Object Storage Management Architectures

Alexandros Biliris[1] *and Jack Orenstein*[2]

[1] AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974, USA.
Email: `biliris@research.att.com`

[2] Object Design Inc., One New England Executive Park, Burlington, MA 01803, USA.
Email: `jack@odi.com`

Abstract. This paper examines the architectural issues in building storage systems for object-oriented database management systems (OO DBMSs) and persistent languages. We survey techniques for placing small and large objects on disk and disk space management, and we present client-server architectures for OO DBMSs. We describe alternatives in making a programming language persistent and in particular, we discuss pointers and three pointer dereference mechanisms: import/export, software dereference, and hardware dereference.

1 Introduction

Relational database systems cannot meet the requirements of advanced database applications such as electronic CAD, mechanical CAD, geographic information systems and computer-aided publishing [28]. These applications are characterized by extremely complex data structures, and complex patterns of computation and navigation. They are implemented in 3GLs such as FORTRAN, C, and C++, and because these applications are both computationally intensive and interactive, the performance demands are severe. The requirements of these applications are quite different from "traditional" ones, in which high throughput for simple manipulations of simple data models is the goal.

The data management requirements are similar to those of more traditional applications in that both persistence and transaction management are essential. Associative queries and related features such as views are useful but not essential. It is often the case that large volumes of existing code, (typically in FORTRAN and C) need to be supported, and of course there are no queries in this code. There is, of course, associative access, but it is coded in very low-level terms, e.g. a binary search subroutine. The acceptance of query processing, provided in the database language or in some sort of collection libraries, is a slow process by the developer's of these non-traditional applications.

Relational databases are not a good fit for the needs of these non-traditional applications. The modeling and performance requirements differ, as noted above. The separation of logical and physical schemas, supported by relational databases, is not so important for CAD applications - performance requirements are so stringent that concerns about representation are part of the application developers conceptual view. There is also a “cultural barrier”. Developers of non-traditional applications either have no experience with database systems, or have been disappointed by the performance and modeling problems of relational database systems, (i.e. problems in applying these systems to their applications). Because of these problems, many developers have relied on “home-grown” data management systems. Many of these developers are now using or investigating OO DBMSs because they solve the same problems but have advantages in ease of use, performance, safety, or cost.

Like any relational DBMS, an OO DBMS offers persistence, transaction management, and associative queries. An OO DBMS also has the benefits of a programming language, namely generality (as opposed to an incomplete languages such as SQL or QUEL), low-level efficiency, and object-orientation [4].

In this paper we examine the architectural issues in building storage systems for object-oriented database systems and persistent languages. We start in Section 2 with techniques of placing small and large objects on disk, and disk space management. Section 3 describes client-server architectures. In Section 3.3 we discuss mechanisms for caching objects or larger units of data on the client workstation. Section 4 describes alternatives in making a programming language persistent. Section 5 discusses pointer dereference mechanisms. Section 6 summarizes our work.

2 Organizing Objects on Disk

Most database systems store objects on *slotted pages* [3, 37]. The basic idea is as follows. A slotted page contains a header and a variable size array of slots at the beginning and at the end of the page, respectively. The offset of each object from the beginning of the page is kept in one of the slots. When the first object is inserted into an empty page, it is placed right after the page header and slot[0] points to that location. Subsequent records are inserted right after the previous one, and their offsets are stored in the next slots (toward the beginning) of that page. So, as the page gets filled up, records and their slots grow towards each other and the free space is squeezed. Details for organizing and administering slotted pages vary considerably from system to system. For instance, slots can grow from left (right after the fixed-size page header) to right and objects from right (starting at the end of the page) to left.

The physical address of the object consists of the page number and the disk volume in which the object is stored and the slot number that contains the object's offset within the page. This way, objects can be shifted within the page, e.g., to make room for new objects when existing ones are deleted, without changing their address.

Object ids (oids) uniquely identify objects for the life of the database (sort of). In some systems, oids are the physical address of the objects. Other systems use logical oids, i.e., a value that maps through some kind of table to the actual location of the object. Physical oids are large and fast (no need for translation to locate an object) and logical oids are more flexible (allow free movement of objects). With physical oids, object can be moved too; however, the new address of the object must be kept in the original place so references to it can be forwarded to the new address. There are also combinations. In Mneme for example [30], oids always name objects within a single file. So, if an object needs to refer to another one in the same file the oid of the referenced object would suffice to identify it. References to objects in other files are made with one level of indirection. In Section 4.1 we discuss the relationship between programming language pointers and oids.

In any storage system, there must be a mechanism that gathers "related" objects together; database *files* serve this purpose. On the physical level, a file consists of a number of pages and/or segments – a number of physically contiguous pages. Usually, pages and the objects stored on them belong to one file only but there are systems that allow object replication in many files [21]. File organizations are classified according to the technique used to insert objects in them: in *unsorted* files a new object is placed in any of the file's pages or it is appended at the end of the file; in *key-based* files – such as sequential files, hash files, B-tree files – an attribute value of the object is used to determine the page in which the object is stored [19]. Note that in all key-based file organizations an object may have to be moved to a different page than it was originally assigned to. This limits their practicality because pointers that were pointing to the relocated objects become dangling. To avoid this problem, many systems use unclustered indices; i.e., indices that store a pointer to the object rather than the object itself.

At minimum, files must provide facilities for sequencing through the objects they contain. In addition, good object stores and the OO DBMS language should allow applications to exercise some control over the physical placement of objects in a database. Although there are proposals for automatically clustering objects, what seems in practice to have most impact on performance is user specified clustering hints. Clustering hints may take the form "put the object near this object", "put the object on that disk volume", "do not put more objects on the page you place the new object", etc. For example, the latter may be useful in reducing contention because of locking for frequently accessed pages.

2.1 Large Objects

An object that can't fit entirely in a single page is termed *large*. Multimedia applications operate on very large objects such as digital images, continuous media (digital audio and video), documents and books. To display images, show movies, or play digital sound recordings, they require DBMSs to manipulate large objects as efficiently as possible [1]. Large objects may also appear in other non multimedia kind applications.

There are several functional requirements imposed on a storage manager for manipulating large objects. First, ideally, the storage manager must have been designed in a way that can support objects of virtually unlimited size (within the bounds of the physical storage available). Second, the large object abstraction must support operations that deal with a specific number of bytes within the object: *read*, *write*, *insert*, *delete* bytes starting at arbitrary positions within the object, and *append* bytes at the end of the object.

There are also several performance requirements. First, the cost of allocating a large number of disk blocks must be minimal; this will reduce the cost of creating a large object. Ideally, the allocation cost should be 1 disk access regardless of the size of the requested space or the database size. The performance of successive appends at the end of the object is of particular importance since this is the expected way of creating very large objects.

Second, to perform a byte range operation we must first seek to a specific byte in the large object which requires efficient random access, i.e., the cost of locating any given byte within the object must be independent of the object size. This requirement by itself rules out simple solutions such linking in a linear fashion the pages on which the object is stored. After the first byte is located, we need to efficiently sequentially access the remaining bytes of the range to perform the operation. In turn, good sequential access performance means that the I/O rates in reading/writing a large chunk of bytes must be close to transfer rates. In other words, the cost of byte range operations must depend on the number of bytes involved in the operation rather than the size of the entire object. For this to happen, disk seek delays must be minimized which in turn requires that disk space is allocated in large units of physically adjacent disk blocks, rather than on a block-by-block basis.

Finally, regarding updates, small changes should have small impact; e.g., inserting few bytes in the middle of the object should not cause the entire object to be re-organized.

In early solutions to large object management [3, 20, 13], single disk blocks were used to store consecutive byte ranges of the object. The problems with the above schemes is lack of support for unlimited size objects and the loss of sequentiality. As a result, reads are slow because virtually every disk page fetch most likely results in a disk seek. More recent solutions store the object in segments of physically adjacent blocks [11, 27, 5]. From the latter three, EXODUS [11] uses fixed-size segments, Starburst [27] uses segments of fixed pattern of growth, and EOS [5] uses variable-size segments. Comparative performance results of the last three architectures are presented in [6].

In EOS [5], large objects of any size are stored in a sequence of variable-size segments allocated using the scheme discussed in Section 2.2. These segments are pointed to by a "count" B-tree-like structure. Each node of the tree contains a sequence of (page-no, cnt) pairs, indicating the child page id and the number of bytes stored under this child from the beginning of the node. Thus, the rightmost pair of a node gives the total number of bytes stored below it, and if the node is the root this value provides the total object size. When the object's size is known in advance, maximum size segments are used to hold the field. Otherwise, successive segments allocated for storage double in size until the maximum segment size is reached; then, a sequence of maximum size segments is used until the entire large object is stored. In either case, the last segment is trimmed, i.e., its unused blocks at the right end are freed. When updates (byte range deletes and inserts) are performed on the large object, segments may have to be broken up into smaller ones. If during an update operation, the two parts of a split segment are smaller than some user specified number of blocks, the segments are merged into a larger one.

2.2 Disk Space Management

Disk space management addresses the problem of allocation and deallocation of disk blocks to database files. There are two broad techniques, block-based and extent-based allocation.

In *block-based* allocation schemes each block is addressed individually (no notion of physical contiguity). Free blocks can be managed by a simple linear linked list. This is the approach taken by the design of Unix. The file directory contains an array (*inode*) of 13 pointers: Pointers 0 through 9 contain the addresses of the first 10 blocks. Pointer 10 points to a block that contains the addresses of the next n blocks, where n is the number of block addresses that can fit in a block. Pointer 11 points to a block that contains n pointers each of which points to a block with n block addresses (total n^2). Similarly for pointer 12. Total addressability: $10 + n + n^2 + n^3$. The expected performance for databases is going to be poor for two reasons. First, there is no notion of physical contiguity and second the scheme may lead to an unbalanced directory for large files.

In *extent-based* allocation schemes, disk space is allocated in chunks of contiguous blocks. An example of an extent-based allocation scheme is the one used in EOS [5] and it is based on the binary buddy system [25]. Starburst uses a similar scheme [27].

A disk area is partitioned into a number of equal-size *extents*. An extent is a disk section of physically adjacent disk blocks. There is an allocation map directory associated with each extent that encodes the status (free or allocated) of each block in the extent. Disk space allocation is performed in terms of *segments* – variable-size sequences of physically adjacent disk blocks taken from one of the extents. For example, a 128-Mbyte disk volume could be partitioned into sixteen 8-Mbyte extents. This allows allocation of segments up to 8-Mbyte long.

In the binary buddy system, segments of a given size can start only at blocks whose block number is divisible by the size of the segment. For example, a segment of size 8 can start only at blocks $0, 8, 16, ...,$ etc. Suppose that a free segment of size $n = 2^t$ exists in the extent. In searching for this free segment, the buddy system always starts by checking the status of segment $S = 0$. If S is of size $m \neq n$, searching continues recursively at segment $S = S + \max(n, m)$ until the desired segment is located. Thus, in order to locate a free segment of a given size, there is no need to check every single byte of the allocation map.

If there is no free segment of size 2^t we find the smallest free segment of size 2^j such that $j > t$. Then this segment is split in half into two buddies each of size 2^{j-1}. One of these 2^{j-1} block segments is marked as free and the other is split up into two 2^{j-2} block segments. This process continues recursively until a segment of size 2^t is finally made up.

Conversely, on deallocation of a segment of size 2^t, the allocation map is updated to reflect the change. To avoid fragmentation, the buddy of the just deallocated segment is examined for possible coalescing. The buddy of a segment can easily be found by simply taking the exclusive OR of the segment address with its size. For example, the buddy of segment $6_{10} = 0110_2$ of size $2_{10} = 0010_2$ is segment $0110_2 \oplus 0010_2 = 0100_2 = 4_{10}$. If both of these 2-block buddies are free, they are merged into the larger free segment 4 of size 4.

Notice that whereas segments are internally managed as if their sizes are some integral power of 2, an application may request the allocation of a segment of any size. Details of the algorithms can be found in [5].

3 Client Server Architectures

In a client-server architecture, the database resides on the database server machine. Objects in the shared database are accessed over computer networks by applications programs running on client workstations. Client/server architectures could be classified according to the way they perform the following functions.

– Method execution site.
  This refers to whether database queries and object functions in general are evaluated on the server or client site or both. Traditionally, relational systems do what is called *query shipping*: queries are shipped to the server and the query is executed there. On the other hand, most OO DBMSs employ the *data shipping* approach: data is shipped to the client where queries as well as navigational type of operations are executed.
– Unit of data transfer between the server and client.
  Depending on the unit of transfer, we have *object servers* and *page servers* where the unit of transfer is an object and a page, respectively. In *file servers*, the client and the server communicate via a remote file system such as NFS [32].
– Caching and unit of data replication.
  The goal of caching is to reduce the number of messages sent to and the need to obtain data from the server in the first place. Thus, during normal transaction execution, a portion of the database as well as locks are cached on the clients. Consequently, several copies of a shared object can exists in more than one application cache at the same time. The terms *intra-transaction* and *inter-transaction* caching refer to caching within a transaction and between transactions, respectively. If we keep in the cache objects of committed transactions to be used by subsequent transactions we need to address the *cache consistency* problem. This is because between the time one transaction committed (and therefore all its locks are released) and a new one starts, the copy of the object on the server may have been updated by another transaction and so the object cached on the client becomes invalid.
– Unit of locking and recovery.
  Access to shared data must somehow be synchronized to ensure the ACID properties of transactions. Synchronization may occur at various granules such as files, pages, and objects.

Although the above functions are conceptually more or less orthogonal, the choice between the above approaches has significant impact on the overall design and implementation complexity of the system as well as performance. Usually, but not necessarily, the unit of transfer, locking and cache consistency is the same. For example, object servers employee object-level locking and cache consistency protocols, while virtually all existing page servers support locking and cache consistency protocols at the level of a page. Other combinations however are possible. For example, [29] proposed a scheme in the context of shared disk systems where two-phase locking is used on objects to ensure serializability and physical locks on pages for cache maintenance. Moreover, algorithms with adaptive granularity have been proposed in the context of shared disk systems [23] and main memory databases [18] that, properly modified, can also be used in a client server architecture. These algorithms use the idea of *lock de-escalation*: locks are obtained at some coarse granule and if, later on, data contention increases locks are de-escalated into finer-grained locks at the page or object level.

OO DBMSs using the object server approach include the MCC's Orion1 [24], early versions of GemStone [14], a prototype of O_2 [39], and Versant [40]. Storage systems that can be classified as page servers are EXODUS [12], EOS [8], ObjectStore [26], O_2 [15], GemStone [9]. In the Ontos system [34], users can choose between the object and page server approach; the choice is indicated statically on a per collection basis. Finally, Objectivity [33] is a page server using NFS to transfer pages.

3.1 Query Execution Site

For a server to execute functions on objects it must understand the way objects are stored in the database as well as know about the types of objects. Basically, the functionality of the DBMS is replicated in both the server and the client. Since the server is aware of objects, it is capable to retrieve a query from the client that selects some objects from a collection of objects, process the query locally, and return the result to the client. The advantage of executing functions at the server is that if the result of the query against the collection is a small subset of the collection, it avoids the communication cost of transferring the entire collection to the client. As an example, consider an application that requires a full scan of all the fingerprints in the database so they can be compared with a target fingerprint. Transferring the target fingerprint to the server so it can be compared there is more efficient that transferring the available fingerprint collection to the client.

There are however several problems with this approach. First, for the server to apply methods on objects, user code has to be linked with the server's code which makes this scheme practically difficult to employ. Second, if methods can be applied on both the client and the server, their caches must be synchronized. This is because an object update performed on the client site must somehow become known to the server before the server applies methods to this object cached in its own pool. Third, if the server is not powerful enough, the scheme may face scalability problems. The server may become the bottleneck of the system, since most of the computations are carried out at the server side, while at the same time workstations remain underutilized.

3.2 Unit of Data Transfer

Page servers deal with pages only thus, knowledge of object structure or behavior is unnecessary. When the client needs an object it first searches its cache. If the object is not found, the page in which the object resides is requested from the server. The server sets the appropriate lock on the page, searches its own buffer pool and if the page is not there it is retrieved from the disk; then, the page is sent to the client. In this scheme, the server's cache consists of page frames. The client's cache can be organized in terms of objects, pages or both.

Object servers respond to object-based requests and thus they must understand the way objects are stored in the database. When the client needs an object it first searches its object cache. If the object is not found, the request is sent to the server. The server searches its own buffer pool and if the object is not there, the page in which the object resides is retrieved from the disk. The object is then extracted from the cached page and sent to the client.

One disadvantage of this scheme is increased communication cost for accessing objects on the client that might have been clustered in the same page. For instance, assume a program running at the client that retrieves all employee objects from a database collection and displays some of their fields on the screen. Although, there may be many such objects in the pages of the collection, the client has to initiate a request to the server for each one of them. Also, object-level transfer may not be appropriate for large objects. Even if the application intends to access a subset of the object's bytes, the entire object would need to be transferred. This is the case for example, when the object is a digitized movie stored at the server's side, and the client wants to display on the screen a frame of that movie at a time.

Finally, in file-servers the DBMS runs on one machine and it access databases residing on a remotely mounted file system accessed via a remote file service such as NFS. Since NFS is part of the operating system, the DBMS can use this service directly to access database pages. The server however still has to provide concurrency control and recovery which are not usually provided by operating systems services. Thus to read or write a page, there must be two message requests. One message to the DBMS server to perform locking and recovery related activity, and the other to NFS to actually read or write the page, respectively. The latter – writing a data page on disk – actually involves an additional I/Os because the `inode` pointing to the block is written too.

A study that compares the performance of the above three architectures can be found in [16].

3.3 Caching - Data Replication

During transaction execution, objects fetched from the server as cached in a buffer pool. Typically, locks acquired on the server are also cached in some internal structures of the client. When the transaction commits, data and locks could be held for subsequent transactions. Caching items and/or locks on a client machine between transactions is generally referred in the literature as *inter-transaction* caching [43, 41, 12, 17, 35]. The following paragraphs elaborate on techniques with or without inter-transaction cashing. We assume page-level locking.

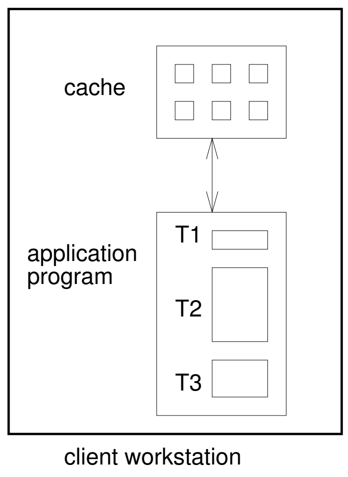

Fig. 1. Caching.

2PL with no Inter-Transaction Caching

Let's assume that an application running on a client workstation consists of three transaction blocks, as shown in Figure 1, that are executed sequentially. Suppose 2PL is used for concurrency with no intra-transaction caching. First time page requests require a round-trip message interchange between the client and the server. When the server receives the request, it first places the appropriate lock (read or write) on the page – i.e., a page requests implies a request for page locking so these two messages are combined into one. The client caches both the page and the lock mode acquired on it. Subsequent requests for pages that have already been cached do not require any interaction with the server as long as the lock mode held on the page is the same or stronger than the one required for the current access. Locks are held until the transaction terminates at which point the locks are released at the server and the client's cache is cleaned. Deadlocks are detected at the server using some centralized deadlock detection mechanism.

Referring to Figure 1, when the transaction T1 commits, its locks are released at the server so that other transactions may access them. When the next transaction T2 begins, it sees a clean cache. Objects needed by this transaction must be requested from the server even if they were cached for the previous transaction. The server will have to acquire the appropriate locks on the objects before it transmits them to the client.

Data/No-Lock Cache 2PL

In this algorithm, pages cached in the client are retained across transaction boundaries. However, locks acquired during a transaction are released at the end of the transaction. When a transaction accesses, for the first time, a page cached in the client's pool by a previously running transaction, a read or write lock for this page must be requested from the server. The client is blocked until the server replies back that the lock is acquired. The page itself does not have to be sent to the client as long as the two copies on the client and the server are the same. If the server discovers that its local copy of the page is more recent than the one cached on the client, it sends the fresh copy to the client along with the reply.

For this scheme to work, for each page cached on a client workstation, the server must know the client that performed the last update on the page. If a client requests a lock on a page that has been modified by another client, the server attaches the page itself to the reply message.

This technique does nothing more than to essentially extend the available server pool to include the buffer pool space of all clients connected to the server. Thus, compared to non-caching 2PL this method reduces the number of I/Os for pages cached in the client but swapped out of the server's pool.

Data/Lock Cache 2PL (Callback Locking)

In *callback locking* when a transaction terminates, pages used by the transaction as well as their lock modes are retained in the client's cache to be used for the next transaction. No interaction with the server is needed when a client accesses a cached page and the required lock is covered by the lock already held on the page. However, when a stronger lock is needed or a page is not present in the local cache, the server is contacted to place the lock, fetch the page or do both.

When the server gets such a message from a client, it broadcasts to all clients holding incompatible locks on the page, to give up their locks. If the client does not need the lock on the page – i.e., non of the currently running transactions holds the lock on the page – it releases the lock. On the other hand, if the lock is needed the client holds the lock until the transaction terminates. This mechanism is used in the Andrew File System [22] and ObjectStore database system [26].

Optimistic Locking

This technique assumes that a page found in the local cache is valid. That is, the application continues executing without being block when accessing cached pages. Lock upgrade requests are still sent to the server. At commit time, the server checks if conflicts developed during the normal transaction processing in which case the transaction is aborted [42]. In this algorithm, a transaction may continue executing even if the server knows the transaction will abort because of conflicts.

In a variation of the above algorithm, when the server receives a page updated by a committed transaction, it notifies all other clients holding copies of this page in their cache that the copy is invalid. This reduces the likelihood to abort a transaction at commit time. The server may also send the valid copy of the page along with the notification message.

4 Persistence and Programming Languages

The best OO DBMSs combine influences from databases and programming languages in a clean way. Ideally, the same type system applies to both transient and persistent data. That is, any type may have both transient and persistent instances. This means that the application developer works with one data model (or type system) and language, not two - one for transient data such as C and another for persistent such as SQL. Finally, this means that there need be no translation between in-memory and on-disk representations. Or if there is such a translation, it is the responsibility of the OO DBMS and not the application developer or the developer of each class that requires persistence.

An OO DBMS can be viewed as a conventional programming language extended with persistence, (and other features). This view leads to the question: what is involved in adding persistence to a programming language?

In FORTRAN the basic types supported are numeric and string types, and the only structuring facility is provided by arrays. FORTRAN (up through FORTRAN-77) does not have pointers. Thus, it would be easy to add persistence to FORTRAN.

Adding persistence to PL/I is more complex, because in addition to the types provided by FORTRAN, there are pointers to deal with. However, some pointers can be handled easily: Objects allocated in an “area” which refer to one another via “offsets” can be made persistent easily. The entire area can be written to disk and read back. The pointers, which are relative to the beginning of the area, never have to be adjusted following retrieval. Ordinary pointers cannot be handled so easily.

In Smalltalk and Lisp pointers are not accessible by the programmer. A runtime system providing persistence has much latitude in how inter-object references are represented and manipulated. Absolute addresses (i.e. ordinary pointers) can be used; an area and offset scheme can be used; and there are other possibilities. It is even possible to use more than one approach, since the implementation is hidden in a way that is not possible with a language such as C.

In Pascal, Ada, C and C++, the programmer has direct access to pointers. Adding persistence to these languages is difficult. CAD applications spent much of their time traversing pointers, and the approach to persistence should not require any significant change in the way programmers use pointers, or increase the cost of dereference. What this means is that the languages most important for CAD and many other applications are precisely the languages that are most difficult to extend with persistence.

4.1 Pointers and Object ids

Pointers and object ids serve similar purposes - both serve to identify objects - but in different contexts. A pointer is valid only during the execution of a program; it specifies a location within a 32-bit address space; and operations on pointers are extremely efficient since they are supported directly by hardware. In order to be useful in an OO DBMS, an object id has to be valid for the lifetime of the object, potentially beyond the lifetime of the process that creates the object. The address space is much larger, typically 64 to 128 bits, and access is usually slower. Object ids are not supported in hardware, so software mediation is required, and the dereference may involve an access to a disk or a request sent over a network.

In conventional application, (i.e., applications that don't use an OO DBMS), connections between objects are represented by pointers, and networks of objects are traversed by dereferencing pointers. In OO DBMS applications, objects are connected by object ids, and these object ids (oids, for short), have to be dereferenced. There is no question that oids, not pointers, must be stored in a database. The question is what the programmer deals with during the execution of an application. If the programmer is aware of oids, there are consequences for performance and ease of use. The programmer has to use an oid to refer to any potentially persistent object. If the programmer deals with pointers, then there must be a translation from oids to pointers and this raises questions about how in-memory and on-disk representations compare.

4.2 Object Layout

The in-memory and on-disk representations of an object may be the same or different. The former is determined by a compiler, (since the application is written in the language supported by the compiler), while the latter is under control of the OO DBMS. The OO DBMS may choose to store objects in some "neutral" format, (i.e. different from the layout generated by every compiler used to create an application accessing the database), or the layout may be identical to that of one compiler, (i.e. different from the layout generated by every compiler but one).

In many cases, the issue is decided by the approach taken to oids and pointers. If oids are stored in the database and pointers are used in the application, and if oids and pointers occupy different amounts of storage, then it is likely, but not certain, that the in-memory and on-disk representations differ. In general, some per-object processing is required in bring an object into memory. This processing may range from a mere transfer in the best case, to generating of an object with a different layout in the worst case. An intermediate case is one in which no reformatting is required, but some fixup is needed.

4.3 Retrieving and Updating Objects

Another key architectural issue is object retrieval. It is rarely a good idea to fetch a single object at a time. As we discussed in section 3, the request for an object typically may go out over a network and have to be serviced by retrieving an object from disk. Doing all this work for a single object is wasteful, so this raises the question of what other objects should be retrieved. The possibilities include the following: the requested object and connected objects, a logical cluster containing the requested object, or a physical unit containing the required object.

The retrieval may be implicit, triggered by a dereference, or explicitly requested by the programmer. When an object (or some larger unit) is going to be updated, it is necessary to record the fact for purposes of concurrency control and recovery. Ideally, the programmer would simply update the object and the OO DBMS would note the update. Also, if writes have to be noted explicitly, this not only diminishes source compatibility, but introduces an opportunity for subtle errors.

One approach, followed in [26], is to detect updates automatically as a result of virtual memory protection violation, see section 5.3. Another approach, followed in [2], is to have the compiler detect when an object becomes dirty, e.g., during an assignment, and pass flags to the underlying storage system.

5 OO DBMS Architectures

An OO DBMS architecture has to address all the issues identified above: what are the roles of oids and pointers? how are object retrieval and update specified and implemented? what are the in-memory and on-disk object representations? These issues are not completely independent of one another. This section will survey three OO DBMS architectures to examine how these problems have been dealt with. The architectures to be surveyed are the following:

– Import/export object managers: Each type has functions for translating between in-memory and on-disk representations.
– Software dereferencing: Function call interface to all data management operations, possibly hidden by syntactic sugar.
– Hardware dereferencing: Data management functions triggered by hardware interrupts.

For each architecture, we will describe how it solves the problems of dereferencing, retrieval, update, and layout. We will describe the consequences of the solution:

– *Source compatibility.* How does a program that manipulates persistent data differ from the corresponding program for transient data? This issue is strongly related to ease of use. If persistent data and transient data are handled differently, then writing new code and migrating existing code will both be more complicated than would otherwise be the case.

- *Binary compatibility.* Does code have to be recompiled?
- *Performance.* When during execution does the overhead for persistence show up?

5.1 Import/export Object Managers

An import/export object manager depends on the presence of a pair of functions for carrying out translations between in-memory and on-disk representations. This approach is common in "home-grown" systems and, for example, the NIH class library.

Persistence is defined by reachability. That is, if object A refers to object B, and A needs to be persistent, then B has to be persistent also. (In some cases, the definer of type A may decide that the import/export functions do not need to propagate across certain pointers, e.g. if B is purely transient data that is only of interest within the scope of one process.)

In this architecture, only the import and export functions deal with oids, by turning them into ordinary pointers to ordinary transient objects. This means that object layout on disk is determined by the export functions. Retrieval is requested by the programmer, and the extent of the retrieval is determined by the import functions. Obviously, the finer the granularity, the more frequent are the requests for retrieval by the programmer. The request from the programmer may offer some advice about how much to fetch. Updates are also requested explicitly; updated objects are written to disk by running the export functions on updated objects.

Concurrency control, if implemented at all, is typically coarse-grain. Recovery is often not supported.

This approach to persistence treats class implementers and class users differently with respect to source compatibility. A class implementer must provide import and export functions and must therefore be aware of both representations. A user of this class need not be aware of the different representations, but need only initiate the retrieval or update. One aspect of representation that is relevant to the user of a class is the extent of the retrieval - when a retrieval occurs, the user must know how far traversal can proceed before another retrieval has to be issued. Binary compatibility is not an issue. Once an object has been imported, then already compiled code manipulating the object can be run without recompilation, provided the retrieval brings in all the objects that will be touched. For example, consider a linked list, composed of individual nodes. If there is a function that traverses the entire list, (e.g. mapcar), then this function can be applied to a persistent list following a retrieval that fetches the entire list. If retrieval of a node does not result in retrieval of a successor node, then the mapcar implementation will fail.

The performance of this approach depends on the granularity of retrieval. Import functions, which are written per type, determine the granularity of access, and this granularity may be too fine for some applications and too coarse for others. Each retrieval carries an overhead cost due to network and disk access costs, so if granularity is too fine, then performance will suffer due to the accumulation of these overhead costs. If granularity is too coarse, then performance can suffer due to unnecessarily high transfer times and the cost of reformatting objects.

5.2 Software Dereferencing

In a software dereferencing scheme, there are distinct types for pointers to transient objects and pointers to potentially persistent objects. The programmer must be careful to use the correct kind of pointer. A pointer to a potentially persistent object is dereferenced in software. This operation, sometimes called *swizzling*, checks to see if the target object is present in the applications address space [31]. This involves at least a hash table lookup. If the target is not present, it is fetched. Once the object is present, the ordinary dereference, (the kind that occurs for ordinary transient pointers), can take place. In some implementations, the physical address is cached to optimize later dereferences.

In languages that do not provide direct access to pointers, (e.g. Smalltalk and Lisp), the distinction between transient and persistent pointers can be hidden. This is not possible in C and C++ because programmers have direct access to pointers. Persistent pointers appear to the programmer as a distinct type. Retrieval is triggered by the dereference of a persistent pointer. There are various policies that determine what should be retrieved. The options include retrieval of just the required object; objects in the same physical container, (e.g. a page); and objects reachable from the requested object.

The in-memory and on-disk object layouts are usually the same, but the architecture does not require this.

Transaction management relies on two-phase locking or optimistic concurrency control, and the unit of locking may be as small as a single object. Read sets are maintained during the software dereference. Write sets are more difficult to maintain this way, and if this can't be done, then the user has to indicate which objects have been modified by a transaction.

The software dereferencing scheme provides very poor source compatibility since it requires every variable and argument that could be bound to a persistent object to be declared as a persistent pointer. This is a serious problem when porting existing code. Binary compatibility is also poor, since code compiled to handle transient objects cannot manipulate persistent objects. Persistent pointers are typically larger than transient pointers. Typical CAD databases are so full of pointers that this results in a significant increase in storage requirements which in turn may lead to more I/Os [10]. There is a time penalty also, as software dereferencing is slower than ordinary dereferencing.

One advantage of the software dereferencing scheme is that it permits dynamic reclustering. Because persistent pointers indirect through at least one table, when an object is moved, only the tables need be updated, not each pointer to the object.

5.3 Hardware Dereferencing

A hardware dereferencing scheme does not require custom hardware; just the ordinary virtual memory management hardware. In hardware dereferencing, there is only one kind of pointer, and it can be used to refer to both transient and persistent objects.

There are two variations of this idea. The simplest implementation just maps an entire database into virtual memory, (e.g. as if it were one big PL/I area). Pointers may have to be located and adjusted for the base address. This limits database size to the size of virtual memory, (typically no more than 2^{32} bytes). A more complex but more practical scheme maps portions of the database into virtual memory dynamically [36, 26]. This allows arbitrarily large databases, but limits the amount of storage that can be accessed in a single transaction to the size of virtual memory. The rest of this discussion will focus on the second variation.

The dynamic mapping scheme relies on the manipulation of memory protection bits. Objects that have been fetched reside in unprotected memory. Objects that have not been fetched but are pointed to by memory-resident objects lie in read-protected memory. An attempt to dereference such a pointer results in a protection violation. An interrupt handler fetches the protected unit of memory, e.g. a page, reduces protection to write- protection, and resumes the offending instruction. It may be necessary to read-protect more memory, to accommodate pointers on the retrieved page.

If, during a dereference, the target object has already been fetched, there is zero runtime penalty. The machine code sequence implementing the dereference is the same one used for an ordinary transient pointer. Indeed, transient and persistent pointers are identical.

In-memory and on-disk object representations are identical. It might seem that a pointer stored in the database would have to be larger than 32 bits, (since the address space of a database is not limited to 32 bits). This is not the case - conceptually the value stored in the pointer is used as a key to a table which contains the full oid. The implementation does not require a table with an entry for each object; large chunks of memory can be handled by a single table entry.

Transaction management is based on two-phase locking or optimistic concurrency control. Read and write sets are easy to maintain automatically. Violation of read protection yields read set information. The first write to a write-protected region of memory generates another interrupt. The handler for this interrupt reduces protection to unprotected, and maintains the write set.

Hardware dereferencing schemes provide nearly ideal source compatibility. There is only one pointer type, so no code that manipulates pointers has to be rewritten. Code written and compiled for ordinary transient data will also work on persistent data, as long as the access takes place in the scope of a transaction. The only caveat is that code doing allocation might have to be modified so that allocation in the database can be implemented. Write sets are maintained automatically, so no source modifications are required to indicate dirty data.

Hardware dereferencing swizzles each retrieved pointer exactly once, when the page is fetched. All subsequent dereferences proceed at full speed, (i.e. as if they were ordinary transient dereferences). In hardware dereferencing schemes, clustering is determined at the time of allocation. If dynamic reclustering is required, then it is implemented above the basic dereferencing scheme, in a library or in an application.

5.4 The C++ Hidden Pointers

Besides the problem of pointers discussed above, there are additional problems in making the C++ programming language persistent. In particular, some C++ objects contain special pointers which point to function tables implementing runtime dispatch of function calls. Such pointers are contained in C++ objects of types that have virtual functions or virtual base classes. The pointers are called *hidden* pointers [7] because they are not accessible to the user. In the case of virtual functions, the hidden pointer points to a virtual function table that is used to determine which function is to be called. In the case of virtual base classes, the hidden pointers are used for sharing base classes [38].

Virtual functions are used for late binding. For example, a `Person` pointer `p` may point to a `Student` object. Assume that class `Person` defines a function `address` as virtual, and class `Student` defines its own version of `address` with a different body. Then when `p->address()` is invoked, the actual function body that is executed is determined at run time. If `p` points to a `Person` object, the `Person`'s address function is invoked. If `p` points to a `Student` object, the `Student`'s address function is invoked. The hidden pointers have to be handled specially, since the function tables are transient.

The basic scheme followed in [7] is to apply an overloaded `operator new` to invoke a constructor on an object that is just fetched from disk (and therefore the hidden pointers it contains are bad) to fix the hidden pointers. Note that the constructor must not initialize the values of the data members; it must have null body. For this to happen, the compiler generates a function for each class that sets the value of a global boolean variable to true before invoking the overloaded version of the `new` operator (the one that does not allocate any storage). Constructors are modified so that if this global variable is true they do not execute any user specified code in the constructor body. The effect of the constructor invocation is simply that the hidden pointers are assigned the right values.

In ObjectStore [26] the schema records the layout of each object. This information is used when a page is mapped into the client's address space. Pointers are relocated, as described in section 5.3, so that they are consistent for the current address space. Schema generation is the process of analyzing class declarations (or some equivalent, e.g. debug information), to understand object layout. During schema generation, two notable events occur. First, pointers to virtual tables are noted. Second, a table mapping type identifiers to virtual table pointers is created. This table is filled in at link time. During relocation, virtual table pointers are filled in by consulting the table.

6 Summary

In this paper, we examined the architectural issues in building storage systems for object-oriented database systems and persistent languages. We discussed techniques for placing small and large objects on disk, disk space management, client-server architectures and mechanisms for caching data on the client workstation. Finally, we have described three pointer dereference mechanisms – import/export, software dereference, and hardware dereference.

References

1. Special Issue on Digital Multimedia Systems. *Communications of the ACM*, 34(4), April 1991.
2. R. Agrawal and N. Gehani. ODE (Object Database and Environment): The Language and the Data Model. In *Proceedings of ACM-SIGMOD 1989 International Conference on Management of Data,* Portland, Oregon, June 1989.
3. M. M. Astrahan et al. System R: Relational approach to database management. *ACM Transactions on Database Systems*, 1(2):97–137, June 1976.
4. M. Atkinson, F Bancilhon, D. DeWitt, K. Dittrich, D. Maier, and S. Zdonik. The object-oriented database system manifesto. Technical Report 30-89, ALTAIR, September 1989.
5. A. Biliris. An efficient database storage structure for large dynamic objects. In *Proceedings of the Eighth International Conference on Data Engineering,* Tempe, Arizona, pages 301–308, February 1992.
6. A. Biliris. The performance of three database storage structures for managing large objects. In *Proceedings of ACM-SIGMOD 1992 International Conference on Management of Data,* San Diego, California, pages 276–285, May 1992.
7. A. Biliris, S. Dar, and N. Gehani. Making C++ objects persistent: The hidden pointers. *Software Practice and Experience*, 23(12):1285 – 1303, December 1993.
8. A. Biliris and E. Panagos. EOS User's Guide, Release 2.0. Technical report, AT&T Bell Laboratories, May 1993.
9. P. Butterworth, A. Otis, and J. Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):51–63, October 1991.
10. M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proceedings of ACM-SIGMOD 1993 International Conference on Management of Data,* Washington, D. C., pages 12–21, May 1993.
11. 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 Databases,* Kyoto, Japan, pages 91–100, August 1986.
12. M. J. Carey, M. Franklin, M. Linvy, and Shekita. Data caching tradeoffs in client-server DBMS architectures. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data,* Denver, Colorado, pages 357–366, May 1991.
13. H-T. Chou, D.J. DeWitt, R.H. Katz, and A.C. Klug. Design and implementation of the Wisconsin storage system. *Software Practice and Experience*, 15(10):943–962, October 1985.
14. G. P. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of ACM-SIGMOD 1984 International Conference on Management of Data,* Boston, Massachusetts, June 1984.

15. O. Deux et al. The O_2 system. *Communications of the ACM*, 34(10):51–63, October 1991.
16. D. J. DeWitt, D. Maier, P. Futtersack, and F. Velez. A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems. In *Proceedings of the Sixteenth International Conference on Very Large Databases,* Brisbane, pages 107–121, August 1990.
17. M. Franklin, M. Carey, and Livny M. Global memory management in client-server DBMS architectures. In *Proceedings of the Eighteenth International Conference on Very Large Databases,* Vancouver, British Columbia, pages 596–609, August 1992.
18. V. Gottemukkala and T. Lehman. The design and performance evaluation of a lock manager for a memory-resident database system. In *Proceedings of the Eighteenth International Conference on Very Large Databases,* Vancouver, British Columbia, pages 533–544, August 1992.
19. J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques.* Morgan Kaufmann, San Mateo, California, 1993.
20. R. L. Haskin and R. A. Lorie. On extending the relational database system. In *Proceedings of ACM-SIGMOD 1982 International Conference on Management of Data,* Orlando, Florida, pages 207–212, May 1982.
21. M.F. Hornick and S.B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Transactions on Information Systems*, 5(1):70–95, January 1987.
22. J. H. Howard et al. Scale and Performance in a Distributed File System. *ACM Transactions on Computer Systems*, 6(1):51–81, February 1988.
23. A. Joshi. Adaptive locking strategies in a multi-node data sharing environment. In *Proceedings of the Seventeenth International Conference on Very Large Databases,* Barcelona, Spain, pages 181–191, September 1991.
24. W. Kim, J. Garza, N. Ballou, and D. Woelk. Architecture of the ORION next-generation database system. *IEEE Transactions on Knowledge and Data Engineering*, 2(1), March 1990.
25. P.D. Kotch. Disk file allocation based on the buddy system. *ACM Transactions on Computer Systems*, 5(4), November 1987.
26. C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):51–63, October 1991.
27. T.J. Lehman and B.G. Lindsay. The Starburst long field manager. In *Proceedings of the Fifteenth International Conference on Very Large Databases,* Amsterdam, Netherlands, pages 375–383, August 1989.
28. D. Maier. Making database systems fast enough for CAD applications. In W. Kim and F. Lochovsky, editors, *Object-Oriented Concepts, Applications and Databases*, pages 573–582. ACM and Addison-Wesley, New York, NY, 1989.
29. C. Mohan and I. Narang. Recovery and coherency-control protocols for fast intersystem page transfer and fine-granularity locking in a shared disks transaction environment. In *Proceedings of the Seventeenth International Conference on Very Large Databases,* Barcelona, Spain, pages 193–207, September 1991.
30. J.E.B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.
31. J.E.B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.
32. *Network File System: Version 2 Protocol Specification.* Sun Microsystems, Inc., Mountain View, California, 1988.

33. Objectivity Inc. *Objectivity/DB Documentation V2*, 1993.
34. ONTOS Inc., Burlington, Massachusetts. *Ontos DB 2.2 Reference Manual*, 1992.
35. M. T. Ozsu, U. Dayal, and P. Valduriez. An introduction to distributed object management. In M.T. Ozsu, U. Dayal, and P. Valduriez, editors, *Distributed Object Management*, pages 1–24. Morgan Kaufmann, San Mateo, California, 1994.
36. V. Singhal, S. V. Kakkad, and P. R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Proceeding of the Fifth Int'l Workshop on Persistent Object Systems,* San Miniato, Italy, pages 11–33, September 1992.
37. M. Stonebraker, E. Wong, P. Kreps, and G. Held. The design and implementations of Ingres. *ACM Transactions on Database Systems*, 1(3):189–222, September 1976.
38. B. Stroustrup. *C++ Programming Language.* Addison-Wesley, Reading, MA, 1987. 2nd ed.
39. F. Velez, G. Bernard, and V. Darnis. The O2 object manager: an overview. In *Proceedings of the Fifteenth International Conference on Very Large Databases,* Amsterdam, Netherlands, pages 357–366, August 1989.
40. Versant Object Technology, Menlo Park, California. *VERSTANT System Reference Manual, Release 1.6*, 1991.
41. Y. Wang and L. A. Rowe. Cache consistency and concurrency control in client/server DBMS architecture. In *Proceedings of ACM-SIGMOD 1991 International Conference on Management of Data,* Denver, Colorado, pages 367–376, May 1991.
42. D. Weinreb et al. An Object-Oriented Database System to Support an Integrated Programming Environment. *IEEE Database Engineering Bulletin*, 11(2):33–43, June 1988.
43. K. Wilkinson and M. A. Neimat. Maintaining Consistency of Client-Cached Data. In *Proceedings of the Sixteenth International Conference on Very Large Databases,* Brisbane, pages 122–133, August 1990.

This article was processed using the LaTeX macro package with LMAMULT style