AN INTEGRATED PERSISTENT OBJECT MANAGER (IPOM) : A MODEL TO SUPPORT PERSISTENCE AND DATA SHARING IN OBJECT-ORIENTED DATABASE SYSTEMS

By

TEH-CHEN SHEN

Bachelor of Science

National Taiwan Institute of Technology

Taiwan, R.O.C.

1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
December, 1992

Thesis
1992
S5466

AN INTEGRATED PERSISTENT OBJECT MANAGER

(IPOM) : A MODEL TO SUPPORT PERSISTENCE

AND DATA SHARING IN OBJECT-ORIENTED

DATABASE SYSTEMS

Thesis Approved:

[illegible]
Thesis Advisor

Ronald D Fisher

Huizhu Lu

Thomas C Collins
Dean of the Graduate College

ACKNOWLEDGMENTS

I wish to express my utmost appreciation to my advisor Dr. K. M. George. I appreciate not only the many hours of time he spent on my thesis but also the continuous encouragement and insight he gave me through my graduate study in OSU. I extent sincere thanks to Dr. D. D. Fisher, my thesis committee member and mentor, for his warm and constant guidance in addition to valuable suggestions and comments on this thesis. I would also like to express my deep gratitude to Dr. Huizhu Lu for her assistance and for being my thesis committee.

My special thanks go to my mother, Tao-Mei Shen-Wu for her understanding and support. My deep appreciation is extended to my wife, Hsiu-Feng Chang and to my son, Ian for their love and patience.

Teh-Chen Shen

OSU

December, 1992

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION

The next-generation (advanced) database applications are expected to handle objects with complex data types and complex relationships among them. Some of those applications such as design databases for computer-aided design (CAD) and computer-aided manufacturing (CAM) also have the characteristics of long-duration and data sharing among multiple users. According to some research, traditional database systems are not adequate to support the requirements of these applications due to their limited modeling power, supporting only simple data types, and short-transaction oriented characteristics [BER91, DEU91, BEM89, BLO87, MAI89]. Therefore, different research efforts have been exploring alternative approaches to existing methods in order to meet the needs of those new applications. One of those approaches is to extend object-oriented programming languages (OOPLs) in the direction of database languages.

An OOPL provides many new features not present in the database languages (including general computation languages, e.g. C and PL/1 etc., and interactive query languages, e.g. SQL, QUEL, and QBE etc.) in traditional database systems.

These features include powerful modeling capability, inheritance, encapsulation, reusability, rich data types [STR86], etc. However, the objective of the OOPLs is to support general purpose programming, that is, a "computational model". Several properties necessary for supporting database applications are not parts of OOPLs.

Figure 1 shows a classification of most of the database functionalities for the next-generation database applications. Among these functionalities, persistence and data sharing are two fundamental concepts to databases. The need for the inclusion of these two concepts to the OOPL paradigm to support an OOPL to be used as a basis for database implementation and programming has been recognized for quite some time [ATK83, COC83, BLO87, FOR88]. With the above mentioned as a goal, numerous persistent object storage models have been proposed to directly support persistent objects in object-oriented programming languages [CAR89, LAM91, FOR88, HOR87, KIM89, DEU91, DIX89].

However, efficient and flexible mechanisms are still being investigated. So, it is important to investigate a persistent object storage model that provides support for persistence and data sharing and also provides support for access to large and persistent complex objects efficiently for the underlying target languages.

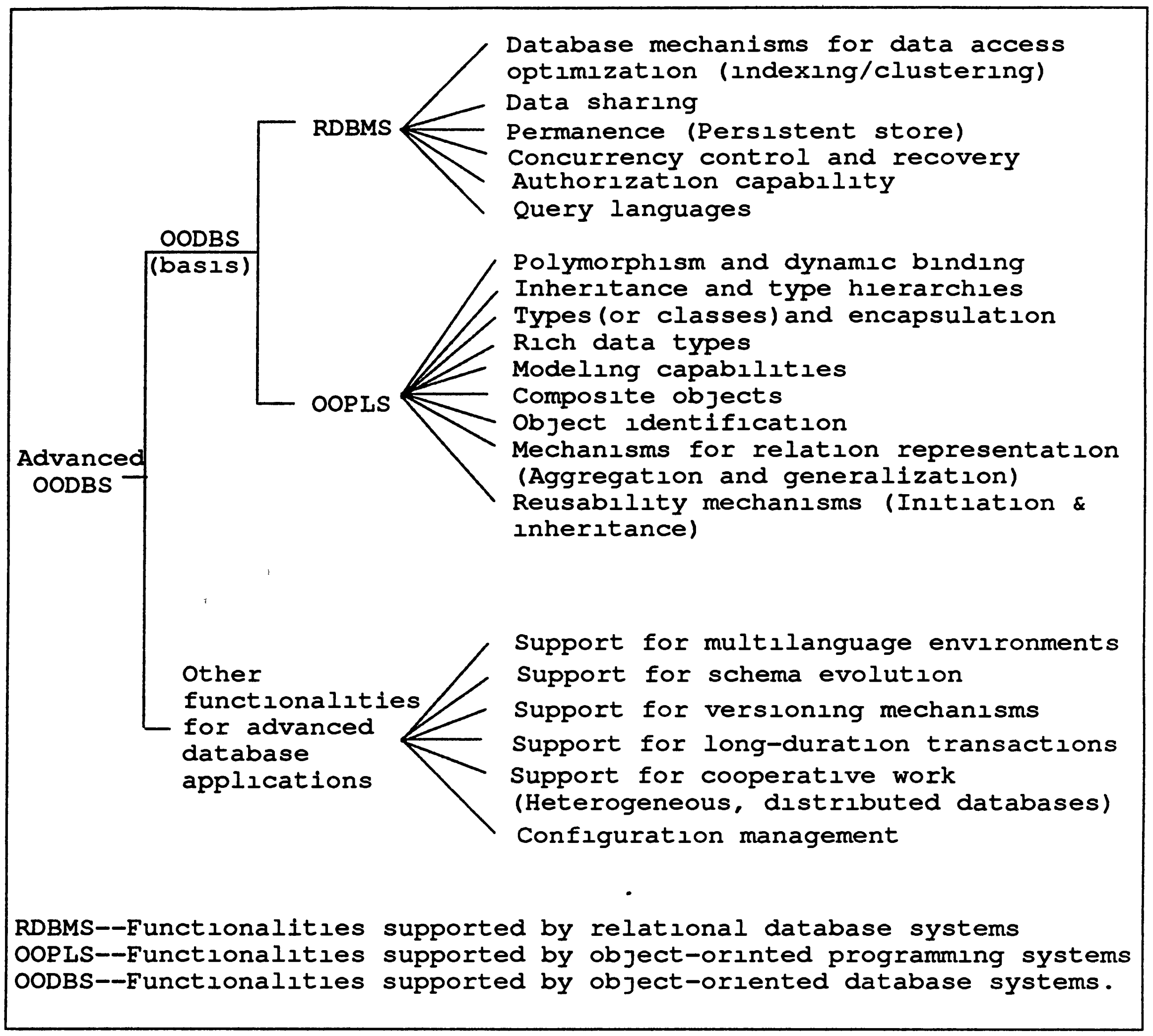

Advanced OODBS
OODBS (basis)
RDBMS
Database mechanisms for data access optimization (indexing/clustering)
Data sharing
Permanence (Persistent store)
Concurrency control and recovery
Authorization capability
Query languages
OOPLS
Polymorphism and dynamic binding
Inheritance and type hierarchies
Types(or classes)and encapsulation
Rich data types
Modeling capabilities
Composite objects
Object identification
Mechanisms for relation representation (Aggregation and generalization)
Reusability mechanisms (Initiation & inheritance)
Other functionalities for advanced database applications
Support for multilanguage environments
Support for schema evolution
Support for versioning mechanisms
Support for long-duration transactions
Support for cooperative work (Heterogeneous, distributed databases)
Configuration management
RDBMS--Functionalities supported by relational database systems
OOPLS--Functionalities supported by object-orinted programming systems
OODBS--Functionalities supported by object-oriented database systems.

Figure 1. Functionalities and Requirements for Advanced Database Applications

Motivation for Studying OODB Systems

The emergence of object-oriented database systems (OODBSs) as a promising alternative to conventional database systems capable of supporting the next-generation database applications can be attributed to its powerful modeling capability and rich data type definition mechanisms. With OOPLs, most real-world complex objects with complex relationships between their sub-objects can be modeled naturally in OODBSs. A complex object representing an object with one or more complex states (e.g. set-valued attributes) which may be complex objects themselves are often encountered in the real-world. For example, a simple complex object (memo object) representing a real-world memo entity is given in Figure 2. This object consists of some

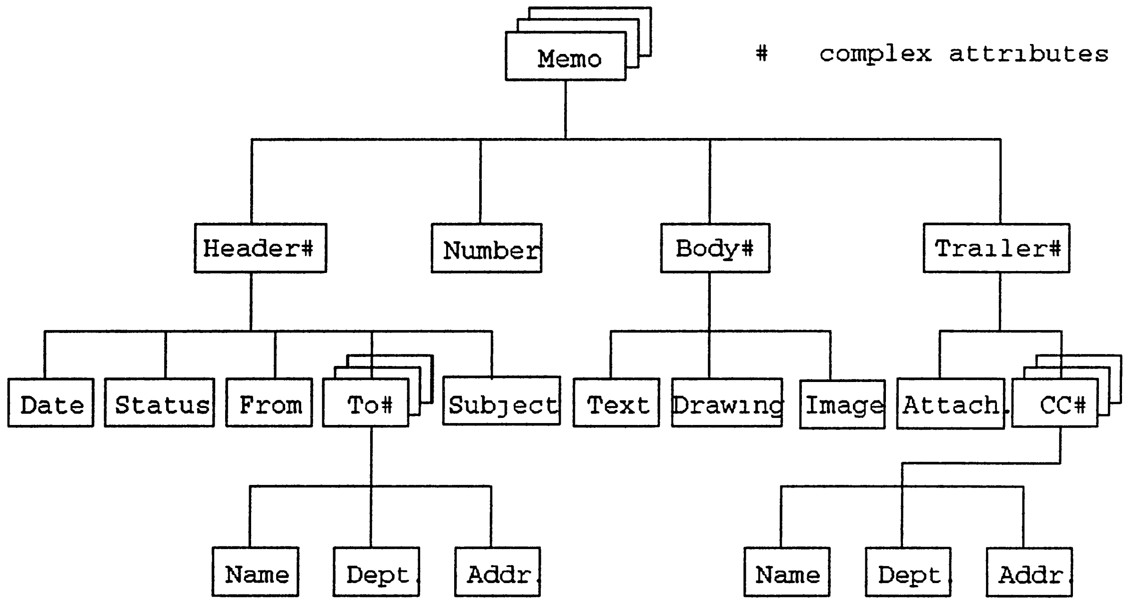


Figure 2. The Memo Object

complex states (states depending on other object(s)) such as the set of persons to send this memo to and objects with user-defined data types such as text, bitmap, images. Without sufficient modeling power, the relational model has to resort to the join-after-decomposition (decompose and then join) scheme. Such an approach to modeling real-world complex objects is likely to lead to unnatural interpretations of objects and unnecessary overhead due to expensive join operations [ELM89].

The OODBS proposes to fulfill the needs of advanced database applications by combining its powerful modeling capability with the advantages of traditional database systems such as persistence and fine-grained data sharing, etc. [JOS91, MCL89, BLO87, SIL91]. However, many researchers [BEM89, MCL89] have observed that the next-generation database systems largely differ from the traditional database systems in the following: first, the domains of their applications; second, the types of real-world objects to be modeled (thus, the need of new data types such as text, bitmaps etc.); third, the existence of complex relationships between these objects (thus, "the need to capture complex semantics of interpreting and updating the data" [MAI89]). These differences make it imperative for object-oriented database system designers to rethink the following critical issues. One is the "external modeling functionalities" for a database system such as the declarative and modeling powers of the underlying database programming languages [JOE87]. Another is the "internal functionalities" of a database system that are pertinent to persistence and data sharing [COP84, BLO87, CAR89]. These functionalities are related to storage structures, indexing and grouping, buffer management, and concurrency control and recovery mechanisms of a database system. Therefore it is necessary to have more research in these fields.

Problem Statement

As mentioned earlier, in order to combine the advantages of object-oriented programming languages and traditional databases to meet the next-generation database applications, at least two critical extensions to OOPLs are necessary [COP84]. One is the "external functionalities" extension such as powerful type constructs, persistent mechanisms, and declarative query or browsing functionalities in the OOPL itself and its interface. The other is the extension that incorporates the most important database functionalities, persistence and data sharing into an OOPL's underlying environment.

To achieve the above goals, the main concerns are as follows:

1) to provide sets or other high-level language constructs for easily modeling complex objects and set-oriented operators or iterators for efficiently accessing these objects;
2) to provide some declarative constructs in the target OOPLs;
3) to provide persistent mechanisms to bind persistent variables in the programming environment to persistent objects in databases at compile time or run time (Shen and George [SHE92] have described a classification of persistent mechanism into four types, that is, reachability-based, type-based, universal, and inheritance-based persistent

mechanisms);

4) to provide an integrated persistent object storage system that provides necessary functionalities to support persistence and maintain the consistency of the databases under the multi-user environment.

The first three are involved in extending existing language syntaxes. They are called the "external extension" of an OOPL. The last one is involved in providing functional interfaces to the compiler or run time systems. Supporting persistence and data sharing is called the "internal extension" of an OOPL (more precisely, OOPL's underlying environment).

The persistent object storage system of an OODBS serves as the basis for supporting the external extension of its underlying target OOPL. It is used to implement interfaces to create and store objects in persistent storage, move objects between the main memory and persistent storage, and to enforce concurrency control and recovery. Maintenance of indexing and grouping is also included in some designs [DEU91, BUT91]. Therefore it is the persistent object storage system that makes an OODBPL suitable as the database programming and/or interactive query language of an OODBS.

The framework of an OODBS includes the persistent storage model as an important component. Thus, research related to persistent storage model has been reported in the literature [CHO85, HOR87, PUR87, FOR88, DEU90]. However, efficient and flexible persistent object storage models that provide support for persistence and data sharing are still being investigated. In this thesis, a model of a persistent storage system, namely an integrated persistent object manager (IPOM), is proposed. The objective is to investigate a persistent object storage model that provides support for persistence and data sharing as part of the environment of an OOPL. Also support for efficient access to large and persistent complex objects for the underlying target languages is another concern.

The IPOM storage model proposed in this thesis consists of five modules:

1) the persistent object interface module (POIM);
2) the storage and grouping manager module (SGMM);
3) the transaction and lock manager module (TLMM);
4) the recovery and log manager module (RLMM); and
5) the buffer manager module (BMM).

The design features of the IPOM include the follows:

1) direct support index for complex attributes;
2) storage structures supporting "total-retrieval" and "partial retrieval" of complex objects or attributes;
3) uncopy-based buffer interfaces with the "cache strateg transaction mechanism"; and
4) local least-recently-used (LRU) buffer allocation and replacement scheme with a simple hint.

Outline of the Study

This thesis is organized as follows. In this Chapter the motivation and the statement of problem are addressed. In Chapter 2, the spectrum of persistence is discussed. Having identified the definition and spectrum of the persistence attribute, in Chapter 3, some related work is reviewed. In Chapter 4, a proposed model of a persistent object storage system is presented and the underlying architecture is introduced. The proposed approach is compared against existing models and architectures in Chapter 5. Finally, conclusions of the thesis and suggestions for future study are given in Chapter 6.

CHAPTER II

PERSISTENCE AND DATA SHARING

Definition of Persistence

"Persistence" is one of the essential concepts in traditional databases. However, the term "persistence" is rarely referred to explicitly in the traditional database and programming language literature. It was first referred to in the persistent programming paradigm a few years ago [ATK83, COC83]. As such there has been confusion concerning the terminology and definition of this concept.

From the programming language perspective, persistence is a property of an object that determines how long it should be kept. From this point of view, persistence has been defined as the ability of an object to exist as long as needed and the lifetime of the object beyond the lifetime of the process that created or manipulated it [ATK83]. This introduced the persistent object concept into the programming environment. The major objective of a persistent programming language is to manage the movement of a persistent object between the persistent storage and programming environments. The movement of an object between these environments occurs automatically through persistent mechanisms without the efforts of programmers.

The persistent mechanism is used in binding a persistent variable or identifier in the programming environment to its corresponding persistent object in the persistent storage environment. Thus, the persistent programming languages and OODB systems have at least one common goal. This common goal is to eliminate the gap between persistent and non-persistent (transient) objects. That is, from the user's point of view there should be no difference between in-memory objects and on-disk objects.

Furthermore, Kazerooni-zand and Fisher [KAZ88, KAZ89] described a classification of the persistency of an object into two types from non-persistent programming's point of view. The first one is the existence persistency (Eper). The other is the version persistency (Vper). Eper allows the lifetime of the object beyond the life cycle of the program that created it. In this type of persistency, only one version of the object is saved. Any change to an object results in a replacement of the old object by a new one. On the other hand, Vper allows different versions of one object to co-exist, and each version is marked using a timestamp or version number. The lifetime of a version may exceed the lifetime of it's ancestor. However, in current OODBS's versioning is considered to be an orthogonal issue to the persistence from databases' point of view since its absence from traditional database system (that is, the traditional database systems do not support versioning).

The notion of "existence persistence" as viewed by the programming language is implicit in the database environment. For example, in a traditional database system, a tuple object of a relation can always be identified through it's key identifier (if the key value is not allowed to be modified) at each run of different application programs or the same program. The existence of this kind of object is independent of the lifetime of the processes that create and manipulate it. Note that in the database environment, the movement of a persistent object between the main memory and persistent storage is automatically accomplished by and under control of the DBS. Furthermore, the database system guarantees that persistent objects can survive from either software or hardware failure. The term "persistence" traditionally also has been associated with both the notion of "recoverability" and the notion of "resilience" (permanence) in the database environment [BER87, SIL91]. That is persistence means the ability of a "database object" to be consistent under software failure (recoverability) and the ability to endure hardware failures (resilience). *In this thesis, the notion of persistence in an OODBS is the combination of the notion of the existence persistency (Eper) as viewed by the persistent programming languages and the concepts of recoverability and resilience from the database systems.*

Spectrum of Persistence

Following [KHO86], while treating persistence from the programming perspective, there are at least two dimensions involved in the spectrum of persistence, that is, the "representation dimension" [KHO86] and the "lifetime dimension" of an object. The lifetime of an object denotes the time interval between the time it was created and the time it becomes inaccessible (either by explicit destruction or by accident). Figure 3 illustrates this persistence space. Some general purpose software systems and OODBPLs in the spectrum are given. The representation dimension can be classified as the data value identity, the user-defined name identity, the built-in identity, the physical surrogate identity, and the logical surrogate identity.

The lifetime dimension can be classified as the existence identity of an object within an expression evaluation, within a procedure or sub-transaction activation, within a program or a transaction, between various versions of transactions (an example of this is UNIX™ shell variables which survive between various versions of processes), or beyond the lifetime of the program that created it (e.g., database objects). Based on these two dimensions, persistence is a property of an object which is associated with a persistent variable or identifier. Unlike transient variables in general purpose programming environments, persistent variables are maintained by the systems that support persistence. With its persistent variable, the object can be referenced throughout the system and may exist beyond the scope of the process that created or manipulated it.

Persistence in OOPLs

According to the discussion of persistence above, there should exist a binding between a persistent variable (or a persistent identifier of an object) in the computational environment and it's corresponding persistent object on the storage environment either at compile time (static binding) or at run time (dynamic binding). However, most of the traditional object-oriented programming languages such as C++ [STR86], Smalltalk-80 [GOL83], and CLOS (the common Lisp object system [KEE89]) are RAM-based. Even though they provide powerful data modeling and computational capabilities, they do not attempt to support persistence and data sharing, as illustrated in Figure 3. They only provide computational environments for general purpose programming on top of the file system of the underlying operating system. Therefore, there is no notion of persistent variables of persistent objects in such environments. The temporary identifiers of objects (i.e. user-supplied names, such as local variable names or global variable names) are temporarily mapped to objects in question in the storage environment through interface software of the traditional file systems. These identifiers no longer exist when the process that created and manipulated them terminates.

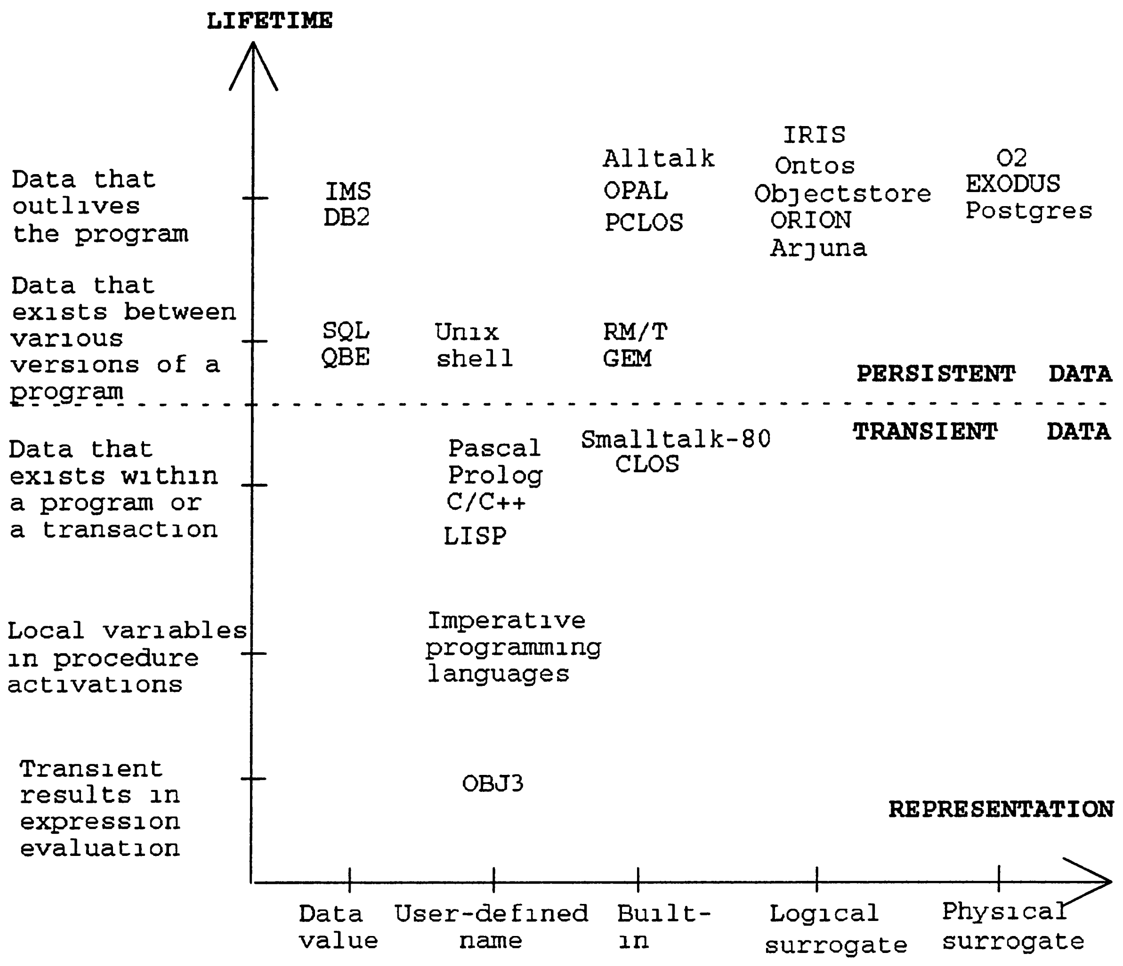


Figure 3. Languages in Persistence Space (Adopted and modified from [KHO86])

In addition to the problem of lacking the notion of persistent variables, there occurs the problem of mismatching the representations of objects between the OOPL's computational environment and its storage environment. This is called "structural mismatch" in [COP84]. In a traditional database environment, there is a uniform data structure (e.g. the relation, a set of tuples with the same data type) in both its "computational environment" and "storage environment". In a general purpose programming system, there is a rich set of data structures to represent objects in its computational environment to facilitate computation through efficient algorithms. On the other hand, there is only one type of object, the file object, in the storage environment. Since the file object consists of typeless byte-strings, there exists only the mapping between temporary variables and byte offsets in file objects. The lack of persistent variable notation and inconsistency in object representations between their computational and storage environments incur two problems in OOPLs. One is that its compiler can not bind temporary variables directly to persistent objects in a storage environment. The other is that the movement of a persistent object can not be performed automatically by the underlying file system. Therefore, traditional OOPLs like their imperative counterparts have to resort to the programmer's coding effort to preserve the states of objects created or manipulated on volatile memory. To reuse these objects, application programmers must convert their "in-file" representations back to their "in-memory" representations again. According to Atkinson [ATK83], typically 30% of the program space and programming effort is required to map or translate the representations of objects in both environments. Also reported in [JOE87], about 70% of the code for a typical access method in INGRES [STO76] database system is needed to map the representations between computational and storage environments.

Persistence in DBS's and OODBPL's

Persistence in the database system seems to be a necessary concept to achieve some important objectives in a database system such as data independence, data abstraction, and support of multiple user views [DAT86, ELM89]. To achieve these objectives, the database management system (DBMS) of a database system (in particular, the storage system of a DBMS must take care of all accesses and stores to the database objects. That is, the persistent storage system of a DBS needs to function like a persistent mechanism in the persistent programming paradigm. Thus, this capability to abstract away or hide storage details from users of the database not only eases the coding effort but also increases productivity of application programmers [JOE87, JOE89]. This is also one of the major functionalities that makes a database system powerful and different from traditional file systems [DAT84, ELM89, MAI89]. The reason to support persistence in OODB's is to achieve the same goal and thus allow an OOPL to be a basis for database programming and implementation. Programming with persistence support is then called an object-oriented database programming language (OODBPLs) [JOS91].

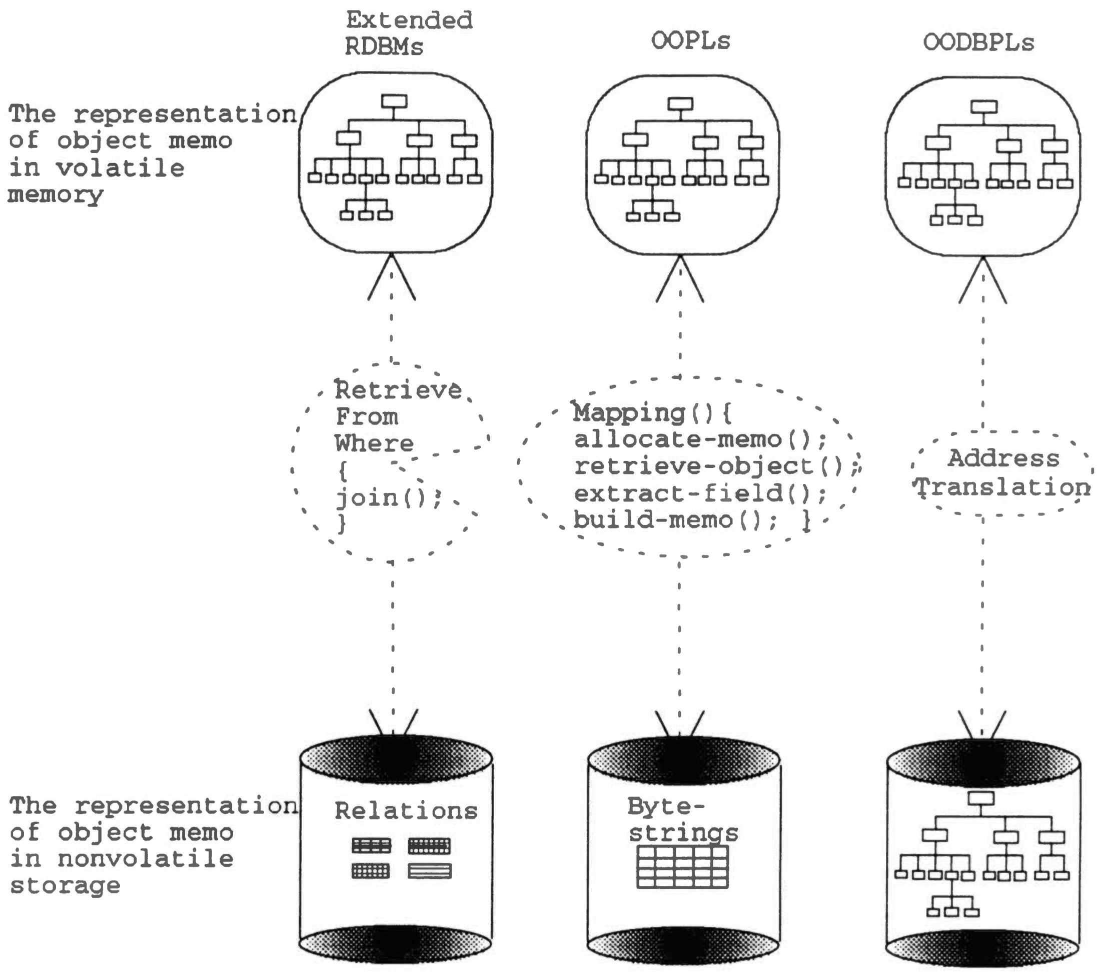

Extended RDBMs
OOPLs
OODBPLs
The representation of object memo in volatile memory
Retrieve From Where { join(); }
Mapping(){ allocate-memo(); retrieve-object(); extract-field(); build-memo(); }
Address Translation
The representation of object memo in nonvolatile storage
Relations
Byte-strings

Figure 4. Mapping in the Extended RDBM's, OOPL's, and OODBPL's Environments

To illustrate the differences between traditional OOPLs and object-oriented database programming languages (OODBPLs) which support persistent objects, Figure 4 shows an illustration of building a memo object in traditional OOPLs and OODBPLs environments. The data members (states) of each sub-object of the memo object have types. To store this typed data, the programmer's code must explicitly map these typed data to a stream of untyped byte-strings on a typeless storage (typically, a file) by coding effort. This involves coding that issues calls to conventional file system interfaces and manually handling offset, length, and type indicator information (e.g. length indicators or delimiters [FOL87]). To bring a memo object into the computational environment (volatile memory) from typeless storage, the programmer's code must also create a memo object, explicitly pick fields out of byte-strings from the storage and copy them into data members of the memo object. This requires the programmer's knowledge of the details of the storage organization which the memo object reside in [DAT86].

On the other hand, in an object-oriented database programming language environment which supports persistent data, the compiler or run-time system is responsible for the binding between persistent variables and persistent objects. Only address translation (or "swizzling" [JOE89]) is needed to access any type of persistent objects in a persistent storage and this is performed automatically through the persistence support of the underlying persistent object storage system [ATW91]. Note that traditional DBMSs provide support for only one type of persistent object, the relation object. This type of object consists of a set of tuples with limited base data types in their fields. In order to support complex objects such as the memo object, an extension effort should be made to the traditional DBMSs. This is another alternative to cope with the requirements imposed by next-generation database applications by extending the relational data model to support complex objects [CAR90] and is given here for comparison purpose.

Data Sharing in OODB Systems

One of the primary purposes of a database system is to allow multiple users to use the correct database. So, data sharing is supported in traditional database systems [JOS91, SIL91]. In a database framework, "data sharing" means allowing simultaneous use of database objects by multiple users and ensuring the consistency of the objects stored in the database [DAT86, ELM89]. The users of a database object could be thought of as concurrently executing transactions. This involves concurrency control and recovery activities [BER87].

On the other hand, traditional OOPLs generally do not deal with the multi-user environment issue. The meaning of data sharing in both paradigms is inconsistent. In the object-oriented paradigm "data sharing" means the support and maintenance of the references to shared objects. The users of database objects are themselves objects in the sense that an object may be shared by many other objects. As such, objects must have some ways to refer to each other through unambiguous references. Therefore a strong notion of object identity is imposed in object-oriented paradigms [KHO86].

However, this notion of object identity is quite different from that of database paradigms. In the relational data model, for example, a tuple is identified by its contents within a relation and does not have explicit identifiers. This identification content of a tuple is unchangeable and must be used with the relation name; otherwise, it's identity in the database is lost [MAI89]. Besides, the notion of data sharing through object identity also facilitates relationship representation in object-oriented paradigms by storing the object identifier of the related object. In this way, complex relationships among objects can be represented easily and updates to attributes of an object do not affect its object identifier. Therefore, "referential integrity" [DAT86] is ensured. In this thesis, data sharing concept in OODB's is considered to be the same meaning as that of traditional database systems.

Persistent Object Storage Systems

The above study of persistence and data sharing from both programming and database perspectives reveals the following information. First, traditional programming languages as well as OOPLs provide computational environments for general purpose programming. They deal with persistent objects (files) through explicit coding effort and interfaces of traditional file systems. This imposes a heavy coding burden on the application programmers that use such languages without appropriate support for persistence. Second, the traditional database systems support persistence in a somewhat limited sense. There is only one type of persistent object (the relation object consisting of a set of tuples of the same type with limited base data types such as integer, string, etc. in their fields) supported. This makes it impossible to meet the requirements of advanced database applications involving complex objects without further extension. Third, the lack of data sharing capability among many different programs in OOPLs makes it unsuitable for a database environment without data sharing extension.

Accordingly, the combinations of the advantages of powerful modeling capabilities in OOPLs and the persistence and data sharing functionalities in traditional database systems will benefit the advanced database applications. It is also clear that instead of a traditional file system, a persistent object storage system for an OODBS is needed to provide support for persistence and data sharing as well as support for access to large and persistent complex objects efficiently.

CHAPTER III

LITERATURE REVIEW

Introduction

In the previous Chapter we have indicated that the general approach to building an object-oriented database system has been to take the concepts of OOPLs and enrich them with persistent features (external extensions). Then, the persistent storage system performs the management of persistent objects such as providing interface facilities for retrieval and storage of persistent objects, enforcing concurrency control and maintaining the consistency of the system (internal extensions). Considerable research related to persistent object storage design has been reported and numerous storage models of OODBSs have been proposed. In the next section these related systems will be reviewed.

Related Work

The EXODUS/E Storage System

The object storage manager of EXODUS [JOE87, CAR89] is a persistent object storage system proposed to support persistence and data sharing in the E programming environment. One of the objectives has been to ease the design of application-specific database systems for database implementors [HAN91]. EXODUS supports large objects which span pages by using positional B-trees, in which indexed keys are positions in the large objects. It supports neither complex objects nor complex attribute indexing directly. EXODUS adopted a single buffer management scheme along with least-recently used (LRU) replacement algorithm. This single buffer scheme (also known as the locate mode [FOL87]) makes it possible to avoid the cost of copying large objects in the system buffer to the application address space. This scheme improves system performance substantially.

The ObjectStore DBMS

The ObjectStore database system [LAM91] like the E programming language is a C++-based object-oriented database system along with C++ extension libraries to support ad hoc queries. ObjectStore differs from all other OODBSs in that it's persistent object storage system uses a memory-mapped scheme (e.g., the single-level memory scheme originally adopted by the Multics operating system in early 1970) to map portions of the database used by an application into virtual memory and "fault-in" the necessary pages when there is a page fault.

The O2 DBMS

The persistent storage system in O2 [DEU90, DEU91] is a modified WISS (Wisconsin storage system [CHO85]). WISS supports storage structures such as long data items, sequential files, and B-tree indices which are used by O2 to implement complex objects such as tuples, sets, and lists or insertable arrays. Since O2 provides method execution support in the storage server (in [JOS91], this kind of server is called a type-based object server), the objects (the message receivers) in the server to be applied for method execution should be materialized or instantiated, that is, retrieved from the secondary storage into the server. The cost of copying objects from the WISS buffer pool into the server's object buffer have been benchmarked; a drastic degradation of system performance in read/write-intensive applications is reported [DEU90].

The Gemstone DBMS

In Gemstone [PUR87, BUT91], the "Stone subsystem" is a persistent object storage system. The Stone object storage model supports five storage formats for objects including indexed formats for large arrays and non-sequenceable collections such as bags and sets. Gemstone is based on a pure object model. Thus objects in Gemstone consist of small Smalltalk objects. Stone is also responsible for clustering collections of related objects together on the secondary storage, and concurrency control and recovery.

Gemstone also supports indexing on collections of objects.

The Arjuna OOPS

Arjuna [DIX89] is a distributed object-oriented programming system (OOPS) that supports persistence and data sharing. Arjuna supports persistence by using the inheritance property of object-oriented programming [STR86]. The persistent object store in Arjuna is implemented via the file system of the UNIX operating system. Consequently, each simple object (no support for complex objects) has to be kept in a file and the management of persistent objects is supported by traditional file system interface software. This causes severe performance penalties [DIX89].

The ORION DBMS

The ORION DBMS [KIM89] is an OODBMS that supports complex objects (called composite objects in ORION) directly in its data model. In the ORION data model, class has the meaning of both specification and extension. This means that a class automatically has its own extent (a system-maintained extent). Therefore, unlike Gemstone which supports indexing on user-maintained extents called collections, ORION supports indexing on classes instead of collections of objects. The storage subsystem in ORION uses a dual buffer with an LRU replacement scheme and copy-based interface.

The ONTOS DBMS

ONTOS [AND91] is an OODBMS and a successor of the Vbase [AND87]. Like Arjuna, ONTOS supports persistence by employing the inheritance property of the OOPL. The storage server is also built on top of the file system of its underlying UNIX operating system.

The ENCORE/Observer DBMS

The Observer object server [HOR87] is the persistent object storage system of the ENCORE DBMS. The Observer is a "typeless object server" according to [JOS91] and supports only the notion of simple objects. The Observe is intended to operate in a client-server network environment. Thus, objects are clustered into segments which reside in the database files and a segment is the unit of transfer between the workstation and server in order to reduce transfer overhead. The ENCORE/Observer database system does not address complex object issues. However, Observer provides a novel set of lock modes including notify locks to support data sharing in the client/server environment.

CHAPTER IV

THE INTEGRATED PERSISTENT OBJECT MANAGER (IPOM)

Introduction

The integrated persistent object manager (IPOM) is a persistent object storage system to support both persistence and data sharing in an OODBMS. The motivation for the design has been to investigate the architecture and mechanisms of a persistent object storage system in the context of support for persistence and data sharing in object-oriented programming languages. The IPOM storage model proposed in this thesis differs from other models of object storage system mainly in its design schemes. These schemes include:

1) a direct support index for complex states (attributes);
2) storage structures supporting "total-retrieval" and "partial retrieval" of complex objects or attributes;
3) uncopy-based buffer interfaces with selectively copy-based option (cache strategy transactions); and
4) local allocation and replacement buffer scheme with

a simple hint (either keep or discard) to the buffer manager.

Direct support indexing on complex states (attributes) and grouping storage structures provides efficient fetch and store of a whole complex object or any state of a complex object without retrieving the whole object into the memory (this is especially useful when the complex attributes are very large). The use of non-copy-based interfaces provides efficient retrieval and direct manipulation of objects without further copying cost. The buffering scheme with a simple hint (either KEEP or DISCARD) gives more flexibility to the nontraditional database applications with dominant "chain reference" (access sub-object via embedded object identifiers rather than join) access patterns. The cache strategy transaction mechanism provides support for computation-intensive applications without incurring excessive interfaces of crossing of a database system.

All these design features are expected to provide appropriate support for large persistent complex objects, efficient retrieval of entire or partial complex objects or attributes, and access patterns that arise from advanced database applications. A comprehensive scheme is presented in this chapter which combines new ideas and adaptations of some well-established concepts. That is, the proposed model draws heavily from ideas developed by the research community of traditional and object-oriented database systems in the past. The uncopy-based interface scheme is borrowed from the EXODUS storage system (originally from the system R [AST76]) and extended with copy-based option (cache strategy transactions) to solve the problem of excessive calls to the persistent storage system in EXODUS. Especially, the concurrency control and recovery modules are adopted from that of the traditional database systems as other storage models do. The only exception is that the buffer strategy argument can be specified in the Trans_Begin command to support cache strategy transactions.

The Generic Object Model

Since the proposed storage model is to be used as a vehicle to implement an underlying target object-oriented model, a generic object model is presented here. This generic model combines the most common features of many of the object-oriented models proposed in the past few years. These common features include object identity, strong typing, type constructors, and object references. In this model, every instance of an object owns a system-wide object identifier (OID) that can not be changed. The object identifier is used by the system to reference its corresponding object. This model also supports basic types such as integer, string, float, bits, etc., two collection type constructors, sets and lists, and the tuple constructor. Each instance of the constructors is the first-class object that owns a unique object identifier.

In this model, a unique set is defined as a collection of objects with the same type. A **"set-valued"** attribute is an attribute whose value is a set. A list is an object with a sequence of elements of the same type and each element is of atomic type. A **"sequence-valued"** attribute is an attribute of list type; a tuple consists of a set of attributes that are of different atomic types. A **"tuple-valued"** attribute is an attribute of tuple type. Attributes that are not of atomic type are called complex attributes. A complex object may consist of any combination of atomic attribute (atomic-valued), complex attribute (e.g., set-valued, sequence-valued, or tuple-valued attribute), and complex sub-object. In this sense, a complex object is a tuple that has at least one non-atomic attribute which itself may be a complex object. The object reference concept allows a complex object to be a nested object. That is the attribute of an object or an element of a set can be itself an object. While a tuple can be viewed as a special case of the complex object type, a relation or table in traditional databases can be viewed as a tuple-valued set.

To illustrate the concept of complex object, the object memo given in the chapter I can be used as a simplified example of a complex object. It contains three complex sub-objects, "Header", "Body", and "Trailer". The complex sub-object "Body" of memo consists of "sequence-valued" attributes such as "Image" and "text" attributes. The complex sub-object " Header" contains atomic attributes such as "Date", "Status", "From", and a "set-valued" attribute

"TO" which consists of a set of tuple-valued elements.

The System Architecture

Figure 5 shows the general system architecture of the IPOM. The functionalities of the IPOM can be included in IPOM modules linked with the high-level language run-time system layer or included as parts of the application run-time system. The IPOM consists of five modules: the persistent object interface module (POIM), the transaction and lock manager module (TLMM), the storage and grouping manager module (SGMM), the recovery and log manager module (RLMM), and the buffer manager module (BMM). The IPOM is designed to be built on top of the physical I/O module similar to the UNIX I/O system call interface level. The five modules of IPOM are described below in top-down fashion.

Persistence and Data Sharing in IPOM

Before introducing the IPOM, let us first informally describe how the IPOM realizes the support for persistence and data sharing. As mentioned in Chapter II, external and internal extensions are needed to support persistence and data sharing in OOPL environments. We use the Figure 5 to describe the scenario of supporting persistence and data sharing under the IPOM model. The process of supporting persistence and data sharing can be classified into five phases as follows:

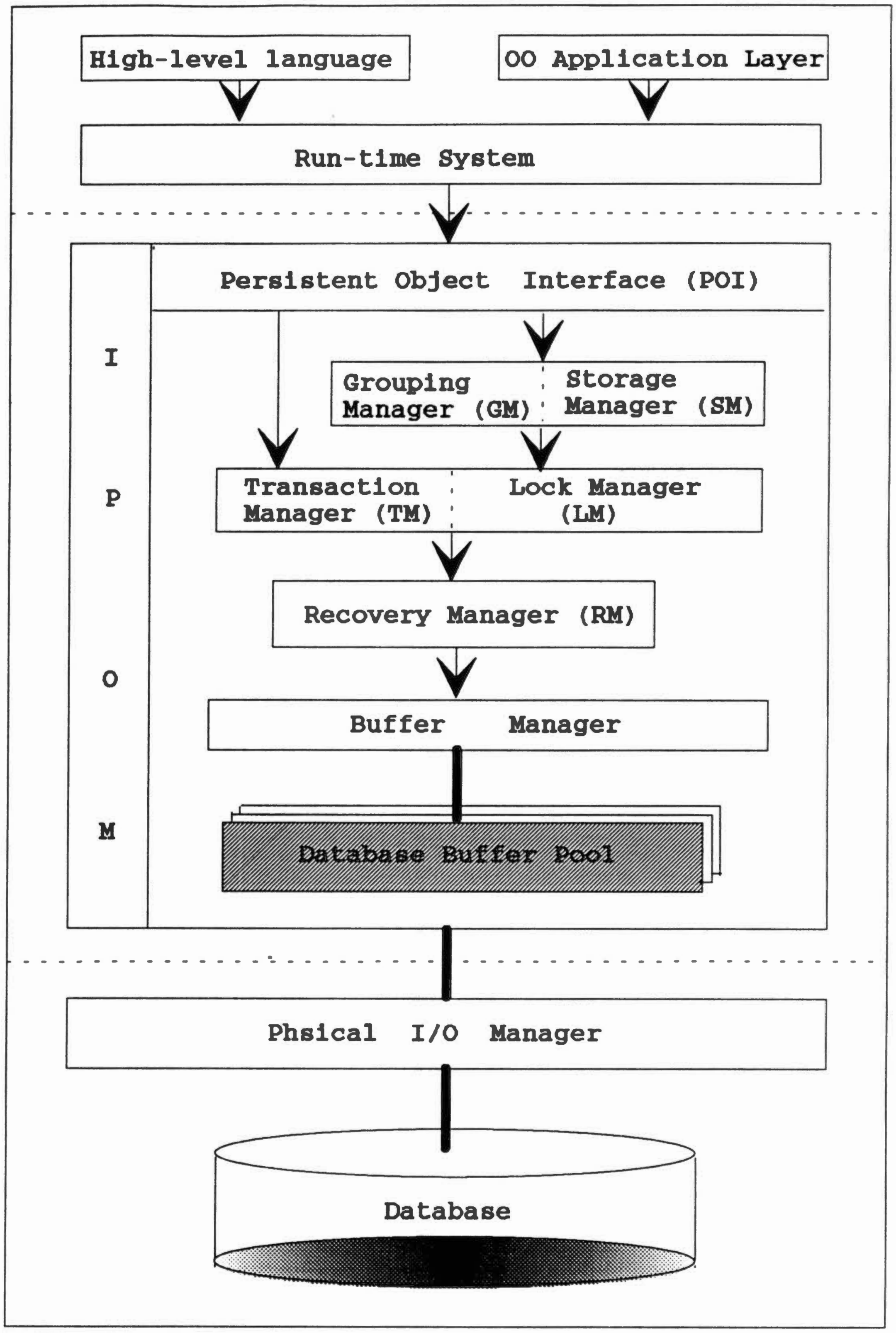


Figure 5. The IPOM Architecture

1) the binding phase: In this phase, the binding between persistent variables (names) in application programs or query statements and persistent objects in databases is established. This is always done by persistent name server by searching persistent name dictionaries and mapping a persistent name to a unique identifier (UID) during the compiling phase. The compiled codes or query evaluation plans will contain the primitives provided by the persistent object interface, in our case this will be the persistent object interface module (POIM).
2) the execution phase: The run-time system dispatches compiled codes or query evaluation plans and executes primitives provided by POIM and SGMM when the object access or manipulation is needed.
3) the concurrency control phase: The object retrieval and storage operations (read and write) invoked by POIM primitives are sent to the transaction and lock manager (TLM). The TLM enforces the concurrency control protocol and checks access conflicting.
4) the locating and fetching phase: Before fetching the desired object into the database buffer pool, its location in the database must be located. This involves mapping UID to PID (physical object identifier) in case that logical identifiers are used. The buffer manager module (BMM) performs the

buffer allocation and replacement tasks. Then the physical I/O operations provided by the underlying operating system are issued and the page containing the desired object is brought into the database buffer pool.

5) the extracting or isolating phase: The location and boundary of the desired object in the database are determined according to object template information (schema information). A pointer to the object in the buffer is returned to the user or query processing algorithms that invoke the fetch or store operations. In case of copy-based interface, the desired object in the buffer must be translated from its on-disk format into in-memory format and then be copied into the user address space.

Note that persistence and data sharing support by the IPOM is transparent to the user. Persistence is supported as in the persistent programming environment in which the programmers specify which object is to be accessed or manipulated without explicit coding how to do it. In addition, data sharing is supported as in a traditional database system.

Persistent Object Interface Module (POIM)

The objective of the POIM is to provide primitives for the run-time system or compiler as an interface to access and manipulate persistent objects in the database. Since the high-level object model supports complex objects with arbitrary levels, the POIM should support primitives to access and manipulate various types of large complex objects with potentially unlimited size. This imposes several different invariants for access methods according to the semantics of the operations on different types of complex attributes or objects (tuples, sets, lists, etc.). Therefore, the POIM provides a set of primitive operations on tuples, sets, and lists to access and manipulate these types of persistent objects. This is based on the three types outlined earlier.

In addition to the basic operations on complex objects in a file or group such as retrieval, insertion, deletion, and creation of an entire complex object, the POIM also supports primitives to operate on portion(s) of a complex object. These "partial-object" operations include the following primitives: retrieve only selected attributes of a complex object (or a large tuple); update only single or selected attribute(s) of a complex object; retrieve only selected elements of a set-valued attribute of a complex object; delete only selected elements of a set-valued attribute of a complex object; update only selected elements of a set-valued attribute of a complex object; and insert elements into a set-valued attribute of a complex object. A set of primitives (operations) on complex objects proposed in our design of POIM is depicted in Table 1. These primitives allow portion(s) of an entire complex object to

TABLE I

THE PRIMITIVES OF THE POIM

Primitives of the Persistent Object Interface Module	
Primitive name	**Functions**
CREATE_TUPLE()	create a new tuple
RETRIEVE-TUPLE()	retrieve an entire tuple
PROJECT_TUPLE()	retrieve only selected attributes of a tuple
DELETE_TUPLE()	delete a tuple
UPDATE_ATTR()	update the content of an attribute
RELEASE-TUPLE()	unfix a tuple in the buffer
CREATE_SET()	create a new set
INSERT_ELEMENT()	insert an element into a set
DELETE_ELEMENT()	delete an element of a set
UPDATE_ELEMENT()	update the content of an element
RETRIEVE_SET()	retrieve an entire set
RETRIEVE_ELEMENT()	retrieve an element of a set
SCAN_ELEMENT()	scan all the elements of a set
EXIST_TEST()	test for the presence of a particular element in the set
GET_RANGE()	get the elements which are qualified in the specified range
APPEND_LIST()	append elements to a list
UPDATE_LIST()	replace elements of a list
DELETE_LIST()	delete elements of a list

be retrieved. This kind of partial-retrieval of a complex object can not be done by storage systems that rely on traditional file systems or relational storage systems which do not support complex objects. Usually, the whole complex object, implemented as a file, has to be retrieved from secondary storage and translated into its in-memory format record-by-record.

Storage and Grouping Manager Module (SGMM)

Object Representations

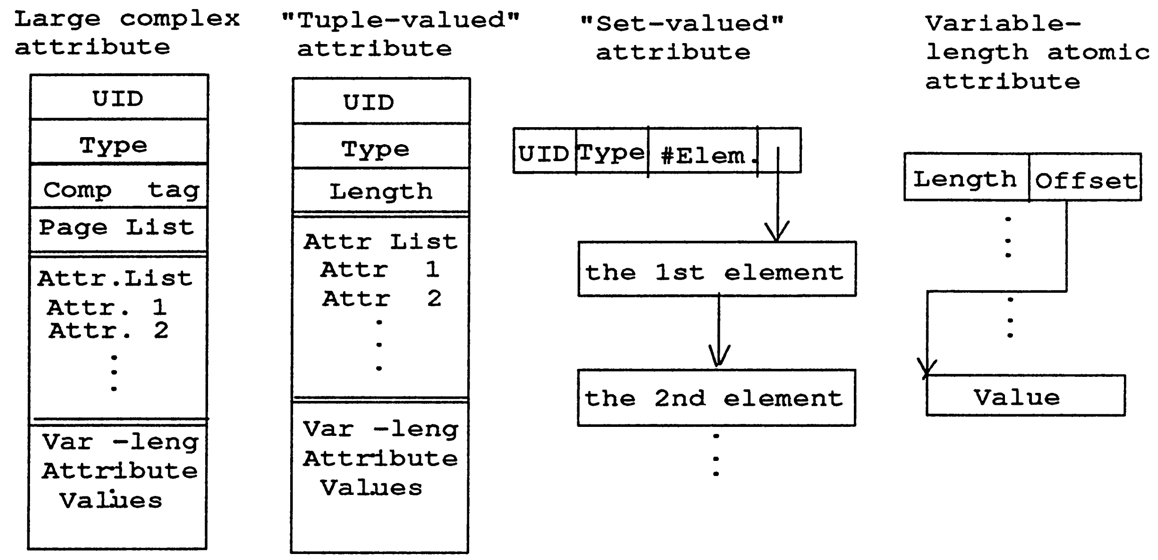
The advanced database applications are expected to handle complex objects with complex relationship among them. Also the attributes of a complex object may not be like that of simple objects with fixed and small size and of atomic-valued attributes. The object representations of storage structures for complex objects should consider the characteristics of these complex objects. These characteristics include objects of different types, objects of variable-length, and objects with potentially unlimited size. This implies that extended storage structures are needed. Therefore, there are different object representations for different type of objects in the IPOM storage model (refer to Figure 6). The basic type of object representation is that of the basic type construct tuple, as illustrated in Figure 6(b). There are three parts in a tuple object: the tuple prefix, the attribute list, and variable-length attribute value parts. The tuple prefix contains the following information:

1) the unique object identifier for the tuple,

2) the type (or class) of the tuple (object),

3) the length of the tuple (object), or a tag to indicate that this attribute (object) is a large complex attribute (object), or/and

4) the page list of the tuple if the tuple is large.



(a) A Large Complex Attribute (b) A Tuple Attribute (c) A Small Set Attribute (d) A Variable-length Atomic Attribute

Figure 6. The Object Representations

The attribute list part contains all the attribute information of the tuple. This part contains the following information:

1) the value of fixed-length atomic types such as integer, float, and char etc.,

2) the information of each variable-length attribute

including its size and the offset in the variable-length attribute part, and

3) the information of small set-value attribute or UID's of large complex attributes. The former includes the type of the set, the number of elements in the set, and the pointer to the first element in the set.

The variable-length attribute part contains the values of the variable-length attributes and a list of set elements of small set-value attributes that link each other together.

Grouping Manager (GM)

One of the major goals of the SGM is to improve the performance of retrieval of an entire large complex object or part of it actually needed from secondary storage. We call these "total-retrieval" and 'partial-retrieval" of complex objects, respectively. Storing a complex object in a file as in existing CAD/CAM applications [KAT90] would require that whole complex object be retrieved from the secondary storage and then the actually needed portion of that complex object be extracted. This will lead to the waste of expensive disk I/O and the waste of main memory space. Therefore, the storage structures of large complex objects are important in the context of "partial retrieval" of large complex objects.

As mentioned above, the generic model supports complex objects possibly with "set-valued", "sequence-valued", and "tuple-valued" attributes. In order to reduce disk I/O in retrieving an entire complex attribute, an internal grouping strategy is needed. This grouping technique is used to facilitate the "total-retrieval" of a large complex attribute. It also will group the instances of a complex attribute in the same storage extent (an extent is a number of contiguous blocks on the secondary storage) as possible. This grouping technique is a kind of "inter-object" grouping strategy that groups objects of the same type (e.g. a class) together. The SGMM module supports internal grouping. That is instances of a complex attribute (a set-valued, a sequence-valued, or a tuple-valued) are grouped together. This grouping hint is given by the user when an element is to be inserted into a group with a "near hint" in the argument of the insert primitive. The SGMM module then will try to insert that element into or near the indicated group as possible. If a group is very large "set-oriented I/O" such as "scatter read/write I/O" supported by IBM/370 systems is useful [KAT86]. However, in some environments such as UNIX this kind of benefit is not available currently. It is also not guaranteed that contiguous blocks can be allocated on disk. The alternative approach is to use a page list in the prefix part of each group root page. The page list is a list of physical page numbers of the entire complex attributes. This page list can be used to facilitate the prefetching of the complex object.

A complex object with potentially unlimited size may have large complex attributes that themselves are complex objects. Thus, the number of elements of a "set-valued" attribute may be very large. For example, in CAD design, a VLSI chip consists of 25 sections which contains 164 cells and each cell contains about 2000 transistors. Each transistor may contain 40 to 100 bytes. In addition, the number of elements of a "sequence-valued" attribute may be unlimited in size. For example, in multimedia database in office information applications, a bit map of a digitized 8.5" X 11" image can consume up to 4 Mbyte of storage [WOE86]. Also in clinical databases or medical databases, a large patient or gene sequence record with hundreds of attributes is possible [SIL91].

With the large complex attributes as discussed above, the efficient retrieval of small portion of or one single element of such large complex attribute depends on indexing techniques. Direct indexing on a complex attribute of a complex object provides a way to efficiently access any instance (state) of a complex attribute. However, the criteria for indexing on instances of a complex attribute is according to its size. If the size of a complex attribute is larger than a disk page then it is suitable to be indexing according to its type characteristic. For example, a large "set-valued" attribute (an unique large set) can be represented as a B+ tree index [COM79]. A large "sequence-valued" attribute (there are different terms in literature such as a list, a sequence, an indexible list, insertable array, variable-length array [CAR88], or ordered collection [LAM91]) can be represented as a positional B tree in which the search keys are the positions of the elements within the list [CAR89]. It is used to model ordered complex objects such as texts or documents (type of strings), bitmaps (type of bits) etc. The above two indexing techniques have been used by some current OODB systems that support "sets" such as O2 and Objectstore.

However, the representation of a large tuple with large amounts of attributes has not been proposed. The SGMM module adopts the positional B-tree to represent a large tuple. In this case, we use the attribute number of each attribute in a large tuple as the search key instead of the position of an element of a list. The storage structure of a large tuple adopted by SGMM is shown in Figure 7.

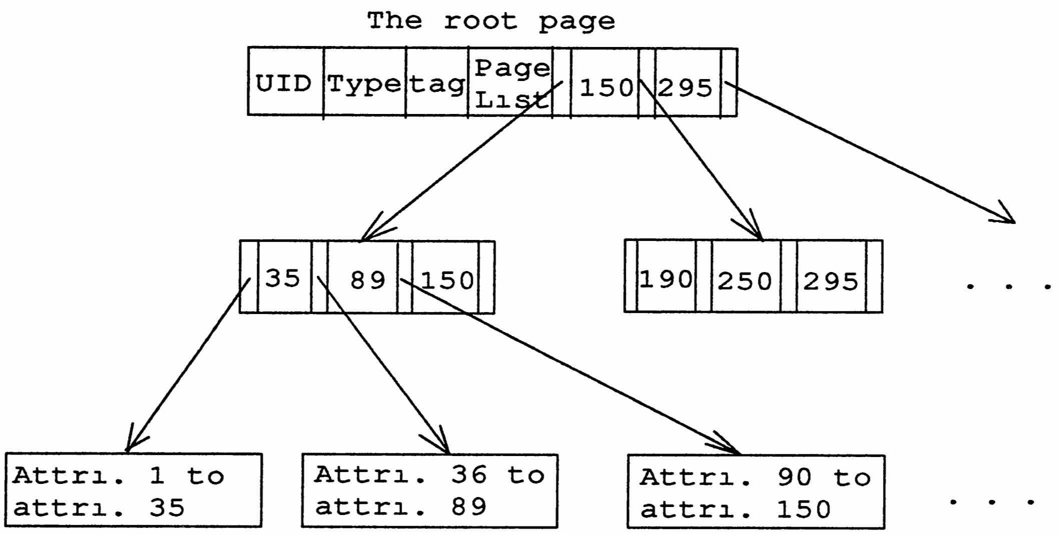


Figure 7. A Large Tuple Attribute

To illustrate the object representation on disk, the storage structure for a complex object memo is shown in Figure 8. The size of a complex attribute is smaller than a disk page, the instances of the complex attribute are organized as a linked list on the disk. For example, in Figure 8 the instances of the complex attribute "CC" of the complex object "memo" are linked together. On the other hand, if the size of a complex attribute is larger than the size of a disk page, the complex attribute is grouped according its type. In Figure 8, for example, the complex attribute "To" of the complex object "memo" is grouped as a B+ tree. Since the SMM understands the structures of complex attribute, or object, it serves the request for retrieving or updating any portion(s) of a complex object or a complex attribute or object. To achieve this, the SMM provides support for sequential single-element scan or range scan to a group. The Module also provides primitives for POIM and SGMM to update or insert an element into a group.

Transaction and Lock Manager Module (TLMM)

Data sharing is one of the most fundamental concepts and handled well in traditional database systems. The DB system provides data sharing capability while ensuring database integrity. To achieve this, the DB system must ensure that each transaction concurrently executed to be executed atomically. Another major concept in databases is to support database consistency in the presence of

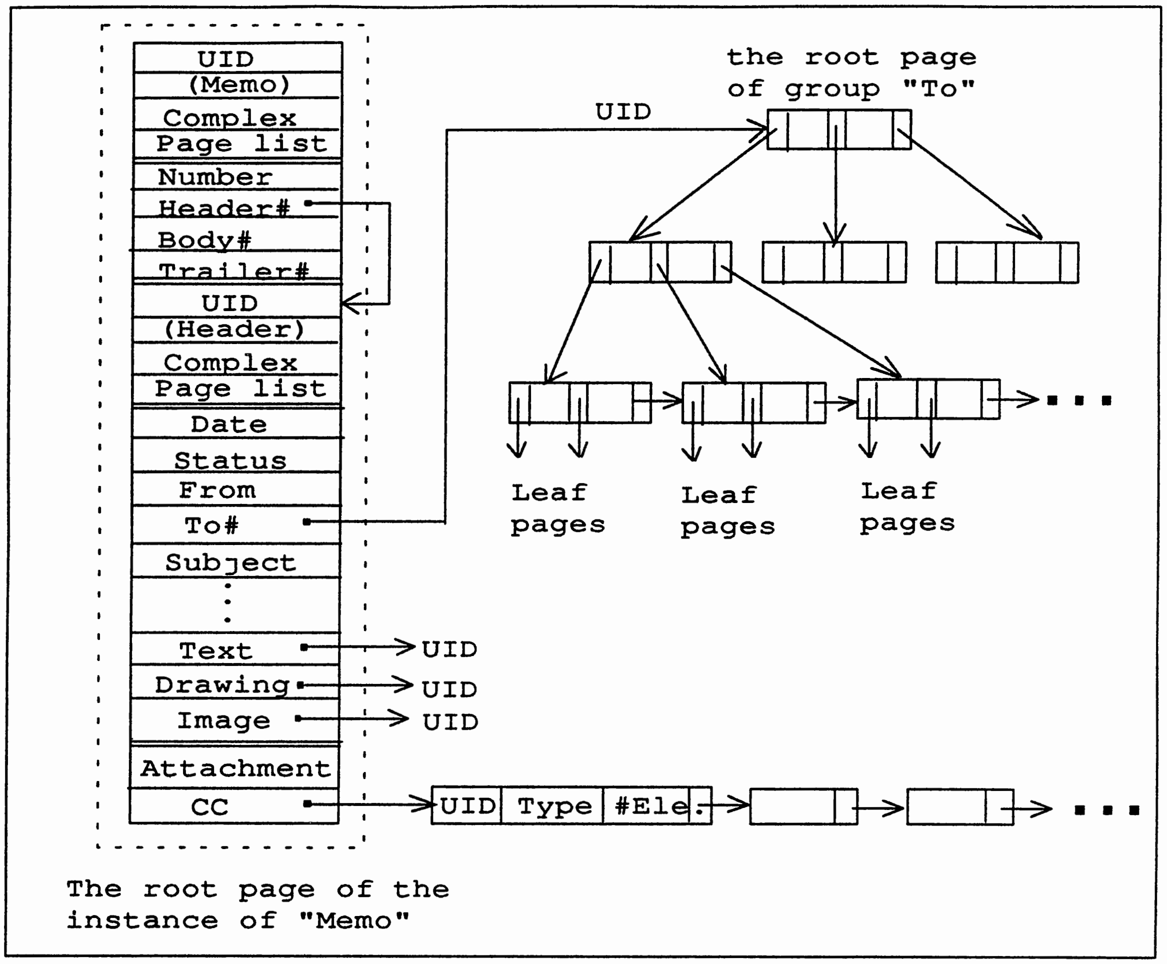


Figure 8. The Storage Structure of an Object

concurrency and system failures, either software or hardware. The latter is the notion of "strictness" and the former "serializability". The strictness or "recoverability and cascadelessness" ensures that the results of partially completed transactions will not be visible to other transactions [BER87}. The serializability ensures that each concurrently executed transaction accessing shared data does not interfere with each other. Therefore, database models should incorporate concurrency control and recovery mechanism to enforce some protocols such as locking protocol to achieve above requirements.

Operation Scheduling Abstraction

The traditional concurrency control and recovery techniques such as locking and logging have a good foundation. So, the transaction and recovery manager module (TRM) of the IPOM adopts the locking and logging schemes from traditional DB systems to support concurrency control and recovery in multi-user database environment. This module is illustrated for the sake of completeness of IPOM design and to show its relationship with other modules. To ensure serializability, operations issued by transactions are scheduled based on the "strict two-phase locking protocol". As shown in Figure 9, the operation scheduling abstraction can be described as follows:

1) The transaction manager (TM) receives the operation requests from the high-level, persistent object interface module (POIM). The primitives of the POIM are called by high-level software such as query processing algorithms, programming language operations, or end user's queries. These primitives rely on the underlying system's persistent mechanism to map the UID of the object to be read or written to its location on disk. There are only two database operations, read() and write, to be issued to the TM. The run-time system also can issue transaction operations based on the

behavior of the user's application programs or users themselves. The transaction operations supported are Trans_Begin (Buffer-strategy), Trans_Commit(), and Trans_Abort().

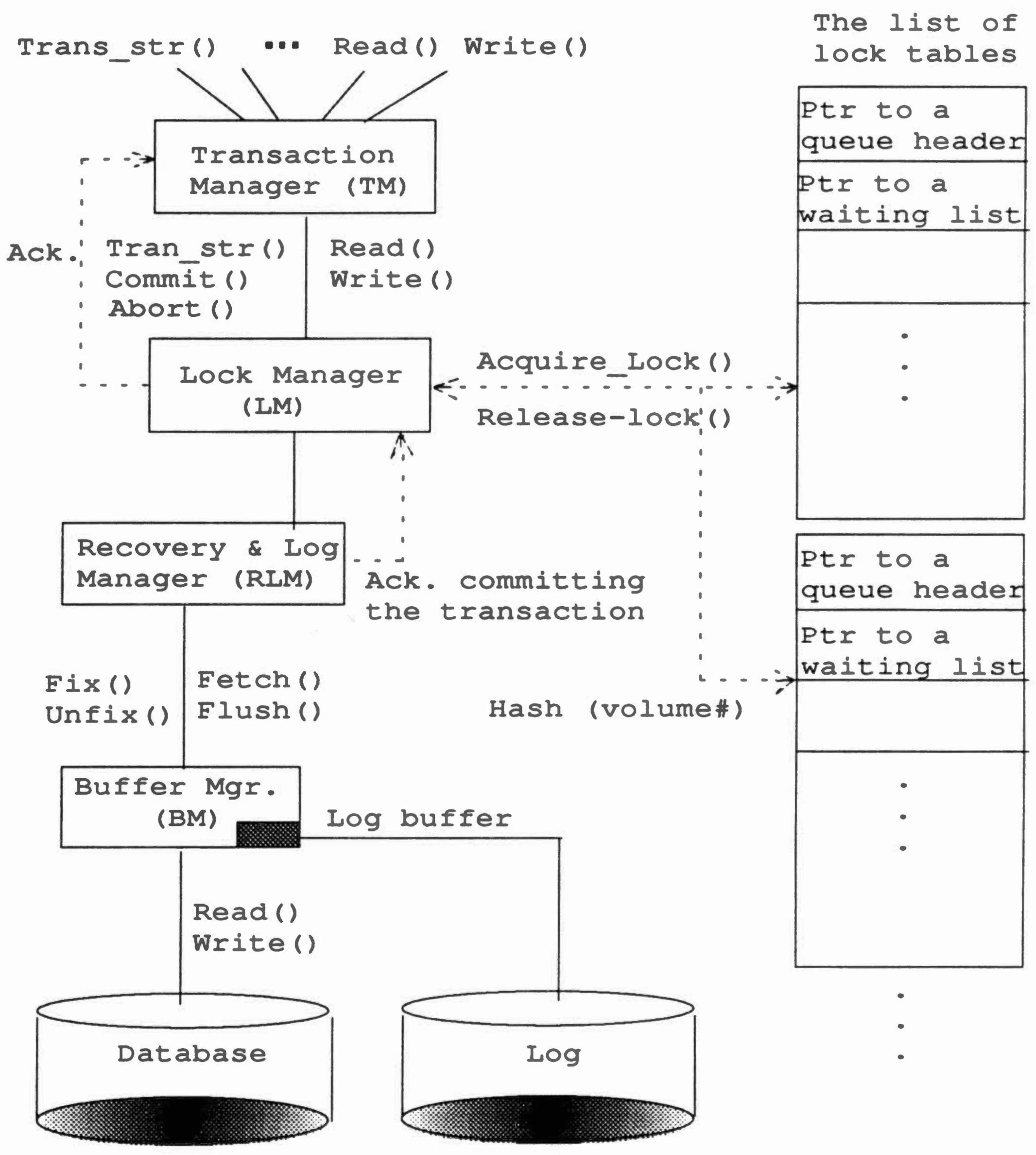


Figure 9. The TLM and RLM Modules

2) The TM performs required preprocessing and adds an appropriate transaction identifier to each operation. When it receives a database operation, it sends the appropriate lock request for the database operation to the lock manager (LM); when it receives a transaction operation (Trans_Commit or Trans_Abort), it requests LM to release all locks (logical locks) held by the transaction.
3) The LM is responsible for maintaining a set of lock tables, which shows the locks that each active transaction holds or is waiting for. It also provides locking(Trans_ID, OBJ, Mode) and unlocking (Trans_ID, OBJ) operations for objects. The OBJ can be an object, a group, or a storage unit. The LM either accepts or rejects the lock request from the TM according to the lock compatible and lock table information. If the lock request does not conflict with any lock hold by another transaction, the LM accepts the lock request and sends the accepted database operation to the recovery manager. Otherwise, it reports rejecting the lock request to the TM and puts the lock request into its appropriate lock waiting list.
4) The recovery and log manager module (RLMM) is responsible for transaction commit and abortion. It is also responsible for initiating the database operations upon receiving them from the TLMM. The

RLMM interface is defined by four procedures:

1. RLMM_Fetch(Trans_ID, OBJ): Fetch the object OBJ,
2. RLMM_Flush(Trans_ID, OBJ, PTR_OBJ1): Store the OBJ1 into OBJ,
3. RLMM_Commit(Trans_ID): Commit the transaction with transaction identifier Trans_ID, and
4. RLMM_Abort(Trans_ID): Abort the transaction with transaction identifier Trans_ID.

Complex Object and Index Locking

The TLM module also uses the granularity-hierarchy locking protocol proposed by Gray [GRA78] with strict two-phase locking protocol. This is to control all concurrent accesses and manipulations to complex objects. A complex object can be viewed as a structural collection of sub-objects. It is possible to form an abstract structure hierarchy by constructing the set of its sub-objects. For example, the complex object "memo" form a hierarchy of lock granules (a directed acyclic graph), as shown in Figure 10.

To minimize the locks to be set in accessing a complex object, it is better to set one lock for the entire memo object rather than one lock for each sub-object. There are five lock modes provided, that is, shared (S), exclusive (X), intention share (IS), intention exclusive (IX), and shared intention exclusive (SIX). According to [GRA78, BER87], for a given dag of locking hierarchy G, the locking protocol for the lock manager (LM) to set and release locks

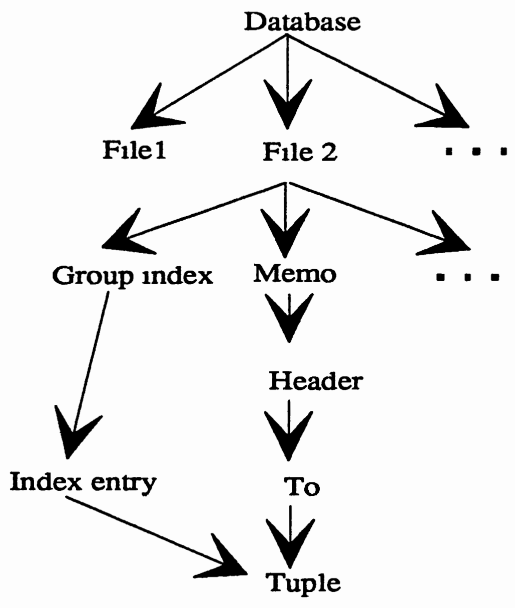


Figure 10. A Hierarchy of Granules

for each transaction, Trans_ID, is described as follows:

1) If a lockable granule g is not the root of G, then to set S or IS lock on g, Trans_ID must set an IS or IX on all direct ancestors of g first.

2) If g is not the root of G, then to set X or IX lock on g, Trans_ID must set an IX lock on all direct ancestors of g.

3) To write g, Trans_ID must have an X lock for some ancestor of g, for any path from the root of G to g. To read g, Trans_ID must have an S or X lock on some ancestor of g. Locks must be set in root-to leaf order.

4) Trans_ID must release all logical locks in leaf-to-root order before commit or in any order after transaction commit.

Since object locking granularity can be as low as the storage unit level, lock table contention may occur. When this situation takes place a number of lock requests may be blocked waiting for the lock semaphore for the lock table. We propose to use a list of lock tables, each for a single device and handled by one semaphore server. When a lock is requested by the TM, a hash function is applied using the volume number of an object identifier as hashing key. Then the semaphore server responsible for that volume takes care of the lock request. With such an approach, lock request congestion can be avoided.

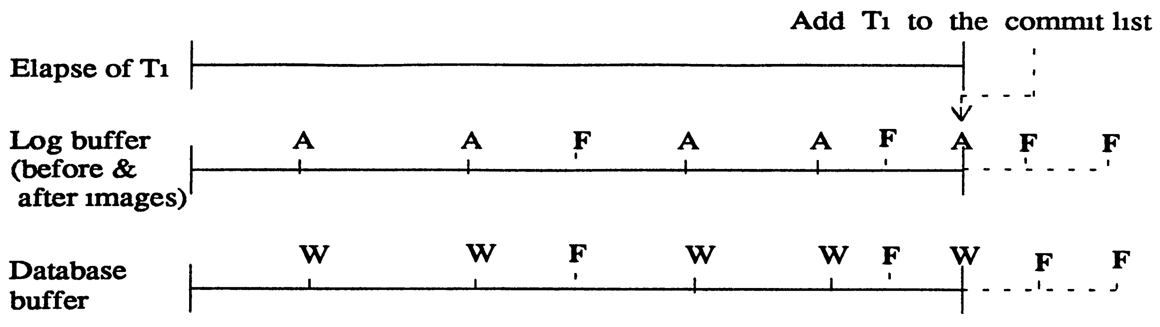
Though the two-phase locking protocol is used by the TLMM, it is not suitable for some search structures. The reason is that it does not take advantage of the predictable access patterns of search structures. So it is too restrictive to be suitable for search structures such as B tree or B+ tree. In addition, in advanced database applications, the access paths may become "hot spot" that would result from two-phase locking (2PL) hot resources [PAU87]. When hot spot objects occur a number of transactions may be blocked waiting for these hot spot objects. To increase the degree of concurrency, the non-two-phase index locking protocol (lock coupling or top-down locking) proposed by Bayer [BAY77] is used for searching and updating these indexing structures.

Recovery and Log Manager Module (RLMM)

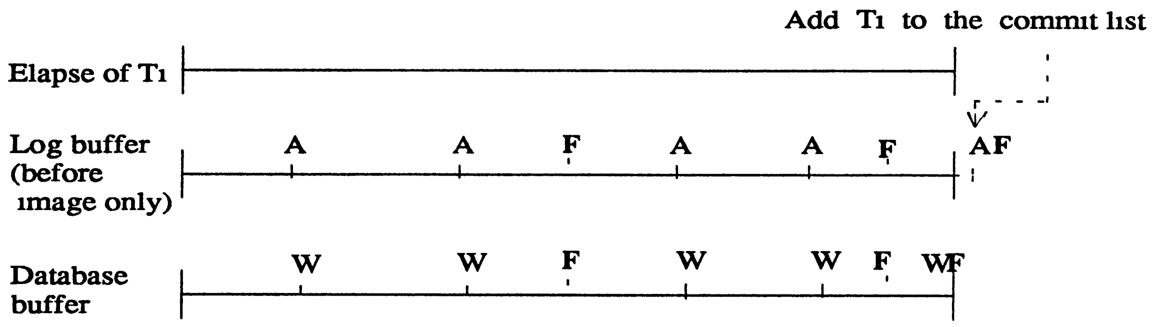
As for the recovery mechanism, "strictness" (or loosely speaking "recoverability") can be enforced at different levels. The recovery mechanisms of OODB's are similar to those of conventional DB systems [JOS91]. So OODB's designers do not concern themselves with this issue. Current commercial and prototype OODB's use variants of write-ahead-log (WAL) or shadowing mechanisms as the conventional DB systems do. The selection of recovery mechanisms may be based on the application-orientation concerned. According to [HAE83, BER87], there are four categories of recovery schemes:

1) STEAL/~FORCE (UNDO/REDO),
2) STEAL/FORCE (UNDO only),
3) ~STEAL/~FORCE (REDO only), and
4) ~STEAL/FORCE (~UNDO/~REDO).

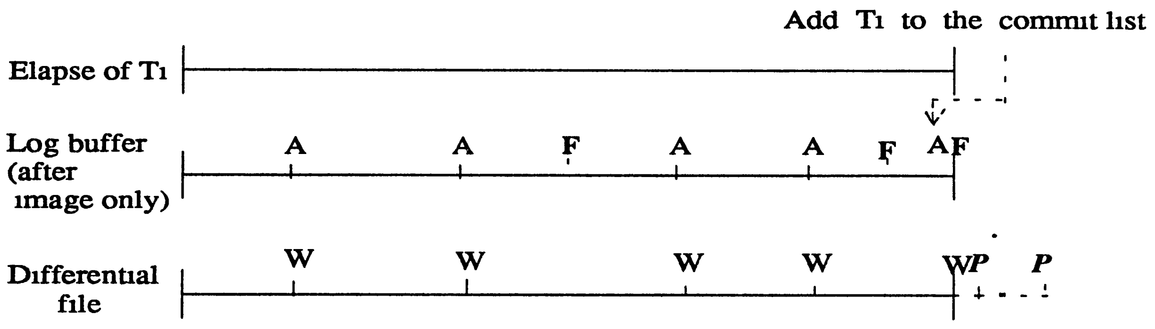
These schemes are illustrated pictorially, as shown in Figure 11. The STEAL/~FORCE scheme allows modified pages be flushed and propagated at any time, as shown in Figure 11(a). The buffer manager flushes modified pages according to the buffer occupation. This scheme complicates recovery processing since pages modified by incomplete transactions may be flushed to the stable storage. Thus, before image (for undo purpose) and after image (for redo purpose) loggings are required.



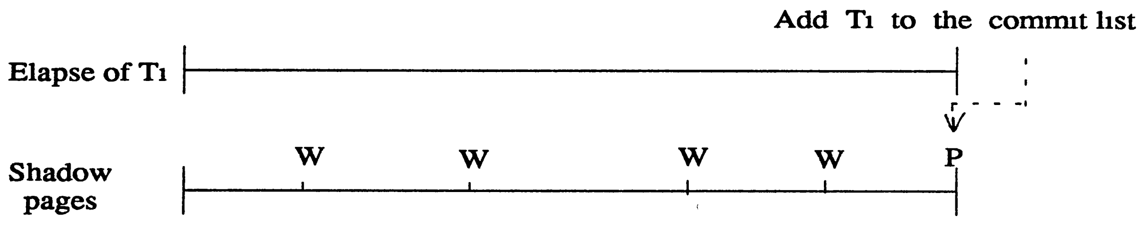
(a) The STEAL/~FORCE (UNDO/REDO) Scheme



(b) The STEAL/FORCE (UNDO/~REDO) Scheme



(c) The ~STEAL/~FORCE (~UNDO/REDO) Scheme



(d) The ~STEAL/FORCE (~UNDO/~REDO) Scheme

Legend: A. Append the log entry
W: Update in the database buffer, differential file, or shadow page
F: Flush the dirty page(s)
P: Propagation (not in-place)
P In-place propagation

Figure 11. The Recovery Schemes

The STEAL/FORCE scheme flushes all pages modified by transaction Ti before adding Ti to the commit list, as shown in Figure 11(b). That is, if transaction Ti is committed all pages modified by Ti are already in the stable storage. Thus, after image for redo purpose is not required, but before image for undo is needed in case that pages modified by incomplete transactions have been flushed into stable storage. The above two schemes also use an update-in-place approach where modified pages are flushed into the same blocks. Therefore flushing dirty pages and propagating control structures take place at the same time.

The ~STEAL/~FORCE scheme may requires redo but never requires undo. That is pages modified by uncommitted transactions are not flushed into the stable storage until the end of transaction. To keep all dirty pages of uncommitted transactions in the database buffer, a very large database buffer would be required. The alternative is using a "differential file" that records all modified pages. Then, propagation can be repeated as often as wished, as shown in Figure 11(c).

The ~STEAL/FORCE scheme is to avoid redo and undo operations. This requires that none of the pages modified by transaction Ti can be flushed into the stable storage before Ti is committed and all of these pages must be flushed into the stable storage by the time Ti is committed. To achieve the above goal, the shadow pages are needed to preserve the old state of the materialized database and all modified pages are written into their new blocks (not update-in-place). The major drawback of this scheme is that it potentially destroys the physical grouping (clustering) that have existed in the database.

In pragmatic sense, the last scheme is not appropriate for IPOM since it violates the principle of grouping complex attributes (objects) for total-retrieval of complex objects. The ~STEAL/~FORCE (REDO only) scheme is not adopted in IPOM since a very large database buffer is required (for long-duration update transactions). However, supporting rich data types in OODB's can influence their recovery mechanisms. For example, in case that multimedia objects such as texts, graphics, and bitmaps are supported, the cost of logging the before and after images of every changed multimedia object is expensive. Therefore, for implementation simplicity, we prefer the STEAL/FORCE (UNDO only) scheme to the others.

The RLMM is responsible for the logging of changes to objects (create, delete, and update) within a transaction. It keeps only the UNDO log (before image) of transaction Ti in the log buffer when Ti is active. When transaction Ti commits, the RLMM sends FLUSH(Ti) operation to the buffer manager to flush the changes to objects within transaction Ti to disk. Then the RLMM appends the Ti to the commit list (appends the commit record to the log buffer). The RLMM then forces the log to the disk and acknowledges the transaction commit to the TM.

It is not necessary to log every change to "hot spot data", index pages such as B tree or B+ tree search structures everytime during update operations on these structures. Thus, a special log mechanism is needed to avoid excessive logging. The traditional operation-oriented (logical) log [BER87] can be used to avoid logging every affected change to such search structures in case that a page split or concatenation is necessary to maintain the search structure invariants. However, undo and redo procedures to reverse the insertion and deletion operations must be provided in case of transaction abort.

Buffer Manager Module (BMM)

The buffer manager module (BMM) is one of the most important modules in the IPOM. It maintains a sufficiently large database buffer pool of pages as the final destination of objects to be processed except for "cache strategy transactions". In the IPOM model, it is involved at the locating and fetching phase of the persistence and data sharing process. When an object fetch request is issued to the buffer manager module (BMM), the buffer manager locates the desired page in the database buffer pool; otherwise it fetches the desired page from the disk if there is a buffer fault. It also enforces replacement policies specified by higher level softwares, e.g. the query processing routines or clients (clients can specify the Keep/Discard replacement policy in the transaction attribute when Trans_Begin()

command is explicitly coded in a user's application programs). In addition to specifying the buffer replacement policy, the application programmer can specify which buffering interface (uncopy-based or copy-based) to use to retrieve a large complex object into the user's address space. This is to avoid performance problems with excessive interface crossing between application programs and the IPOM. We will describe this problem below.

Cache Strategy Transactions

According to [STO81, DEU90], it is expensive to copy and translate every object from the database buffer pool into the application address space and later translate and perform a copy back. This overhead is the side-effect of environments such as heap-based programming and traditional computational-intensive environment due to copy-based interfaces in traditional file systems. The copy-based and uncopy-based interfaces are illustrated in Figure 12.

However, sometimes there are some computation-intensive applications in advanced database applications that require intensive computation on some objects. In object-oriented database systems with noncopy-based buffer interface such as EXODUS/E, the cost to call persistent object storage interfaces excessively is high. It is inefficient to repeatedly fix an already resident page when an object is referenced frequently in a program loop. With copy-based buffer interfaces, most heap-oriented database programming languages such as Smalltalk-based or CLOS-based database programming in Gemstone and ORION do not have this problem.

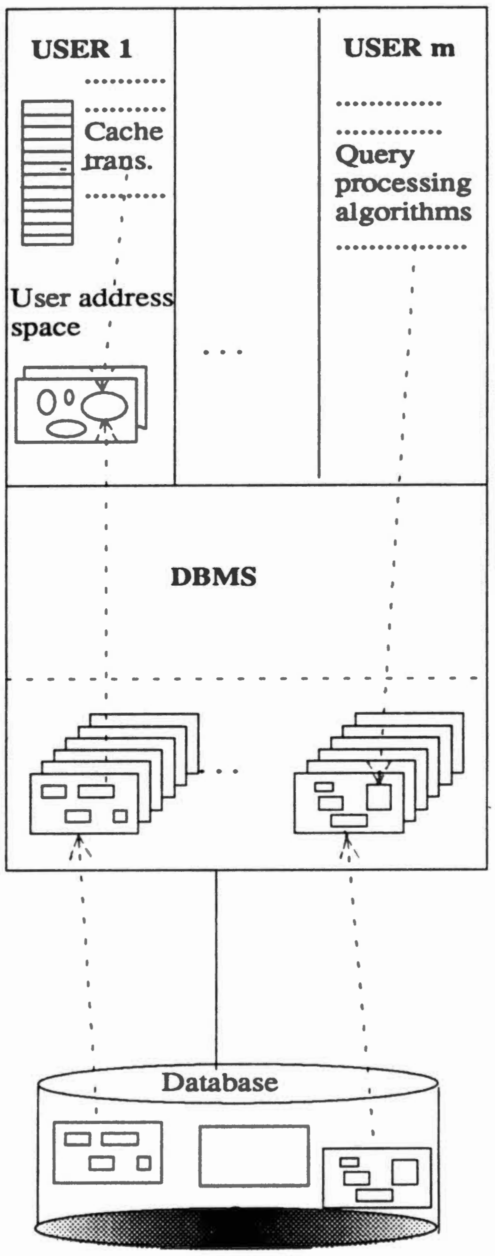


Figure 12. The Copy-base and Uncopy-base Interfaces

The objective of a copy-based interface is to avoid fixing (setting a physical lock on the page which the object being accessed or manipulated is resident in) a large number of pages in the database buffer pool during the potentially long-duration of computation. Therefore, on the one hand we want to reduce the CPU costs of copying objects from the database buffer pool into the application address space and delay propagating updated pages into the database buffer pool. On the other hand, we want to avoid excessive calls to fixing page frames in the database buffer pool that arises from repeatedly referencing the same object in a program loop.

Given the above seemingly conflicting objectives, we propose a cache strategy to solve the above problem. The buffer manager module retains the single-buffer (uncopy-based) scheme as the default scheme. It also allows a copy-based scheme, the cache strategy, to be used by explicitly specifying it in the variable declaration. The scenario of a cache strategy is illustrated in Figure 13. This can be accomplished by using the embedded object cache construct provided by the underlying object-oriented database programming language (OODBPL) to declare the object to be cached. The handle of the object to be cached must be declared with the keyword CACHE, for example, **CACHE** OBJ_TYPE *obj_handle. Then the preprocessor automatically enables the object initiation routines to generate the attribute part of a complex object and issues calls to the POIM to

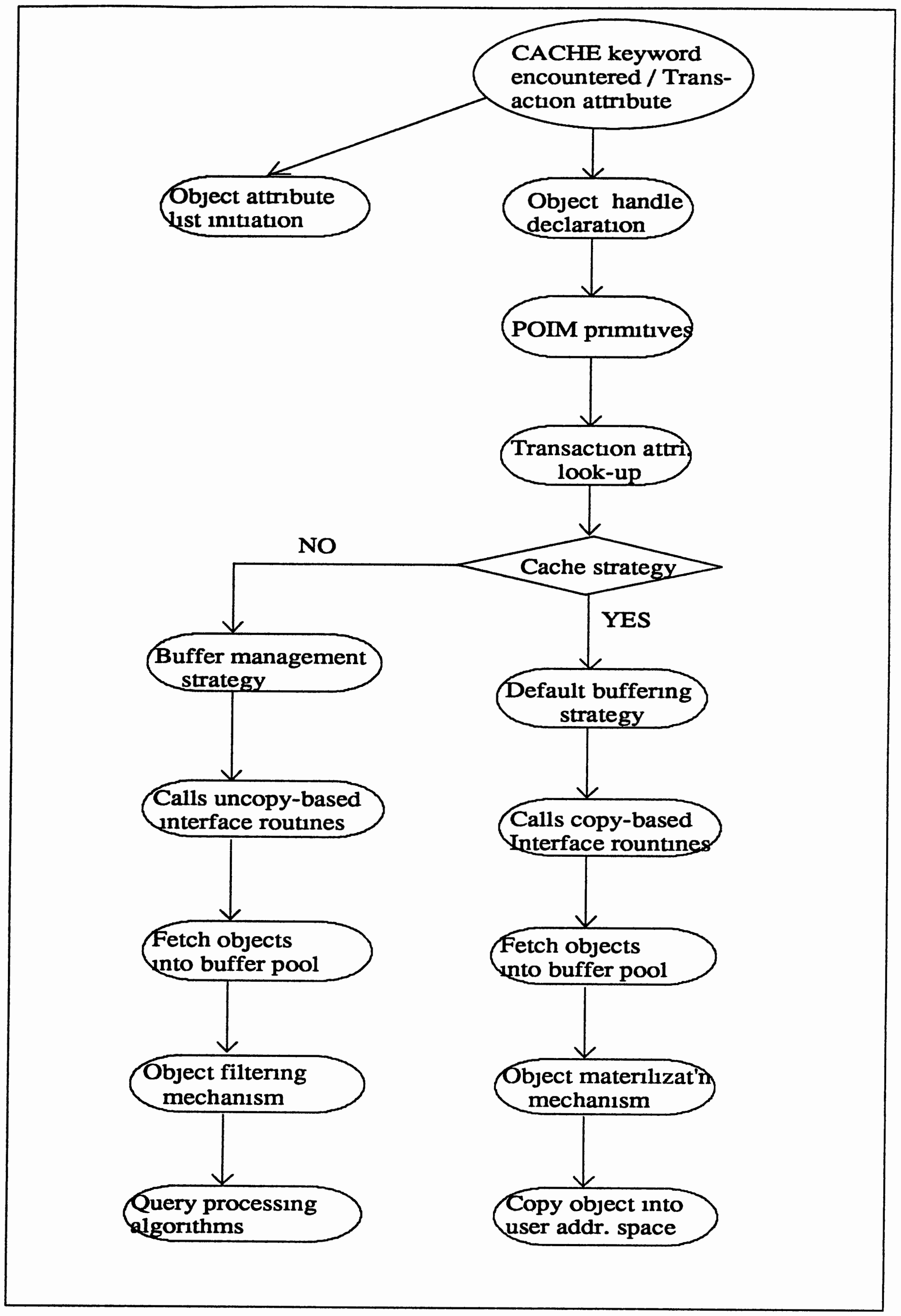

CACHE keyword encountered / Transaction attribute
Object attribute list initiation
Object handle declaration
POIM primitives
Transaction attri. look-up
Cache strategy
NO
YES
Buffer management strategy
Calls uncopy-based interface routines
Fetch objects into buffer pool
Object filtering mechanism
Query processing algorithms
Default buffering strategy
Calls copy-based Interface rountines
Fetch objects into buffer pool
Object materilizat'n mechanism
Copy object into user addr. space

Figure 13. The Cache Strategy

retrieve the entire complex object. The buffer pool to stage the objects to be cached will be unfixed after the desired objects are copied into the application address space. When the cached objects are retrieved and copied into the user address space, they are locked in read mode. When the cache strategy transaction is to be committed, the changes to any object must be first written to the log before they are written into the disk.

The Buffer Replacement Scheme

In the traditional database systems, query optimizations frequently use sophisticated join that may span several tables. As such the buffer replacement scheme of a database system plays an important role in query optimization processing based on the disk I/O cost. However, in object-oriented database systems, the traditional join approach based on matching attribute value is less important since a complex object can be viewed as a pre-computed join. Rather the object access via reference chains is dominant and the indexing structures are to maintain the information about the reference chains. Therefore, in OODB systems the reusage patterns of indexing in optimizing query processing tends to be more straightforward than those of traditional DB systems.

However, B trees or B+ trees will be used frequently in the databases as base index structures. The "keep-the-root-strategy" or "keep-the-highest-levels-strategy" replacement policy is identified to be useful [FOL87]. Therefore, high-level index pages are better to be ignored by the buffer pool replacement policy (usually, the LRU or MRU replacement scheme). In this way either temporarily or permanently hot pages (e.g., index pages used often within a transaction or root index pages of B+ trees that are always hot) are kept in buffer pools as long as possible when they are not physically locked.

We propose a local buffering scheme with simple hint, either keep or discard. The original local buffer allocation and replacement algorithm is proposed by Sacco [SAC86, pp 489-490]. This algorithm does not take into account the difference between "hot spot" resources and regular data. The simple "KEEP/DISCARD hint" scheme is added to the algorithm, as shown in Figure 14 and Figure 15. When buffer pages are allocated to the transaction that requests buffer pages, "the KEEP hint" can make these pages be put into the top of the local LRU stack. Thus these page will remain in the local LRU stack as normal LRU replacement scheme. When the "DISCARD hint" is issued at the time that buffer pages are requested and allocated by the buffer manager, these buffer pages will be put into the bottom of the local LRU stack. In this case, these pages will remain in the LRU stack for a short time and are ready to be replaced. The size of the buffer pool to be allocated is either specified by the client or from the query evaluation routines similar to the "hot-set size" in the hot-set model

```
   PROCEDURE  Alloc_Replace_Algorithm (
        TID, Buffer_strategy, Page_Request )

   VAR  TID_Stack_List, Free_Buffer_List,

   BEGIN
1      IF (TID is not in TID_Stack_List) DO
       BEGIN
             PUT TID to an empty TID_Stack_list slot,
             SET TID_Request_page = Page_Request;
             SET TID_no_Page_Alloc = 0,
       END   /* End of 1 */

2      IF (TID is already in the TID_Stack_List) DO
       BEGIN
2.1        IF (TID_Page_Request > TID_No_Page_Alloc ) DO
           BEGIN
2 1.1          IF ( page is in the Local LRU stack) DO
               BEGIN
                   IF ( Buffer_strategy != DISCARD ) THEN
                        PUT page to the top of the local LRU stack,
                   ELSE
                        PUT page to the bottom of local LRU stack,
               END      /* End of 2 1 1 */

2 1 2          IF (page is in the Free_Buffer_List ) DO
               BEGIN
                   IF (Buffer_strategy != DISCARD ) THEN
                       PUT page in the top of the local LRU stack,
                   ELSE
                       PUT page to the bottom of the local LRU stack;
                   UNLINK it from the Free_Buffer_List,
                   TID_NO_page_Alloc += 1,
               END       /* End of 2 1 2 */

2 1 3          IF (page fault occurs) DO
               BEGIN
2 1 3 1            IF (Free_Buffer_List = empty) DO  /* Replacement */
                   BEGIN
                       IF (Buffer_Strategy != DISCARD) THEN
                           PUT the bottom page in the local LRU stack
                           into the top of the local LRU stack,
                       ELSE
                       KEEP the bottom page as repacement page;
                   END    /* End of 2 1 3 1 */

2 1 3.2            IF (Free_Buffer_list != empty ) DO
                   BEGIN
                       IF (Buffer_strategy != DISCARD ) THEN
                           PUT the bottom page of the Free_Buffer_List
                           into the top of the local LRU stack,
                       ELSE
                           PUT the bottom page of the Free_Buffer_List
                           into the bottom of the local LRU stack,
                       TID_No_page_Alloc += 1,
                   END    /* End of 2 1 3 2 */
               END        /* End of 2 1 3 */
           END            /* End of the 2 1 */
                                                  (Cont')
```

Figure 14. The Buffering Algorithm with Simple Hint

```
2.2     IF (TID_Page_Request <= TID_No_Page_Alloc ) DO
        BEGIN
2.2.1       IF ( page is in the Local LRU stack) DO
            BEGIN
                IF ( Buffer_strategy != DISCARD ) THEN
                    PUT page to the top of the local LRU stack,
                ELSE
                    PUT page to the bottom of local LRU stack,

            END   /* ENd of the 2 2 1 */

2.2.2       IF (page is found in the Free_Buffer_List ) DO
            BEGIN
                PUT bottom page of the local LRU stack
                into the top of the Free_Buffer_List,

                IF (Buffer_strategy != DISCARD ) THEN
                    PUT found page in the top of the local LRU stack,
                ELSE
                    PUT found page to the bottom of the local LRU stack,
                UNLINK found page from the Free_Buffer_List,

            END    /* End of the 2.2.2 */

2.2.3       IF (page fault occurs) DO   /* Do replacement */
            BEGIN
                PUT the bottom page in the local LRU stack
                into the top of the Free_Buffer_List,

                IF (Buffer_Strategy != DISCARD) THEN
                    PUT the bottom page of the Free_Buffer_list
                    to the top of the local LRU stack,
                ELSE
                    PUT the bottom page of the Free_Buffer_List
                    to the bottom of the local LRU stack,

            END      /* End of the 2 2 3 */
        END          /* End of the 2 2 */
    END              /* End of 2 */

END                  /* End of the procedure */
```

Figure 15. The Buffering Algorithm with Simple Hint (Con't)

[SAC86]. The elegance of this scheme is that a hint can be given when an access pattern is known by either the high level software or the user and does not incur the complexity of the buffering algorithm.

CHAPTER V

DISCUSSION AND COMPARISON WITH OTHER MODELS

The Features of the IPOM Model

In this Chapter, a comparison between the design schemes of the IPOM and that of other storage models is presented. The design decision of the IPOM is made by first surveying related models of storage systems, second identifying the pros and cons of these models, and then investigating possible design solutions to solve existing problems. Based on this design approach, the design of the IPOM is motivated by the following observations:

1) limited modeling power: Some of current OODB systems suffer from limited modeling power because they do not directly support large and complex objects of arbitrary levels.

2) indexing internally: Some of current OODB systems do not support indexing objects internally. This may contribute to the side-effect of supporting only simple objects in their models.

3) partial-retrieval capability: None of current OODB systems provide the flexibility for both "total-retrieval" and "partial retrieval" of large complex

attributes (objects) for efficiency.

4) inefficient file systems: Some of current OODB systems suffer from poor performance due to inefficient storage systems based on traditional UNIX™ file systems.

5) copy-based interface: Most of OODB systems use copy-based interfaces [STO81] that affect performance dramatically. It is expensive to copy and translate every object in the database buffer pool into the user address space.

6) buffering schemes: The buffer allocation and replacement schemes (e.g. the global LRU replacement scheme) of most OODB systems do not consider the access patterns of some "hot spot objects" such as indexing structures.

7) limited application domains: Most of current OODB systems limit their application domain to load/work/save (LWS) applications only. Query processing capability is either not supported or performed poorly. The latter is due to excessively copying of every object between the database buffer pool and query processing algorithms.

The IPOM storage model proposed differs from other models of object storage systems mainly in its schemes to solve the above problems. These schemes include the following:

1) direct indexing support for complex attributes,

2) storage structures supporting "total-retrieval" and "partial retrieval" of complex attributes including tuples with large amounts of attributes,
3) uncopy-based buffer interfaces with cache strategy option (selectively copy-based interfaces),
4) local allocation and replacement algorithm with simple hint (KEEP/DISCARD) scheme.

A summary of comparison between the IPOM and other models is given in Table 2. The rationale for IPOM to adopt above schemes are discussed (largely analysis) in the following sections.

Architecture

The design of the architecture of the integrated persistent object manager (IPOM) is inspired by those of system R [AST76], EXODUS [CAR89], O2 [DEU90], ORION [KIM90], Zeitgeist [FOR88]. These can be summarized as follows:

1) the research storage system (RSS) of the system R:
   a. the relational storage interface (RSI) provides simple record-at-a-time operators on relations (the SQL cursor concept),
   b. the transaction manager (TM) provides transaction management concepts (transaction consistency and locking, recovery), and
   c. the database manager (DM) provides cache Management, mapping persistent database objects into main memory objects.

2) the storage manager of EXODUS provides support for buffering, concurrency control and recovery, and interfaces for manipulation of simple objects and large objects.

3) the storage subsystem and transaction subsystem in ORION provide support for persistent object management and data sharing:

   a. the transaction subsystem consists of deadlock manager, lock manager, recovery manager, and log manager.

   b. the storage subsystem consists of access manager, object buffer manager, page buffer manager, and storage manager.

4) the persistent object store (POS) of Zeitgeist consists of client interface, object translation mechanism, the transport subsystem, the transaction subsystem, and the storage server(s).

5) the storage system of O2 is an extension of the Wisconsin Storage System (WISS) which provides support for persistent structures, transaction, and write-ahead log for recovery.

The latter three models of storage have the same feature that employs a translation mechanism to translate the database objects into in-memory objects.

TABLE II

COMPARISON OF IPOM AND OTHER MODELS

SYSTEM	Persistent Object Interface	Complex Attribute Indexing	Buffer Allocation & Buffering Schemes		Concurrency Control	Recovery Scheme
IPOM	Complex & set-oriented	Internally grouping indexing	Local with a keep/ discard hint	Single buffer copy & cache strategy	2PL	Undo log
EXODUS/E	Simple & large objects	No	Local	Single buffer non-copy	2PL	Undo/Redo log & shadowing
ObjectStore	Simple & complex objects	No large tuple indexing	Global	Memory mapped scheme	2PL	Write-ahead log
Gemstone	Simple & complex objects	Collection indexing	Global	Dual buffers copy-based	Optimistic & pessimistic (2PL)	Shadowing mechanism
O_2	Simple & complex objects	No large tuple indexing	Global	Dual buffers copy-based	2PL	Redo log
Arjuna	Simple objects	No	Global	Dual buffers copy-based	2PL	No
ORION	Simple & complex objects	Single class indexing	Global	Dual buffers copy-based	Extended 2PL	Undo log & shadowing
Zeitgeist	Simple objects	No	Global	Dual buffers copy-based	2PL	No
ONTOS	Simple & complex objects	No large tuple indexing	Global	Dual buffers copy-based	Optimistic & pessimistic (2PL)	Checkpoint
ENCORE/ Observer	Simple objects	No	Global	Dual buffers copy-based	Extended 2PL & comm. modes	No

Storage and Grouping Module

It is obvious that storage models that do not support complex objects will have limited modeling power. For example, EXODUS/E, ENCORE/Observer, Arjuna, and Zeitgeist, support only the simple object concept. As a consequence, application programmers must take substantial coding efforts to simulate the complex object notation. At the same time, application programmers have to take care of the internal structures of complex objects and indexing details related to these complex objects in these systems.

Without directly indexing on complex states, some models also suffer from waste of buffer space and I/O bandwidth when retrieving the whole large complex state within a complex object. We think support for direct indexing on complex attributes is important due to the following reasons:

1) First, such a scheme makes it possible to retrieve the parts of a complex object that are actually needed for applications without incurring unnecessary disk I/O bandwidth and main memory consumption;
2) Second, since the index pages and data pages can be distinguished by the storage system, it is possible to employ more efficient concurrency control protocol such as non-two phase concurrency control mechanisms;
3) Third, in advance database applications, a large

tuple with large number of attributes is possible, direct indexing can benefit retrieval and update of any attribute of such structure.

The instances of a set is represented with a linked list structure on disk in ORION. With large sets of instances of a complex attribute, retrieving a specific instance or some range of instances in the set's keyed indexing structures such as B+ trees is more efficient than it would be with only linked list structures which are employed in ORION. None of the other models address or handle the large tuple case. IPOM supports indexing on all kinds of complex attributes including large tuples.

The Buffer Manager Module

One of the major objectives of the buffer manager module of the IPOM is to avoid copying every object from the database buffer pool and the user application space. We adopt the uncopy-based buffer interface, as illustrated in Figure 16, from the research system R [AST76]and EXODUS/E [CAR89] with extension to support complex objects and the cache strategy (selective copy-based interface). The copy-based interface is illustrated in Figure 17. There are three major drawbacks to using the copy-based interface (dual buffer scheme):

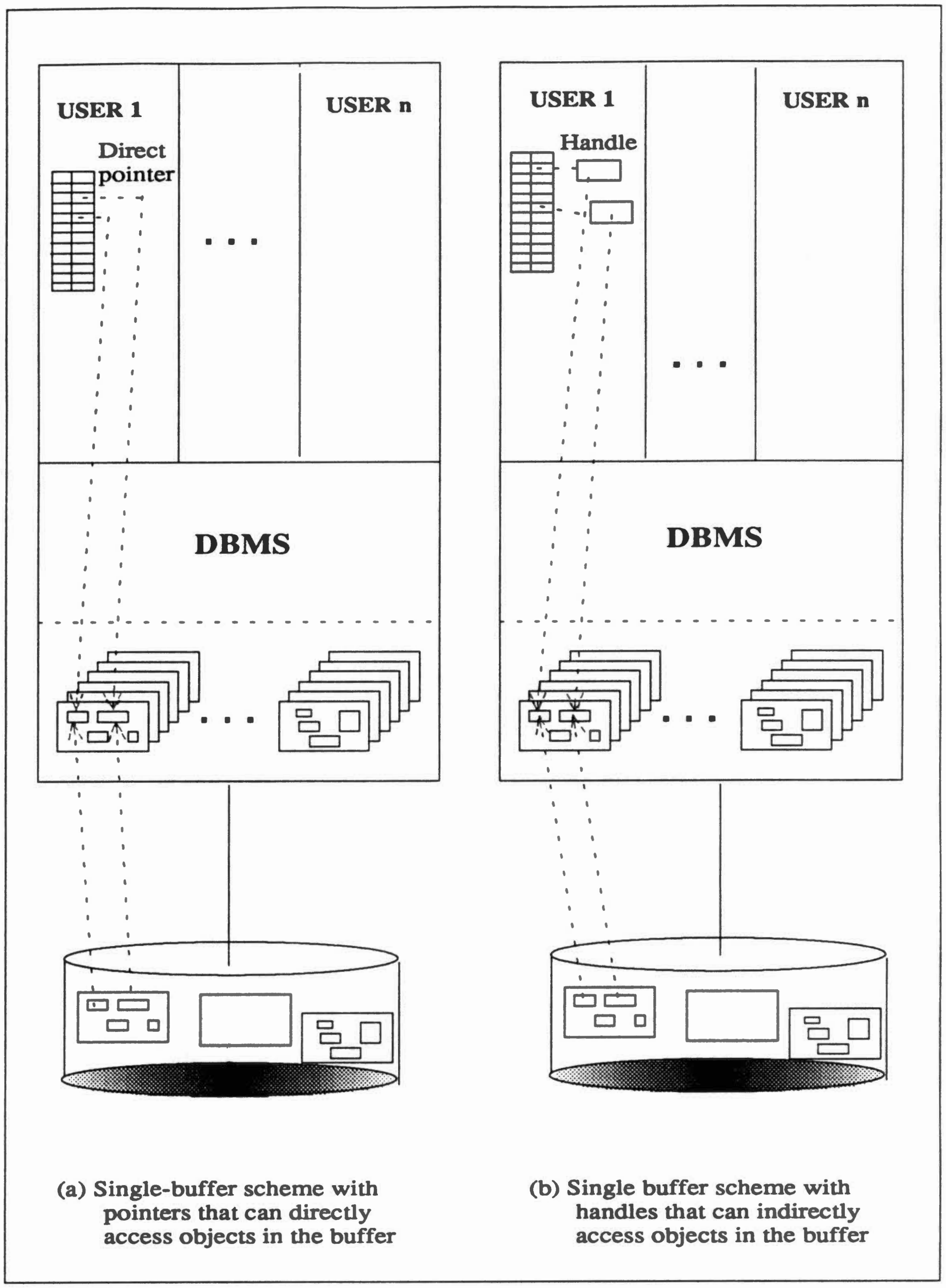

USER 1
Direct pointer
. . .
USER n
DBMS
. . .
(a) Single-buffer scheme with pointers that can directly access objects in the buffer
USER 1
Handle
. . .
USER n
DBMS
. . .
(b) Single buffer scheme with handles that can indirectly access objects in the buffer

Figure 16. The Single Buffer Scheme without Copy-based Translation

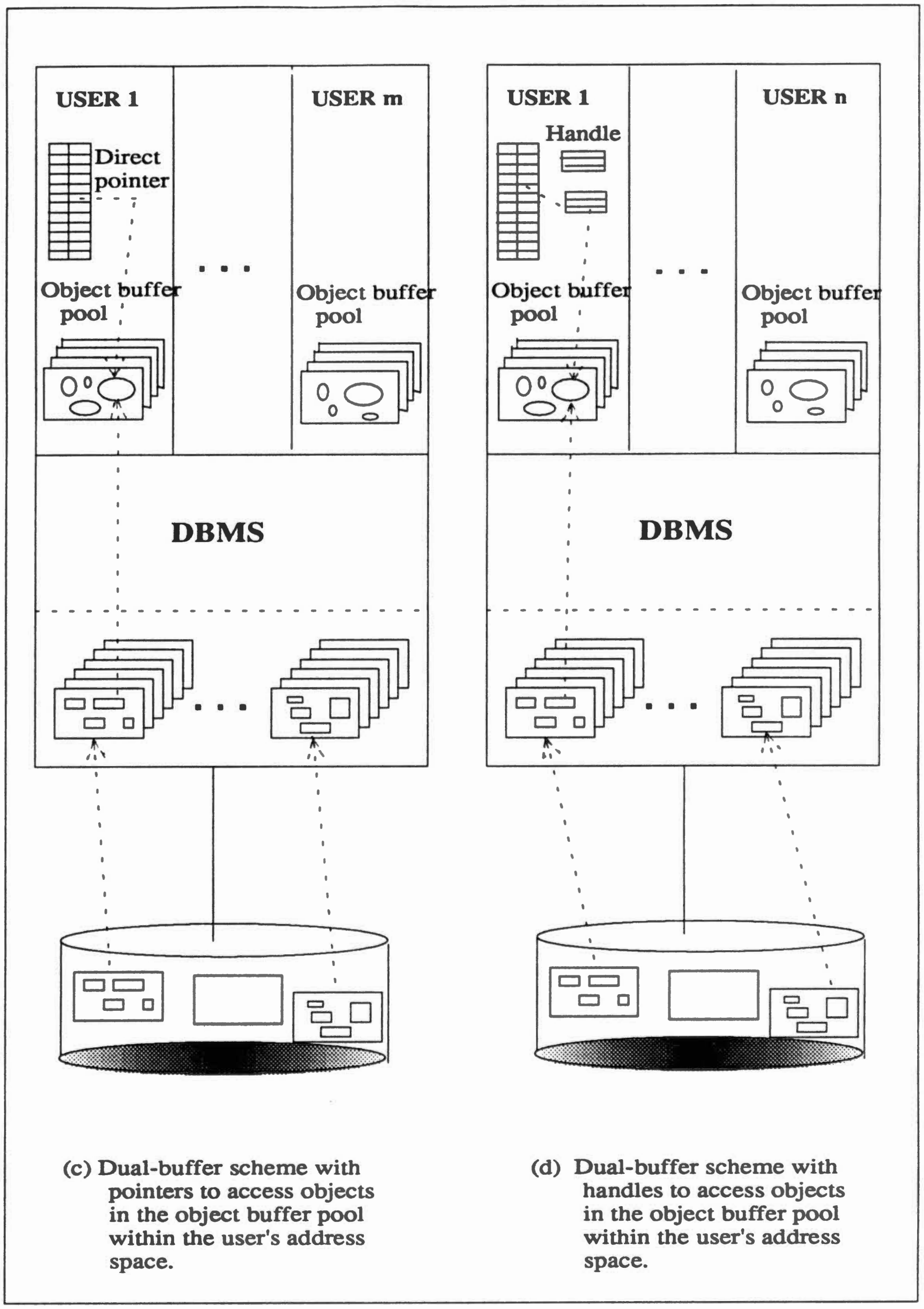

USER 1
USER m
Direct pointer
Object buffer pool
Object buffer pool
DBMS
(c) Dual-buffer scheme with pointers to access objects in the object buffer pool within the user's address space.
USER 1
USER n
Handle
Object buffer pool
Object buffer pool
DBMS
(d) Dual-buffer scheme with handles to access objects in the object buffer pool within the user's address space.

Figure 17. The Dual-buffer Scheme with
Copy-based translation

1) The cost of copying objects from the system buffer pool into the user application space is expensive. According to Stonebreaker [ST081], the cost of copying one byte (512 bytes) from the system buffer pool into the user cache is about 1800 (5000) instructions in PDP-11/70 running UNIX. Suppose a fast CPU with 50Mhz clock rate (the system can retrieve one instruction from the DRAM in 20 nanoseconds) is available, then it will take approximately 36 ms to copy one byte. This is about one disk I/O cost (average 25-50 ms). More precise simulation results also have been reported. For example, Kim [KIM88] reported that the cost of copying an object with a size of 30-150 bytes from database buffer pool to user address space is approximately the same as that of a disk retrieval.
2) The second major drawback of the dual buffer scheme as indicated in [KIM90], is that query evaluations must evaluate predicates twice, once in the object buffer pool, the other in the database. This makes the evaluation algorithms complicated because of the different object formats in the object buffer and database.
3) The third major drawback is the conversion cost (translate the on-disk format of the whole complex object into in-memory format). It loads a complex object into memory page by page. The incremental

transformation technique is not used in this case.

All other models except EXODUS/E use copy-based approach partially due to their heap-based programming environments. EXODUS/E does not support the complex object concept and suffers from the problem of excessive calls to its storage systems. IPOM not only extends uncopy-based interface with the complex object concept but also uses the cache strategy scheme to avoid unnecessary interface crossing.

Finally, the traditional global LRU allocation and replacement algorithm does not take into account database access patterns. Stonebreaker [STO81] argued that the LRU algorithm is not suitable as a database buffer replacement algorithm and that some form of advice from the database system is necessary. Our approach is to give this advice from the system software (query evaluation plans) or users. This hint is to KEEP the requested pages either on the top or in the bottom of the local LRU stack. This scheme not only makes buffer management more efficient but also does not add too much complexity to the buffering algorithm.

CHAPTER VI

CONCLUSIONS AND FUTURE WORK

In the past few years, OODB systems have received great popularity in database community. A number of new data models have also been proposed. Unfortunately, there is little agreement about what an object-oriented database system should be. Generally, OODB systems are criticized for doing a poor job in terms of performance [JOS89]. This is largely due to disk I/O, excessive interface crossing between application programs and DB systems, without partial-retrieval capability, and object translation cost. The object translation cost includes the copy cost between the database buffer pool and the object buffer pool in the user address space. These problems are related to the internal functionalities of an OODB system, that is, the design of the persistent object storage system.

In this thesis, a persistent object storage model, namely the integrated persistent object manager (IPOM) that can be integrated into OOPL environment, is proposed. The objective of IPOM design is to investigate a persistent object storage system that provides support for persistence, large and complex objects, efficient object management in the buffer, and data sharing. We believe that the following design factors can improve the performance of an OODB system: first, the efficient and selective retrieval of sets of complex objects; second, reduction of unnecessary copy cost between the database buffer pool and the user address space; third, an appropriate buffering scheme reducing buffer contention between hot spot resource and regular data with a simple hint algorithm. All these are pertinent to the design of an integrated persistent object manager (IPOM). The major contribution of this thesis is largely in its analysis of how some ideas are useful for solving some problems that arise in the context of the design of an OODB system. The future work is to do some simulation studies and subject to the results of these studies to implement a prototye of the IPOM that uses these design schemes. At the same time, the external functionality extension of the target OOPL such as programming constructs to support the generic model, the binding mechanism, and the user's interface also need to be investigated.

It should be pointed out while the design of the persistent object storage system is deemed important, there are some important performance related research topics. For example, query processing and selection of access paths in OODB systems is one of the important directions. Parallel processing in a multipropcessor environment is also an important performance issue. Another research topic is to investigate a single-level store scheme with potentially unlimited main memory to improve performance.

BIBLIOGRAPHY

[AND87] Andrew, T. and Harris, C. "Combining Language and Database advances in an Object-Oriented Development Environment," Procs. ACM OOPSLA'87, 430-440, (1987).

[AND91] Andrews, T., Harris, C., and Sinkel, K. "ONTOS: A Persistent Database for C++," in Gupta R. and Horowitz, E., editors, Object-Oriented Databases with Applications to CASE, Networks, and VLSI CAD, 387-406, Prentice-Hall, Englewood Cliffs, NJ, 1991.

[AGR89] Agrawal, R. and Gehani, N. "ODE (Object Database and Environment): The Language and the Data Model," Procs. ACM SIGMOD Conf. on Management of Data, 36-45, Portland, Ore., (May 1989).

[ATK83] Atkinson, M. P., Chisholm, K. J., Cockshott, W. P. and Marshall, R. "Algorithms for a Persistent Heap," Soft. Pract. Experience, 13, 259-271 (1983).

[ATW91] Atwood, T. M. "The case for object-oriented database," IEEE Spectrum, 44-47 (February 1991).

[AST76] Astrahan, M. M. et. al. "System R: Relational Approach to Database Management," ACM Trans. on Database Systems, 1(2), 97-137 (June 1976).

[BEM89] Bemstein, P. A. et al. "Future Directions in DBMS Research," ACM SIGMOD, 18(1), 17-26 (March 1989).

[BER91] Bertino, E. and Martino, L. "Object-Oriented Database Management Systems: Concepts and Issues," IEEE Computer, 33-47 (April 1991).

[BLO87] Bloom T. and Zdonik S. B. "Issues in the Design of Object-Oriented Database Programming Languages," Procs. ACM OOPSLA'87, 441-451 (October 1987).

[BUT91] Butterworth, P., Otis, A., and Stein, J. "The Gemstone Object Database Management System," CACM, 34(10), 64-77 (October 1991).

[CAR89] Carey, M. J., Dewitt, D. J., Richardson, J. E., and Shekita, E. J. "Storage Management for Objects in EXODUS," in Kim, W. and Lochovsky, F. H., editors, Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, NY, 341-370, 1989.

[CAR90] Carey, M. and Haas, L. "Extensible Database Management Systems," ACM SIGMOD Record, 19(4), 54-60 (December 1990).

[CHO85] Chou, H-T., Dewitt J. D., Kate, R. H., and Klug, A. C. "Design and Implementation of the Wisconsin Storage System," Soft. Pract. Experience, 15(10), 943-962 (October 1985).

[COC83] Cockshott, W. P. "Orthogonal Persistence," Ph.D. Thesis, University of Edinburgh, February 1983.

[COM79] Comer, D. "The Ubiquitous B-tree," ACM Computing Surveys, 11(2) 121-137 (June 1979).

[COP84] Copeland, G. and Maier, D. "Making Smalltalk a Database System," Procs. ACM SIGMOD Conf. on the Management of Data, 316-325 (1984).

[DAT86] Date, C. J. An Introduction to Database Systems. Volume 1, 4th ed., Addison-Wesley, 1986.

[DEU90] Deux, D. et al. "The Story of O2," IEEE Trans. on Knowledge and Data Engineering, 2(1), 91-108 (March 1990).

[DEU91] Deux, D. et al. "The O2 System," CACM, 34(10),34-48 (October 1991).

[DIX89] Dixon, G. N., Parrington, G. D., Shrivastava, S. K., and Wheater, S. M. "The Treatment of Persistent Objects in Arjuna," Computer J., 32(4), 323-332 (1989).

[EFF84] Effelsberg, W. and Haerder, T. "Principals of Database Buffer Management," ACM Trans. on Database Systems, 9(4), 560-595 (December 1984).

[ELM89] Elmasri, R. and Navathe S. B. Fundamentals of Database Systems. Benjamin/Cummings, CA, 1989.

[FOL87] Folk, M. J. and Zoellick B. File Structures: A Conceptual Toolkit. Addison-Wesley, 1987.

[FOR88] Ford, S. et. al. "ZEITGEIST: Database Support for Object-Oriented Programming," Proc. Second Int. Workshop on Object-Oriented Database Syst., Lecture Notes in Computer Science, Springer-Verlag, 23-42, 1988.

[GRA78] Gray, J. Notes on Database Operating Systems. Operating Systems: An Advanced Course. Spring-verlag, New York, (1979).

[GOL83] Goldberg, A. and Robison, D. Smalltalk-80: The Language and its Implementation. Addison-Wesley, Reading, MA 1983.

[HAN91] Hanson, E. N., Harvey, T. M., and Roth M. A. "Experiences in DBMS Implementation Using an Object-oriented Persistent Programming Language and a Database Toolkit," Procs. ACM OOPSLA'91, 314-328, (1991).

[HAE83] Haerder, T. and Reuter, A. "Principals of Transaction-Oriented Database Recovery," ACM Computing Surveys, 15(4), 287-317(December 1983).

[HOR87] Hornick, M. F. and Zdonik, S. B. "A Shared, Segmented Memory System for an Object-Oriented Database," ACM Trans. on Office Info. Syst. 5(1), 70-95 (January 1987).

[JOE87] Joel, E. R. and Carey, M. J. "Programming Constructors for Database system Implementation in EXODUS," Procs. ACM SIGMOD Conf. on Management of Data, 208-219 (May 1987).

[JOE89] Joel, E. R. and Carey, M. "Persistence in the E Language: Issues and Implementation," Soft. Pract. Experience, 19(12), 1115-1150 (December 1989).

[JOS91] Joseph, J. V., Thatte, S. M., Thompson, C. W., and Wells, D. L. "Object-Oriented Databases: Design and Implementation," Procs. IEEE, 79(1), 42-64 (1991).

[KAT86] Katzan H. Operating System: A Pragmatic Approach. Van Nostrand Reinhold Company Inc., 1986.

[KAT90] Kate, R. H. "Toward a Unified Framework for version Modeling in Engineering Databases," ACM Computing Surveys, 22(4), 375-408 (December 1990).

[KAZ88] Kazerooni-Zand, M. and Fisher, D. D. "Space-efficient Persistent B-tree," Procs. ACM Second Workshop on Applied Computing'88, Oklahoma, 295-318 (March 1988).

[KAZ89] Kazerooni-Zand, M. and Fisher, D. D. "Deletion on Persistent B-tree," Procs. ACM Workshop on Applied Computing'89, Oklahoma, 90-96 (1989).

[KEE89] Keene, S. E. Object-Oriented Programming in COMMON LISP, Addison-Wesley, Reading, MA, 1989.

[KHO86] Khoshaflan S. N. and Copeland, G. P. "Object Identity," Procs. ACM OOPSAL'86 Conf., 406-416 (1986).

[KIM88] Kim, W., Chou, H., and Banerjee, J. "Operations and Implementation of Complex Objects," IEEE Trans. on Software Eng., 14(7), 985-996 (July 1988).

[KIM89] Kim, W., Kim, K-C., and Dale, A. "Indexing Techniques for Object-Oriented Database," in Kim, W. and Lochovsky, F. H., editors, Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, NY, 341-370, 1989.

[KIM90] Kim, W. Introduction to Object-Oriented Databases. The MIT Press, MA, 1990.

[LAM91] Lamb, C., Landis, G., Orenstein, J., and Weinreb, D. "The Objectstore Database System," CACM, 34(10), 50-63 (October 1991).

[MAI89] Maier, D. "Making Database Systems Fast Enough for CAD Applications," in Kim, W. and Lochovsky, F. H., editors, Object-Oriented Concepts, Databases, and Applications, ACM Press, New York, NY, 573-582, 1989.

[MAI89] Maier D. Why Isn't There an Object-Oriented Data Model? Computer Science TR CS/E-89-002, Oregon Graduate Center, May 1989.

[MCL89] Mcleod, D. "1988 VLDB Panel on "Future Directions in DBMS Research: A Brief, Informal Summary," ACM SIGMOD Record, 18(1), 27-30 (March 1989).

[PAU87] Paul, H.B. et al. "Architecture and Implementation of the Darmstadt Database Kernel System," Procs. ACM SIGMOD Conf. on Management of Data, 196-207, 1987.

[PUR87] Purdy, A., Schuchardt, B., and Maier, D. "Integrating an Object Server with Other Worlds," ACM Trans. on Office Info. Syst. 5(1), 27-47 (January 1987).

[SAC86] Sacco G. M. and Schkolnick, M. "Buffer Management in Relational Database System," ACM Trans. on Database Systems, 11(4), 473-498 (December 1986).

[SHE92] Shen, T-C. and George, K. M. "A Taxonomy of Object-Oriented Database Systems Based on Persistence and Data Sharing," Procs. International Association for Computer Information Systems, New Orleans, August 1992.

[SIL91] Silberschatz, A., Stonebraker, M. and Ullman, J. "Database Systems: Achievements and Opportunities," CACM, 34(10), 110-120 (October 1991).

[STA84] Stamos, J. W. "Static Grouping of Small Objects to Enhance Performance of a Paged Virtual Memory," ACM Trans. on Computer Systems, 2(2), 155-180 (May 1984).

[STO76] Stonebraker, M., Wong, E., and Kreps, P. "The Design and Implementation of INGRES," ACM Trans. on Database Systems, 1(3), 189-222 (September 1976).

[STO81] Stonebraker, M. "Operating System Support for Database Management Systems," CACM, 24(7), 412-418 (July 1981).

[STR86] Stroustrup, B. The C++ Programming Language. Addison Wesley, 1986.

[WOE86] Woelk, D., Kim, W., and Luther, W. "Object-Oriented Approach to Multimedia Database," Procs. ACM SIGMOD Conf. on Management of Data, 311-325, May 1986.

APPENDIX

GLOSSARY

After image. The after image of object obj with respect to transaction Ti is the (last) value written into obj by Ti. It can be used to perform a redo operation.

Arjuna. Arjuna is an object-oriented programming system.

Atomicity. A series of database operations has an all or nothing effect on the database: either all operations of a transaction succeed or fail.

Bag. A bag is a set (collection) of elements of the same data type with duplicates.

Binding. The process of connecting or applying the description of the data (object) to the data (object) itself.

Before image. The before image of a write(obj) operation is the value of obj just before this operation executed. It can be used to perform a undo operation.

Cascadelessness. A synonym for "avoiding of cascading aborts."

Checkpointing. An activity that writes information to stable storage during normal operation in order to reduce the amount of work restart has to do after a failure.

Database operations. Operations on object that are supported by a database system, typically read(obj) and write(obj).

Class. A class is a set of objects that share a common structure and a common behavior. It is an implementation of an abstract data type.

Complex attribute. A complex attribute consists of any combination of simple and complex attributes (set-valued, tuple-valued, or sequence-valued attributes).

Complex object. A complex object is an object that is composed of a number of component objects, each of which may in turn be composed of other component objects.

Data value identity. An object is identified by its content as in relational databases where tuple objects are identified by primary or secondary keys.

Dirty pages. A page whose dirty bit is set (iff the value of the object(s) stored in that page was updated since it was last flushed) is called a dirty page.

Encapsulation. Encapsulation is the process of hiding all of the details of an object such as the structure of an object and the implementation of its methods.

Extent. A number of contiguous physical storage blocks in secondary storage.

Fixing(x). The buffer manager operation that makes a buffer slot x unavailable for flushing.

Flush. The buffer manager operation that writes an object from a (dirty) page to stable storage.

FORCE. The FORCE scheme means that all modified pages are written and propagated during end of transaction processing. In this case no redo logging is required.

~FORCE (NO-FORCE). The ~FORCE scheme means that no propagation is triggered during end of transaction processing. In this case a redo logging is required in case there is a system crash before the propagation is completed.

Granularity-hierarchy locking protocol. A locking method where different transactions can lock different granularity objects (data items).

Hot spot. A portion of the database that is accessed very frequently.

Inheritance. Inheritance is a mechanism for sharing properties and methods among classes, subclasses, and objects automatically. The subclass of a class (superclass) inherits properties and methods from its superclass.

KEEP/DISCARD hint. An advice from the system or user to the buffer manager indicating that the requested pages should be kept either in the top or bottom of the local LRU stack.

Lifetime dimension. The lifetime dimension of an object denotes the time interval between the time it was created and the time it becomes inaccessible.

Lock coupling. The tree locking technique whereby a transaction obtains locks on a node N's children before releasing its lock on N.

Logical surrogate identity. The object identifier of an object contains no information about location on secondary storage (e.g., <node-ID>, <class-ID>, instance-ID)

Notify locks. A notify lock is issued by a transaction that modifies an object locked by another transaction. Then, this notification can be used by the lock holder to either trigger a reread of the object or necessary operations to resolve any inconsistencies.

Object. Generally, a conceptual entity is modeled as an object. An object has state, behavior, and identity. In our generic data model, an instance of tuple-valued, set-valued, or list-valued data type is an object.

Persistence. The property of an object by which exists beyond the scope of the process that created or manipulated it. That is, the maintenance of object over long periods of time, independent of any programs that access the object.

Physical surrogate identity. The object identifier of an object is a persistent object identifier such as a record identifier or tuple identifier that represents stable storage location (e.g., volume-ID, page/segment-ID, slot-Id, <unique-ID/timestamp>).

Pinning(x). See fixing(x).

Positional B tree. A positional B tree is a B tree index on byte position with a large object and is used to represent the large object .

Propagation. If dirty pages are not written to the same blocks (not update-in-place), the procedure that writes an updated control structure for mapping logical updated page(s) to new block(s) into a stable storage after writing dirty pages into new blocks, is called propagation. If dirty pages are stored in different blocks (update-not-in-place), propagation can be repeated as often as wished.

Propagation-in-place. If dirty pages are always written to the same blocks (update-in-place), the control structure for mapping logical updated pages to the same physical blocks is not changed. Thus, writing dirty pages into the same physical blocks implicitly is the equivalent of propagation.

Redo scheme. The redo scheme states that before a transaction can commit, the value it produced for each object it wrote must be in stable storage (e.g. in the stable database or the log).

Recoverability. Recoverability means that the results of partially completed transactions will not be visible to other transactions. That is transaction Ti cannot commit until all transactions that wrote values read by Ti are committed.

Replacement strategy. The criterion according to which the buffer manager chooses a page to flush in order to make room for an object being fetched.

Representation dimension. The representation dimension of an object denotes the mechanism to make an object distinguishable from other objects. This can be a data value identity, a user-defined name identity, a logical surrogate identity, or a physical surrogate identity.

Resilience. The ability of an object to survive hardware crashes and software errors without sustaining loss or becoming inconsistent.

Reusability. In object-oriented paradigm, instantiation and inheritance are two reusability mechanisms that make it possible to reuse the same definition to generate objects with the same structure and behavior.

Serializability. The serializability means that the result of two interleaved transactions is as if one ran to complete before the other started.

Simple attribute. A simple attribute is an attribute with integer, string, Boolean, or float value.

Shadow page scheme. The shadow page scheme maintains two copies of page tables. One is the current page table, the other is the shadow page table which preserves the old state of the database. New pages are created to reflect the changes of a transaction and written to new blocks. If the transaction aborts, the current page table is discarded and the shadow page remains intact. When the transaction commits the current page table replaces the shadow page table to reflect the current state of the database.

Simple object. A simple object is a tuple-valued object where each attribute is of atomic-value attribute.

STEAL. Modified pages may be flushed into stable storage and/or propagated at any time. In this case undo logging is required in case that the transaction is aborted.

~STEAL (NO-STEAL). Modified pages are kept in buffer at least until the end of the transaction. In this case no undo logging is required.

Strict 2PL. A two-phase locking protocol where the lock manager releases all of the transaction's locks together, after the transaction commits or aborts.

Strictness. A transaction is strict if it is recoverable and cascadeless (avoiding cascading aborts.)

Structural mismatch. The data manipulation language of a database does not support the same data types as a general-purpose computational language.

Swizzle. The procedure that translates the physical identity format of an object into a virtual memory address format is called "swizzling".

Typeless storage. In traditional file system, the file object has no notion of data types except the notion of uninterpreted byte-strings.

Two-phase locking (2PL) protocol. The locking protocol in which each transaction obtains a read (or write) lock on each object before it reads (or writes) that object, and does not obtain any locks after it has released some locks.

Undo scheme. The undo scheme states that if an object's location in the stable database presently contains the last committed value of the object, then that value must be saved in stable storage before being overwritten in the stable storage by an uncommitted value.

Unfix(x). The buffer manager operation that makes a previously pinned page x again available for flushing.

Unpin(x). See Unfix(x).

Write-ahead-log (WAL). The WAL protocol requires undo information be written to the log file before the corresponding updates are written to the stable storage. If a transaction is incomplete, the undo log is used to rollback the transaction.

Zeitgeist. An object-oriented database system developed by Texas Instruments.

VITA

TEH-CHEN SHEN

Candidate for the Degree of

Master of Science

Thesis: AN INTEGRATED PERSISTENT OBJECT MANAGER (IPOM): A MODEL TO SUPPORT PERSISTENCE AND DATA SHARING IN OBJECT-ORIENTED DATABASE SYSTEMS

Major Field: Computer Science

Biographical:

Personal Data: Born in Taiwan, February 29, 1960, the son of Lo-Sheng Shen and Tao-Mei Shen-Wu.

Education: Graduated from Ming-Hsin Engineering College, Taiwan, in June 1980; received Bachelor of Science Degree in Mechanical Engineering from National Taiwan Institute of Technology, Taiwan, ROC, June 1985; completed requirements for the Master of Science Degree at Oklahoma State University in December 1992.

Professional Experience: Mechanical Technician, Department of IC Production, RCA Ltd. Corp., Taiwan, ROC, June 1982, to July 1983; Mechanical Engineer, Department of Operation, Shen-Ao Steam Power Plant, Taiwan Power Company, Taiwan, ROC, June 1985, to July 1990.