipulate the byte streams used by the object server. The storage manager deals with the semantics of objects through the class definitions by interacting with the object memory module of the prototype.

4.3.2.2 Storage Concepts

All objects in the database are implicitly *persistent*, and it is not any concern to the user whether the object being accessed resides in main memory (object memory), or in a disk file. It is rather the storage manager's responsibility to install the object into object memory if the referenced object is not already installed, and to store it into the secondary storage when the session closes or memory needs to be compacted. The current implementation installs a session-specified number of objects into object memory when a session is opened, and saves the objects to the persistent store when session closes. However, these initial load and final dump operations are implemented by using atomic, single object transfers, which can be issued at any time the session is open.

Objects are the basic unit of information stored and manipulated by the

storage manager. Since the objects are represented in the class hierarchy of the data model, an object may have many components that belong to different classes with different implementations. For this reason an object x of the class A cannot be viewed as a single structure, but is indeed a collection of different structures each belonging to the corresponding class in the super class chain of A . From here on, the private memory of an object instance (x) corresponding to a class (A) will be called a *chunk*, and the term *object* will refer to the aggregate of all chunks that make up x . To illustrate the representation of an object in the object memory (main memory) consider the following example [31].

There are three user defined classes with the following class definitions:

CLASS	SUPER CLASS	INSTANCE VARIABLES
PersonName	CLASS	first_name : string last_name : string
TitledName	PersonName	title : string
TitledNameWithLetters	TitledName	letters : string

To create objects that represent person names with titles another class TitledName is created as a subclass of PersonNames. The instances of the TitledName class will automatically have instance variables first_name and last_name and an additional instance variable title to hold the title. Then, another class, TitledNameWithLetters is created as a subclass of TitledName. This new class has the additional instance variable letters .

Now, when a new instance of TitledNameWithLetters is created, three chunks will be allocated (assuming that the superclass of PersonNames is the class Class). Figure 4.4 shows the allocated chunks for the name "Dr.John Smith,OBE". In this example it is assumed that all the instance variables are string objects and S1, S2, S3, S4 are the object-oriented pointers (oops) of these objects. C1, C2, C3 are the oops of the classes and oop1, oop2, oop3 are the oops of the instances of the classes.

Object Identity is provided by assigning a non-recycled, unique identifier to each object in the database upon its creation. This identifier is called an object-oriented pointer (OOP), and will be used in all future references to that

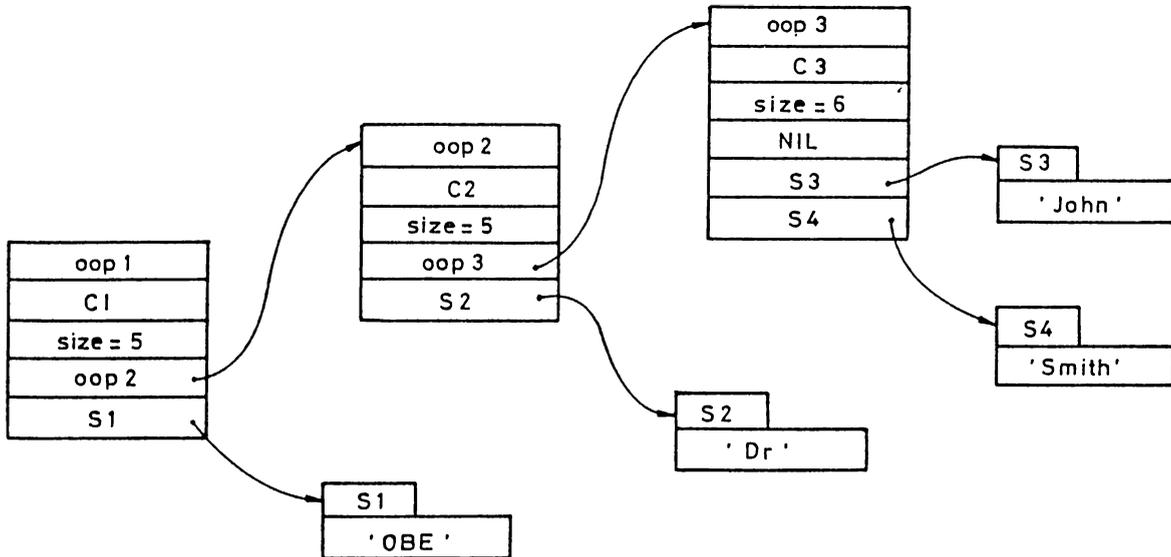


Figure 4.4: Allocated chunks for a memory object

object. Indeed, instantiation of an object will result in the creation of many chunks used to implement the object as specified by the inheritance structure of the class the object belongs to, and each of these chunks are assigned OOPs as well. Each of these chunks is accessible through the operations defined on the corresponding class, and thus can be viewed as an independent abstract object.

4.3.2.3 Storage Mapping

OOPs are essentially symbolic pointers which are converted to physical addresses when accessing the objects. The conversion takes time, but this mechanism provides location independence and solves the referential integrity problem found in conventional database management systems. Also, an instance variable of an object can contain an OOP, that is another object as its value, which results in a multi-dimensional representation of data.

When accessing the secondary storage, the OOP provides individual access to any object. The storage manager employs two types of unique identifiers: *external OOPs* and *internal OOPs*. An external OOP provides the user with a constant reference to a database object. When the storage manager dereferences a valid external OOP, there results an internal OOP, manipulated by the system to locate an object physically. The mapping from external to internal OOPs is one-to-one, that is, no object replication is allowed in the

secondary storage. Deleted objects point to a special OOP when their external OOPs get dereferenced so that the system recognizes them as deleted objects, and detects dangling references among objects.

4.3.2.4 Storage Structures

The secondary storage is composed of four distinct file structures, namely, *object-file*, *system-file*, *method files* and *index files*. The object-file contains the user defined objects and is implemented as a stream file. The system-file captures the meta information in the database by storing the class defining objects, the instance variable definitions and their associations with the user defined classes, and the class hierarchy. Methods belonging to classes are stored in a separate text file. Finally, for each index maintained there is a separate index file, organized as a B-tree.

The Container Structure

When manipulating objects in the secondary storage, efficiency becomes the principle design criterion. Efficiency is closely related with eliminating extra physical I/O and OOP dereferencing (which might involve a disk access itself) by providing a suitable secondary storage organization technique. One of these techniques, which is adopted by the prototype implementation, is *clustering* related groups of data. The *container* provides this facility. The container is a directly accessible, variable sized, recursively defined structure. Each container can be viewed as a segment holding one object with the current values of all of its instance variables. The main objective is to hold together individual chunks of an object contiguously on disk. Since a container is the unit of transfer between secondary storage and object memory, retrieval of the object with the OOP *oop1* will actually cause all the chunks that collectively define the object specified by *oop1* to be retrieved, instead of retrieving the chunk that *oop1* maps to alone. It is assumed that retrieving a chunk into main memory would most likely reference other chunks of the same object due to inheritance and thus retrieving an object in its entirety is important for eliminating single chunk disk retrievals, that is, eliminating extra physical I/O. Also, some OOP dereferencing will be eliminated, since chunks linked with the super-object-chains of a chunk are immediately available and no physical address look-up is necessary. These properties conform well with the efficiency criteria stated above.

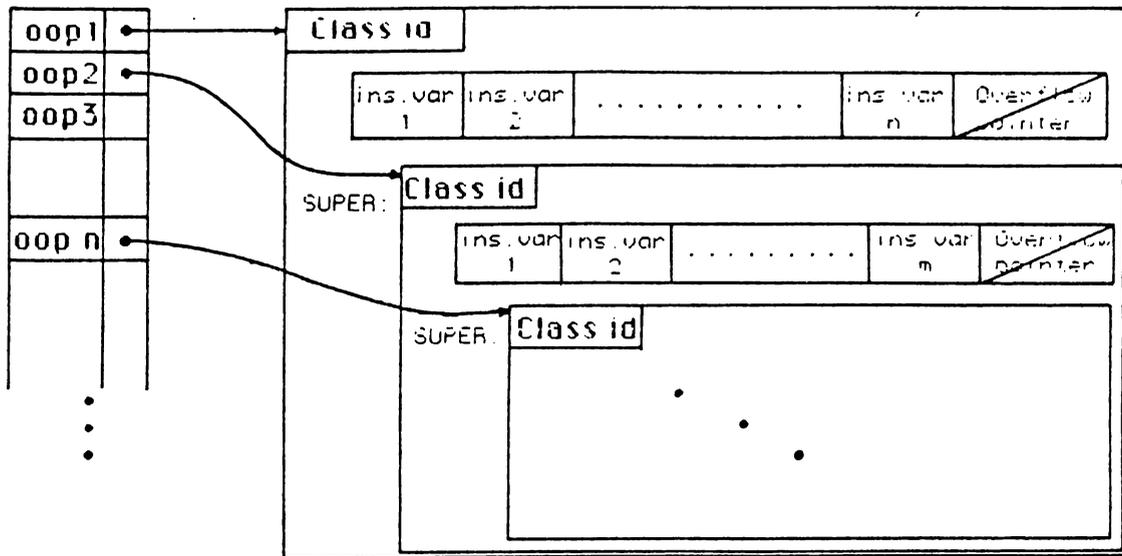


Figure 4.5: The abstract view of a variable sized container

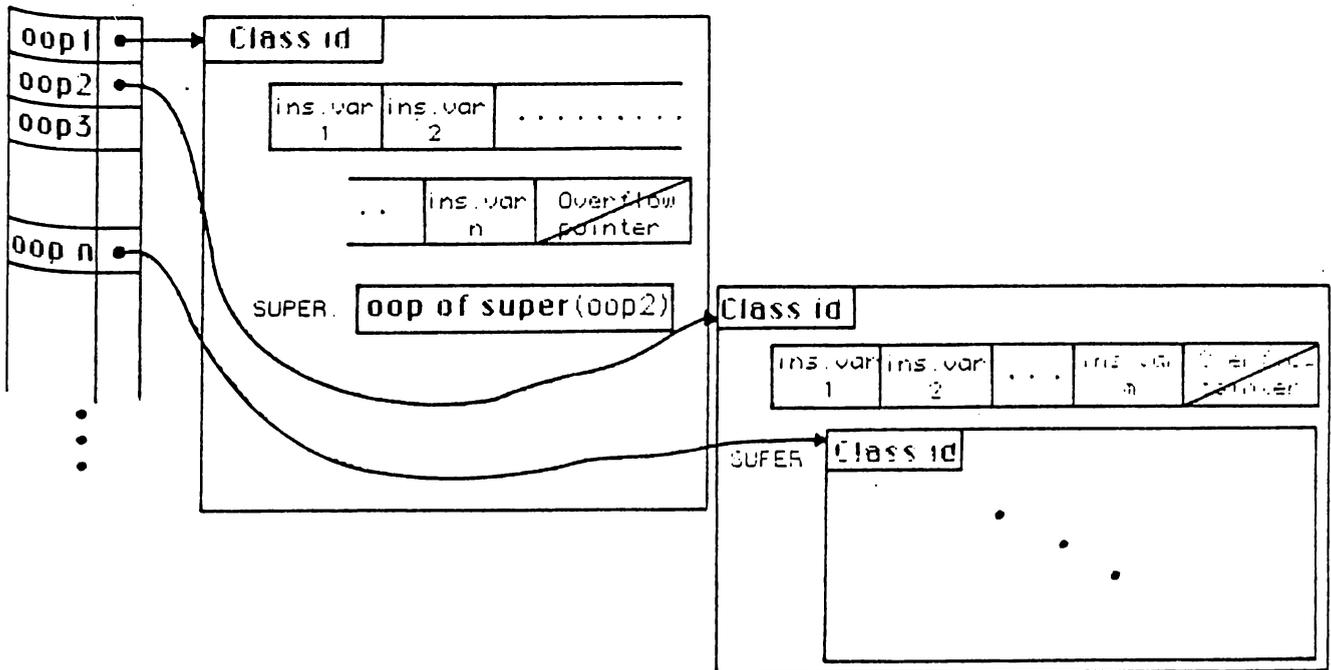


Figure 4.6: The abstract view of a variable sized container with external super-part

The container has three parts; the header , the data part and the super-object part. The *header* carries information about the class the object belongs (i.e., the oop of the class); a delete flag; and the object's oop. This oop is used for recovery from a crash of the Object Location Table, which maps OOPs to physical addresses, and used for the identification of objects included in a collection. The *data part* contains a set of triplets and an overflow-pointer. The first component of the triplet is a byte which identifies the instance variable within the class definition of current object's class; the second byte is a code for the type of the instance variable and it informs the object-server about what sort of data to expect as the third component of the tuple. The third component might be either an atomic value; (like an integer, character), or an oop, (which means that the value of this instance variable is an object with the given oop); or a container (the indirection is not necessary so the value of the instance variable which is an object is immediately accessible); or a nesting block which contains a collection object followed by the objects or their oops in the collection, terminated by a pointer to the overflow file. If new members should be added to the collection they make use of this pointer. Notice here that, the data part for objects in the same class can be quite different in size and complexity (it may have several objects, and/or collections in it); and also note that if the instance variable is of object-type, then it can belong to any class, since type information (in header) is always present in the container that contains the object. In this respect the secondary storage module is less restricted than the main-memory data model. Finally, the overflow-pointer is used to virtually expand the data part of the container, to be able to incorporate new instance variables with the existing objects. The data part in the overflow blocks are accessed as though the container and the overflow blocks were contiguous in main memory. The *Super-object part* contains either the oop of the super object or the container holding the super-object. If the class of the super object is the class CLASS, then no more nesting can occur and this is the stopping condition for the recursive structure. It is complete since every object must have an object in its super chain, and since the class hierarchy is restricted to be a tree, this chain of super objects is a linear chain and must stop at the class CLASS.

An abstract view of a container is given in figure 4.5 . The instance variable values can be atomic or structured, that is, an object itself or a reference to it. An object, which is indeed a collection of chunks each of which corresponds to a separate class in the hierarchy/inheritance path of that object, is stored in its entirety in exactly one container. A container, on the other hand, may contain more than one object in it, since according to the

s1	
s2	
oop1	
oop2	
s3	
s4	
oop3	

OLT

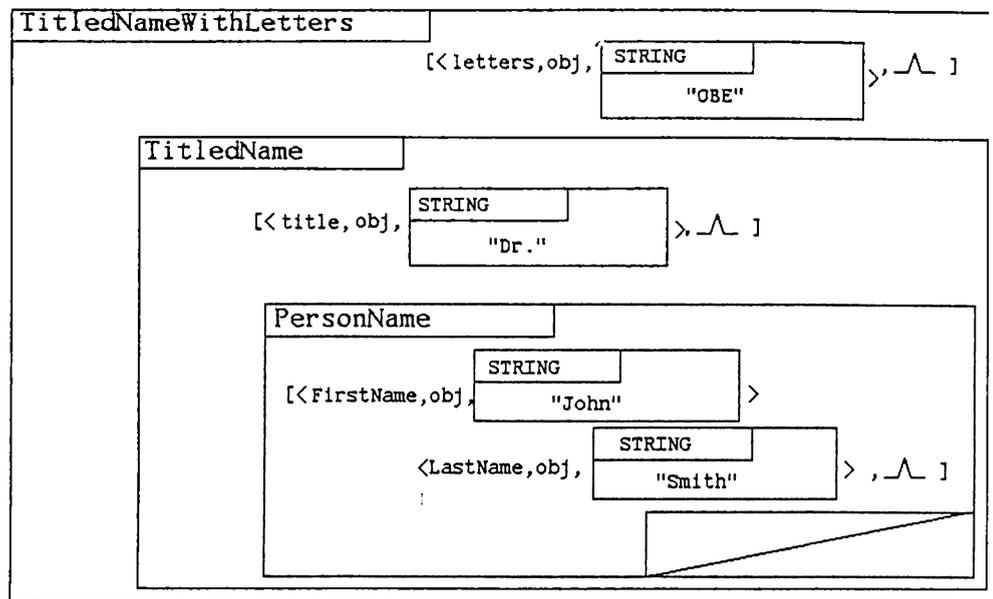


Figure 4.7: Container for TitledNameWithLetters object

context, a subset of an object's hierarchy path can be treated and accessed as a less specialized object, or an instance variable's value may be a collection of instances of a class, etc. This definition of a container actually implies a nested structure for containers. Figure 4.6 shows a container which represents the same object as in figure 4.5, however, the super object resides in another container and is accessible by indirection, then, it is only necessary to store its OOP which can be used to access the container holding the super object. To give a more concrete example consider the object memory representation of the example of figure 4.4. A possible container representation of the "Dr. John Smith,OBE" is given in figure 4.7.

A formal definition of a container is given in Appendix A, it can be seen that the container can be shown as a regular expression and indeed it is stored physically with special tokens delimiting the container's header, values, nesting blocks, subcontainers and overflow pointers. The storage of system tokens with each object involves some storage overhead, but brings flexibility, and better clustering possibilities.

Object File

Object-file is the file where all of the user defined objects are stored as a single file stream which is logically partitioned into *containers*. There is also an *overflow file* which is similar in structure to the object file. The overflow file serves as a temporary storage for objects that have undergone some kind of a schema evolution; like the addition of a new instance variable,

change in the type of an instance variable, etc.; or for objects with collection type instance variables; when a new object is added to the collection. The purpose of the overflow file is to defer the re-organization of the database after a schema modification.

The prototype uses unique surrogates (OOPs) to refer to objects, and for the secondary storage, an *Object Location File* is maintained by the Storage Manager to map OOPs to their physical locations. The physical address is a relative byte-offset from the beginning of the object-file. Each time a new object is stored into the database it goes through a registration process, and the OOP is registered into the *Object Location File* with its relative byte-offset. We actually see the object-file as a big array and access objects with their unique offset addresses. Using this technique provides us flexibility in storage allocation with no limitation on block sizes. It also makes it possible to use low-level Unix primitives as a typeless backend. This approach, however, leaves all the buffer management to the Unix file system, and clearly overlooks the associated performance related issues.

System Files

The *Object Memory Module* keeps several tables while providing primitive functions in the development and the operation of the whole system [31]. These tables are the *object table* (OT), the *instance variable definition table* (IVDT), *method definition table* (MDT), and *instance access table* (IAT). The tables and their functions have been given in [31]. These tables are active during the session and for smooth operation of the system they have to be stored while closing the session and restored when a new session starts. The storage manager flushes the contents of these tables to the system file at the end of a session. When starting a new session this file is read and the tables get initialized by the storage manager.

Method Files

For each method definition in classes, there is a method file in the secondary storage where the code implementing the method is kept. Their management is quite simple in that they are either read and compiled, or overwritten with the updated code to reflect the most recent status of the database.

Index Files

The system provides a B-tree based indexing and each separate index

component is stored in a special file. The indexing problem will be explained in detail later in this chapter.

4.3.2.5 Clustering

The storage manager has two views for clustering. One is the contiguous storage of the chunks of an object together in the same container. The other view is storing the objects that the members of a collection together in the same container, and similarly storing the object that is the value of an instance variable inside the container of the referencing object. Both views are based on the assumption that when an object is referenced then it is very likely that the inherited instance variables will also be needed, which means that other chunks in the hierarchical implementation of the object will need to be retrieved. Therefore, preloading these chunks contributes to higher performance by eliminating expensive pending single chunk disk transfers. As to the collection objects and structured instance variables, the assumption is that, the semantics of the operations that has required the logical grouping of these objects will tend to access them and process them together.

4.3.2.6 Dynamic Schema Evolution

The storage manager allows a restricted set of schema evolution functions, such as, addition, deletion, or modification of an instance variable and addition of a new class. Such schema updates can be handled through the use of the overflow file. When a new instance variable is introduced, the overflow-pointer of the data-part is instantiated to a pointer which enables the object-server to go and find the implementation of that instance variable in the overflow file. Deletion of an instance variable can be achieved by simply setting type component of the instance variable triplet to 'deleted'.

4.3.2.7 An Example

Consider figure 4.8 with the class definitions for Person, Student, UnivStudent and Employee. The figure also shows the object memory allocation of the object *U1*, which is a university student at M.E.T.U., as a 5th year student with student id = "36932" and name = "ALI". Please note that the string

```

CLASS Person (Super:UserDefinedClasses)
  name : string
CLASS Student (Super:Person)
  year : integer
  id# : string
CLASS Univ_Student (Super:Student)
  university : string
CLASS Employee (Super:Person)
  salary : real

```

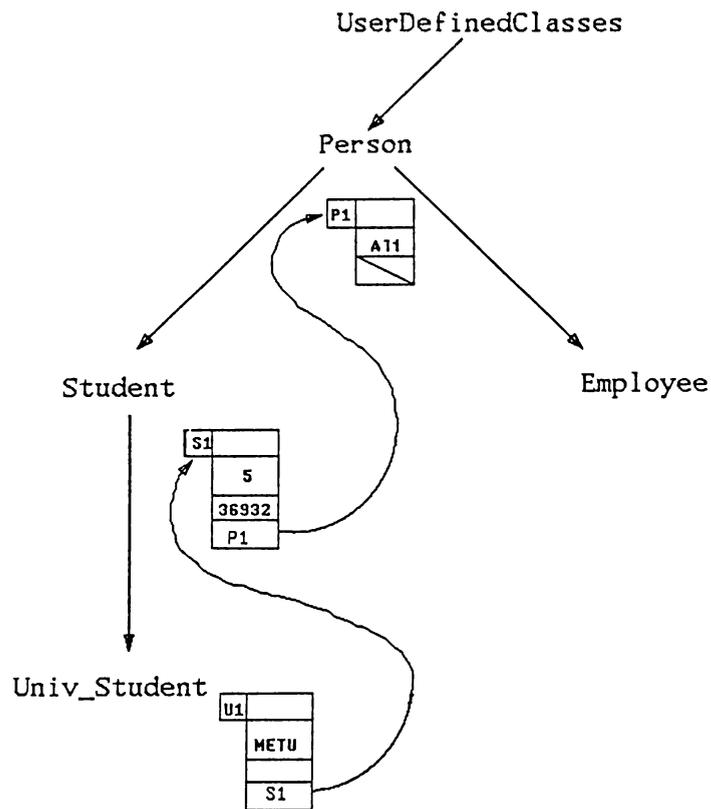


Figure 4.8: Class definitions and allocated chunks for a memory object

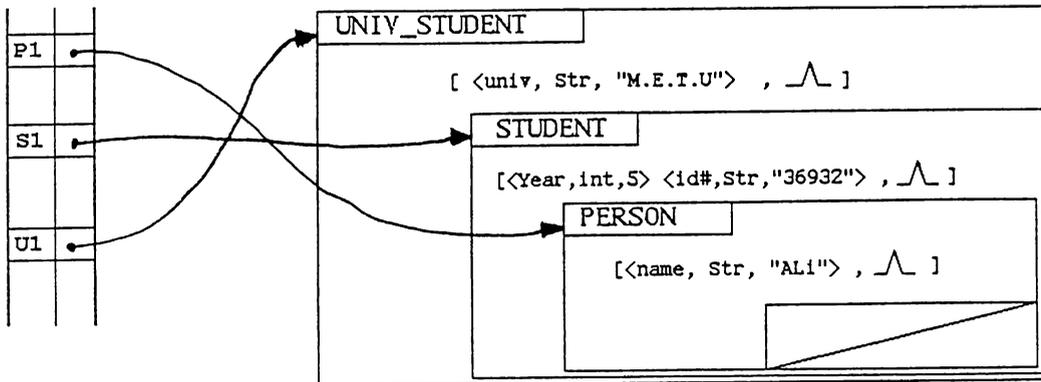


Figure 4.9: Secondary Storage representation of a memory object

objects have been shown as primitive data types for the sake of simplicity. The *U1* object can be most ideally represented in the secondary storage as is shown in figure 4.9. It is considered ideal because, it conforms with the clustering conventions of the prototype.

Now consider the container organizations in figure 4.11 representing the object memory of figure 4.10. First thing to notice is the non-clustered chunks, *U1* is located in a container different from *S1* and *P1* and is holding a reference to the container of *S1* and *P1* via the Object Location Table. Another point to mention is the Employee object *E1* with 300,000 and same identity with the Person object "ALI". This situation could have occurred when Ali finishes / leaves university and becomes an Employee. Identity of *P1* Person object remains unchanged, and depending on the context, we can view him as University Student, Student or Employee. Note that the chunk associated with *P1* is stored only in one container and the other references to *P1* are via the indirection through the Object Location Table.

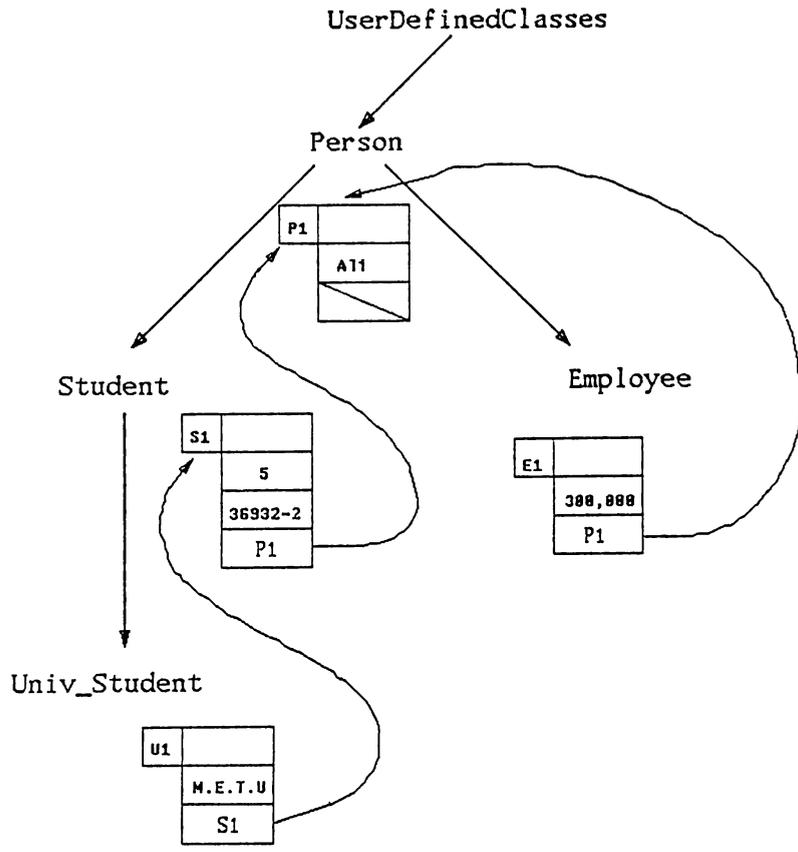


Figure 4.10: Allocated chunks for a memory object

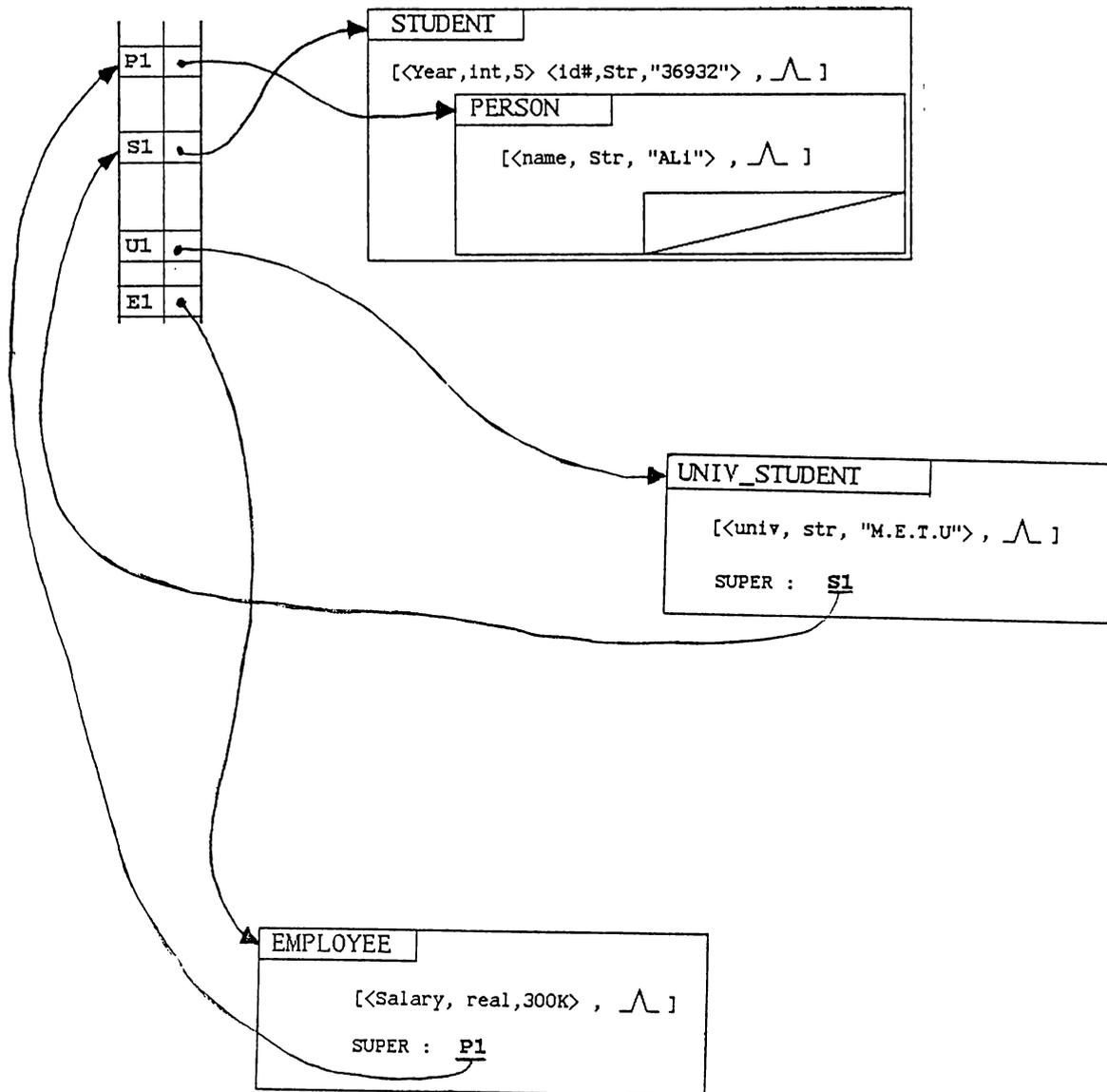


Figure 4.11: Class definitions and allocated chunks for a memory object

4.3.3 Implementation of the Storage Manager

4.3.3.1 Functional Specifications

The Storage Manager is responsible for the storage, retrieval and update of objects that reside in the object-file, while maintaining the integrity between the main memory data model and its mapping to the secondary storage. The server is responsible for registering new objects into the object database by assigning them unique identification tickets (byte-offsets) . One of the principal functions of the server is to maintain the correspondence between the surrogates of the objects and the chunks of memory.

The Storage Manager also guarantees that an object is stored in exactly one container of the object-database, and that all the future references to that object will be directed to this container. The way to guarantee this is by using the registration mechanism whenever a save request is made and if the object is already registered then new save requests will be handled as update requests. This will preserve object identity in the database.

Storage Manager is also responsible for the initialization of the tables used by the Object Memory, and the class hierarchy. The initialization is done while a session is opened by utilizing the system-file where the meta information about the database is encoded. The system-file should also be brought up-to-date when closing the session by the Storage Manager.

4.3.3.2 Interactions with other Modules

The Storage Manager provides simple protocols to its clients. These protocols are associated with retrieval, storage and update of objects in the secondary storage. The server responds to the requests of its clients by actually performing the action rather than returning a block and distributing the task with the client. The disadvantage of this approach is that, it makes the Storage Manager dependent on the object-memory module since it directly employs the object-memory routines in handling the request. Actually, the Storage Manager grabs global control over the whole system while handling the request and clearly violates the module boundaries. However, by compromising on this loss of independence, we eliminate a huge number of inter-module message exchanges , and a complex protocol. The reason for this compromise is that the object-file is not a flat file of fixed-size records, but instead it is a

complex stream that needs to be decoded properly. All the client does, is to send a request to the Storage Manager, and then wait to get the work done. If all goes well, the main memory and the secondary storage gets updated according to the desire of the client, otherwise, an error condition will be reported to the client.

An example request to the Storage Manager could be a retrieval request for an object with a supplied oop. The Storage Manager then maps this oop to a physical disk address and starts decoding the stream making use of the Object Memory primitives and invoking the protocols for installing the object into the main-memory. While installing the object, the server accesses and updates the necessary system tables utilizing the existing protocols. After a success report to the client, the object becomes available for internal processing in main-memory.

The finest *granularity* of the data that the Storage Manager operates on is an object. This level of granularity brings about a side-effect for the object protocols. Accessing an individual chunk of an object has the side-effect of bringing all the chunks of that object. Indeed, this constraint is imposed on the secondary storage by the object model of the main memory. In the object-memory, each chunk has a super-object pointer that must be instantiated to an instance of its super class unless its super class is the system-defined CLASS class. By induction, a connected chain of chunks must exist for each object. If this object model would be extended to handle dangling super-object references, then Storage Manager's granularity can be reduced to a chunk. However, the current granularity level is in conformity with the clustering objectives of our secondary storage module, and bringing all chunks of an object into object-memory in a single request, will save us from very likely physical I/O requests related with the same object.

Another side-effect of the Storage Manager's request handling is the following: when a retrieval request is issued, passing an oop to the server, all the objects that are referenced by that object will also be automatically installed into the object-memory. This side-effect, however is optional and the client may request the server not to cascade the retrieve operation to instance variables of type object. In the default case, however, when the Storage Manager sees an instance variable whose value is an object, if any, it will generate a recursive request to install that object in the object-memory, before it proceeds on to the next instance variable.

4.3.3.3 The protocols of the Storage Manager

The external interface of the Storage Manager is composed of the following functions:

SaveUserDefinedClasses() : Saves all class objects along with their linked structures (instance variable definition lists) to a special system file. This is usually performed while closing the session, or after updating a class-definition.

ReadUserDefinedClasses() : At the start of a session, all user defined classes with their definitions are read from the system file where this information is stored. Also, all of the system tables will be initialized and the class hierarchy tree is generated.

SaveObject (oop) : Saves the object with the given surrogate (oop) in the object file. While doing this, main memory functions are extensively used. This is a recursive function which stores the chunk with the given oop to disk storage and then recursively calls a SaveObject request to save its parent object, unless the parent is of CLASS class. An algorithm for this routine is given in the figure 4.12 . Since this function applies to new objects only, it does not require any overflow handling.

RewriteObject(oop) : This function updates the disk copy of an object with the surrogate oop in-place , also, if the object is resized than this function will issue overflow handling requests.

RetrieveObject(oop) : This routine is used to bring an object from secondary storage to the object-memory. This function is also recursive in nature, and will install other objects in the super-object chain of the object whose surrogate is given in oop. As a side-effect of this function, objects that are referenced via an object-type instance variable will be installed in main memory. The algorithm for this routine is given in figure 4.13 .

4.4 INDEXING

The index handler is one of the most important modules of the secondary storage subsystem. The need for indexing arises in database management

SaveObject (oop)

```
begin

    if is_registered ( oop )
        return;      /* already saved */

    else
        register ( oop );      /* make an entry in the OLT */

    for each instance variable IV of oop

        if simple_type ( IV )
            write_triplet ( IV, type, value )

        if complex_type ( IV )      /* IV is another object */
            if is_registered ( value_of ( IV ) )
                write_triplet ( IV, type, value )

            else
                write_triplet ( IV, type,
                    SaveObject( value_of ( IV ) ) )

        if is_collection_object ( value_of ( IV ) )

            for each member mem_oop in the collection

                if is_registered ( mem_oop )
                    write_stream ( mem_oop )
                else
                    SaveObject ( mem_oop )

            write_stream ( null, end_collection )

    Super_oop = Get_Super_Oop ( oop )

    if Super_oop = Class CLASS      /* terminating condition */
        return;

    if is_registered ( Super_oop )
        write_stream ( super_token, Super_oop )
    else
        write_stream ( super_token, SaveObject(Super_oop))

end
```

Figure 4.12: Save Algorithm

Retrieve (oop)

```
begin

    Get_Physical_address of ( oop )

    Install_object( oop );    /* in the object memory */
                             /* use object memory routines */

    For each triplet with instance variable ( IVs )

        read_triplet ( IV, IV_Type, value )

        if IV is simple
            set_value_of ( IV , oop, value )

        if IV is object_type
            retrieve ( value_of ( IV ) )

        if IV is Collection Type
            install_object ( value )

            while ( not ( end_collection ) )

                retrieve_and_install ( member_object )

    Read_stream ( super_oop )

    if Not null then

        retrieve ( super_oop )

    else

        return

end
```

Figure 4.13: Retrieve Algorithm

systems from the desire to provide efficient associative accesses to objects. In the prototype DBMS, the language does not support direct access to an object when we know the value of an instance variable and want to access that object (or set of objects). Also in the data model, when an object's instance variable has an object as its value we can only access that object from the owner object. References from one object to another are unidirectional. A navigation in the reverse direction is possible only by an exhaustive search on the target class, which may be too cumbersome. Providing two way links in the data model would induce a lot of overhead, too, since multiple references to an object may exist, and keeping the reverse paths of each link would be impractical for all objects. Still, it is desirable to provide a mechanism to facilitate alternate access paths under such cases, where those paths are requested by the application.

4.4.1 Design Considerations

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. One thing to notice here is the path expression construct for both of the components. The formal definitions of these path expressions 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

$A_1.A_2...A_n$ where $A_i \in \{\text{user defined classes}\}$ and

A_i is a subclass of A_{i+1} for $i = 1..n - 1$ and there does not exist any i such that $A_i = \text{CLASS}$ class and the indexed instance variable is among the instance variables of A_n .

An instance variable index path is a string of the form

$V_1.V_2....V_n$ where $V_i \in \{\text{instance variables of the Class of } V_{i-1}\}$ for $i = 2..n$. If $n = 1$ then V is called a simple index path.

Indexing a path $A_1.A_2....A_n$ on the instance variable path V will associate the oops of the objects found in class A_1 with the value of V in the corresponding object, i.e., given a value for V , all oops of objects in class A_1 associated with that value of V are returned.

If V is a simple index path, that is, it is an immediate instance variable of class A_n then this is a *one-level index*. The index handler supports multilevel indexing, too. A *multilevel index* means that we build an index on the instance variable (V_n) which is referenced by the object via an oop as the value of its local instance variable (V_1). A multilevel index is specified by providing a path with the instance variable component of the index specification. To make the indexing mechanisms clearer consider the following examples:

Example :

Consider the following class definitions:

Class Person	Class Agent	Class Vehicle
name: string	name: string	brand : string
age : integer	manager: Person	make : integer
car : Vehicle	.	boughtFrom : Agent
.	.	.
.	.	.

Building an index on the instance variable *name* in Person

< Person, name >

is an example of a one-level index. Since name is an immediate instance variable of Person, this is a valid index specification, and what it does is, it associates the value of its name instance variable with the oop of each object. These value-ooop pairs are inserted into a B-tree keyed on the value. Thus, when a value is specified we directly find the oop or set of oops whose name instance variables have that value.

Now consider a one-level index specified by the index expression:

< Student.Person, name >

where class Student is a subclass of class Person, with the following class definitions :

Class Student

```
    Idno : integer
    year : integer
    .
    .
```

With this index specification the Student objects are indexed over the instance variable, name, which every student object inherits from the class Person. There will be one entry in the B-tree associating each distinct name in the class Person, with a set of oops of the Student objects with this name.

There is one thing to notice here, those Person objects who are not Students will not be entered into the B-tree since we can not associate any Student objects with them.

The last example illustrates a more complicated multilevel indexing. Consider the

< Person, car.boughtFrom.manager.name >

index specification. Given the name of a person who is the manager of a car-selling agent all the oops of person objects who have bought cars from this agent are associated.

4.4.2 Implementation

Each index specification is independently represented by a B-tree. Supporting a one-level index is straightforward, since we have the value immediately available and the value of the indexed instance variable with the associated oop are simply inserted into the B-tree. The index maintenance is relatively easy since we are confined to a single class in detecting any state changes of the object.

Multilevel indexes have been designed by a sequence of index components, one for each link along the path. Indeed each intermediate link indexes the path in reverse-direction by identity. Then at the terminal class, a second index is used for value based access. An advantage of this form is that it allows sharing between path indexes where two paths have a common portion.

4.5 Problem Areas and Directions for Future Research

Performance is one of the most important properties of any product, very few people tolerate poor performance in return for increased functionality. Still, performance is a problem common to most existing object-oriented systems. The performance problem of the object oriented systems is essentially due to the fact that they are interpreted in nature, rather than compiled. There is another performance related bottleneck in object-oriented database systems. Since objects can exist independently from one another and can be arbitrarily related to other objects, we have a case of nonflat views of objects, and the biggest problem of the secondary storage techniques have been to handle this type of data. The concept of normalization in the relational model has been applauded to bring a formalism to eliminate this kind of structuring. In one respect, object-orientation can be more advantageous over the relational model; in the object-oriented model there is no need for most of the joins used in relational systems, as these joins often serve to recompose entities that were decomposed for data normalization. In the object-oriented model entities are not decomposed in the first place, and most of the joins are replaced by path-tracing.

The most valuable contribution to object-oriented data modeling will be the founding of a theoretical ground for explaining it and bringing standardization to it. Actually the power of the relational model, which has been the

field of utmost research for years, comes from the fact that it has a solid theory behind it and it is understood what constitutes a well-designed relational database schema.

Other problem areas of the object-oriented database management systems are *schema evolution*, *version management* and *manipulation of composite and dependent objects*.

One problem encountered in the design of the secondary storage module implementation of the prototype is that it is very important to be able to do good clustering, yet, with any arbitrary choice of clustering, there is still the problem that retrieval time for objects are not uniform. The retrieval time is dependent on many factors, from the temporal order of instantiations of objects to their inter-referencing characteristics. This fact complicates the development of a performance model for the secondary storage.

The object-oriented database management system prototype developed and implemented at Bilkent University supports the basic object-oriented concepts such as object identity, classes, inheritance and message passing but there are some open problems, which could not be handled within the scope of the thesis and are left aside for future research.

The implemented prototype is a single-user system so it may be extended to support multiple users. In order to do this, the transaction concept, authorization control, concurrency control and data integrity checks should be incorporated within the prototype.

The system does not support versions. To be able to do version management some storage structures need modification.

The system allows basic schema evolution functions such as adding a new class to the system, adding a new instance to a class, deleting an existing class and deleting an instance of a class. The system may be extended to support all schema evolution functions.

Another open problem area for object-oriented database management systems is indexing. There are many different ways of approaching the indexing issue and a careful evaluation of these approaches can result in good performance and query processing facilities.

5. CONCLUSION

Although existing record-oriented database management systems fulfill many of the requirements of traditional database applications, they seem inappropriate and incapable of providing facilities well-suited to applications in office information systems, design engineering databases, and artificial intelligence systems. This fact has led to the emergence of a trend in information management from record-based orientation to object-based orientation, which is essentially an approach that provides mechanisms to model the environment in a natural way that is closer to the human understanding and perception.

Object-oriented design allows a designer to introduce a property in its most general form by defining it on a general type and later refine the property definition for more specialized subtypes. This approach, which is known as stepwise refinement by specialization results in reduced application development efforts.

The major problems with object-oriented systems are related with the lack of theory in the field and with non-unified approaches to describing the paradigm. There is no commonly accepted description of what an object is and what its properties should be. Every approach has undertaken a different approach and there is no justification mechanism or basis to compare them.

Performance appears to be the fundamental problem with most object-oriented programming languages and database management systems. The performance problem is related with the late-binding and polymorphism properties of languages that needs run-time support and with clustering in databases. Another reason for relatively poor performance of object oriented systems is that the commercially available products are new and very few in number, and performance related areas have not been the fields of adequate research. Other problem areas of the object-oriented database management systems are schema evolution, version management and manipulation of composite and dependent objects.

The implementation efforts of the prototype has led to a good understanding of these problems and justification of others' design preferences. Some of the initial design preferences of the prototype had to be changed during the implementation as the gained insight increased.

The object-oriented database management system prototype developed and implemented at Bilkent University supports the basic object-oriented concepts such as object identity, classes, inheritance and message passing. It supports the storage of variable sized data objects by introducing the *container* concept, and uses a first-fit strategy in allocating the containers and currently no garbage collection is done in the secondary storage.

The implemented prototype is a single-user system so it may be extended to support multiple users. In order to do this, the transaction concept, authorization control, concurrency control and data integrity checks should be incorporated within the prototype.

The system does not support versions. To be able to do version management some storage structures should be modified.

The system allows basic schema evolution functions such as adding a new class to the system, adding a new instance to a class, deleting an existing class and deleting an instance of a class. The system may be extended to support all schema evolution functions.

Another open problem area for object-oriented database management systems is indexing. There are many different ways of approaching the indexing issue, and a careful evaluation of these approaches can result in good performance and query processing facilities. The prototype supports multi-level indexing, and provides language support for associative retrieval.

Finally, the problems that have been observed in object-oriented database systems do not seem to be unsolvable ones, and they are there mostly because research in object-oriented systems is new. The advantages of object-orientation are so promising for today's highly intelligent and data-intensive applications that no further discussion is necessary to explain why it is one of the popular research areas.

A. APPENDIX

```
<object> ::= B00<object_body>E00 | <oop_type><OOP>

<object_body> ::= <OOP><class_type><obj_ins_var_list><super_list>

<obj_ins_var_list> ::= B0D<ins_var_list>E0D<Overflow_ptr>

<super_list> ::= SUPER<object>E_0_SUPER

<ins_var_list> ::= null | <ins_var_name><type_value_pair><ins_var_list>

<type_valu_pair> ::= <integer_type><int_value>
                    | <real_type><r_value>
                    | <boolean><b_value>
                    | <shared><null>
                    | <default><null>
                    | <derived><null>
                    | <oop_type><OOP>
                    | <nested_type>BEGIN_NEST<collection_object>
                                <nested_obj_list>END_NEST<NEST_ptr>
                    | <string_type><string>

<OOP> ::= legal integer

<overflow_ptr> ::= byte(offset)

<integer_type> ::= -1
<real_type> ::= -2
<boolean> ::= -3
<shared> ::= -4
<default> ::= -5
<derived> ::= -6
<oop_type> ::= -7
<nested_type> ::= -8
<string_type> ::= int>=0
```

```
<collection_object> ::= {"{o}}bject>  
<nested_object_list> ::= null | <object><nested_obj_list>  
<nest_ptr> ::= byte offset
```

REFERENCES

- [1] 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.
- [2] 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.
- [3] Beech, D., *Groundwork for an Object Database Model*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 317-354.
- [4] Copeland, G., and D. Maier, *Making Smalltalk a Database System*, Proc. ACM / SIGMOD International Conference on the Management of Data, 1985.
- [5] Date, C.J., *An Introduction to Database Systems*, Addison Wesley, Vol.2, 1983, pp. 181-210.
- [6] Fishman, D.H. et al., *Iris: An Object-Oriented Database Management System*, ACM Transactions on Office Information Systems, Vol.5, No.1, Jan.1987, pp. 48-69.
- [7] Goldberg, A., and D. Robson, *Smalltalk-80:The Language and Its Implementation*, Addison-Wesley, 1983.
- [8] Hailpern, B., and V. Nguyen, *A Model for Object-Based Inheritance*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 147-164.

- [9] Hammer, M. and D. McLeod, *Database Description with SDM: A Semantic Database Model*, ACM Transactions on Database Systems, Vol. 6, No. 3, Sept. 1981, pp. 351-386.
- [10] 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, Jan. 1987, pp. 70-95.
- [11] Khoshafian, S.N., and G.P. Copeland, *Object Identity*, ACM OOPSLA'86 Proceedings, Sept. 1986.
- [12] Laff, M.R., and B. Hailpern, *SW2-An Object-Based Programming Environment*, Proc. of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 1985, pp. 1-11.
- [13] Lyngbaek, P., and V. Vianu, *Mapping a Semantic Database Model to the Relational Model* ACM 1987.
- [14] Maier, D., and J. Stein, *Indexing in an Object-Oriented DBMS*, Proc. of the Workshop on Object-Oriented Database Systems, Sept. 1986.
- [15] Maier, D., A. Otis, and A. Purdy, *Object-oriented Database Development at Servio Logic*, Database Engineering, IEEE, Vol.8, No.4, Dec. 1985.
- [16] 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.
- [17] Nierstrasz, O.M., *What is the 'Object' in Object-Oriented Programming?*, Objects and Things, ed. D. Tschritzis, Centre Universitaire D'Informatique, Université de Genève, March 1987, pp. 1-13.
- [18] Özelçi, S., N. Kesim, M. Karaorman, E. Arkun, *An Experimental Object-Oriented Database Management System Prototype*, To appear in The Third International Symposium on Computer and Information Sciences in Çeşme.
- [19] Pascoe, G.A., *Elements of Object-Oriented Programming*, Byte, August 1986, pp. 139-144.
- [20] Penney, D.J., and J. Stein, *Class Modification in the GemStone Object-Oriented DBMS*, ACM OOPSLA'87 Proceedings, 1987.
- [21] Skarra, A.H., and S.B. Zdonik, *Type Evolution in an Object-Oriented Database*, Research Directions in Object-Oriented Programming, ed. B.

- Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 393-415.
- [22] Stefik, M., and D.G. Bobrow, *Object-Oriented Programming: Themes and Variations*, AI Magazine, Jan. 1986, pp. 40-62.
- [23] Synder, A., *Inheritance and The Development of Encapsulated Software Components*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 164-188.
- [24] Zaniolo, C., et al., *Object-Oriented Database Systems and Knowledge Systems*, 1st International Workshop on Expert Database Systems, 1985, pp.1-17.
- [25] Christian, K., **The Unix Operating System**, John Wiley and Sons, 1983.
- [26] **Commands Reference Manual**, Sun Microsystems Inc., 1986.
- [27] Cox, Brad J., **Object-oriented Programming An Evolutionary Approach**, Addison-Wesley, 1986.
- [28] **Getting Started with Unix: Beginner's Guide**, Sun Microsystems Inc., 1986.
- [29] Özelçi, M.S., *Message Passing in an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.
- [30] Kelley, A., I. Pohl, **A Book on C**, The Benjamin / Cummings Publishing Company Inc., 1984.
- [31] Kesim, N., *An Object Memory for an Object-Oriented Database Management System*, M.S. Thesis, Bilkent University, Ankara, July 1988.
- [32] Nierstrasz, O.M., *A Survey of Object-oriented Concepts*, **Active Object Environments**, ed. D.Tsichritzis, Centre Universitaire D'Informatique, Université de Genève, July 1988.
- [33] Zdonik, S.B., *Maintaining Consistency in a Database with Changing Types*, ACM SIGPLAN Notices 21:10, Oct 1986, pp.120-127.
- [34] Zdonik, S.B., *Why Properties are Objects or Some Refinements of 'is-a'*, ACM/IEEE Joint Computer Conference, 1986, pp.41-47.

- [35] Hammer, M.M.,McLeod, P.J.,*The Semantic Data Model: A modelling Mechanism for Database Applications*, Proceedings of the ACM SIGMOD International conference on the Management of Data, 1978
- [36] Brachman, R.J.,*On the Epistemological Status of Semantic Networks: Representation and Use of Knowledge by Computer*, Academic Press, 1979
- [37] Mylopoulos, J.et.al.,*A Language Facility for Designing Database Intensive Applications*, ACM Transactions on Databases (5:2)June 1980, pp.185-207.
- [38] Skarra, H.A.,Zdonik S.B.,*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.
- [39] Ege, A.,Ellis, L.A.,*Design and Implementation of GORDION, an Object Base Management System*, Proc.of the Third International Conference on Data Engineering, Feb 3-5 1987 L.A.,USA pp.226-234.
- [40] 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.183-194.
- [41] Carey, M.,et.al., *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.
- [42] Özgüç, B., *Thoughts on User Interface Design for Multiwindow Environments*, SERC Report, Bilkent University, CIS-8703
- [43] Kaehler T.,Krasner, G., *LOOM-Large Object-oriented Memory for Smalltalk-80 Systems*, Smalltalk-80,Bits of History, Words of Advice, G.Krassner,Ed., Addison Wesley 1983, pp.251-270.
- [44] ObjectLisp User Manual, LMI, Cambridge MA, March 1985.
- [45] Reference Guide to Symbolics-Lisp, Symbolics, Cambridge, MA, 1985.
- [46] Ossher,H., *A Mechanism for Specifying the Structure of Large, Layered Systems*,Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 218-251

- [47] Dahl, Ole-Johan, *Object Oriented Specification*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 561-576
- [48] Wegner, P., *The Object-Oriented Classification Paradigm*, Research Directions in Object-Oriented Programming, ed. B. Shriver, and P. Wegner, MIT Press Series in Computer Systems, 1987, pp. 478-559.
- [49] Diederich, J., ODDESSY: An Object-Oriented Database Design System, Proc. of the Third International Conference on Data Engineering, Feb 3-5 1987 L.A., USA pp. 235-245.