Storage Management for Object-Oriented Database Management Systems: A Comparative Survey

David Dueck, Yiwen Jiang, and Archana Sawhney

Contents

1 Introduction **71**

2 The O_2 Object Manager **71**
2.1 Object Representation 72
2.2 Indexing and Object Referencing 73
2.3 Clustering 74
2.4 Updates and Recovery 75

3 The EXODUS Storage Component **75**
3.1 Object Representation 76
3.2 Indexing and Object Referencing 77
3.3 Clustering 77
3.4 Updates and Recovery 78

4 LOOM—Large Object Oriented Memory for Smalltalk-80 Systems **78**
4.1 Object Representation 79
4.2 Indexing and Object Referencing 81
4.3 Clustering 82
4.4 Updates and Recovery 82

5 The Postgres Storage Manager **82**
5.1 Object Representation 83
5.2 Updates and Recovery 83
5.3 Indexing and Object Referencing 83
5.4 Clustering 84

6 Mneme: Persistent Data Management **84**
6.1 Object Representation 85
6.2 Updates and Recovery 85
6.3 Indexing and Object Referencing 85
6.4 Clustering 86

7 Distributed Object Manager for Smalltalk-80 **87**
7.1 Object Representation 87
7.2 Updates and Recovery 87
7.3 Indexing and Object Referencing 88
7.4 Clustering 89

8 Comparison **89**

9 Conclusion **91**

1 Introduction

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 paper 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 O_2, LOOM, EXODUS, Postgres, Mneme, and a distributed object manager for Smalltalk-80.

2 The O_2 Object Manager

The O_2 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 O_2 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 OM is divided 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 O_2 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 O_2, 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).

2.1 Object Representation

The O_2 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 O_2 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 O_2, 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;

∗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 O_2 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 O_2, but have been proposed for the next version.

2.2 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 O_2 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 O_2 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]

2.3 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.

2.4 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 "savepoints", a mechanism to prevent the loss of large amounts of work.

3 The EXODUS Storage Component

EXODUS is an extensible database management system developed at the University of Wisconsin. This section describes the design of the object-oriented storage component of EXODUS. The main intent of the designers of this system was to come up with a modular and modifiable system rather than a complete database system. The EXODUS storage system is the lowest level and the only "fixed" component of EXODUS. "Fixed" means that the storage component of EXODUS is used in its original form by all the applications running on EXODUS; the design of upper level modules, some of which interface with the storage system, is application-specific. Application-specific access methods, operations, and version management layers are built using the primitives provided by the storage system. One of the significant features of this storage system is *minimal semantics*, that is, minimal information about the conceptual schema. In order to the keep the system extensible it does not seem feasible for the storage system to have information about the conceptual schema. On the other hand, semantics are often useful for performance reasons. The solution adopted is to keep schema information out of the storage system, but to allow *hints* to be provided. These hints can help in making decisions that influence performance in important ways. For example, clustering hints can be passed as parameters to the object creation routine to place the new object "near" another object. [CDRS86]

As mentioned earlier, the layer above the storage system provides the access methods for a given EXODUS application. This layer is likely to change from one application to another. The EXODUS storage system provides a procedural interface to this higher layer. This interface includes procedures to: create/destroy file objects; open/close file objects for file scans; create/destroy storage

objects within a file object; and begin, commit, and abort calls for transactions.

3.1 Object Representation

Storage Objects. The basic unit of stored data in the EXODUS storage system is the *storage object*, which is an *uninterpreted byte sequence* of virtually unlimited size. By providing capabilities for storing and manipulating storage objects without regard for their size, a significant amount of generality is obtained. Storage objects can grow and shrink in size. Insertions and deletions are allowed anywhere within the object (that is, not limited to occur at the end of an object). Storage objects can be either small or large, although this distinction is hidden from clients of the storage system.

Large Storage Objects. Conceptually, a large object is an uninterpreted sequence of bytes; physically, it is represented on disk as a B+ tree index on byte position within the object and a collection of leaf (data) blocks. The root of the tree (the large object header) contains a number of (*count, page #*) pairs, one for each child of the root. The count value associated with each child pointer gives the maximum byte number stored in the subtree rooted at that child. The count for the rightmost child pointer is also the size of the object. An absolute byte offset within a child translates to a relative offset within its parent node. The leaf blocks in a large storage object contain pure data. The size of a leaf block is a *parameter* of the data structure and is an integral number of contiguous disk pages. For often-updated objects, leaf blocks will probably be one page in length to minimize the amount of I/O and byte shuffling that must be done on updates. For less frequently updated objects, leaf blocks may be several contiguous pages long. Each internal node of a large storage object corresponds to one disk page. The root node corresponds to at most one disk page, or possibly just a portion of a shared page. Such a two-level tree with a leaf block size of 1 page can support an object size range of 8KB–2MB.

File Objects. In the EXODUS storage system, *file objects* are *collections* (sets) of storage objects. These collections are useful for grouping objects for several purposes and according to various criteria. The EXODUS storage system provides a mechanism for sequencing through all the objects in a file. Thus, related objects can be placed in a common file for sequential scanning purposes. Also, objects within a given file object are placed only on the disk pages allocated to the file. The representation of file objects in EXODUS is quite similar to the representation of large storage objects. The issue of *sorting* a file object is addressed in the next section.

Versions. Versions of storage objects are supported by this storage system. Support is rather basic—one version of each storage object is retained as the current version, and all of the preceding versions are simply marked (in their object headers) as being old versions. Version support is conducive to widely different applications, each with its own notion of versions. Versions of large storage objects are maintained by copying and updating the *pages* that differ from version to version. Various versions of the same object share pages that are common among them. Only dissimilar pages are owned separately. These independently owned pages are pointed to by the unique object headers of each version. [CDRS86]

3.2 Indexing and Object Referencing

Small storage objects reside on a single disk page, whereas large storage objects occupy multiple disk pages. In either case, the object identifier (OID) of a storage object is of the form (*page #, slot #*). The OID of a small storage object is a pointer to the object on the disk. For a large storage object, the OID points to a *large object header*. This header can reside on a page with other large object headers and small storage objects and contains pointers to other pages involved in the representation of the large object. All other pages (the non-header ones) in a large object are private to the object, rather than shared with other storage objects. Pages, however, may be shared between various versions of the same object, as explained later. When a small storage object grows to the point where it can no longer be accommodated on a single page, the EXODUS storage system automatically converts it to a large storage object, leaving its object header in the place of the original small object.

The indexing scheme of *large storage objects* is explained above, along with their representation.

A file object is identified by its OID, a pointer to the root page of the file object. Storage and file objects are distinguished by a bit in their object headers. Like large storage objects, file objects are represented by an index structure similar to a B+ tree with the key for the index being the *disk page number*. This helps to gather information about the pages of a file at one place. This facilitates scanning of all the objects within a given file object *in physical order* for efficiency. It also allows fast deletion of an object with a given OID from a file object. Creation of a file object allocates the file object header.

Keeping the file objects sorted is an aid in object referencing within a given file. The storage system has no semantic knowledge of the contents and data types of the fields for a given file object. The EXODUS storage system thus provides a generic file object sorting routine. One of the arguments to this sort routine is a procedure parameter for an object comparison routine; the sort utility calls this routine to compare storage objects as it sorts the file.

3.3 Clustering

As has been mentioned above, the EXODUS storage system does not have sufficient semantics to implement any kind of clustering on its own. However, there is a provision for providing *hints* to achieve clustering. For example, when an object is created within a file object, the object creation routine can be invoked with an optional hint of the form "place the new object near X" (where X is the OID of an existing object within the file). If this hint is present, the new object is inserted on the same page as X. If there is not enough space on *X's* page, then a new page near *X's* page is allocated for the newly inserted object and its page number is inserted into the file object B+ tree; the OID of the file object is recorded on the newly allocated page.

3.4 Updates and Recovery

The EXODUS storage system supports a number of updating operations on large storage objects and file objects. These include *search, insert, append*, and *delete*. All of these operations permit the updating of any number of bytes at *any position* in an object. All take the size and starting position of the data as input parameters. The deletion algorithm of EXODUS is different from the traditional B+ tree deletion in that it allows *bulk* deletions as opposed to single record deletions. This deletion algorithm has two phases. The first phase removes the entire range of bytes specified. The second phase balances the B+ tree index (which was left unbalanced by the first phase).

Version Updates. All the updating routines provide a *versioning option*. If an application wishes to update/create versions, it invokes these routines with the versioning option *on*. When a storage object is updated with the versioning option on, the old object header (or the entire object, in the case of a small storage object) is copied to a new location on disk as an old version of the object. A new header is written over the old version of the object header. The OID of the old version is returned to the updater and the OID of the new version is the OID that was originally passed to the update routine. In order to save copying costs, the old version is placed on the same page of the file object as the new version. If this is not possible, a nearby page is selected. For creation of new versions of large storage objects, the updating operations of insert, append, delete, and write are invoked with the versioning option turned on. A "smart" algorithm, for deletion of a version of an object, exists. This algorithm ensures the safety of the object's pages that are shared by other versions of the same object. The EXODUS storage system provides both concurrency control and recovery services for storage objects. Two-phase locking is used on byte ranges. An option is also provided for "locking the entire object".

For recovery, small storage objects are handled using before/after image logging and in-place updating at the object level. Recovery of large objects is handled using a combination of *shadows* and *logging*; updated internal pages and leaf blocks are shadowed up to the root level, with updates being installed atomically by overwriting the old object header with the new header. Prior to the installation of update at the root level, the other updated pages are forced to the disk. The name and parameters of the operation that caused the update are logged, with the log sequence number of the log record for the update being placed on the root page of the object. This ensures that operations on large storage objects can be *undone* or *redone* as necessary in an idempotent manner. For versioned objects, the same recovery scheme is used. In this case, however, the before-image of the updated large object header is first copied elsewhere, to be maintained as the version before the updates.

4 LOOM—Large Object Oriented Memory for Smalltalk-80 Systems

LOOM is different from the other systems described in this paper 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 a local area network. The design, however, can be extended to 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.

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 2^{31} 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 *noLambda* bit, and the object faulting code follows the advice of the hint (explained further

under Indexing and Object Referencing below).

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 *noLambda* bit set and that field refers to a non-resident object, then a leaf is created. Thus the noLambda 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 long Oops 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 short Oops. 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 short Oops. 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.

4.3 Clustering

There are no clustering facilities in LOOM.

4.4 Updates and Recovery

There are no updating and recovery facilities in LOOM.

5 The Postgres Storage Manager

The Postgres data base system was constructed at the University of California at Berkeley and was designed by the same team that designed Ingres. The design of the storage manager for Postgres was guided by three goals:

- To provide transaction management without the necessity of writing a large amount of specialized crash recovery code.
- To accommodate the historical state of the data base on a write-once-read-many optical disk in addition to the current state on an ordinary magnetic disk.
- To take advantage of specialized hardware. [Sto90]

The Postgres storage manager is a no-overwrite storage manager. This provides two useful features. First, the time to abort a transaction can be very short because there is no need to process the log in order to undo the effects of the updates; the previous records are readily available in the database. In case of a system crash, little time is needed to recover the database to a previously consistent state since no *rollback* is necessary. Second, it is very convenient for the user who wishes to reference history records since these records are stored in the database.

However, the no-overwrite policy is not problem-free. The migration of history data from magnetic disk holding the current records to the archive where historical records remain is unstable under

heavy load. This is because the backup is run by an asynchronous daemon and, when there are many processes to run, the daemon may not get hold of the CPU. Therefore, the size of the magnetic disk database increases and performance degrades. [Sto90]

5.1 Object Representation

Postgres views a relation as an object. When it is created, Postgres allocates a file to hold the records for the object. However, there is no way to determine the size of the objects or the records. Therefore, for those records which cross disk block boundaries, Postgres allocates continuation records and chains them together using a linked list.

When an object is read into main memory, Postgres reads the pages of the object in a pre-determined order. Each page contains a pointer to the next and the previous logical page. [Sto90]

¿From time to time, an asynchronous daemon, the *vacuum cleaner*, sweeps records which are no longer valid to the archive system and reclaims the space occupied by such records. It generally sweeps a chain of several records to the archive at one time for efficiency reasons. [Sto90]

5.2 Updates and Recovery

The updating and recovery techniques are implemented in such a way so that less disk space is required and recovery after a system crash is easier.

The original value of a record is stored uncompressed and called the *anchor point*. An updated record is differenced against the anchor point and only the actual changes are stored. A forward pointer to the record is altered on the anchor pointer to point to the updated record (called the *delta record*). A delta record can only be accessed by obtaining its corresponding anchor point and chaining forward. Successive updates generate a one-way linked list of delta records off an initial anchor point. [Sto90]

For recovery, minimal (or no) *rollback* is needed because of the updating technique. The pointers to the records are merely moved to the end of the linked list. Therefore, less time is required.

The vacuum cleaner concatenates the anchor point and a collection of delta records and writes the resulting block to the archive as a single variable length record. It does the vacuuming in three phases. First, it writes an archive record and its associated index records to the archive media. Then, it writes a new anchor point in the current data base. Finally, it releases the space occupied by the old anchor point and its delta records back to the disk media. [Sto90] Due to this three-step procedure, “vacuuming” should be done infrequently. This will also cut down on the number of anchor points that occur in the archive, saving space.

5.3 Indexing and Object Referencing

Records can be referenced by a sequential scan of an object. After reading the object into main memory, Postgres can scan a relation by following the forward linked list. On each page, there is a *line table* containing pointers to the starting byte of each anchor point record on that page. Once an anchor point is located, the record that the user wishes to reference can be reconstructed by following the pointer and decompressing the data fields.

An arbitrary number of secondary indexes can be constructed for any base relation. Each index is maintained by an *access method* and provides keyed access on a field or a collection of fields. Each access method must provide all the procedures for the Postgres-defined abstraction for access methods. The Postgres run-time system will call the various routines of the appropriate access method when needed during query processing. [Sto90]

When a record is inserted, an anchor point is constructed for the record along with index entries for each secondary index. Each index record contains a key and a pointer to an entry in the line table on the page where the indexed record resides.

When an existing record is updated, a delta record is constructed and chained onto the appropriate anchor record. If no indexed field has been modified, then no maintenance of secondary indexes is required. If an indexed field has changed, an entry is added to the appropriate index containing the new key and a pointer to the anchor record. There are no direct pointers that point to delta records in secondary indexes. [Sto90]

When it is running, the vacuum daemon inserts secondary index records for any indexes defined on the archive relation. An index record is generated for the anchor point on each archive secondary index. Moreover, an index record must be constructed for each delta record in which a secondary key has been changed.

5.4 Clustering

If a user wishes the records in a relation to be approximately clustered on the value of a designated field, Postgres will attempt to do so by creating a clustered secondary index. [Sto90]

6 Mneme: Persistent Data Management

Mneme (NEE-mee, the Greek word for memory) is a persistent store that has been developed as part of a long term study into integrating object-oriented programming languages and object-oriented databases. Mneme is intended to be a tool that can be used with more than one language and that will take advantage of existing storage managers and network servers. It also is to allow study of the performance problems related to accessing and manipulating small objects on demand.

Mneme provides three interfaces: the client interface, the storage manager interface, and the policy interface. The client interface is made up of a number of routines which may be invoked by

programming languages such as Smalltalk, Trellis/Owl, C++, or Ada. This interface provides a simple and efficient abstraction of objects and allows Mneme to reliably provide persistent, shared objects to the client. Mneme objects are defined without a type system or code execution model in order to keep them general enough so that any language may access them. The storage manager interface provides the actual persistent storage and sharing of data. The basic requirements for a storage manager to be used in conjunction with Mneme are that it stores and retrieves data in segments (sequences of bytes) and that it provides some concurrency control and recovery features. The policy interface provides access to policy modules. Objects are grouped into *pools*, with each pool having an associated *policy module* (described later). These modules, which may be customized by the user, indicate how the pools of objects are to be managed. [MS88]

6.1 Object Representation

Mneme objects are made up of three main components: an array of slots, an array of bytes, and a set of attribute bits. Each slot is the size of a pointer (typically 32 bits) and may contain one of three types of values: it may hold a reference to another object, it may be empty (a 0 value), or it may hold an *immediate value*. "Immediate values are a concession to heap based languages that allow integers and other non-pointer data within slots" [MS88, page 9]. Each byte is a simple uninterpreted 8-bit quantity. Attribute bits are used to mark objects as having properties such as read-only.

An object is presented to the client via its client identifier (CID). Within Mneme, however, an object is known by its persistent identifier (PID). Thus, some conversion is required between CIDs and PIDs. This is discussed below.

6.2 Updates and Recovery

A *transaction* is the atomic unit of recovery within Mneme. The three basic update and recovery concepts used by Mneme are *volatile objects*, *object logs*, and *event notification*. "A volatile object is accessible to other clients and may be changed whenever the client does not have a handle on it, regardless of transactions" [MS88, page 12]. Uncommitted changes by one client may be seen by others, with handles (discussed later) providing mutual exclusion on volatile objects. Object logs are provided to support the use of volatile objects and are used to record past or intended changes to an object. When a transaction commits or aborts, Mneme can complete or undo its manipulations of these objects. Event notification is implemented so that clients waiting for resources (such as a handle being held on an object) may be notified of availability. This eliminates the need for busy waiting or polling.

6.3 Indexing and Object Referencing

Each object is accessible to the client via a *client identifier* (CID) allowing an application to reference up to 2^{30} objects at a time. The reference value stored within the slot of a Mneme object

is called a *persistent identifier* (PID) and is not the exact CID value seen by the client. This is due to the fact that the overall object space is not bounded (at least not conceptually), preventing objects from being assigned a short unique identifier. Since the internal identifier for an object differs from the client view, some conversion must be made between CIDs and PIDs. This is done by segmenting the CID address space into contiguous non-overlapping *chunks* and assigning each file (the main unit of modularity within Mneme) one of these chunks. Each chunk is described by the first CID in the chunk, called the *base*, and the number of CIDs in the chunk. To convert a PID to a CID, the base of the file is added to the PID. A similar conversion is done from CID to PID: determine the corresponding file (a table search) and then subtract the file's base from the CID.

To ensure that CID to PID conversion is not done more than once for a given object, a *handle*, which stores a more time-efficient run-time representation of the object than a CID, is created. This handle is requested by supplying Mneme with the appropriate CID and has the side benefit of providing logically exclusive access (that is, locking) to that object.

Objects within the same file can refer to each other simply by using the corresponding PIDs. In order to begin tracing all the paths of object references within a file, the file is opened and the file's *root object* (which may be defined and changed by the client) is identified. Any object within the file can then be referenced via the root by following the appropriate path from the root. Thus, the notion of a root object and the objects that may be referenced from it corresponds to that of the root directory and its sub-directories in a file system.

When an object refers to an object in a different file, a cross-file reference is used. This is implemented by having the object reference a *forwarder*, a special object in the current file. The forwarder contains the necessary information to identify the desired file and object within that file.

To aid in the handling of object faults (when objects are not found in main memory, they must be retrieved from secondary storage), Mneme partitions the space of PIDs within a file into units of 1024 PID values called *logical segments*. All objects that have PIDs with the same high-order bits belong to the same logical segment. The segments transferred between Mneme's buffer cache and the storage managers are called *physical segments*. "Every physical segment has some set of logical segments assigned to it, and a logical segment is associated with exactly one physical segment"[MS88, page 14]. A *logical segment table* is used to indicate which physical segment contains the logical segment, and, if the physical segment is memory resident, where the logical segment may be found within the physical segment. A null pointer in this table indicates a non-resident physical segment and results in a segment fetch upon access.

There are no explicit indexing techniques for Mneme.

6.4 Clustering

The unit of data modularity within Mneme is the *file*. A file contains a collection of objects, with each object in the file having a unique PID. Objects within the same file can refer to each other simply by using the corresponding PIDs. Files may be further subdivided into one or more *pools*, or

collections, of objects. The objects in each pool are managed with the same storage allocation and object management strategies, permitting the most effective strategy for a given client application. For example, if an application has a small number of frequently updated objects and a large number of objects that are changed infrequently, a different strategy could be used for each collection of objects.

7 Distributed Object Manager for Smalltalk-80

The purpose of the distributed object manager project in [Dec89] was to design a distributed object manager to replace the original object manager of the Smalltalk-80 system. This would allow users of several Smalltalk-80 systems to share objects, perform remote execution on objects, or to store objects in a centralized repository.

To accommodate this distribution concept, the object manager has been removed from the byte-code interpreter of the original Smalltalk-80 system, becoming its own entity within the system and distributed over several nodes of a local area network. The network manager at each node is the controlling process, executing remote accesses on remote objects, handling local access requests from other network managers, and controlling local processes. Main memory management activities such as relocation, swapping, free space management, and garbage collection are performed by the main memory manager while the secondary memory manager manages secondary storage, including the Smalltalk object descriptor table and the object space.

7.1 Object Representation

All objects residing in secondary storage have an entry in the *object descriptor table*. This entry, called a descriptor, describes the object by containing such information as the object size, object class pointer, internal and external reference counts, object location pointer (in persistent memory), and a set of flags. Included in these flags are a garbage collection indicator and a proxy flag indicating whether or not this object is a proxy object (proxy objects are discussed later).

When an access fault is encountered by the main memory manager, objects are automatically loaded into main memory. This occurs in a manner similar to segment moves in a typical segmented virtual memory. A mapping structure within main memory, taking the form of a hash table, is used to determine whether or not an object is resident and, if it is, the address of its object entry. An *object entry* is used to describe each object in main memory and includes pointers to the object descriptor (which is composed of the same fields as in secondary storage) and to the value block. This entry also contains state information for the descriptor and value block (present, absent, or being loaded) and whether either has been modified [Dec89].

7.2 Updates and Recovery

The described distributed object manager has no explicit update and recovery policies. Replication of objects is not considered, even though this would appear natural in a distributed environment. Garbage collection is performed via a local and distributed reclamation scheme in which objects are marked reachable and checked periodically for zero local and remote internal and external reference counts (described below).

7.3 Indexing and Object Referencing

Naming provides a mechanism for referencing objects without concern for their physical location, including actual site of reference. Although transparent to the user, access to objects within this distributed system is either local or remote. When remote, object access may be handled either by performing an effective remote access with no need for object migration or by executing a local access and forcing object migration. This decision depends on communication load, object size, and application requests and is not explicitly handled by the object manager. [Dec89]

The first scenario (remote access with no migration) is addressed by a particular type of object, called a *proxy* object, which locally represents a remote object. The proxy belongs to the private data of the object manager and contains two fields: "the resident site of the remote object and the virtual pointer to the object in the resident site"[Dec89, page 492].

The second way to handle a remote access is to transform the access into a local access and migrate the desired object to the local site. This involves creating a proxy object on the remote, or source, site and replacing the proxy object on the destination site (if it exists) with the real object. Local and remote reference counts must be updated in order for correct processing to occur. All proxy objects which point to the original site must be updated, but this is deferred until actual remote access is attempted with the proxy objects.

When an object references a local, or real, object, no special processing needs to occur. The result pointer (that is, a pointer to the object being referenced) is set to the local address of the real object and processing continues. Similarly, if an object references a proxy object, the result pointer is set to the local address of the proxy object. However, if a proxy object references another object, the problem of how to represent the result pointer arises. Three different cases are possible. First, the proxy object may reference a real object on the same site as the proxy object. In this case, the result pointer is simply set to the address of the local real object. Second, the proxy object may be referencing a real object on the site to which the proxy object is already pointing. Another proxy object must be created on the local site, with the result pointer being set to this local address (the result pointer must always be local). The third case is when the proxy object references a real object on a third site (via a proxy object on a remote second site). Here, a proxy object must be created on the local site having the same information as the proxy object on the second site. The result pointer then is set to point to this newly created object.

Similar situations (such as those encountered setting result pointers in remote references) occur when sites are dynamically disconnected. The disconnecting site may currently support objects,

owned by other sites, which must be sent back to their owners in order to ensure a graceful disconnection[1]. Secondly, objects owned by the disconnecting site may currently reside on other sites. The objects in question could be sent back to the owner before it disconnects. However, this reduces the sharability of objects. Also, if an object migrates when the owning site is disconnected, the owning site is not informed. Thus, when a site is reconnected, its proxy objects must be updated to reflect the current state of the network.

There are no explicit indexing techniques for this distributed object manager.

7.4 Clustering

There are no clustering features described for the distributed object manager for Smalltalk-80 systems.

8 Comparison

Four issues related to storage management in object-oriented data base management systems have been discussed. These issues have been presented with references to O_2, EXODUS, LOOM, Postgres, Mneme, and a distributed object manager for Smalltalk-80. Each system emphasizes different aspects of these four issues. Now, a comparison between the different storage managers for these OODBMS will be made.

All storage managers but Postgres view a tuple as an object with objects being able to reference other objects. Postgres, however, views a relation as an object. All but EXODUS understand what an object is. EXODUS views the objects as uninterpreted byte sequences. EXODUS represents a large object on disk as a B+tree index on byte position within the object and a collection of leaf blocks.

Mneme understands what an object is, and how it is internally structured. The distributed object manager for Smalltalk-80 knows the internal structure of the object and its location in memory. It handles "object faulting" in the same manner as virtual memory handles a page fault. O_2 has a very good understanding of objects. They are grouped into tuples, lists, and sets.

The LOOM storage manager understands the internal structure of objects as well as something about object classes. It swizzles disk references into short object identifiers, which are then mapped through a *resident object table* when dereferenced. For memory references, it converts a memory object into a *leaf*, which contains only a small amount of the state of the object (enough to allow it to be expanded later). For non-memory resident object, it uses a special *lambda* value to indicate the reference.

The Postgres storage manager has semantic knowledge about objects and stores them accordingly (that is, it tries to store the tuples contiguously). When the object is too long to be stored in a

[1] No satisfactory solution has been thought of to handle traumatic disconnections such as software and hardware failure.

page, it splits the object up by tuples and chains them together.

The storage managers use different update and recovery techniques. Mneme puts a *handle* on an uncommitted object to enforce mutual. It notifies the waiting clients of the availability of the object using *event notification*. It recovers the objects by using the *object logs* after a crash.

The distributed object manager for Smalltalk-80 has no explicit update and recovery policies. LOOM does not support these two policies either, since it is only a virtual memory system with no managerial capabilities.

The O_2 object manager uses an exclusive lock when updating an object, but recovery is not implemented. The EXODUS storage system uses two-phase locking on byte ranges when updating objects. It also provides an option for locking an entire object. It uses before/after image logging and in-place updating at the object level for small object recovery. For large object recovery, it uses a combination of shadows and logging.

The storage manager for Postgres does not store entire updated objects. Instead, it stores them as *deltas* (differences) from previous versions, and reconstructs them when they are referenced later.

The storage managers use different object referencing techniques. Mneme accesses the client via a *client identifier* (CID), whereas when an object references another object, a *persistent identifier* (PID) is used. A PID can be converted to a CID, and vice versa.

O_2, EXODUS, and LOOM also use identifiers to reference objects. O_2 identifies persistent objects by their *object identifiers* (OIDs), which are the physical address of objects on disk. In memory, a persistent object is referenced through the *record's identifier* (RID). The temporary records that exist in virtual memory are referenced through a TempID (which is made up of a virtual memory address and a machine tag). However, a major problem is moving objects on disks or memory without changing their identifiers.

EXODUS identifies both storage and file objects by their *object identifiers* (OIDs). The OID for small objects is a pointer to the object on disk. For large objects, it contains pointers to other pages which store the object.

LOOM identifies an object by its *object pointer* (Oop). Oop is an index into the *resident object table* (ROT), which contains the actual main memory address of each resident object.

The distributed object manager for Smalltalk-80 references objects by their names, without concern as to their physical location. A *proxy object* is used to represent a remote object locally.

The storage manager for Postgres references records by scanning the relation sequentially. If an updated version of the record is referenced, it is reconstructed by merging the original copy of the record with the delta version of the record. Any number of secondary indexes can be constructed for any base relation.

Since many objects can be referenced from many places, not every object can be clustered with the objects it references. Therefore, some of the storage managers for OODBMS do not provide clustering. For example, the distributed object manager for Smalltalk-80 has no advanced clustering features besides what is provided in a centralized Smalltalk-80 system. LOOM does not have to

consider any clustering issues since it is a virtual memory system.

The storage managers for EXODUS and Postgres do not provide any clustering on their own, unless the user instructs them to try to put the records together. This allows users to decide which sets of records should be clustered together, giving them more control.

Mneme and O_2 provide clustering. Mneme divides files into one or more pools of objects. The objects in each pool are managed with the same storage allocation and object management strategies. O_2 clusters a composite object and its object or value components using *cluster trees*.

The above six storage managers are of the interpreter-storage manager architecture. But this architecture is not the only one that has been proposed to support persistent objects. Persistent memory can be another kind of storage architecture for OODBMS. In this point of view, an object persists independently of its type or the storage medium on which it resides, so long as it cannot be garbage collected. This extends the concept of automatically managed, garbage collected virtual memory to a storage architecture that also includes database objects. In this structure, all processes and objects share a single virtual address space. Therefore, it is possible to share objects via pointers. The structure also provides a recovery scheme which itself is at the virtual memory level. It is based on timestamp and sibling page techniques and has low space and time overheads. It eliminates the distinction between transient and persistent objects. This architecture has not be implemented by any existing systems. [Tha86]

9 Conclusion

This paper has addressed 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. Six different object storage models (O_2, LOOM, EXODUS, Postgres, Mneme, and a distributed object manager for Smalltalk-80) were examined to determine how they addressed these four areas. The techniques employed by these models were then compared and contrasted.

Future research into the area of storage management for OODBMSs is, and should continue to be, focussed on the topics of indexing and clustering. These two topics are related in that indexing is the basis for clustering objects. Proper indexing techniques, which will aid in the decision of how to cluster related objects, have yet to be developed. Clustering and storage polices to handle growth (the addition of new objects) in the database also need to be researched. Persistent memory has been studied but needs to be implemented.

References

[CAC+90] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent Object Management System. In S. B. Zdonik and D. Maier, editors, *Readings*

in Object-Oriented Database Systems. Morgan Kaufmann Publishers, Inc., San Meteo, California, 1990.

[CDRS86] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the Twelfth International Conference on Very Large Data Bases*, August 1986.

[Dec89] D. Decouchant. A Distributed Object Manager for the Smalltalk-80 System. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. Addison-Wesley Publishing Company, New York, New York, 1989.

[KK90] T. Kaehler and G. Krasner. LOOM: Large Object-Oriented Memory for Smalltalk-80 Systems. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, Inc., San Meteo, California, 1990.

[Mos90] J. E. B. Moss. Design of the Mneme Persistent Object Store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[MS88] J. E. B. Moss and S. Sinofsky. Managing Persistent Data with Mneme: Designing a Reliable, Shared Object Interface. Technical Report COINS TR 88-67, University of Massachusetts, July 1988.

[SRH90] M. Stonebraker, L. A. Rowe, and M. Hirohama. The Implementation of POSTGRES. In *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[Sto90] M. Stonebraker. The Design of the Postgres Storage System. In S. B. Zdonik and D. Maier, editors, *Readings in Object-Oriented Database Systems*. Morgan Kaufmann Publishers, Inc., San Meteo, California, 1990.

[Tha86] S. M. Thatte. The Design of the Postgres Storage System. In *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, September 1986.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 Object Manager: an Overview. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, 1989.

[VDD+90] F. Velez, V. Darnis, D. DeWitt, P. Futtersack, G. Harrus, D. Maier, and M. Raoux. Implementing the O_2 Object Manager: Some Lessons. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, September 1990.

[ZM89] S. B. Zdonik and D. Maier, editors. *Readings in Object-Oriented Database Systems*, chapter 4, pages 237–241. Addison-Wesley Publishing Company, New York, New York, 1989.