Chapter 3

3  Synergistic Buffer Management and Storage Management

3.1  Introduction

In the modern world the need of organized data is increasing rapidly but to retrieve the information from huge data source is still a big problem. Today we have large number of data source (Database), but how can we minimize the time in accessing the information from the complex database. Unfortunately: There is no silver bullet Due to the daily advance in the computing power of the hardware it become more complex to find the solution of the problem. Despite the dominance of relational database management systems (RDBMS) in the database market, object-oriented database management systems (OODBMS) continue to play an important role in complex data management. An object database (also object-oriented database management system) is a database management system in which information is represented in the form of objects as used in object-oriented programming.

When database capabilities are combined with object-oriented programming language capabilities, the result is an object-oriented database management system (OODBMS). OODBMS allow object-oriented programmers to develop the product, store them as objects, and replicate or modify existing objects to make new objects within the OODBMS. Because the database is integrated with the programming language, the programmer can maintain consistency within one environment, in that both the OODBMS and the programming language will use the same model of representation. Relational DBMS projects, by way of contrast, maintain a clearer division between the database model and the application. As the usage of web-based technology increases with the implementation of Intranets and extranets, companies have a vested interest in OODBMS to display their complex data. Using a DBMS that has been specifically designed to store data as objects gives an advantage to those companies that are geared towards multimedia presentation or organizations that utilize computer-aided design.

3.2  Importance of synergetic buffer in OODBMS

The ever increasing demand for fast complex data storage and retrieval makes a strong case for OODBMSs' (Object Oriented Data base Management Systems) survival as an important database management technology. OODBMSs are particularly suited to the management of
complex data since they provide fast navigational access, efficient storage of class methods and efficient and natural storage of many-to-many relationships.

Storage management is an important issue in the design of any object-oriented database management system (OODBMS). In fact, most object-oriented database management systems are composed of two main subsystems, an interpreter and a storage manager.

The interpreter provides the operational semantics as seen by the user; it understands the details of the data model, enforces object encapsulation, and executes methods. It calls the storage manager for physical data access and manipulation. The storage manager, in turn, concerns itself with the placement of objects on secondary storage, movement of data between secondary storage and main memory, creation of new objects, recovery, concurrency control, and sometimes indexing and authorization.

Complex data is most often accessed via navigation. Relational Data Base Management Systems (RDBMSs) are poorly suited for fast navigational access since simple object graph navigations can often turn into joins of multiple tables when converted to queries on the relational schema.

Object-Relational Database Management Systems (ORDBMS) offer better navigational performance by storing references between objects inside the relational tables. Object navigations can then proceed by de-referencing these references instead of executing multiple joins. However this approach still does not perform as well as OODBMSs which often store objects traversed together on the same disk page, thus generating less IO. In contrast, ORDBMSs typically do not store objects traversed together on the same disk page, they are often stored as tuples on different relational tables instead. Typically, tuples of the same table are stored together on disk. Thus one of the most attractive characteristics of OODBMSs is fast navigational access.

The ability for OODBMS to provide fast navigational access is conditioned on efficient main memory caching, which is made more important by the fact that disk IO performance improves at only 5-8% per year whereas CPU performance doubles approximately every 18 months. An example of a fast disk is the Seagate Cheetah 18 with an average access time of 8.19 ms. a consequence is that disk IO is likely to be a bottleneck in an increasing number of OODB applications. Thus the focus of the proposed study is on reducing the effects of disk IO on the performance of OODBMSs.
It should also be noted that much recent research on performance optimization of RDBMSs has been focused on the main memory bottleneck instead of the disk IO bottleneck [Ailamaki et al. 1999; Chen et al. 2001; Rao and Ross 1999; Rao and Ross 2000]. This is due to main memory becoming cheaper and sophisticated techniques for hiding disk IO latency in RDBMSs. Some database users now choose to set up their system so that the entire database fits in memory.

This decision is typically based on cost/performance trade-offs. However, disk continues to remain cheaper than memory, and so in any cost/performance analysis, scalability will ultimately dictate the use of disk. In addition, in many database applications a very high percentage of data accesses are directed at a very small portion of the database. In such cases, it is more cost-effective to only store a small portion of the database in memory (the portion which has a very high percentage of data access).

When the database is larger than memory, techniques for hiding disk IO are needed to ensure the system is not bottlenecked at the disk. Existing techniques for hiding disk IO in OODBMSs do not perform as well as their RDBMS counterparts. This is because navigational data accesses (often used in OODBMSs) are much harder to predict than index and table accesses in RDBMSs. The disk IO bottleneck in OODBMSs is thus a pressing research problem.

Effective buffer management is the key to reducing the disk IO bottleneck in OODBMSs. There has been much existing work, namely in the areas of: static clustering; dynamic clustering; buffer replacement; and prefetching. All of these techniques can be used together in a complimentary Manner. Most existing research has focused on finding the best solution for each area with little regard on how solutions from the different areas affect each other. We believe synergy exists between the areas, and that exploiting the synergy leads to the best overall solution. We focus on demonstrating synergistic techniques are both feasible to implement and outperform their non-synergistic counterparts.

### 3.3 Storage management for OODBMS

Storage management is an important issue in the design of any object-oriented database management system (OODBMS). In fact, most object-oriented database management systems are composed of two main subsystems, an interpreter and a storage manager. The interpreter provides the operational semantics as seen by the user; it understands the details of the data model, enforces object encapsulation, and executes methods. It calls the storage manager for physical data access and manipulation. The storage manager, in turn, concerns itself “with the
placement of objects on secondary storage, movement of data between secondary storage and main memory, creation of new objects, recovery, concurrency control, and sometimes indexing and authorization” [ZM89, page 237].

This section addresses four important issues in managing storage for object-oriented database management systems: object representation, updates and recovery, indexing and object referencing, and clustering. The object representation issue deals with how an object is represented in memory versus how it is represented on secondary storage. It also includes the representation of different versions of objects. Updates and recovery are related issues in that how updates to objects are handled by the system influences the recovery schemes that are available. The hierarchical nature of object-oriented systems makes object identification vital since objects may reference one or more other objects and OODBMSs must be able to follow these reference paths (pointer chasing) efficiently. Indexing may provide more efficient access to related objects. Frequently, it may be helpful to cluster, or group, objects physically in storage to increase system performance. Thus, clustering is another important issue in storage management.

In discussing these four issues, six different storage management models will be examined. These include the storage management strategies that are found in O2, LOOM, EXODUS, Postgres, Mneme, and a distributed object manager for Smalltalk-80.

### 3.3.1 O2 Object Manager

The O2 object manager has a server/worker architecture. The workstation component is single user, memory based, while the server is multi-user, disk based. The object manager (OM) is the software module of O2 that handles persistent and temporary complex objects, each of which has an identity. Furthermore, objects are shared and reliable and move between workstations and the server. The OM is used by all the upper modules of the system. The O2 Misdivided into four layers, each of which is responsible for implementing the various features of OM. These layers are: (I) a layer which copes with the manipulation of O2 objects and values, and with transactional control (involves creation/deletion of objects and structured types, retrieval of objects by name, support for set, list, and tuple objects), (ii) a Memory Management layer (involves translation of object identifiers to memory addresses, handling object faults, managing space occupied by objects in main memory), (iii) a Communication layer which takes into account. Object transfers, execution migration (from workstations to
server and back), and (IV) a Storage layer devoted to persistence, disk management, and transaction support (this feature is on the server). The last layer is implemented by the underlying WiSS, the Wisconsin Storage System. The WiSS runs under Unix System V but bypasses the Unix File System and does its own buffering. [VBD89] In O2, the distributed architecture is visible to the application programmer. The programmer may explicitly specify the machine desired to execute a message passing expression. The unit of transfer between the server and a workstation is an object. Each object is encoded (from disk format to memory format or vice-versa) before each transfer and decoded at the other side. The following process layout is adopted. On the workstation, an application and the workstation version of the OM form one unique process. For each process running on a workstation, a mirror process runs on the server. The mirror process contains the code to be executed in the server in case an execution migration arises. In this case, if the selectivity ratio is high and the set to be searched is large, running the method on the server may result in a better performance. Two other notable features of the OM are: persistence is implemented with a simple composition-based schema in which deletions are implicit; clustering issues are clearly separated from the schema information and specified by the DBA in the form of a subset of a composition graph (defined later).

3.3.2 Object Representation

The O2 data model distinguishes between values and objects. Objects have identity and encapsulate values and user-defined methods. Values can be set, list or tuple structured, or atomic. Each value has a type which describes its structure. A named object or value is an object or value with a user-defined name. These objects or values are the roots of persistence. The OM of O2 does not distinguish between objects and values. It deals with only objects and atomic values. Structured values are given an identifier and are managed as “standard” objects. The system supports both the primitives for manipulating values as well as the message passing mechanism for objects. In the OM there are primitives to distinguish oids (object identifiers) denoting objects from oids denoting values. Tuples. On disk, a tuple is represented as a record stored in a page. When a tuple outgrows a disk page, it is switched to a different representation suitable for storing long records. This representation is the Long Data Item (or LDI) format provided by WiSS. The object identifier of the tuple is unchanged. In main memory, tuples are represented as contiguous chunks containing the actual values. These chunks hold pointers to the proper locations of the strings of the tuple. The strings are stored at a different location,
away from the main chunk. This way the strings may grow or shrink without requiring the entire object to change location. In O2, a tuple may have exceptional attributes, that is, attribute values not declared in its class. Consider the following example:

Class Person type tuple (name: string, age: integer); O2 Person x;

Storage Management for OODBMS 73

*x = tuple (name: “john”, age: 28, my-opinion: “nice fellow”);

Here, the attribute my-opinion is the exceptional attribute for the given tuple. In the case of tuples with exceptional attributes, the tuple object may grow in length. When such a tuple grows, a level of indirection for the entire tuple value is generated if an in-place extension is not possible. Lists. These may be considered as insertable arrays and are represented as ordered trees in this system. An ordered tree is a kind of B-tree in which each internal node contains a count of the nodes under it. The node-counts have to be kept updated at all times. This structure efficiently stores small as well as large lists. Sets. A set of objects is itself an object containing the object identifiers of its members. The representation of large sets is required to be such that (I) membership tests are efficient and (ii) scanning the elements of the set is also efficient. B-tree indices (provided by the underlying WiSS) are used to represent large sets. It could be costly to index a small set. Hence, under a limit size, a set is represented as a WiSS record. The value of the limit size is the maximum record size in WiSS. Small sets are kept ordered. This ensures that binary operations (such as unions and differences) on the sets take advantage of the ordered representation. Note that the larger sets are automatically ordered. Multimedia Objects. Two types of multimedia objects are implemented: unstructured text and Bitmap. From the user point of view, they are instances of the predefined classes Text and Bitmap. The predefined methods in these classes are display and edit. Text is represented as an atomic object of type string and bitmaps are atomic objects of type bytes, an unstructured byte string preceded by its length. Persistence is defined in the O2 model as reachability from persistent root objects. This is achieved by associating with each object a reference count. An object persists as long as this counter is greater than 0. Persistence is user controlled: in order to make an object persistent, the user has to make it a component of an already persistent object. Versioning and authorization have not been addressed in the first version of O2, but have been proposed for the next version.
3.3.3 Indexing and Object Referencing

Objects are uniquely identified and accessed by object identifiers (oids). The (persistent) identifiers are “physical” identifiers, that is, reflecting the location on disk. An object is stored in a WiSS record and the object identifier is the record’s identifier, an RID. Such identifiers may have a performance edge over “logical” identifiers, in the sense that they may save an extra disk access for retrieving the “object table” (which maps the logical identifiers to the physical addresses). This disk access is a necessity in the case of logical identifiers. A major problem, though, is moving objects on disks without changing their identifiers. The solution adopted in O2 is to use “forwarding markers”. In this scheme, the old physical address of a “moved” object contains a pointer to its new location.

An RID is coded in 8 bytes: a volume identifier (2 bytes), a page identifier within a volume (4 bytes), and a slot-number (2 bytes). In contrast to the persistent identifiers, the virtual memory records are identified by TempIDs. Each TempID is made up of a virtual memory address and a machine tag. A machine tag indicates if the virtual memory address is a workstation address or a server address. Both the workstation and the server maintain caches of recently accessed objects. The server has a dual buffer management scheme: a page buffer implemented by WiSS and an object buffer pool, the object memory. Objects in the page buffer are in the disk format. In the object memory, they are in their memory format. The disk format is more compact. Both of the server caches are shared among all concurrent processes. There is a Memory Management Module to: (I) translate object identifiers into memory addresses (includes handling object faults for objects requested by the application but not currently in memory); (ii) manage the space occupied by objects in main memory. On the server, an object fault implies reading a WiSS record and transferring it between the page buffer and the server object memory. On every object fault, all the valid records on the same page as the object in question are transferred into the object memory. This “read-ahead” strategy is based on the fact that objects which have a strong correlation between them are clustered on the same or nearby pages and reading an entire page will accelerate further processing. Objects are transferred from the server to the workstations via the Communication Manager. Both on the server and on the workstations, the memory address at which an object is stored never changes until the object migrates to another machine or is written out to disk.

While an object is in memory, an object table maps its identifier to its true location in memory. This table is hashed on identifier values and contains entries for resident objects only. The
policy of having physical addresses as object identifiers poses some problems for temporary records. As such records are only in memory, they are forced to remain at the same address at which they were stored at the time of creation. This has some problems. First, the promotion of records from temporary to persistent storage must be delayed until the transaction commits. Second, it does not allow use of well-known garbage-collection techniques. Records cannot be freed from memory until commit time unless a temporary reference count scheme is built. In fact, as a record may be referenced by the O2 variables of a method, it should exist as long as it is pointed to by a variable, even if the reference count of the record is 0. It has been observed that typical applications running on top of the OM create temporary records at a furious rate and that these records never get deleted explicitly before commit. [VDD+90]

3.3.4 Clustering

Newly created persistent objects are given a persistent identifier when they are inserted in a file at transaction commit. The mapping of objects to files depends on control data given by the Database Administrator (DBA). These data describe the placement of objects: the placement trees (also termed cluster trees). These trees express the way in which a composite object and its object or value components will be clustered together. The main heuristic used to postulate that two objects will be used together frequently is their relationship through composition structure. For example, if a class A is a child of class B in a placement tree and an object a of class A is a component of object b of class B, the system attempts to store a as close as possible to b. Cluster trees are traversed at commit time to determine the file into which the record is to be inserted. Furthermore, one has the option of specifying the RID of the parent record r in the insert routine to compel WiSS to try and store the new record in the same page as r or in a nearby page.

3.3.5 Updates and Recovery

When the workstation needs to update a persistent object, an explicit exclusive lock request on the page the object resides is made. This is done with the help of the RID, which also contains the page identifier. In addition, all the objects which are represented by ordered trees will have their node counts updated after every insertion or deletion. Recovery and rollbacks are not implemented in the current version of WiSS. This feature is proposed for the next version. Also proposed are “save points”, a mechanism to prevent the loss of large amounts of work.
3.4 LOOM—Large Object Oriented Memory for Smalltalk-80 Systems

LOOM is different from the other systems described in this research in the sense that it is not really an object storage manager. It is a virtual memory system designed and implemented for Smalltalk-80 systems. It does not have any notion of the “semantics” of the objects stored. It does not support operations like updates, versioning, recovery, or concurrency control. It may be viewed simply as a memory system storing objects on primary and secondary memory. The most important feature of LOOM is that it provides virtual addresses that are much wider than either the word size or the memory size of the computer on which it runs. It is a single-user, virtual memory system that Operates without assistance from the programmer. LOOM is currently intended for use over local area network. The design, however, can be extended too many users and many machines. The major issues in the LOOM design and implementation are:

- representation of resident and secondary memory objects;
- translation between representations;
- Identification of times when the translations must occur.

3.4.1 Object Representation

There are two different name spaces in LOOM: one for main memory and the other for secondary memory. The same object is identified by names from different spaces when it resides in different parts of the system. The identifier of an object is called an Oop, which stands for “object pointer”. Main memory has a resident object table (ROT, or sometimes called an OT), which contains the actual main memory address of each resident object. In main memory, each object has a short (16 bit) Oop as its identifier. This short Oop is an index into the ROT, so that an object’s starting address can be determined from its Oop with a single addition and memory reference. The ROT entry also has reference-count bits. The body of each object contains a word for the length of the body, a pointer to the object’s class, and the object’s fields. Each field is either a pointer to another object or a collection of “bits”. In the following discussion, only pointer fields are dealt with. Each field (as well as the class pointer) that refers to another resident object contains the short Oop of that object. Fields that refer to non-resident objects (objects on secondary storage) contain a short Oop of one of two types, a leaf or a lambda (described later). In addition to these fields, resident objects in a LOOM system have three extra words.
Two of these words contain the long (secondary memory) Oop of that object. The third word, known as the delta word, contains a delta reference-count (described later under object referencing). The short Oop is the result of a hash function applied to that object’s long Oop. In secondary storage, object representation has some different features than in main memory. Secondary memory is addressed as a linear space of 32-bit words. Objects start with a header word that contains 16 bits of length and some status bits. Each pointer field in the object is 32 bits wide. Non-pointer fields (such as bytes in Strings) are packed, with 4 bytes in each 32-bit word. The long Oops in pointer fields are 31-bit disk pointers, addressing as many objects as will fit into 231 disk words (32-bit words).

Fields of objects on secondary storage always refer to objects in secondary storage and do not change when the object to which they point is currently cached in main memory. No information about primary memory is ever stored in the secondary memory. [KK90] When an object is brought into main memory, its fields must be translated from the long form to short form. The object is assigned an appropriate short Oop (one to which its long Oop hashes), a block of memory is reserved for it, and all of its fields are translated from long Oops to short Oops. Leaves. Leaves are pseudo-objects that represent an object on secondary storage. They have a short Oop hashed by that object’s long Oop and ROT entry, but their image in memory only contains a length word, disk address words, and the delta word. Their image contains no class word or fields. Leaves, therefore, take only up to 4 words of memory, whereas an average object (object in its expanded form) takes up 13. Leaves are created without looking at that object’s image on secondary storage. This is very important, since a major cost in virtual memories is the number of disk accesses. The short Oop of the leaf may be treated as if it were the short Oop of the object; it may be stored into fields of other objects, without ever needing the actual contents of that object. Its reference-count can be incremented and decremented just like a fully expanded object’s reference count. An object will be in the main memory: (i) in its entirety; (ii) as a leaf. If none of the above is true, then the object will be on disk. Lambdas. Lambdas are the second way to represent fields of resident objects that refer to objects on secondary storage. This representation reduces the number of leaves in the system at any given time. A lambda is a “slot holder” for a pointer to an object which has not been assigned a short Oop.

It is a pseudo-Oop, a reserved short Oop (the Oop 0) which is not the name of any resident object. Unlike leaves, lambdas do not take up separate ROT entries for themselves. Instead, they are all mapped to the single pseudo-ROT entry at 0. Any object that refers to the field
represented by a lambda, accesses the 0th entry of the ROT table. This signals to LOOM to go back to the secondary storage image of the object containing this field. There it finds the long Oop of the field. The field is fetched into main memory as a leaf or as a full resident object. The most significant feature of lambdas is that they do not utilize main memory storage. This saving can prove important at times. Putting lambdas into fields of objects, which are not likely to be referenced during these objects’ typical stay in memory, saves both space and time needed to create and destroy many leaves. There is, however, an added cost of one disk access for fetching a lambda field. The choice of strategy (to decide between making the fields of an object leaves or lambdas when the object is brought in the main memory) can strongly affect the performance of a LOOM system. Creating a leaf takes more time, uses more main memory, and creates a ROT entry, but does not cause any extra disk accesses. A lambda is easy to create but causes an extra disk access if the field it occupies happens to be referenced. It is suggested to rely on history to make the decision: if a field was a lambda when the object was written to the disk once, it is likely to remain unreferenced during its next trip into main memory. Hence, a lambda must be created for this field when the object is brought into main memory another time. Each pointer field of the disk contains a hint, the no Lambda bit, and the object faulting code follows the advice of the hint (explained further under Indexing and Object Referencing below).

3.4.2 Indexing and Object Referencing

If an object is in main memory in its entirety, then it can simply be referred to by the short Oop of the object. When a field (which in turn may be another object) represented by a leaf is needed, the entire object with its fields has to be brought into main memory. Since the leaf contains the disk Oop, the body is easy to find. After the body is translated into main memory form, its memory address replaces the leaf’s ROT entry and the leaf is discarded. Short Oop references to the object (that is, the references by which the other objects may refer to this object) remain the same when the leaf is replaced by the entire object. Since a leaf can be substituted for an object and vice versa with no effect on pointers to the object, LOOM is always free to make more room in main memory by turning resident objects into leaves. If a lambda represents one of the fields of an object, then LOOM must go back to the object’s image on secondary storage, look in that field for a long pointer, and create a leaf or resident object. This is done in order to discover the actual value of that field.
When a field of an object being brought into main memory has the no Lambda bit set and that field refers to a non-resident object, then a leaf is created. Thus the no Lambda bit may be used to decide between representing any field of an object as a leaf or a lambda when the object is brought into main memory. A leaf is also created when a field of a resident object containing a lambda is accessed. When the interpreter needs to access a field in a leaf, the leaf is expanded into a resident object and its fields are translated from long form to short form. This is called an object fault. The reverse operation of contracting a leaf can be done at any time. The final part of an object’s journey into primary memory consists of destroying the leaf and reusing its short Oop and memory space. This can be done only when there are no longer any fields in any resident objects pointing to the leaf. Lambdas may be resolved into leaves and leaves may be expanded into full objects before they are needed. This operation is called prefetch. Reference counting is used by LOOM for “garbage” identification. LOOM keeps a separate count of references for the short and longOops of every object. This is essential because any object may be only on disk, entirely in memory, or in leaf form in memory. There are three possible sources of reference-count changes. One pointer may be stored over another, a long pointer can be converted to a short pointer, and a short pointer may be converted to a long pointer. The Smalltalk interpreter can keep track of the shortOops. However, whenever a leaf is expanded into complete object or an object is shrunk into a leaf, there arises the need to update the reference-count of the long Oop of that object. The long Oop reference-count is stored on disk with the main body of the object. Hence, updating it would mean a disk-access. In order to reduce the disk access cost for every change in the long Oop count, LOOM keeps a “delta” or running change in the long Oop reference count for each object in main memory.

The true long pointer reference count of any object is the count found on the disk in the object’s header plus the count found in the delta part of the object’s delta word in main memory. Every time a leaf is expanded, the delta of its long count is decremented. This is due to the fact that the object is now in main memory and will be directly referred to by its short Oop (the reference-count of which is simultaneously incremented). The delta count is incremented when an object is translated into a leaf. At any given time, the long Oop reference-count of an object is the sum of its delta count and long Oop reference-count on the disk. The short Oop reference-count of all objects in the memory are stored in the ROT (resident object table) along with their shortOops. This helps to detect when the last short pointer to any object disappears (that is, when the short Oop count of the object goes to zero) so that the short pointer may be reused. Both these reference counts also detect the situation of “no reference” to an object.
3.5 Conclusion and Summary

In this section I have tried to explain the issue of storage management for object-oriented database management systems. Four important areas in object storage management have been examined: object representation, updates and recovery, indexing and object referencing, and clustering. I basically focused on two object storage models (O2, LOOM) as my research work is related to these storage model. The techniques employed by these models were then compared and contrasted. I also put some stress on storage management which I explain in details later in my thesis paper, as we know that Storage management is an important issue in the design of any object-oriented database management system (OODBMS). In fact, most object-oriented database management systems are composed of two main subsystems, an interpreter and a storage manager. The interpreter provides the operational semantics as seen by the user; it understands the details of the data model, enforces object encapsulation, and executes methods. It calls the storage manager for physical data access and manipulation in my work I explain the different buffer management techniques and the best economic deal among them.