The Performance of Three Database Storage Structures for Managing Large Objects

Alexandros Biliris[†]
AT&T Bell Laboratories
Murray Hill, NJ 07974
biliris@research.att.com

ABSTRACT

This study analyzes the performance of the storage structures and algorithms employed in three experimental database storage systems – EXODUS, Starburst, and EOS – for managing large unstructured general-purpose objects. All three mechanisms are segment-based in that the large object is stored in a sequence of segments, each consisting of physically contiguous disk blocks. To analyze the algorithms we measured object creation time, sequential scan time, storage utilization in the presence of updates, and the I/O cost of random reads, inserts, and deletes.

1. INTRODUCTION

Efficient manipulation of large objects is an important issue in many so called unconventional database applications such as geographical, computer-aided design, document processing and publishing, and multimedia presentation. Pictures, books, digitized video and sound recordings are a few examples of large objects that need to be stored in the database. Efficient manipulation of large objects is also important in any object-oriented and extended relational database management system, regardless of the application in which they are being used, in supporting general-purpose advanced data modeling constructs such as long lists or "insertable" arrays. For example, O_2 – a commercial object-oriented database system [Deux90] – uses the large object manager of the Wisconsin Storage System [Chou85] to store large lists of any type of elements.

The management of large objects imposes several requirements on the database storage system. 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). It must support operations that deal with a specific number of bytes within the object: *read* or *replace* a random byte range within the object, *insert* or *delete* bytes at arbitrary positions within the object, and *append* bytes at the end of the object. These piece-wise operations are important for the following reasons. 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; rather, smaller (but sizable) chunks of bytes will be successively appended at the end of the object. Second, applications 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. Lastly, since the large object is stored in the database it must be protected from transaction and system failures.

Given this background it appears that the performance requirements on large objects storage techniques are the following:

- *Object creation (and deletion).* 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. The performance of successive appends at the end of the object is of particular importance since this the expected way of creating large objects.
- *Byte range operations.* To perform a byte range operation we must first seek to a specific byte in the large object, and then read, replace, insert, delete, or append a specific number of bytes. Thus, we require good random access performance meaning that the cost of locating any given byte within the object is independent of the object size. We also require good sequential access performance meaning that I/O rates in reading/writing a large chunk of bytes 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.
- *Storage utilization.* We require the large object to be stored in a way that minimizes internal fragmentation. Ideally, storage utilization must be very close to 100%

The solutions that have been proposed to manage large objects are either *block-based* or *segment-based.* Algorithms of the first kind store the large object in a number of single blocks [Astr76, Hask82, Chou85]. In these schemes, blocks that store consecutive byte ranges of the object are scattered over a disk volume. As a result, sequential reads will be slow because virtually every disk page fetch will most likely result in a disk seek. Segment-based algorithms store the object on physically adjacent blocks [Care86, Lehm89, Bili92]. From the latter three, the first [Care86] uses fixed-size segments, the second [Lehm89] segments of fixed pattern of growth, and the third [Bili92] uses variable-size segments.

Our study analyzes, by means of simulation, the performance tradeoffs of the segment-based algorithms with respect to the operations mentioned above. The work closer to ours is the one reported in [Care86]. Our work differs from this effort in several ways. First, we employ a much more detailed model of reading and writing large data segments, the importance of which will be clear from our results. Second, we present results for object creation and sequential read time. Third, the work in

[†] This work has been performed at Boston University and it was partially supported by the NSF grant CDA-8920936.

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0276...$1.50

[Care86] analyzes the design decisions and tradeoffs of their own algorithm only while our study compares the design of three such algorithms. We know of no other work on the evaluation of large object management techniques and the relative performance of these algorithms is an open question.

The remainder of this paper proceeds as follows. Section 2 provides a review of the algorithms simulated in this study. Section 3 presents our architectural assumptions. Section 4 presents experiments and results, and Section 5 concludes our work. In this paper the terms *record*, *tuple*, and *object* will be synonymous as are the terms *field* and *attribute*.

2. SEGMENT-BASED LARGE OBJECT MANAGEMENT TECHNIQUES

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*, and it is stored in as many pages as necessary to hold the entire object. A kind of directory is created, called *large object descriptor*, that contains addressing information required to access the pages holding the large object. Frequently, we also talk about *long fields* within small objects. The small object holds all short fields along with *long field descriptors* each of which describes one of the object's long field; the long field itself is stored separately from the object.

For example, a `person` object with attributes `name`, `picture`, and `voice`, can be represented as a large database object but it can also be mapped to a small database object that contains the short field `name` and two long field descriptors corresponding to long fields `picture` and `voice`, respectively. Some applications may prefer the second view of objects because it is easier to treat the long fields within the same object in different ways; e.g., they may apply a compression technique that is appropriate for pictures in storing the `picture` attribute, and a different one that is appropriate for audio in storing the `voice` attribute [Ston91]. Nevertheless, "large objects" versus "long fields" is an issue that must be considered by the clients of the storage manager. The algorithms that have been presented in the literature can be used for both. We review these algorithms in the following sections.

2.1. EXODUS Storage Manager (ESM)

EXODUS is an extensible database system that supports large objects of unlimited size on which all byte operations can be applied [Care86, Care89]. ESM stores large objects on data segments. The size of the data segments of all large objects within a file can be fixed by the clients to an integral number of disk blocks. The idea is that for often-updated objects, data segments will probably be one disk block in length so as to minimize the amount of I/O performed by updates; for more static objects, larger data segments will be selected to lower the I/O cost of scanning a large sequence of bytes.

The data segments are indexed by a B-tree-like structure, a generalization of the concept of "portals" proposed in [Ston84] to handle arrays of tuples in a relational system. Figure 1 shows an example of the ESM structure, where the leaves of the tree are 4-page segments. (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, 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. As in B-trees, internal nodes as well as leaf segments are required to be at least half full.

As an example, the size of the object shown in Figure 1 is 1830 bytes. The left child of the root indexes the first 900 bytes and the right child the next 1830–900 = 930 bytes. The right child points to three segments. The first segment contains the first 400 bytes of these 930 bytes, the second the next 650–400 = 250, and the third the remaining 930–650 = 280 bytes.

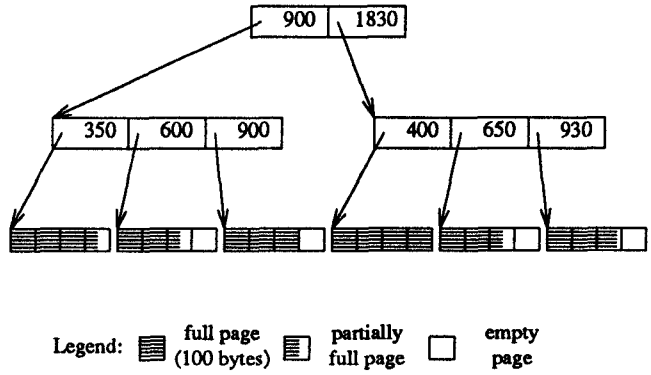


Figure 1. The ESM large object storage structure.

2.2. Starburst

The Starburst long field manager, presented in [Lehm89], uses extent based allocation with extents being organized into a binary buddy system [Knut73, Koch87]. 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. Figure 2 shows an example of a Starburst long field of size 1830 bytes. It consists of 5 segments of size 100, 200, 400, 800, and 330 bytes. The current implementation handles objects up to 1.5 gigabytes long [Lohm91].

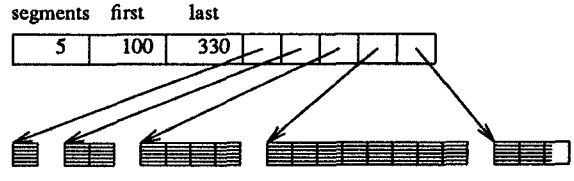


Figure 2. The Starburst long field storage structure.

This scheme can efficiently support (sequential and random) reads, appends, and byte range replace, but it cannot gracefully handle insertion (deletion) of bytes in (from) the middle of the object. The latter operations necessarily change the size of the long field, and because of the particular structure of the long field descriptor, the entire long field (or large portions of it) must be copied to new segments. The applications this long field manager was intended for were large mostly read-only objects – such as video and audio – and not general-purpose objects that would need such length-changing updates.

2.3. EOS

EOS is a storage system for experimental database implementation. The EOS large object mechanism, presented in [Bili92], bridges the two ideas proposed in ESM and Starburst. It can be seen as a generalization of the above structures without the constraint that the size of the segments on which the large object is stored must either be fixed, as in ESM, or follow some predetermined pattern, as in Starburst.

Large objects are stored in a sequence of variable-size segments each of which consists of physically continuous disk pages; segments are allocated using the buddy system. 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 structure, whose internal nodes are identical to the ones proposed in [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 ESM.

When byte range deletes and inserts are performed on the large object, its segments may have to be broken up into smaller ones and thus, the segments that comprise the large object may have sizes that vary drastically. Figure 3 shows how the structure may look like after some byte inserts or deletes have been performed. The size of the object is 1830 bytes. The root has a left and a right child indexing the first 1230 and the remaining 1830–1230 = 600 bytes of the object, respectively. The right child points to two segments. The first segment consists of $\lceil 470/100 \rceil = 5$ pages holding the first 470 bytes of these 600 bytes. The second segment contains the remaining 600–470 = 130 bytes.

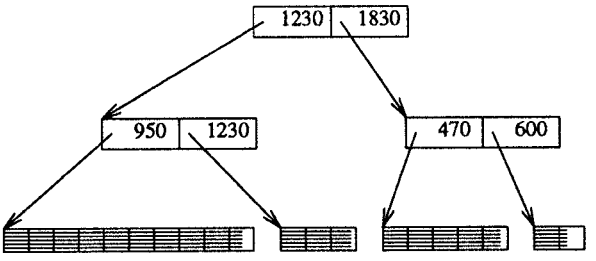


Figure 3. The EOS large object storage structure.

After repetitive inserts or deletes we may end up with a tree whose leaves are single-page segments. To eliminate this problem, EOS allows the client to specify a segment size threshold T establishing 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. Note that the threshold mechanism does not specify fixed size leaf segments neither 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 pages, not in 8 pages. If during an update operation, the parts of a split segment violate the above constraint, pages in neighboring segments have to be shuffled.

3. PROTOTYPING THE LARGE OBJECT MANAGERS

We used the two low-end components of the EOS storage system for the prototyping of the ESM and Starburst large object managers. The first component is responsible for managing disk space and the second provides buffered I/O to the higher level components. It is important, therefore, to analyze the effects of these two components of EOS on the simulation results; we do so in Sections 3.1 and 3.2, respectively. Next, in Section 3.3, we discuss the effect of recovery policy on the performance of the three algorithms, as well as our cost model of performing reads and writes on multi-block segments. Lastly, Sections 3.4 and 3.5 provide some clarifications on the implementation of the ESM and Starburst large object mechanisms, respectively.

All three mechanisms were written in C. The simulation runs on the SunOS operating system on SparcStations.

3.1. Disk Space Allocation Policy

To simplify matters, the disk manager was implemented on top of the UNIX file system. Each database area is assigned to a UNIX file, and its space is managed by the binary buddy system. A database area consists of a number of buddy spaces. Each buddy space is a fixed-length sequence of physically adjacent blocks and a 1-block directory that provides allocation information for all blocks in that space. A *segment* is a sequence of disk pages taken from one of the buddy spaces. 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 maximum segment size that can be supported depends on the block size; with 4K-byte disk blocks, EOS supports at most 32M-byte segments in buddy spaces of at most 63.5 Mbytes [Bili92].

The entire process of allocating and deallocating segments is performed by examining the directory block only. For databases that can fit entirely in a single buddy space, at most one disk access is needed to perform segment allocation and deallocation. However, larger databases will have many buddy spaces and thus, it is possible that the directory block of each buddy space may have to be visited to locate a free 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 maintained by the buddy space manager. Initially, it indicates that each buddy space 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.

To summarize, on a steady state, the cost of allocating and deallocating a segment from a buddy space is going to be at most 1 disk access, regardless of the database size.

3.2. Buffering for Large Objects

For segment-based storage systems, there is the problem of how multi-block worth of data can or should be buffered. Reading a multi-block segment into a block-based buffer a block at a time defeats the purpose of introducing segments in the first place.

One approach is to read and buffer the requested pages of a segment in the pool. This scheme may perform sufficiently good for small byte ranges. However, for large byte ranges (hundreds of kilobytes, or even a few megabytes) it would be either impractical (most pages in the pool will be invalidated or forced out to disk) or impossible (because of the limited space of the pool).

The extreme alternative is to do no buffering on large objects; to fetch, simply copy the object from disk directly to the

user space and vice versa to write. We can think of many applications where buffering does no good at all such as those that scan the object and never go back to previously visited byte ranges. (Again, think of multimedia applications; listening to digital sound recordings, frame-to-frame accessing of a movie, etc.) Of course, this approach has just the opposite problem; it will cause repetitive disk accesses when small amounts of bytes that belong to the same or neighboring blocks are successively requested – think of scanning a long list, element by element. Moreover, copying parts of segments from disk "directly" to the application buffers is not as "direct" as it sounds if the requested range does not match block boundaries. Figure 4 illustrates this point. A client wants to read into its buffer B a byte range that happens to be located within a single segment S of N blocks. If the first (last) byte to be read is located somewhere in the middle of page L (R, respectively), the segment cannot be read directly into B, in one step, without overwriting the client's space just before the beginning and/or after the end of B. There are two rather obvious ways to handle this problem. The first is to read the segment into a N-block space in virtual memory and then copy the desired part of the segment into the application buffer. The second approach avoids in-memory copying of potentially large data segments. It breaks this single I/O request into a 3-step I/O that first reads L and copies its right part at the beginning of B, then reads the $N-2$ blocks following L directly into B, and then reads the block R.

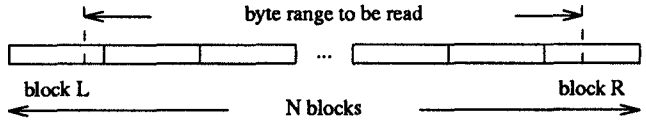


Figure 4. A byte range that does not match block boundaries.

The buffering scheme we used for the simulation is a hybrid approach that works as follows. When the large object manager needs to read a N-block segment, it makes this request to the buffer manager. If the N blocks can be accommodated into the buffer pool, according to some criteria discussed shortly, space in the pool is freed (we start first by freeing the least recently used clean pages followed by dirty pages that, of course, have to be written back to disk), and the segment is read in a single step into the buffer pool. Otherwise, the N-block segment is not buffered, and the large object manager reads the segment from disk into the application space. If there is a boundary mismatch, we follow the 3-step I/O approach discussed above, with the first and last block of the segment being placed in the pool and copied from there into the application buffers.

The buffer manager facilitates a very simple criterion based on buffer availability at run time [Effe84], and a limit on the maximum number of physically adjacent disk blocks that can be read with one I/O call into contiguous buffer blocks in the pool. As it will be discussed shortly, the simulation was performed with a 12-page buffer pool where the aforementioned limit is set to 4 pages. Our goal was to provide a buffering scheme that handles small multi-block segments in a reasonable way. Larger segments are not buffered.

In addition to the above, the buffer manager provides the usual call for clients to fix a page. The call returns a pointer to the control block of the frame holding the desired page. The client must inform the buffer manager that a fixed page is dirty; this can be done when the pointer to the control block is acquired, when it is released, or any time in between.

3.3. Reads and Writes on Multi-Block Segments

Regarding reads on multi-block segments, the smallest unit of I/O request is still a single disk page. Thus, when few bytes need to be read from a segment, only those pages that contain the desired bytes are read, not the entire segment.

Regarding updates, all three algorithms examined in this study assume some kind of shadowing for recovery [Gray79]. Shadowing is a recovery technique in which 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. Thus, applying shadowing on a modified block of a multi-block segment will destroy the physical continuity of the blocks 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. Although our study does not involve transactions, we have decided to include the shadowing cost so we can correctly analyze the effect of the segment size in the simulated algorithms. For example, with no shadowing, the cost of updating a page that belongs to a 2-block segment would be the same with the cost of updating again a single page but which is part of a 64-block segment. With shadowing, the two updates will have substantially different costs (with the second update being approximately 6 to 7 times more costly than the first).

To be precise, all updates on index pages, except the root, are shadowed and the new copy that contains the update is flushed out to disk at the end of the operation that caused the update. Updates on leaf segments are performed as follows. If the update overwrites useful bytes of the leaf[1], the above procedure is used, i.e., copy, update, flush. If the update just appends bytes in the leaf, the segment is not shadowed; instead, the update is performed in place, and the dirty pages of the segment are simply flushed to disk at the end of the operation.

3.4. ESM

The difference between the ESM algorithms and the ones used in EOS is in the way inserts, deletes, and appends are performed in the leaf nodes. Thus, the code that manipulates the tree nodes, other than the leaves, is shared between the two implementations. For example, routines that split/merge/rotate index pages, add/delete pairs in them, etc, are the same.

For byte range inserts, there are two algorithms that appear in [Care86], termed *basic* and *improved*. In the basic algorithm, when an overflow occurs during an insertion in a leaf block L, the bytes of L and the new bytes being inserted in L are evenly distributed in new leaves. The improved algorithm attempts to redistribute the new bytes, the bytes of L and the bytes of one of L's neighbor, if in doing so the creation of a new leaf block is avoided. As it is shown in [Care86], the improved algorithm leads to significant gains in storage utilization with minimal additional insert cost. The results reported in this paper are taken with the improved insert algorithm.

ESM uses fixed-size leaves of some given number of blocks and thus, an update on a leaf that overwrites useful bytes will force the allocation of a new leaf of the same size on which the actual update is performed. When writing a multi-block leaf to disk, only the blocks that are actually dirty are written (sequentially) to disk, not the entire leaf. For example, if after the allocation (and subsequent update) of a 16-page leaf only the first 10 pages contain useful data, only these first 10 pages will be written to disk. In contrast, the preliminary results reported in [Care86]

1. The number of "useful" bytes in a leaf is given in the corresponding (count, pointer) pair of the parent node that indexes this leaf.

for the ESM prototype assumed leaf segments rather than single pages to be the unit of both read and write operations.

3.5. Starburst

The implementation of the Starburst search and append algorithms are straightforward. To implement insertion (similarly, deletion) of bytes in (from) the middle of the object, the segments on the right of (and including) the segment in which the start byte of the operation belongs to are read. The original bytes together with the new ones (or except the ones being deleted) are placed into a new set of segments. We have also included in this copying the segment in which the start byte belongs because of shadowing as discussed above. We assumed the availability of a dynamically allocated 512K-byte virtual memory space through which copying of segments from one disk place to another is performed. The cost associated with such allocation (and possible swapping) is not considered in our simulation.

4. PERFORMANCE EVALUATION

This section presents the performance evaluation results. First, in section 4.1, we present the values of the system and simulation parameters we have assumed throughout the experiments. In order to make direct comparison with the results reported in [Care86], we have chosen these parameters to be the same with the ones used in [Care86]. Next, in sections 4.2 and 4.3, we present the cost results of creating a large object and then performing a sequential scan on the entire object, respectively. Following that, in section 4.4, we present the I/O cost results of performing reads, inserts and deletes at random portions of the object, and the effect of these updates on the storage utilization. Section 4.5 compares our results with the ones reported in [Care86], and finally section 4.6 summarizes the results of the experiments.

4.1. Measurement Approach

The experiments are performed on 4K-byte disk pages. For I/O cost, we separate disk seek time (including rotation time) and data transfer time so we can model sequential disk accesses. We assume disk seek time of 33 milliseconds and a transfer rate of 1K-byte per millisecond. We count a disk seek every time the disk is accessed to fetch or write a segment on disk. For example, the I/O cost of reading a 3-block (12K-byte) segment is 33+4×3 = 45 milliseconds; the cost of reading the same number of blocks with 3 I/O calls is (33+4)×3 = 111 milliseconds. The size of the buffer pool was set to 12 pages and the largest segment that can be read into the pool, in one step, was set to 4 pages. Table 1 summarizes the values of these parameters that were fixed throughout the simulation.

Parameter	Value
Page (block) size	4K-byte
Buffer pool size	12 pages
Largest segment in pool	4 pages
I/O seek cost	33 milliseconds
I/O transfer rate	1K-byte/millisecond

Table 1: Fixed system parameters.

The simulations run on a 10M-byte object. We have also experimented with larger objects and we simply discuss the effect of the object size without presenting graphs. The root of the object is placed in a page with no other objects in it. Pointer and count values of an index page require 4 bytes each. With 4K-byte pages we may store up to 507 pairs in the root and 511 pairs in internal index pages. For EOS, we used threshold sizes of 1, 4, 16, and 64 pages. Similarly, for ESM, leaf segment sizes of 1, 4, 16, and 64 pages were used. Notice that since the buffer manager can handle segments of up to 4 pages, full buffering is performed when the ESM leaf block size is set to 1 or 4 pages.

The database was set up in two database areas both managed by the buddy system, one for the leaf segments holding the bytes of large objects and the second for everything else. The use of two database areas is due purely to our desire to experiment with very large objects on the limited disk space available in our computing installations. It allowed us to use the disk space for all database blocks except the leaf blocks (i.e., the actual large object) that were not actually read from or written to disk. We simply kept track of the number of disk I/O calls (to count disk seeks) and the number of pages involved in each access (to count the cost of transferring the segment).

4.2. Object Build Time

This section presents the time needed to built a 10M-byte object by successively appending fixed-size chunks of bytes. We start with 3K-byte and go up to 512K-byte chunks.

In ESM, with 1-page leaves, a 10M-byte object turns out to be of level 2 – the root, one level of 9 internal nodes, and then 2560 leaves. With 4-page leaves, the object is again of level 2 – the root, 2 internal nodes and 640 leaves. For leaf blocks of 16 and 64 pages, the tree is of level 1. (The level of a 100M-byte object is 2 for 1, 4, and 16-page leaf blocks and 1 for 64-page leaves.) For Starburst and EOS the tree level is always 1. Since the growth of the object in both Starburst and EOS has the same pattern, we treat these algorithms together in this particular experiment. Figure 5 shows the time required to build a 10M-byte object in the three algorithms[2].

The most startling result is the effect on the ESM object creation time of changing the append size by a few kilobytes. Observe that the object build cost for 1-page leaves and for 3K-byte appends is approximately 575 seconds; it drops to 170 seconds for 4K-byte appends and rises up again to 380 seconds for 5K-byte appends. In ESM, this is due to the mismatch of the block boundaries and append size discussed in the previous section, and mainly to the way appends are performed. When an overflow occurs on the rightmost leaf because of an append, the new bytes being appended, the bytes of the rightmost leaf, and the bytes of its left neighbor (if it has free space) are redistributed in such a way that all but the two rightmost leaves are full. The remaining bytes are evenly distribute in the last two leaves, leaving each of them at least 1/2 full. In general, when the append size is not precisely a multiple of the leaf block size, reshuffling as described above is performed which increases the cost of appends.

The block boundary mismatch problem affects also the Starburst and EOS algorithms but to a lesser degree. In these two mechanisms, bytes are simply appended at the end of the rightmost page with no reshuffling. Thus, the cost of an append operation is the one of reading the rightmost page (if it is not full) and flushing to disk the pages containing the new bytes. Also, there are no index pages to write; the tree level is always of level 1. (In EOS, to come up with a tree of level greater than 1, the size of the object being created must be larger than 16 Gigabytes.)

Another interesting result related to ESM is that we can not, independent of the append size, isolate a particular leaf block size as the winner. Referring to Figure 5, for appends of 4K, 16K, 64K, and 256K bytes, the best performance is achieved when the

2. The exact append sizes in the horizontal axis of the graph are the following (in kilobytes): 3, 4, 5, 6, 7, 8, 10, 12, 14, 16, 20, 24, 28, 32, 50, 64, 100, 128, 200, 256, and 512.

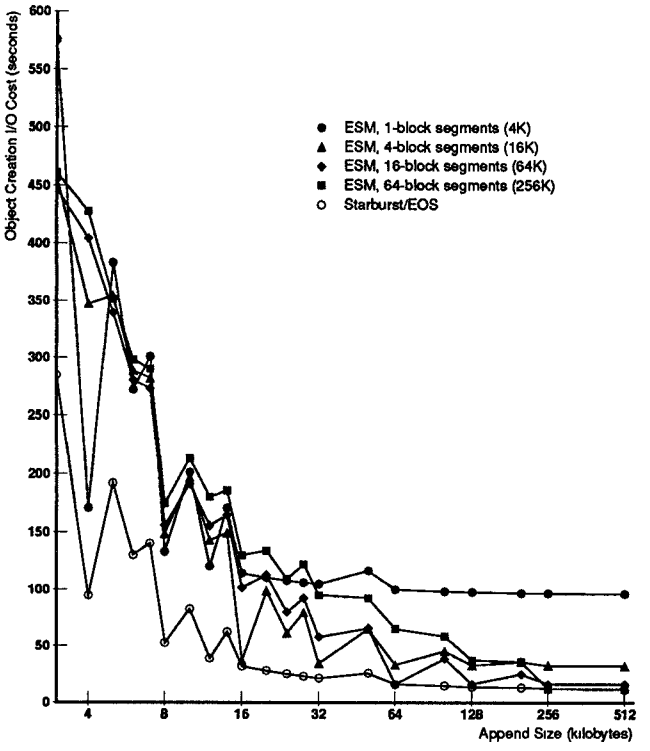


Figure 5. 10M-byte object creation time.

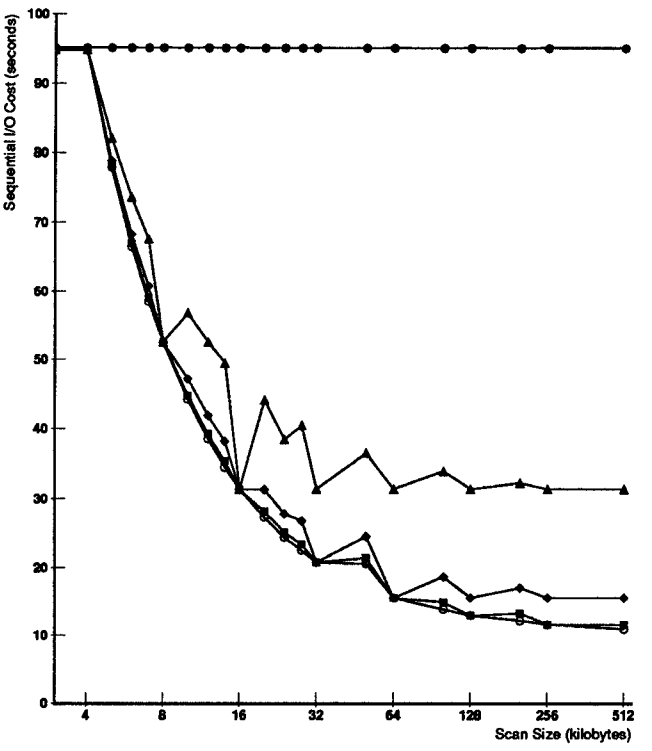


Figure 6. 10-Mbyte sequential scan time.

leaf block size is 1, 4, 16, and 64 pages respectively; i.e., precisely when there is exact match between append and leaf block size. For appends larger than 256K bytes, larger blocks have better performance.

Finally, in all mechanisms, the cost of creating an object grows linearly with the object size. For instance, to obtain the time required to build a 100M-byte object, just multiply the numbers in Figure 5 by 10.

To summarize this experiment, the general trend in both curves is the larger the append size the better the response time. However, small differences in the append size, especially for small appends, may result in very big differences in the total object creation time. In comparing Starburst/EOS and ESM, for the same append size the first algorithms perform the same as or better than the best case of ESM.

4.3. Sequential Scan

This section shows the time required to sequentially retrieve the entire large object from the database. After the 10M-byte object was built in the previous experiment, it was scanned from the beginning to the end in fixed-size chunks of bytes. The n-byte scan was performed on the object created by n-byte appends[3]. With a transfer rate of 1K-byte/millisecond, the best performance that can be achieved is approximately 10 seconds.

As shown in Figure 6, for scans shorter than the page size all three techniques produce the same results; the page being scanned is buffered and all its bytes are read. The differences appear for scans larger than the page size. In ESM, the cost for the 1-page segments case is the worst and is independent of the scan size; all leaf pages of the object are read one by one. Larger segments produce much better results and their performance reach a plateau when the scan size exceeds the segment size. The performance of Starburst/EOS follows the expected pattern; larger scans produce better response time.

In comparing the three algorithms, the same conclusion can be made as for object creation time; i.e., for the same scan size Starburst and EOS perform the same as or better than the best case of ESM.

4.4. Random Reads and Updates

In this section we analyze the effect of updating a 10M-byte object on the storage utilization, and we measure the performance of random reads and updates (inserts and deletes). We assume a mix of 40% reads, 30% inserts, and 30% deletes[4]. The mean operation size is 100, 10K, and 100K bytes, where the actual operation size was varied ±50% about the mean. The operations were randomly run with the above probability and uniformly distributed over the entire byte range of the object. To ensure that the object size remained stable, the size of a delete operation was set to the size chosen for the immediate previous insert.

4.4.1. Storage Utilization

Storage utilization compares the object size with the actual space required to store the object including possible index pages. Since after each update Starburst completely reorganizes the affected segments, only the last page of the large object may have some free space. Therefore, we do not further discuss Starburst in this section because it achieves, unconditionally, the best possible storage utilization.

Figures 7 and 8 show how utilization, which is initially near 100%, in ESM and EOS, respectively, is affected as random insertions and deletions break up the full leaf nodes. Each mark in the graphs represents the utilization after the corresponding number of operations have been performed. We first discuss the

3. This is slightly important in Starburst/EOS because the growth of the segments (and therefore the resulting structure) depends on the size of the first append. For ESM, the resulting structure is independent of the append size.

4. This is a small percentage for reads. However, the results do not depend on the mix rather on the operation size. A larger search percentage will simply require more runs to stabilize the performance curves.

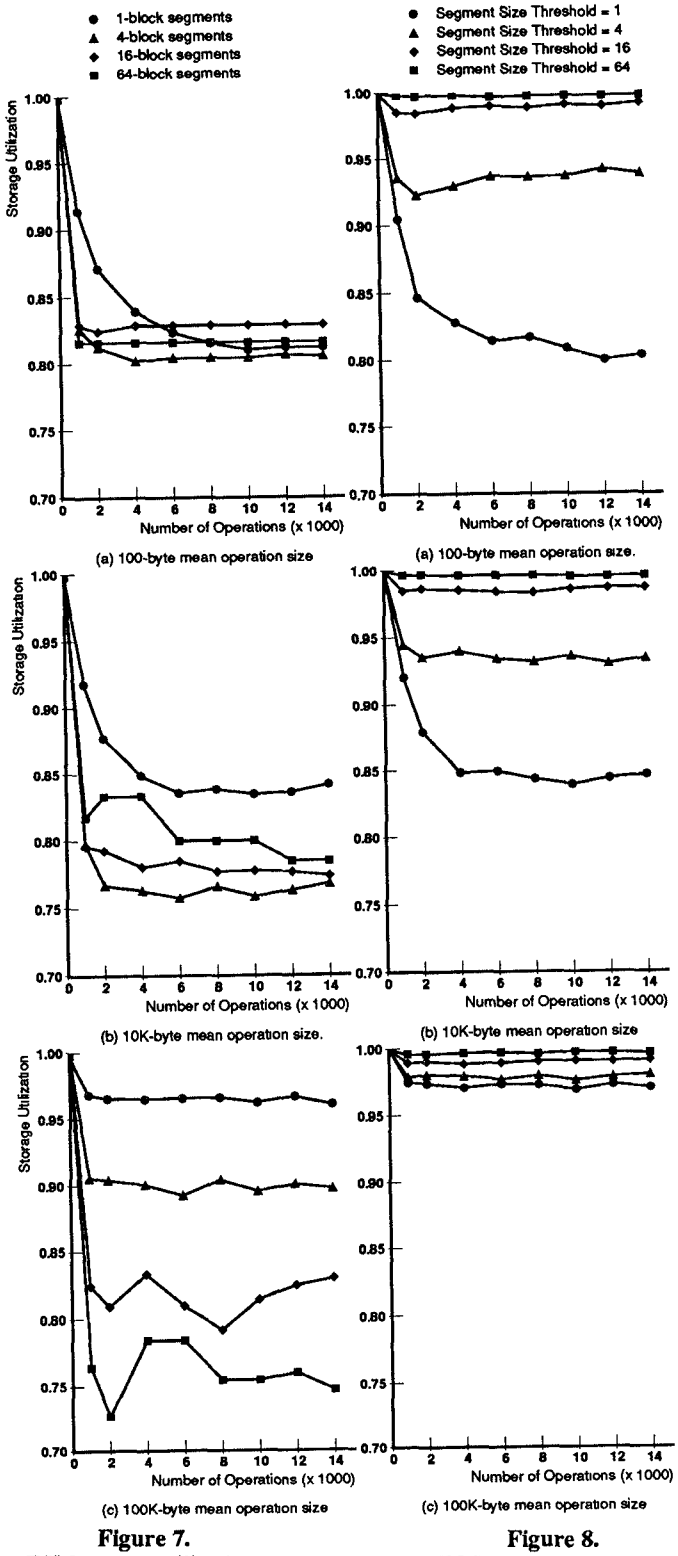


Figure 7.
ESM storage utilization.

Figure 8.
EOS storage utilization.

effect of the operation size on each algorithm and then we compare the two algorithms.

Figure 7.a, referring to small updates in ESM, shows that utilization does not depend on the leaf block size; it stabilizes at pretty much the same level, at the low 80%. As larger updates are performed on the object, Figure 7.b, the 1-page leaf case with 85% utilization starts distinguishing itself from the rest of the cases that are in the range of 80%. For 100K-byte updates, Figure 7.c, there is a clear distinction. The larger the leaf, the worse the utilization, with substantial difference between the best and the worst (from approximately 96% with 1-page leaves, down to on the average 75% with 64-page leaves).

For EOS, the trends are clear regardless of the size of the updates. The larger the segment size threshold, the better the utilization is. This is due to the fact that only the last page in a segment may have unused space. Thus larger segments achieve a better utilization on a per segment basis and thus, for the entire object. We can also observe that a segment size threshold of 16 pages achieves a utilization higher than 98%; with the 64-page case this number becomes almost 100%.

Comparing ESM and EOS, we can see that their performance is approximately the same for the case of 1-page leaves and 1-page segment size threshold, respectively. For larger leaf size settings, the utilization in each algorithm goes the opposite direction. In general, it definitely improves in EOS, while it becomes worse (or in the best case, it remains within the same range) in ESM.

4.4.2. Read Cost

We first present, in Table 2, the average I/O cost results for reads in Starburst. As we have explained before, the Starburst structure is reorganized after every update; therefore, the read cost does not depend on the updates performed before this read.

Figures 9 and 10 show the I/O cost for reads in ESM and EOS, respectively, as random insertions and deletions degrade the large object structure. Each mark in the graph represents the average cost of the read operations performed since the previous mark. For example, the mark at the 10,000 operations indicates the average cost of the reads performed within the last 2,000 operations.

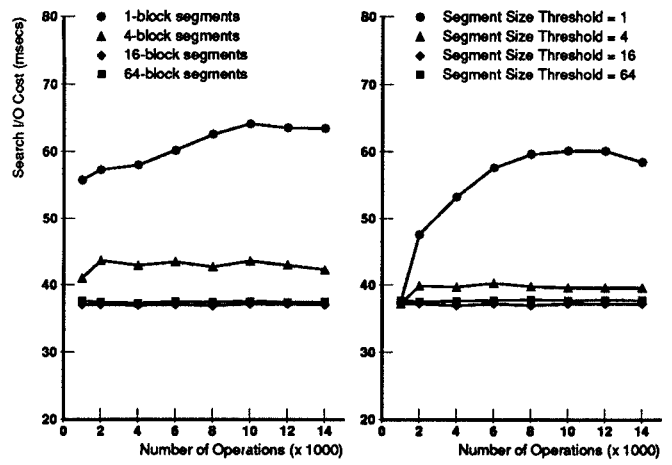
First, a general comment regarding the results for EOS in Figure 10. When the first updates are applied to the object, the I/O cost for reads is independent of the segment size threshold. This is because when the object is initially created, the count tree is just a one-level directory and the leaf segments are large at this point. However, as more and more updates are performed, these segments gradually degrade to about N-page leaves, where N is the segment size threshold.

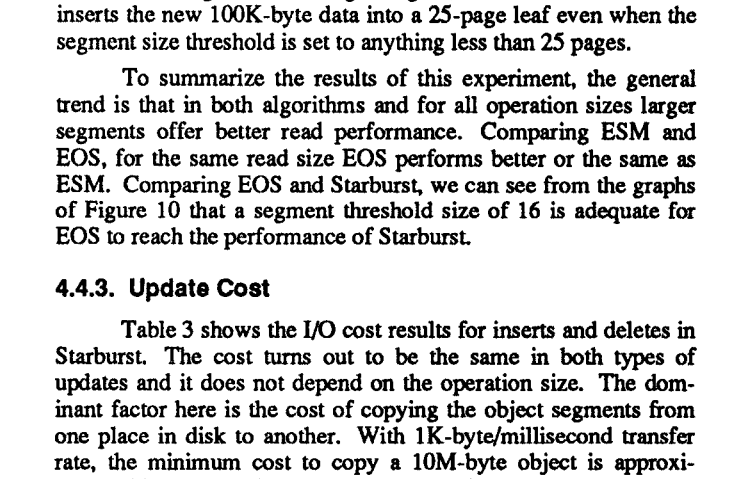
In Figures 9.a and 10.a, 1-page leaves have a tiny disadvantage over the other cases whose cost is practically the same. Although 100-byte reads can almost always be satisfied by accessing a single page, the difference is due to the fact that larger leaves reduce the number of pairs that need to be kept in index pages which in turn minimizes the number of index pages. As we have mentioned in section 4.2, the ESM tree with 1-page leaves has 9 internal nodes, while with 4-page leaves it has 2 nodes. Thus in the latter case, the likehood of an index page missing in the buffer pool is reduced.

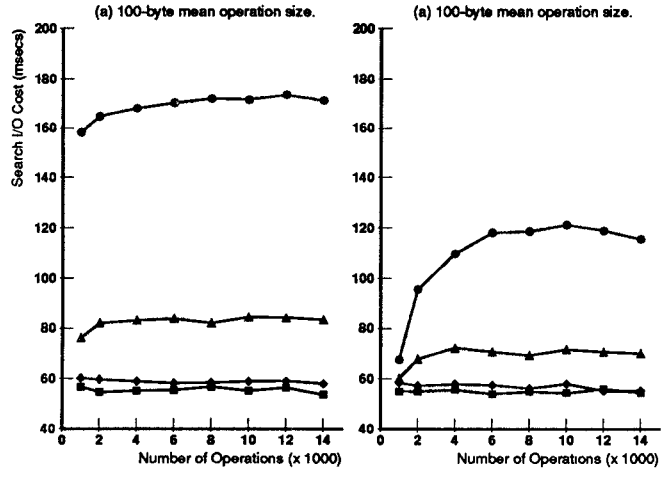
Turning to Figure 9.b which shows the cost of 10K-byte reads in ESM, the cost difference between 1-page and 4-page leaves becomes substantial (approximately, it doubles). This is because roughly 3 pages need to be accessed to read 10K bytes, and the cost of this multi-page read is higher in the first case.

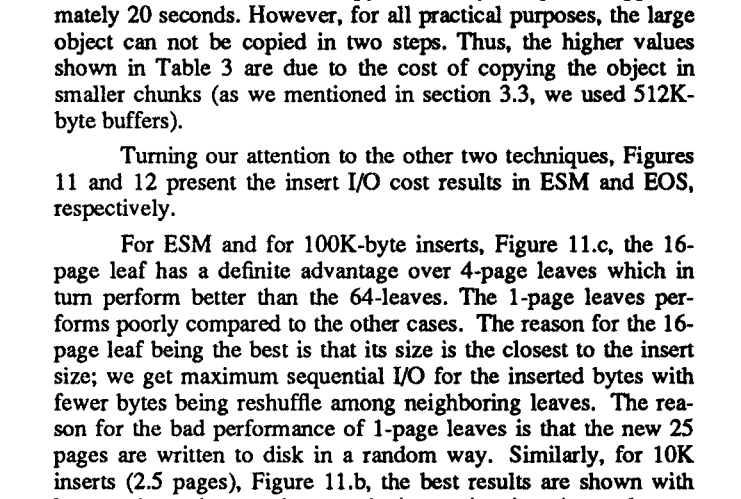
Comparing Figures 9.b and 10.b and for 1-page leaves, the performance of EOS is better than the one of ESM (about 15%). This can be explained by the fact that EOS's leaf nodes are variable-length. The new bytes are inserted into a 3-page (12K) leaf segment, whereas ESM would insert them into 3 separate leaf pages. For larger leaves, the two algorithms have practically the same performance.

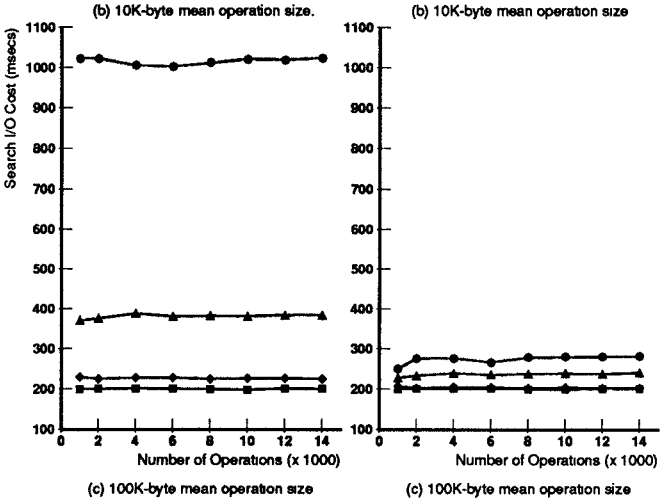
In Figures 9.c and 10.c which show the cost of 100K-byte reads, the differences between the two algorithms for the same











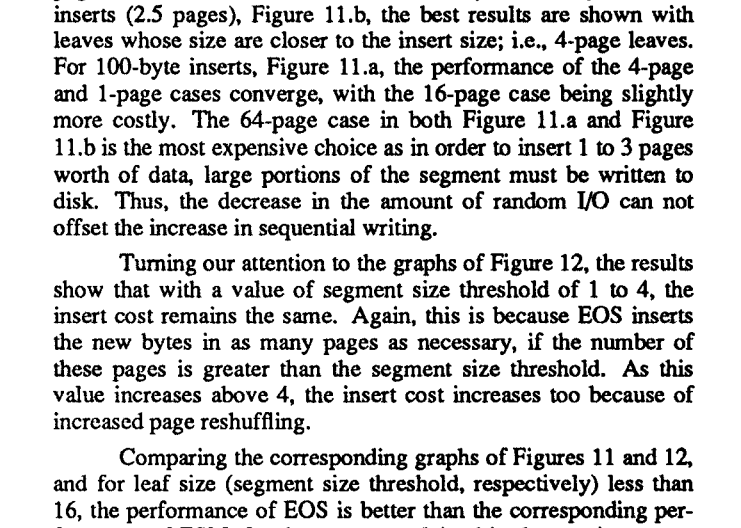


Figure 9.
ESM read I/O cost.

Figure 10.
EOS read I/O cost.

Mean Operation size (bytes)	100	10K	100K
Read I/O Cost (milliseconds)	37	54	201

Table 2: Starburst read I/O cost.

leaf size setting becomes large. Again, the reason is that EOS inserts the new 100K-byte data into a 25-page leaf even when the segment size threshold is set to anything less than 25 pages.

To summarize the results of this experiment, the general trend is that in both algorithms and for all operation sizes larger segments offer better read performance. Comparing ESM and EOS, for the same read size EOS performs better or the same as ESM. Comparing EOS and Starburst, we can see from the graphs of Figure 10 that a segment threshold size of 16 is adequate for EOS to reach the performance of Starburst.

4.4.3. Update Cost

Table 3 shows the I/O cost results for inserts and deletes in Starburst. The cost turns out to be the same in both types of updates and it does not depend on the operation size. The dominant factor here is the cost of copying the object segments from one place in disk to another. With 1K-byte/millisecond transfer rate, the minimum cost to copy a 10M-byte object is approximately 20 seconds. However, for all practical purposes, the large object can not be copied in two steps. Thus, the higher values shown in Table 3 are due to the cost of copying the object in smaller chunks (as we mentioned in section 3.3, we used 512K-byte buffers).

Turning our attention to the other two techniques, Figures 11 and 12 present the insert I/O cost results in ESM and EOS, respectively.

For ESM and for 100K-byte inserts, Figure 11.c, the 16-page leaf has a definite advantage over 4-page leaves which in turn perform better than the 64-leaves. The 1-page leaves performs poorly compared to the other cases. The reason for the 16-page leaf being the best is that its size is the closest to the insert size; we get maximum sequential I/O for the inserted bytes with fewer bytes being reshuffle among neighboring leaves. The reason for the bad performance of 1-page leaves is that the new 25 pages are written to disk in a random way. Similarly, for 10K inserts (2.5 pages), Figure 11.b, the best results are shown with leaves whose size are closer to the insert size; i.e., 4-page leaves. For 100-byte inserts, Figure 11.a, the performance of the 4-page and 1-page cases converge, with the 16-page case being slightly more costly. The 64-page case in both Figure 11.a and Figure 11.b is the most expensive choice as in order to insert 1 to 3 pages worth of data, large portions of the segment must be written to disk. Thus, the decrease in the amount of random I/O can not offset the increase in sequential writing.

Turning our attention to the graphs of Figure 12, the results show that with a value of segment size threshold of 1 to 4, the insert cost remains the same. Again, this is because EOS inserts the new bytes in as many pages as necessary, if the number of these pages is greater than the segment size threshold. As this value increases above 4, the insert cost increases too because of increased page reshuffling.

Comparing the corresponding graphs of Figures 11 and 12, and for leaf size (segment size threshold, respectively) less than 16, the performance of EOS is better than the corresponding performance of ESM, for the reason explained in the previous paragraph. However, for values greater or equal than 16, the comparison yields to mixed results, with ESM being better for small inserts, and EOS being better for larger ones. We have also examined the I/O cost for deletes. Because of space constraints we do not present graphs but we briefly mention that the trends mentioned for inserts are also valid for the delete operations (readers interested in the graphs may refer to the technical report).

To summarize these results, Starburst performs badly when it comes to inserts and deletes. The update costs in both ESM and EOS are well below the corresponding costs in Starburst. Notice

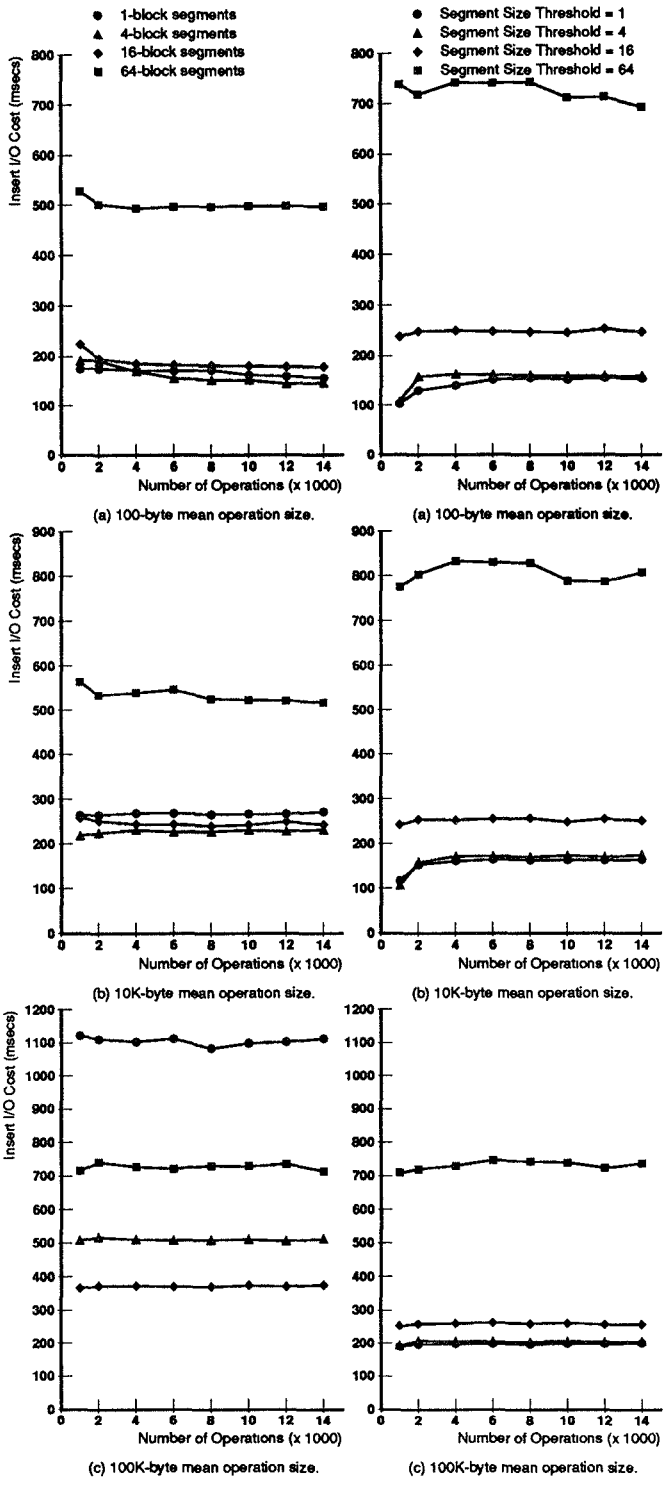


Figure 11.
ESM insert I/O cost.

Figure 12.
EOS insert I/O cost.

Mean Operation size (bytes)	100	10K	100K
Insert, Delete I/O Cost (seconds)	22.3	22.3	22.3

Table 3: Starburst insert and delete I/O cost.

also that the update cost in both ESM and EOS is independent of the object size, while in Starburst this cost depends directly on the object size. For 100M-byte object the difference between ESM/EOS and Starburst becomes dramatic; the cost remains at the same levels in both ESM and EOS, and it rises to approximately 2.5 minutes in Starburst.

4.5. Comparison with Results in [Care86]

The only previous work similar to ours is the study of [Care86] where they present some experimental evidence of the sort of performance that can be expected using the ESM large object manager. They report results on storage utilization, and the I/O costs of reads, inserts, and deletes using 100-byte and 10K-byte operation sizes, and leaf sizes of 1 and 4 pages. Thus, the comparison can be made only for the above cases.

Regarding the storage utilization, our results for the 100 and 10K-byte operations match the ones in [Care86]. Their conclusion, however, that increasing the operation size further will diminish the difference in utilization between the case of 1-page leaves and larger leaves, conflicts with our conclusions as is evident from the graphs of Figure 7.

We attempt to explain this point by comparing the 1-page leaf case with any other case of P-page leaves, where $P>1$. When a P-page leaf is split because of an N-byte insertion, the result is two or more new P-page leaves. The total number of unused bytes in these leaves is less than the number of bytes that can fit in P pages. When a 1-page leaf is split because of the same size insertion, the result is two or more 1-page leaves, and the total free space of these leaves can not be more than a page. So, in general, the new leaves will be better off with respect to utilization with 1-page leaves than with P-page leaves. Now, the effect of the utilization of this particular region of the object on the total storage utilization depends on the size of this region compared to the object size; i.e., it depends on the insert size. Thus, for the same object size, the difference on total object utilization between the 1-page and P-page leaves will increase as the insert size increases.

Regarding the read I/O cost for multi-page leaves, the performance of ESM in our experiments is better than the ones reported in [Care86]. This is because our unit of I/O is a disk page, regardless of the leaf block size. Thus, we were able to see in the graphs of Figure 9 the clear advantage of using larger leaves. In their simulation entire leaf blocks are read even when only few pages within the leaf need to be read which increased the I/O cost of reads for multi-block leaves.

Regarding the ESM I/O cost for inserts and deletes, our results differ slightly with the ones reported in [Care86] in a non significant way. It seems that the cost of shadowing somehow offsets the benefits of partial reads and writes.

4.6. Summary of Results

In this section, we summarize the results of our experiments for each of the three techniques we have examined and we compare their relative performance.

First, we discuss Starburst because the evaluation of its performance is relatively easy. The Starburst mechanism achieves excellent performance in all aspects we have examined in this study except for the operations that insert (delete) bytes at (from) arbitrary positions within the object. For these operations, Starburst performs poorly (the larger the object the worse the performance). Again, the applications this long field manager was intended for were large mostly read-only objects where such length-changing updates are applied infrequently.

ESM introduced a very flexible data structure that can gracefully handle all the desired operations on large objects including any kind of updates, and it can do this independently of the object size. The exact performance of ESM will depend on the leaf block size which is given by the applications as a hint so certain operations are optimized. Our study, however, suggests that it is hard to choose a right value for the leaf block size for the following reasons. The leaf size has diametrically different effects on storage utilization and on the performance of sequential and random reads. Large leaves waste too much space at the end of partially full leaves but offer good search time, and small leaves offer good storage utilization but require doing many I/O's for reads. Thus, in general, storage utilization and read time can not be optimized at the same time. We have also seen a case where a single operation can not be optimized without knowing the operation size (object creation, Figure 5).

The performance of EOS depends on the segment size threshold T. The tradeoffs that need to be examined in order to set this value are the following. Larger segments lead to better storage utilization, lower (sequential and random) read costs and higher update cost. Thus, in contrast to ESM, the tradeoff is simple: the only aspect of the performance that might be affected negatively by larger segments is the costs of inserts and deletes.

The selection process for the optimum value of the segment size threshold in EOS is also simple and specific. *First*, segments less than 4 blocks must be avoided. This is because for smaller segments and regardless of the operation size, (a) storage utilization drops substantially, Figure 8, (b) the cost of reads increases, Figure 10, and (c) the cost of maintaining 4-block segments is approximately the same with the cost of maintaining smaller segments, Figure 12. In other words, with 4-block segments, better storage utilization and read performance comes for free. *Second*, 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 threshold the better the overall performance.

When no length-changing updates are applied on the large object, Starburst and EOS perform exactly the same. If such updates are applied, EOS can achieve the same performance as Starburst with a cost for updates that is well below the corresponding cost in Starburst. For the quite large range of operation sizes we have examined in our study (100 bytes to 100K bytes), it is easy to select a segment size threshold that will justify the above statement; e.g., with a threshold value of 64 blocks, EOS provides the same read and utilization performance as Starburst except that the update cost in EOS is approximately 30 times lower.

5. CONCLUSIONS

In this paper we have examined and analyzed the performance of the three large object management techniques proposed in EXODUS [Care86], Starburst [Lehm89], and EOS [Bili92]. To analyze the algorithms we measured object creation time, sequential scan time, storage utilization in the presence of updates, and the I/O cost of random reads, inserts, and deletes.

Our main conclusions are the following. We found that Starburst provides excellent performance, in many cases the best possible, on all aspects we have examined except for length-changing updates where it performs badly (the larger the object the worse the performance). EXODUS can gracefully handle all operations on large objects including any kind of updates, and it can do so independently of the object size. However, in order for EXODUS to perform well on reads, the segment size must be increased. Our study suggest that this can not be done for free in that large fixed-size segments provide poor storage utilization. Finally, our results suggest that the algorithms of EOS can perform exactly the same as in Starburst except for length-changing updates where the update costs in EOS are well below the corresponding cost in Starburst.

ACKNOWLEDGMENTS

I would like to thank Amir Samad who wrote the simulation driver and run some initial experiments. Thanks are also due to Sue Nagy and Alex Buchmann for their constructive comments. Finally, I am grateful to Toby Lehman of IBM Almaden Research Center for providing many helpful comments for our design as well as for his input regarding Starburst.

REFERENCES

[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.

[Bili92] Biliris, A., "An Efficient Database Storage Structure for Large Dynamic Objects," *Proc. IEEE 8th Int. Conf. on Data Engineering*, Phoenix, Arizona, February 1992, pp. 301-308.

[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 VLDB Conference*, Kyoto, Japan, August 1986, pp. 91-100. Also, TR 638, CS Department, University of Wisconsin-Madison.

[Care89] Carey, M. J., DeWitt, D. J., Richardson, J. E., Shekita, E. J., "Storage Management for Objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W.Kim and F.Lochovsky, eds., Addison-Wesley, 1989.

[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.

[Effe84] Effelsberg W., Haerder T., "Principles of Database Buffer Management", *ACM Transactions on Database Systems*, Vol. 9, No 4, 1984, pp.560-595.

[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. Also, in IBM Res. Rep. RJ288 (Feb. 1978).

[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.

[Ston84] Stonebraker, M., Rowe, L. A., "Database Portals: a New Application Program Interface," *Proc. Int. Conf. on Very Large Data Bases*, Singapore, August 1984, pp. 3-13.

[Ston91] Stonebraker, M. *et. al.*, "Managing Persistent Objects in a Multi-Level Store," Electronics Research Laboratory, University of California at Berkeley, TR M91/16, March 1991.