An Efficient Database Storage Structure for Large Dynamic Objects

Alexandros Biliris[1]

Computer Science Department, Boston University
Boston, MA 02215
Email: biliris@cs.bu.edu

ABSTRACT

This paper presents storage structures and algorithms for the efficient manipulation of general-purpose large unstructured objects in a database system. The large object is stored in a sequence of variable-size segments, each of which consists of a large number of physically contiguous disk blocks. A tree structure indexes byte positions within the object. Disk space management is based on the binary buddy system. The scheme supports operations that replace, insert, delete bytes at arbitrary positions within the object, and append bytes at the end of the object.

1. INTRODUCTION

The manipulation of large objects is becoming an increasingly important issue of many so called unconventional database applications such as geographical, image analysis, computer-aided design (VLSI, mechanical, software engineering), office automation for document processing and publishing, and multimedia presentation. The latter are perhaps the most noticeable applications that demand efficient storage and retrieval of large objects in that they require displaying images, showing movies, or playing digital sound recordings in real time [ACM91]. However, efficient manipulation of large objects is important in any object-oriented and extended relational database management system in supporting general-purpose advanced data modeling constructs such as long lists or ‘‘insertable’’ arrays.

From the perspective of a database storage system that stores objects on pages, an object is *small* if it can fit entirely in a single page; otherwise, the object is *large*. Ideally, objects of virtually unlimited size (within the bounds of the physical storage available) have to be supported and stored in a way that minimizes internal fragmentation, i.e., the storage utilization should be close to 100%. To reduce the cost of creating a large object in the database, the cost of allocating (and symmetrically, deallocating) a large number of disk blocks must be minimal; ideally, 1 disk access regardless of the space size.

Since an object may indeed be very large, the following operations dealing with part of the object must be provided and efficiently implemented: reading and replacing a byte range within the object, inserting bytes into or deleting bytes from a specified offset from the beginning of the object, and finally appending bytes at the end of the object. The last three operations may force the object to grow or to shrink.

There are two reasons why such piece-wise operations on large objects are important. First, there may be physical constraints that would make it impractical or even impossible to build, retrieve or update a large object in one chunk, such as when the address space of the program is smaller than the object size. For instance, it would be unlikely (if not impossible) to create a very large objects in one big step; a more realistic scenario is that smaller (but sizable) chunks of bytes will be successively appended at the end of the object. Second, applications using large objects may want to access only a portion of a large object at a time. For instance, to retrieve an object, one would rather sequentially scan through the object in smaller portions, rather than access the whole chunk in one step − think of playing digital sound recordings, frame-to-frame accessing of a movie, etc. Similarly for the insert and delete operations, in manipulating a long list stored as a large object, elements may be removed from or new ones inserted at any place within the list; in multimedia applications, pictures may be annotated and movie spots may be edited to remove or add frames.

The above suggest that both random and sequential access should achieve good performance. Good random access implies that the cost of locating a given byte within the object is independent of the object size. This requirement by itself rules out solutions based on chaining the pages in which the large object is stored in a linear linked list fashion. Good sequential access means that the I/O rates in accessing a large object (or a large chunk of it) must be close to transfer rates. 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. Physical contiguity is also advantageous in computing environments where objects are moved between client and server machines [Ozsu91]. There is some experimental evidence that it is very important to be able to transfer large chunks of data thereby reducing data movement overhead [DeWi90]. Lastly, 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 this paper we present the design of the large object manager of EOS[2]. EOS is a storage system for experimental database implementation; it has also been used in teaching students implementation techniques for database systems’ components. The principle objectives of the EOS large object manager are summarized in the following:

1. Support for objects of unlimited size (within the bounds of the physical storage available).
2. Support for piece-wise operations: append bytes at the end, read and replace a byte range, insert or delete bytes at arbitrary positions within the object.
3. The cost of the above piece-wise operations must depend on the number of bytes involved in the operation, rather than the size of the entire object. In particular, we want to minimize disk head seeks so that I/O rates are close to transfer rates.
4. Allocation of large physically contiguous disk space should be fast; ideally, 1 disk access regardless of the size of the requested space or the database size.
5. Storage utilization must be very close to 100%.
6. The large object must be protected from transaction and system failures.

1. This research has been partially supported by the NSF grant CDA-8920936.

[2] *E*xperimental *O*bject *S*tore. Also, the goddess of dawn in Greek mythology.

Our disk space allocation policy is based on the binary buddy system [Knut73]. Strangely enough, although this scheme has been around for a long time and has been successfully used in file systems such as the Dartmouth Time Sharing System [Koch87], it has been largely ignored by most designers of database storage systems (with Starburst [Lehm89] being an exception). The buddy system performs fast allocation and deallocation of disk segments that differ in sizes by several orders of magnitude with minimal I/O and CPU cost. Previous work on the performance of the buddy system confirms the above, but it also suggests that this allocation policy is prone to severe internal fragmentation [Selt91]. Our design does not suffer from this problem because the unused portion of an allocated segment is always less than a page.

When a large object is created it is stored in a sequence of large variable-size segments, each consisting of physically contiguous disk pages. Subsequently, when byte range deletes and inserts are performed on the large object, its segments may have to be broken up into smaller ones. Thus, the segments that comprise the large object may have sizes that vary drastically. A B-tree-like structure is built to index byte positions within the object. The data structure is identical to the one proposed in Exodus [Care86]. However, because the leaf nodes of the tree are variable-size segments, the EOS algorithms for insert, delete and append are significantly different than the corresponding algorithms of Exodus.

The remainder of the paper is organized as follows. Section 2 reviews proposed solutions and argues that each of them satisfies a few but not all of the objectives stated above. The buddy system of EOS is discussed in Section 3. Section 4 presents the design of the large object manager. Finally, Section 5 concludes our work.

2. RELATED WORK

Work on large objects started in the context of relational systems. System R supported long fields with lengths up to 32 Kilobytes [Astr76]. The long field was implemented as a linear linked list of small segments, each 255 bytes in length, with the long field descriptor pointing to the head of the list. Partial reads or updates were not supported. An extension of the above solution appears in [Hask82] where long fields were stored as a sequence of 4K-byte pages with support for partial reading and updating. The long field descriptor is an array of entries each containing the address and length of each page. The maximum long field was about 2 Gigabytes.

The Wisconsin Storage System (WiSS) stores large objects in data segments called *slices*, [Chou85]. A directory to these slices is stored as a regular (small) record, and it may grow approximately to the size of a page. It contains the address and size of each slice. Each slice can be at most one page in length. Thus, with 4K-byte pages, the directory can accommodate approximately 400 slices, which gives an upper limit of 1.6 Megabytes to the object size. WiSS has been used as the storage engine of O_2, a commercial database product [Deux90].

The problems with the above schemes is the lack of support for unlimited size objects and the loss of sequentiality at the disk level. Blocks that store consecutive byte ranges of the object are scattered over a disk volume. As a result, reads will be slow because virtually every disk page fetch will most likely result in a disk seek.

Two other database storage systems that greatly influenced our design, are Starburst [Lehm89] and Exodus [Care86].

The Starburst long field manager uses extent based allocation with extents being organized into a binary buddy system. When the eventual size of a long field is not known in advance, 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 long field is stored. When the size of a long field is known in advance, maximum size segments are used to hold the field. In either case, the last segment is trimmed, i.e., its unused blocks at the right end are freed. The long field descriptor contains the size of the first and last segment and an array of pointers to all segments allocated to the long field; the size of intermediate segments are implicitly given by the size of the first segment and the known pattern of growth. The maximum field size that can be supported by this scheme depends on the possible segment sizes; the current implementation supports objects up to 1.5 gigabytes long [Lohm91].

Starburst does not gracefully handle byte inserts and deletes on large objects in that these operations require all segments to the right of and including the segment on which the update is performed to be copied into new segments. The applications this long field manager was intended for were large mostly read-only objects and not general-purpose objects that would need such length-changing updates.

Exodus handles large objects of unlimited size by storing them on data pages that are indexed by a B-tree-like structure, where the key is the maximum byte position stored in a leaf data page. This is a dynamic structure in that it can gracefully support byte inserts and deletes. To improve the performance of reading large chunks of bytes, clients can set the size of data pages of all large objects within a file to be some fixed number of disk blocks. However, this mechanism does not help applications that want to simultaneously optimize both search time and storage utilization because the size of the leaf page has diametrically different effects on them. Large pages waste too much space at the end of partially full pages (but offer good search time), and small pages offer good storage utilization (but require doing many I/O's for reads).

3. BUDDY SYSTEM

This section provides a brief overview of the design of the buddy system of EOS. The buddy system manages a number of large fixed-size disk sections of physically adjacent pages, called *buddy segment spaces* (buddy spaces, for short). *Segments* are variable-size sequences of physically adjacent disk pages taken from one of the buddy spaces. Internally, segments are managed as if their sizes is some integral power of 2; a segment of size 2^t pages is said to be of *type* t. Each buddy space includes a 1-page directory that consists of a *count* array and a page allocation map (*amap*), see Figure 1. The *count* array indicates the number of free segments of each possible segment type. If the maximum segment type is k, this array contains $k+1$ entries, from 0 up to and including k and the value of count[t] is the number of free segments of type t, i.e., of size 2^t pages. The allocation map indicates the status (free or allocated) and the type of each segment in the buddy space. No control information is stored in segments such as linking free segments of the same type. The entire process of allocating and deallocating segments is performed on the directory page only.

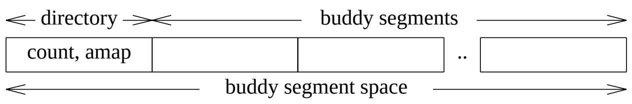

Figure 1. Organization of a buddy segment space.

With an encoding scheme, discussed in 3.1, each byte in the map represents the status and type of at least 4 pages in the buddy space. Since the directory is always 1 page, the maximum buddy space size, as well as the maximum segment size within the buddy space, depend on the page size. For a given page size PS the maximum segment size is $2PS$ pages. As an example, with 4K-byte disk pages, the maximum segment size that can be supported is 2^{13} pages (32 megabytes), with segment types from 0 through $\log_2(2\times4096) = 13$. Thus, assuming 2-byte count entries, the allocation map can be at most $4096-2\times14 = 4068$ bytes long; this allows the support of buddy spaces of at most $4068\times4 = 16,272$ pages (approximately, 63.5 megabytes). The above numbers show upper limits of our design in terms of maximum segment and buddy space sizes. To maximize performance, the buddy space size must be carefully matched to the physical properties of the disk storage such as the number of blocks per track.

3.1. The Allocation Map

The allocation map is a byte string; each byte B encodes the status and size of a segment that starts at one of the 4 pages $4B \cdots 4B+3$. If the left most bit of B is 1, Figure 2.a, a segment of size greater than or equal to 4 starts at page $4B$; the next bit indicates the status of the segment (0 for free, 1 for allocated), and the remaining 6 bits express the type of the segment. The scheme can support segment sizes of up to 2^{63} pages, more than what is really needed. If the left most bit of B is 0, Figure 2.b, the status of pages $4B \cdots 4B+3$ is indicated individually by the last four bits, one for each page. Finally, to indicate that pages $4B \cdots 4B+3$ are part of a segment that does not start at one of these pages, all bits of B are set to zero. The segment that includes those 4 pages is described in the first nonzero byte on the left of B.

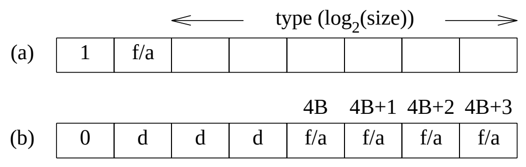

Figure 2. Allocation map encoding scheme.

As an example, Figure 3 shows the first few bytes of the allocation map. Byte 0 indicates that there is an allocated segment of size $2^6 = 64$ that starts at page 0; i.e., pages 0 through 63 are part of this segment. Byte 16 encodes individually the status of pages 64 through 67; pages 64 and 67 are free while pages 65 and 66 are not. Byte 17 indicates a free segment of size $2^2 = 4$ that starts at page 68. Finally, byte 18 encodes a free segment of size $2^3 = 8$ that starts at page 72.

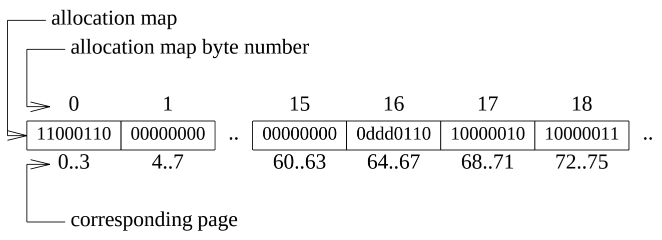

Figure 3. An example of the allocation map.

Segments of a given size can start only at pages whose page number is divisible by the size of the segment. For example, a segment of size 64 can start only at pages 0, 64, 128, 192, $\cdots$, etc. Suppose that a free segment of size $n = 2^t$ exists in the buddy space (this can be determined by examining the value of count[t]). 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. For example, referring to Figure 3, assume that we want to locate a free segment of size 8. We start at segment $S = 0$. The allocation map (byte 0) indicates that segment 0 has 64 pages, not 8. We proceed to check segment $S = S+\max(8,64) = 64$ (byte 16 of the map). Segment 64 is a 1-page segment and thus, we proceed to check segment $S = S+\max(8,1) = 72$ (byte 18 of the map). This is a free segment of size 8, and the searching stops here. 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.

3.2. Allocation/Deallocation of Segments

To allocate a segment of size 2^t, if the value of count[t] is greater than zero, the allocation map is scanned, as described above, and the segment is located. Otherwise, we find smallest type j such that $j > t$ and count[j]>0. Then the *amap* is scanned to locate a free segment of size 2^j, which then is recursively split in half until a segment of the desired size is finally made up.

Conversely, on deallocation of a segment of size 2^t, the allocation map is updated to reflect the change. In addition, however, the buddy of the just deallocated segment must be checked for possible coalescing; otherwise, most segments would quickly end up being decomposed into segments of type 0.

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 $0110_2 \oplus 0010_2 = 0100_2 = 4_{10}$. Symmetrically, just as the buddy of segment 6 of size 2 is 4, the buddy of segment 4 of size 2 is $0100_2 \oplus 0010_2 = 0110_2 = 6_{10}$. If both of these 2-page buddies are free, they are merged into the larger free segment 4 of size 4.

Whereas segments are internally managed as if their sizes are some integral power of 2, a client may request the allocation of a segment of any size, and the request can be fulfilled down to the precision of one block. Also, a client may selectively free any portion of a previously allocated segment, not necessarily the whole segment. The following example illustrates this point.

Assume a client requests the allocation of a segment of size 11. The binary representation of the requested size $11_{10} = 1011_2$ indicates that this segment consists of three contiguous segments of size 2^3, 2^1, and 2^0. The buddy system first finds a free segment of size 2^4, see Figure 4.a. Then, pages 0, 8, and 10 of this 16-page segment are marked as the start of allocated segments of of size 2^3, 2^1, and 2^0, respectively. Similarly, the binary representation of the number of the remaining $16-11 = 5_{10} = 101_2$ free pages indicates, in reverse order, the proper size of the free segments: 2^0 and 2^2. This situation is depicted in Figure 4.b.

Figure 4.c shows the page allocation status after the client frees 7 pages starting from page 3. Now, suppose the client frees page 10; this example demonstrates the iterative coalescing of pages from the situation depicted in Figure 4.c to the one in Figure 4.d. The buddy of the segment 10 of size 1 is $1010_2 \oplus 0001_2 = 1011_2 = 11_{10}$ which is also free and of size 1. Thus, segments 10 and 11 are coalesced into the single segment 10 of size 2. The buddy of segment 10 of size 2 is $1010_2 \oplus 0010_2 = 1000_2 = 8_{10}$ which is also free and of size 2; segments 10 and 8 are coalesced to segment 8 of size 4. Finally, segment 8 and its buddy $1000_2 \oplus 0100_2 = 1100_2 = 12_{10}$, each free and of size 4, are merged into the single segment 8 of size 8. Segment 8 of size 8 and its buddy 0 can not be merged because the latter is not a free segment of size 8.

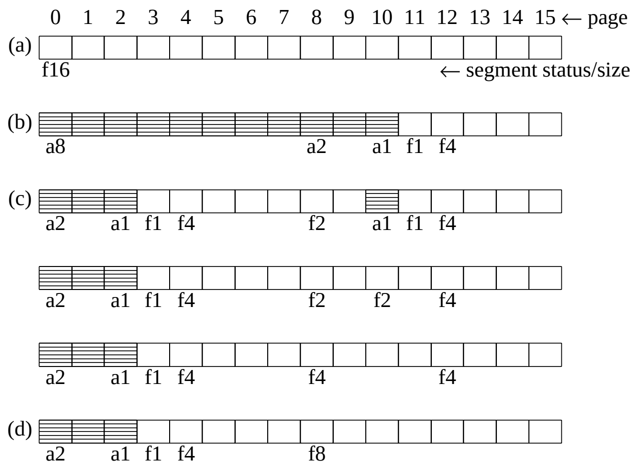

Figure 4. Allocation/deallocation of segments of any size.

3.3. Performance

The entire activity of allocating and deallocating segments is carried out by examining the directory page only which, depending on the page size, may control thousands of pages. Pages in the segment space need not be touched. Thus, for databases that can fit entirely in a single buddy space, at most one disk access is needed to serve block allocation (and deallocation) requests, regardless of the segment size.

However, larger databases will have many buddy spaces and thus, on a space allocation request it is possible that the directory block of each buddy space may have to be visited to locate a segment of the desired size. To avoid this, we make use of a *superdirectory* that contains the size of the largest free segment in each buddy space currently in the database. On a segment allocation request, the buddy system inspects the superdirectory to eliminate unnecessary access to an individual buddy space directory, if the maximum segment size in that space is less than the one requested.

The superdirectory is a main memory structure created at start-up time and subsequently maintained by the buddy space manager. Initially, it indicates that each buddy space available in the system contains a free segment of the maximum size possible. This information may be erroneous. However, as buddy spaces are visited for allocation or deallocation, their corresponding directory is examined and the correct values start being placed in the superdirectory. In other words, the first wrong guess about the maximum segment size available in a particular buddy space will correct the superdirectory information regarding this buddy space. Finally, it is worth mentioning that the superdirectory does not have to be transaction protected (otherwise, it would quickly become a *hot spot*). It is enough to hold a short duration lock (also called *latch* [Moha90]) on the superdirectory during a read or update and release it right after this operation completes; i.e., the lock does not have to be held until the end of the transaction.

4. LARGE OBJECT MANAGEMENT

Large objects are stored as uninterpreted byte strings in a sequence of variable-size segments which are taken from the buddy system using the scheme described in the previous section. There are no holes in each segment in that all of its pages must get filled up except the last one which may be partially full. These segments are then pointed to by a ‘‘positional’’ tree, [Care86], which is a B-tree-like structure in which the keys are the positions of the object’s bytes within the segments. No control information is stored in the data segments since all information needed to access them is kept in their parent index pages. Figure 5 shows a few possible cases of how the structure may look like depending on how the object is created and the kinds of subsequent updates performed on it. (In Figure 5, we have made the unrealistic assumption that pages are of size 100 bytes, just to make calculations in our examples easier to follow.)

Each node N of the tree contains a sequence of $(c[i], p[i])$ pairs, one for each child of N, where $p[i]$ is the page number of the i-th child of N. The number of bytes stored in the subtree rooted at $p[i]$ is $c[i]-c[i-1]$. (Ordering of pairs starts with 0 and by convention $c[-1]=0$.) Thus, if the i-th pair of N is the rightmost pair, $c[i]$ gives the total number of bytes stored below N, and with N being the root, this value provides the total object size.

Each internal node of the tree, except the root, is stored in a single page. As in B-trees, internal nodes are required to be from half full to completely full of pairs. The root has at least two pairs and, in general, can accommodate fewer pairs than internal nodes. Although, EOS manages the internals of the large object root, the placement of the root on a database page is left to the client; e.g., the client may choose to place the root on a page along with roots of other large objects, or in a field of a small object to implement long fields[3]. Lastly, it should be obvious from the nature of the structure that there are no constraints on the size of the large object.

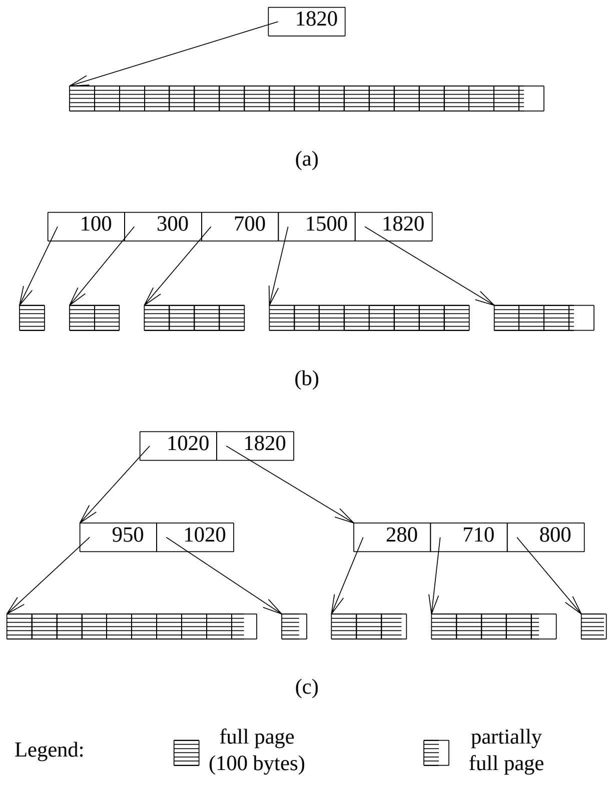

Figure 5. Examples of large objects.

As an example, the size of all three objects of Figure 5 is 1820 bytes because this is the count value of the rightmost pair of their root. The root of the object of Figure 5.a has a single pair pointing to a leaf segment consisting of $\lceil 1820/100 \rceil = 19$ pages.

[3] When a large object is opened for updates, clients may pass a parameter to EOS restricting the maximum size of the root to some given number of bytes.

The root of the object of Figure 5.c has two children. The left child indexes the first 1020 bytes and the right child the next 1820–1020 = 800 bytes. The right child points to three segments. The first segment contains the first 280 bytes of these 800 bytes, the second the next 710–280 = 430, and the third the remaining 800–710 = 90 bytes.

4.1. Append/Create

The append operation appends a number of bytes at the end of a (possibly zero size) large object. Since appends may be applied on the object one after the other, the final size of the object may or may not be known in advance. If the size is known a priori, it is provided as a hint to the large object manager who allocates a segment just large enough to hold the entire object. Then, each chunk of bytes is appended at the end of the previous one with no holes in between them. This is shown in Figure 5.a. If the object is larger than the maximum segment size, a sequence of maximum size segments are allocated to hold the object. When the eventual size of the object is not known in advance, we follow the growth scheme used in [Lehm89]; 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 object is stored. For example, the object shown in Figure 5.b could had been created by successively appending byte chunks of size less than a page.

At the end of these multi-append operations the last allocated segment is always trimmed, i.e., its unused pages (if any) at the right end are given back to the free space. Trimming a segment is trivial because the buddy system of EOS deals with allocation/deallocation of segments of any size with a precision of 1 page.

4.2. Search/Replace

The search operation reads N bytes starting from byte B of a large object (byte 0 is the first byte of the object). Only minor modifications need to be made to the algorithm proposed in Exodus so that variable size leaf segments can be supported. The algorithm is as follows:

(1) Read the root page and call it S.

(2) Save the address of S on the stack and binary search S to find the smallest $c[i]$ such that $c[i] > B$. Set $B = B - c[i-1]$, and $S = p[i]$. If S is not leaf, read S and repeat this step.

(3) S is now a leaf segment. Byte B within S is in page $P = S + \lfloor B/PS \rfloor$ at byte $P_b = B \,\mathbf{mod}\, PS$ within P. If N is less than the number of bytes in S after P_b, read, in one step, all pages from P up to $S + \lfloor (B+N)/PS \rfloor$; the search operation is now complete. If N is greater than the number of bytes in S after P_b, read P and all pages of S on the right of P; use the stack to obtain the rest of the bytes.

Suppose we want to read 320 bytes starting from byte 1470 of the object shown in Figure 5.c. To locate byte B = 1470, we find that $c[1]$ = 1820 is the smallest count of the root that is greater than 1470. Now we have to locate byte B = 1470–1020 = 450 in the child node pointed by $p[1]$. We repeat the same process in the child; i.e., we find that $c[1]$ = 710 is the smallest count greater than 450, and thus, we set $S = p[1]$, and $B = 450 - c[0] = 170$. Byte 170 is in page $S + \lfloor 170/100 \rfloor = S+1$, at byte 70 within that page. We read pages $S+1$ through $S+4$ to retrieve the first $(c[1]-c[0])-(100+70) = 260$ of the desired bytes. Then, the (logically) next segment needs to be retrieved for the remaining 60 bytes.

The cost of the above example operation, including indices except the root, is the cost of 3 disk seeks plus the cost to transfer 6 pages. If we had to perform this operation on the object of Figure 5.a, we would have to read 5 pages within a single segment and the cost of the operation would be 1 disk seek plus 5 page transfers.

The search algorithm can also be used for the byte range replace operation to locate and modify a given byte range within the large object.

4.3. Basic Insert/Delete Algorithms

This section presents the basic algorithms for the insert and delete operations. The next section addresses potential problems of these algorithms and proposes solutions. For a segment S, the following notation is used:

S_c	The number of bytes kept in S.
S_m	The number of bytes in the last page of S.

4.3.1. Insert

The insert algorithm inserts I_c bytes, taken from a buffer I, into a large object starting at byte B from the beginning of the object. Referring to Figure 6, assume that the proper segment S and the page P within S where insertion starts have been identified. Let also P_b be the byte position within P where the first byte of I should be placed. Conceptually, the insertion of the new bytes starting at P_b will create three segments: the left segment L, the right segment R, and a new segment N. The left segment L includes all bytes of S that are on the left of P_b; thus, its address is the address of S, and the number of bytes in L is $L_c = P \times PS + P_b$ The right segment R includes all bytes of S on the right of page P; the address of R is $S+P+1$ and the number of bytes in R is $R_c = S_c - (P+1) \times PS$ or zero if P is the last page in S. The third segment N is a brand new segment that includes the bytes of I followed by the last bytes of P on the right of P_b.

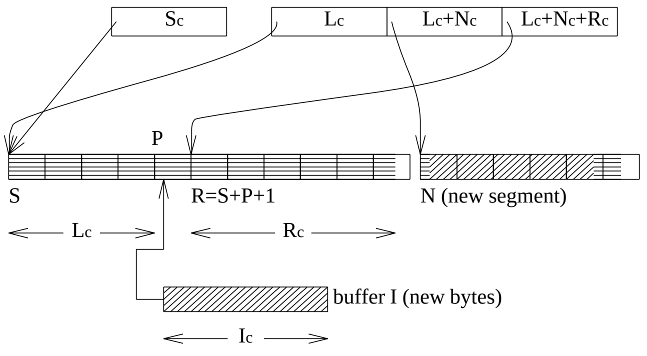

Figure 6. Inserting bytes from buffer *I* in segment *S*.

The above process just computes the number of bytes of the three segments. No leaf pages have yet been read from or written to disk. Since, most probably, the last page of N will have some unused bytes, before we actually write to disk the new segment N, we check if the last P_b bytes of L can be accommodated in the pages of N without overflowing N. If this is the case, then these bytes can be placed at the beginning of N thus eliminating the last page of L (and the waste of space in it). The situation shown in Figure 6 corresponds precisely to this case. If the elimination of the last page of L is not possible, the alternative is to try to balance the free space in L and N by borrowing as many bytes as necessary from the last page of L. Byte reshuffling can also be performed from R to N but only if R has exactly one page. Notice that during byte reshuffling we never overwrite existing segments. The reason for this has to do with recovery and the issue is discussed in section 4.5.

Given this background, we can now describe the insert algorithm as follows:

(1) Traverse the tree as in the search algorithm until the node pointing to segment S that contains the starting byte position is reached. Save the path on the stack. Let B be the desired byte within S.

(2) Preparation. Compute the page P within S that holds byte B: $P = \lfloor B/PS \rfloor$; the byte number P_b within P that insertion starts: $P_b = B\,\mathbf{mod}(PS)$; and the actual number of bytes P_c stored in P as follows: if P is not the last page in S then $P_c = PS$; otherwise, $P_c = S_m$. Set $L = S$ and $L_c = P \times PS + P_b$. Set $R = S + P + 1$ and $R_c = S_c - (P+1) \times PS$ or zero if P is the last page in S. So far, the new segment N has $N_c = I_c + P_c - P_b$ bytes.

(3) Reshuffle bytes of L, N, and R. If $N_m = PS$ skip this step. If there is exactly one page in R and the R_c and N_m bytes can fit in a single page, the R_c bytes become candidates to be placed at the tail of N. If the number of bytes L_m that are in the last page of L and the N_m bytes can fit in a single page, then the L_m bytes become candidates to be placed at the head of N. If both groups of the L_m and the R_c bytes can be moved to N without overflowing the last page of N then move both; otherwise, take the group that is in the segment with the largest free space. If after these operations there is free space at the last page of L, take as many bytes as necessary from L so that the last page of L and the last page of N will have similar amount of free space. Properly update the values of L_c, N_c, and R_c.

(4) Starting from page $S+P$, read one or two pages (read two only if R_c bytes have to be moved to N). Allocate a segment N of as many pages as necessary to hold the N_c bytes. Fill up N with bytes in proper order: first the part of P on the left of P_b (if any), then the I_c bytes, then the part of P on the right of and including P_b (if any), and finally, in case bytes must be taken from R, the first R_c bytes of $S+P+1$. Write the segment to the database.

(5) Fix parent so that it includes a pair for each of the segments L, N, and R whose size is not zero. Split the parent if necessary, and propagate the new counts and pointers up to the root of the tree using the stack built in step 1.

With regard to the I/O cost of the insert algorithm, one or two (physically adjacent) pages from the original leaf segment have to be read. Also, unless N_c is larger than the maximum segment size, the algorithm will add at most two new entries in the parent of the leaf segment.

4.3.2. Delete

The delete operation deletes a number of bytes starting from a specified byte position. This operation can result in either deletion of entire subtrees or partial deletions of leaf segments. Deletion of entire subtrees is performed first. They can be completed without touching a single leaf segment because the address and size of each segment are stored in the corresponding parent index nodes, and they can be given directly to the buddy system. Then, the delete algorithm proceeds to the second phase to perform partial deletions. In the general case, when the leaf level is reached, we will have a situation like the one pictured in Figure 7.a, where the first byte to be deleted is byte P_b in page P of segment S and the last is byte Q_b in page Q of segment S'. Possible segments that may have existed between S and S' have already been deleted during the first phase of the operation. Notice that the two segments of Figure 7.a may have different parents.

To delete all bytes of S on the right of P_b, we simply decrement the counts in the parent of S and free all pages of S on the right of P. There is no need to actually access any page of S. We then proceed by freeing all pages of S' on the left of Q. Now, the bytes of Q on the right of Q_b must be shifted to the left. Since segments cannot have holes, page Q is isolated from the part of segment S' that remains on the right of Q. A new 1-page segment N is allocated to hold the bytes of Q. As in the insert algorithm, before writing N to disk we attempt to balance the free space in N and whatever remained in S and S'.

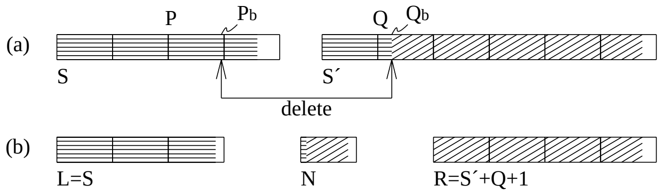

Figure 7. Byte range deletion.

Thus, in the general case, we end up with the following three segments shown in Figure 7.b: the left segment L containing all bytes of S on the left of P as well as those remaining in P itself after the deletion and byte reshuffling; the right segment R containing all bytes of S' to the right of page Q; a new 1-page segment N containing whatever bytes remained in Q and the ones taken from P because of byte reshuffling. Notice that unlike the B-tree algorithms as well as the ones used in Exodus, a partial segment delete may create new entries that need to be added in the parent.

The delete algorithm is as follows.

(1) Traverse the tree to locate segments S and S' where deletion starts and ends, respectively, and save each path to a stack. For each node visited during the tree traversal, if possible, perform deletions of entire subtrees.

(2) Compute the number of bytes P_c and the byte P_b of page P within S where deletion starts. Similarly, compute Q_c and Q_b of page Q within S' where deletion ends. Set $L = S$ and $L_c = P \times PS + P_b$. Set $R = S' + Q + 1$ and $R_c = S'_c - (Q+1) \times PS$ or $R_c = 0$ if Q is the last page of S'. The new segment N has $N_c = Q_c - (Q_b + 1)$ bytes. If $N_c = 0$, go to step 5.

(3) Reshuffle bytes of L, N, and R as in step 3 of the insert algorithm.

(4) Read page $S'+Q$ and if bytes have to shuffled read also the pages of S and/or S' whose bytes must be moved to N. Allocate a segment N to store the N_c bytes. Fill up N with the bytes from P on the left of P_b (if any), followed by those on the right of Q_b. Write N to the database.

(5) Propagate the new counts and pointers up to the root of the tree, using the stacks built in step 1. On each level do the following: if a new entry must be added, do so in the node with fewer pairs; check if a node in one of the two stacks has now less than the allowed number of pairs and if so, merge or rotate with a sibling.

(6) Fix Root. If the root has exactly one child, copy the pairs of this child to the root and repeat this step.

With regard to the I/O cost, deletions where the last byte to be deleted happens to be the last byte of a page in some segment of the large object can be completed without accessing any segment. Deletions of entire subtrees is a special case of the above. Another important special case of the above is object truncation where all bytes from byte B up to and including the last byte of the object are deleted; with $B = 0$ truncation becomes equivalent to deleting the entire object and thus, this operation too does not need to access any segment of the object. Otherwise and if bytes are not shuffled, one leaf page needs to be accessed (the one that contains the last byte to be deleted) and a new segment needs to be created. If bytes are shuffled, in addition to the above, one or two pages will need to be accessed.

4.4. Preserving Clustering

A major problem with the above algorithms is that small insertions and deletions break up the segments on which the operations are performed into (in general, three) smaller segments. It is certain that a reasonable number of such operations evenly distributed over the object will deteriorate the physical continuity of all pages in which the large object is stored, and leaf segments will be just 1-page long. There are two major implications of having leaf segments most of which are of size 1. First, the cost of multi-page reads will substantially increase as a direct consequence of the fact that each page touch will eventually result in a disk seek. Second, more entries are now needed to index the leaf pages. As a result, the tree level will eventually increase which, in turn, will indirectly increase the cost of all operations.

We attempt to eliminate the aforementioned problems by placing constraints on segment sizes of a large object. The segment size threshold T establishes the following constraint: *it can not be the case that a number of bytes are kept in two (logically) adjacent segments, one of which has less than T pages, if they can be stored in one.* It should be clear that the threshold mechanism does not specify fixed size leaf segments. Also, in the general case, it does not specify a minimum number of pages per segment. For example, with $T = 8$, a large object that is 1 page and a half long is kept in two page, not in 8 pages. We say that a segment S is *unsafe* if its size is greater than zero and less than T pages. If during an update operation, a segment is unsafe, we make it safe by taking entire pages from neighboring segments. We call this process *page reshuffling*.

This requires changing the byte reshuffling step of the original algorithm (step 3 of the insert operation) to include page reshuffling. The input of the algorithm is three numbers L_c, N_c, and R_c corresponding to the number of bytes in segments L, N, and R, respectively, and a segment size threshold T. It returns the new number of bytes in these three segments after (page and byte) reshuffling; it works as follows:

Reshuffle segments L, R, and N, with segment size threshold T.

(3.1) If any of the following is true

- a. All three segments are safe, or
- b. L and R are both empty ($L_c = R_c = 0$), or
- c. L or R or both are unsafe, and the number of bytes in the smallest unsafe segment and the N_c bytes cannot fit in a segment of the maximum size,

then go to step 3.4 and do byte reshuffling.

(3.2) If L or R is unsafe, take the smaller of the two and merge it with N, regardless of the N_c value. Go to step 3.1.

(3.3) if N is still unsafe, take as many pages as you can from the smaller of L and R so that N becomes safe. Go to step 3.1.

(3.4) **Reshuffle bytes** of L, R, and N. (This step is the same as step 3 of the insert algorithm.)

With respect to the I/O cost of update operations with page reshuffling, the exact overhead will depend on the value of T. For inserts, the overhead is the cost of transferring some additional pages from within the segment in which insertion occurs (no additional disk seeks). For deletes, in addition to the above, a second page reshuffle invocation may be necessary which results in two disk seeks. In summary, a larger T does increase the I/O cost of the given update operation that performs page reshuffling.

Regarding storage utilization, larger T leads to better utilization for the following reasons. First, the utilization per leaf segment is improved: for segments of size T, the utilization per segment will be on the average $1-1/2T$. For $T = 4$, 16 and 64, this evaluates to utilization of 87%, 97%, and 99%, respectively. (Recall, that only the last page in a segment may have some free space.) Second, for a given object, higher T values mean fewer segments, which leads to an improved utilization of the index pages. Therefore, it must be emphasized that the discussion of the previous paragraph does not reveal the entire picture in the sense that fewer leaf pages for indexing leads to a better tree (with fewer internal pages and shorter) that indiscriminately improves the performance of *all* operations (including updates).

Threshold values can be specified as a *hint* to the storage manager on a per-object or per-file (for all objects in the file) basis. The tradeoffs in selecting the T value are simple: larger T values improve the storage utilization and the performance of append, (sequential and random) read, and replace operations; the only aspect that might be affected negatively by larger segments is the costs of inserts and deletes. For often-updated objects, the T value should be somewhat larger than the size of the search operations expected to be applied on the object so that the amount of I/O performed by both updates and reads is minimized. Again, for more static objects where the cost of updates is of little or no concern, the larger the segment size the better the overall performance.

The threshold value does not have to be constant during the lifetime of a large object. Applications that could not possibly determine access patterns at creation time are allowed to change the T value every time the object is opened for updates. They may also choose to let EOS automatically adjust the value of T for them. In [Bili91a] we show a simple way to do this based on the fan-out ratio of the parent index node of the leaf being updated. The idea is that the closer we are to splitting an index, the higher the value of T should become. When the parent node is indeed going to be split if the child segment is split, the entire node is scanned and and for any two or more logically adjacent segments that have less than T pages, a single larger segment is allocated to accommodate this group of unsafe adjacent segments. Details are given in [Bili91a].

4.5. Concurrency Control and Recovery

Concurrency can be handled either by locking the root of the large object or, for finer granularity, the byte range affected by each operation [Care86]. In addition to locking the large object per se, there is the concurrency control problem associated with freeing a segment in that an update on the allocation status of a segment may propagate to its buddies. A comprehensive solution to this problem is provided in [Lehm89]. When a segment is freed, a (release) lock is placed on the segment and an intention (release) lock is placed on all of the segment's ancestors. As in hierarchical locking, segments that are descendants of a locked segment are also locked, and thus they remain unallocated until the holding transaction releases the locks.

For recovery, one or a combination of logging or shadowing may be used [Gray79]. With logging, updates are performed in place after the old and new values of the updated item have been recorded to the log. With shadowing, a page is never overwritten; instead, a write is performed by allocating and writing a new page and leaving the old one intact until it is no longer needed for recovery. The important issue here is that applying shadowing to a modified page of a segment will destroy the physical continuity of the pages of the segment and therefore the segment itself. To keep together the pages of a segment, the granularity of shadowing must be the whole segment. Thus, if segments are large and updates are small shadowing will be slower than logging.

In designing our insert and delete algorithms, we were careful not to overwrite pages of leaf segments so to avoid the cost of shadowing entire segments. Indeed, an examination of the four update operations presented in this paper − replace, insert, delete, and append − reveals the following. The first modifies the leaf pages without affecting the internal nodes of the tree; writes of this operation are logged. On the other hand, the last three kinds of updates do just the opposite; they modify only the internal nodes of the large object tree without overwriting existing leaf pages. Thus, during an insert, delete, or append, only the modified index pages need to be shadowed. Finally, it is worth mentioning that since no control information is kept on leaf segments, the log record of all updates must contain the operation that caused the update as well as its parameters, and the log sequence number of the update must be placed in the root page of the object to ensure that the update can be undone or redone idempotently [Gray79].

5. CONCLUSIONS

We presented a database storage structure for large objects and algorithms to search for and modify a byte range, insert and delete a sequence of bytes at any place within the object, and append bytes at the end of the object. The solution proposed in this paper satisfies all six principle objectives set in the introductory section. The key characteristic of our solution that made the above possible is the use of variable-size segments as opposed to fixed-size segments used in [Care86] or segments of fixed pattern of growth used in [Lehm89].

Prototype implementations of the large object manager and the buddy system presented in this paper have been completed. We have used the prototype to perform simulation studies to verify our intuition of the effect of the segment size threshold on the performance as discussed in section 4.4, as well as to compare the performance of our algorithms with the ones given in [Care86] and [Lehm89]; the results of this analysis are presented in [Bili91b]. Currently, EOS and the application run on a single process, with no support for transactions (concurrency and recovery). The prototype is written in C, and has been run on the SunOS operating system on SparcStations.

ACKNOWLEDGMENTS

I am grateful to Alex Buchmann of T.H.Darmstadt, and Toby Lehman of IBM Almaden Research Center for providing many important suggestions regarding the content and the presentation of an earlier version of the paper. Thanks are due to Bob Carter and Joe Wells for proofreading this paper.

References

[ACM91] Communications of the ACM, Special Issue on Digital Multimedia Systems, Vol. 34, No. 4, April 1991.

[Astr76] Astrahan M.M., *et al*, ‘‘System R: Relational Approach to Database Management’’, *ACM Transactions on Database Systems*, Vol. 1, No. 2, June 1976, pp. 97-137.

[Bili91a] Biliris, A., ‘‘The EOS Large Object Manager,’’ Boston University, Computer Science TR 91-004, April 1991.

[Bili91b] Biliris, A., ‘‘The Performance of Three Database Storage Structures for Managing Large Objects,’’ Boston University, Computer Science TR 91-011. Submitted for publication.

[Care86] Carey, M. J., DeWitt, D. J., Richardson, J. E., Shekita, E. J., ‘‘Object and File Management in the EXODUS Extensible Database System,’’ *Proc. of the 12th International Conference on Very Large Data Bases*, Kyoto, Japan, August 1986, pp. 91-100.

[Chou85] Chou, H-T., D.J DeWitt, R.H. Katz, and A.C. Klug, ‘‘Design and Implementation of the Wisconsin Storage Systems,’’ *Software Practice and Experience*, John Wiley & Sons, Vol. 15(10), October 1985, pp. 943-962.

[Deux90] Deux, O., *et al*, ‘‘The Story of O_2,’’ *IEEE, Trans. on Knowledge and Data Engineering*, Special Issue on Database Prototype Systems, Vol. 2(1), March 1990, pp. 91-108.

[DeWi90] DeWitt, D.J., D. Maier, P. Futtersack, and F. Velez, ‘‘A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems,’’ *Proc. 16th Int. Conference on Very Large Data Bases*, Brisbane, Australia, August 1990, pp. 107-121.

[Gray79] Gray J., ‘‘Notes on Database Operating Systems’’, in *Operating Systems: An Advaced Course*, R. Bayer, R. M. Graham, and G. Seegmuller, Eds. Spring-Verlag, New York, 1979, pp. 393-481.

[Hask82] Haskin, R. L., and Lorie, R. A., ‘‘On Extending the Relational Database System,’’ *Proc. ACM-SIGMOD Int. Conf. on Management of Data*, 1982, pp.207-212.

[Knut73] Knuth D. E., *The Art of Computer Programming*, Addisson-Wesley, 1973.

[Kotc87] Kotch, P.D., ‘‘Disk File Allocation Based on The Buddy System,’’ *ACM Trans. on Computer Systems*, Vol. 5, No 4, November 1987.

[Lehm89] Lehman, T.J., and B.G. Lindsay, ‘‘The Starburst Long Field Manager,’’ *Proc. 15-th Int. Conference on Very Large Data Bases*, Amsterdam, The Netherlands, August 1989, pp. 375-383.

[Lohm91] Lohman, G.M, B. Lindsay, H. Pirahesh, and K.B. Schiefer, ‘‘Extensions to Starburst: Objects, Types, Functions, and Rules,’’ *CACM*, Vol. 34 (10), October 1991, pp. 94-109.

[Moha90] Mohan, C., ‘‘Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems,’’ *Proc. 16th Int. Conference on Very Large Data Bases*, Brisbane, Australia, August 1990, pp. 406-418.

[Ozsu91] Ozsu, T., and P. Valduriez, ‘‘Distributed Database Systems: Where are we?’’ *IEEE Computer*, Vol. 24, No. 8, August 1991, pp. 68-78.

[Selt91] Seltzer, M., and M. Stonebraker, ‘‘Read Optimized File Sytem Designs: A Performance Evaluation,’’ *Proc. 7th IEEE Int. Conference on Data Engineering*, Kobe, Japan, April 1991, pp. 602-611.