Efficient Storage Management for Large Dynamic Objects

Byungyeon Hwang

Department of Computer Science
Songsim University
Kyonggi, Puchon 422-743, Korea
byhwang@sicom.kaist.ac.kr

Inhwan Jung and Songchun Moon

Department of Information and
Communications Engineering, KAIST
207-43, Cheongryangni, Dongdaemun
Seoul 130-012, Korea

Abstract

In this paper, we propose a new database storage manager, called Buddy-size Segment Storage System (BSSS), to handle large dynamic objects of any size and then implement search, insertion, append, and deletion algorithms used for the storage structure. The internal nodes of the proposed storage manager are identical to the ones proposed in Exodus. However, unlike Exodus that has fixed-size segments for leaf blocks, BSSS has binary buddy-size leaf segments. The proposed storage manager is compared with Exodus through performance results from simulation approach. BSSS performs the same as or better than the best case of Exodus for object creation time, sequential scan time, and random search time. However, the insertion performance of BSSS is the same as or worse than the one of Exodus.

1: Introduction

The manipulation of large objects is an important issue in many unconventional database applications such as geographic information systems, computer-aided design, office information systems, and multimedia presentation. In particular, multimedia presentation requires displaying images, showing movies, or playing digital sound recordings in real time [2, 8, 14]. Efficient manipulation of large objects is also important in any object-oriented and extended relational database management system to support advanced data modelling constructs like long list.

Several storage systems to manage large complex objects have been proposed [7, 12, 13, 15]. The problems in the existing storage managers for large objects are briefly discussed here. The data structure of WiSS [4] and Starburst [11] consists of data segments and a directory for the data segments. Since the directory is a set of sequential entries pointing to the data segments, dynamic insertion and deletion operations of large objects require the reorganization of the directory; the corresponding partial insertion and deletion algorithms are inefficient. Furthermore, the long field of WiSS has a size limit of 1.6 megabytes. In case of Starburst, it is 448 megabytes. Therefore, they cannot store larger data objects than their own limited size.

In general, the pages organizing an extent give good search performance since they are allocated to physically contiguous disk blocks. In WiSS, when a volume is created, the size of an extent, in pages, organizing the file is fixed. Therefore, if the size of a data object is larger than that of the fixed extent, the data object is separately allocated in several extents. Obviously, this implies that the search performance of WiSS is poor when the size of a data object is larger than that of the fixed extent. Exodus [3] can set the size of data pages of all large objects to be some fixed number of disk blocks. However, this mechanism is not suitable to applications that want to simultaneously optimize both search time and storage utilization. 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 disk I/O for read operations.

To solve the problems described above, we propose a new storage manager, Buddy-size Segment Storage System (BSSS), which stores data objects of any size. BSSS has binary buddy-size leaf segments, whereas Exodus has fixed-size leaf segments. The proposed storage manager handles large objects by storing them on data pages that are indexed by the B^+-tree [6] structure, where the key is the maximum byte position stored in a leaf data page. Therefore, the partial insertion and deletion algorithms of our storage manager for large dynamic objects are more efficient than those of WiSS and Starburst which use a sequential directory structure.

We believe that most applications handling large complex objects such as voice, image, sound, or video will read and write whole large objects and not perform update operations that affect only parts of large objects. However, some applications using large objects may want to access only a portion of a large object at a time. That

0-8186-6430-4/94 $4.00 © 1994 IEEE

is, to retrieve an object, one would rather sequentially scan through the object in smaller portions, rather than access the whole chunk in one step. For instance, playing of digital sound recordings or frame-to-frame accessing of a movie is required to access only a portion of a large object at a time. Similarly, for the insertion and deletion operations for a large object, elements may be inserted or deleted at any place within the large object. In multimedia applications, pictures may be annotated and movie spots may be edited to remove or add frames of the movie.

Our storage manager is based on the binary buddy system [10] to manage disk space. The buddy system is known for fast allocation and automatic coalescing of blocks on deallocation, but it has also been reported to suffer from poor space utilization due to fragmentation [5]. Many of the negative reports about the buddy system come from those trying to use the buddy system to allocate a single block of storage rather than a set of blocks. However, when multiple blocks are used, both internal and external fragmentation can be reduced greatly. Since our storage system can supply buddy segment sizes in 2^n units (the units typically being disk pages) and the last allocated segment is trimmed, it is possible to allocate the correct sizes of buddy segments so that the unused portion of an allocated segment is always less than a page.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 presents a new storage manager to manage large dynamic objects, and then describes search, insertion, append, and deletion algorithms used for the storage manager. The performance results for the proposed storage manager and Exodus are shown in Section 4. Finally, conclusions appear in Section 5.

2: Related work

The relational DBMS lacks the efficient support of large complex data types and spatial searches. These new requirements produce a need to redesign the storage manager. A number of researchers have addressed the storage management problems for new applications [3, 9, 16]. Because we are interested in spatial applications particularly, in this section, we briefly describe some storage managers that are related to our work.

2.1: WiSS

The WiSS consists of four distinct layers. The lowest level handles physical I/O and space allocation. The buffer manager is designed to accept user hints. Above this is the storage structure level which includes records, files, and indices. Finally, there is an interface layer that supports scans and large storage objects. A feature of WiSS is that it supports large storage objects which exceed a page in size. This is in contrast to earlier systems that forced the user to implement these structures on top of page sized records. However, the storage manager differentiates long items from small items, so it is not possible to build an index on a file of long data item. A WiSS *Long Data Item* consists of a directory and a set of small records less than one page long. To avoid fragmentation on the last page, the last record used is stored on a page that is shared with other records. There is a limit of 1.6 megabytes on the length of a long data item, which makes it unsuitable for storing huge images. WiSS provides dense B^+-tree indices and hashing as access methods. However, there is no provision made for adding new access methods to the system. The buffer manager uses a least recently used (LRU) replacement strategy with hints consisting of three priority levels. The physical I/O management level improves on the standard UNIX file system by allocating space for files in extents. However, the size of an extent, in page, organizing the file is fixed when a volume is created. Therefore, if the size of a data object is larger than that of the fixed extent, the data object must be separately allocated in several extents and cannot be clustered.

2.2: Exodus

The Exodus storage manager is a part of the Exodus extensible DBMS project. In order to enhance extensibility at the persistent programming level (E language), the emphasis was on minimizing semantics in the storage manager. Conceptually, all data handled by the Exodus storage manager are viewed as variable length byte sequences or *storage objects* that can be arbitrarily long, and are uniquely identified by Object Identifiers (OIDs). The choice of physical surrogates over logical surrogates avoids a table lookup for each reference, but can cause storage fragmentation in the long run. Long data items are managed by a B^+-tree like structure based on byte offset. Objects smaller than a page are stored as single records. With a uniform interface to large and small storage objects, it is possible to index files containing both types of objects. However, indices are not provided in the storage manager. Files in Exodus are collections of storage objects which are managed by using B^+-tree structure based on disk page number. Exodus comes close to meeting our storage management requirements. However, under the circumstance the size of an object increases dynamically, search performance becomes poor since Exodus sets the size of an extent to be some fixed number of disk blocks. Also, the storage manager does not support indices.

2.3: Starburst

The Starburst long field manager was designed to manage large database objects such as voice, image, and video. The long field manager uses the buddy system to manage disk space. Because it is not practical to store a long field directly in the relation, the long field descriptor is stored in the relation. The long field descriptor is a single level directory of disk extents. The variable disk extents, called *buddy segments*, are taken from *buddy space* which are large fixed-size sections of disk that are reserved for long field. Buddy spaces are taken from an even larger portion of disk, labeled *DB space.* A buddy space comprises an allocation page and a data page. The allocation page describes the state size of the individual buddy segments in the buddy space data area. Buddy segments contain only data. The maximum le gth of a long field varies with the maximum buddy segment size since the maximum length of the long field descriptor is fixed. The current design of Starburst, based on a 4 kilobyte allocation page, does not support buddy segments larger than 8 megabytes. Therefore, the Starburst long fields have a size limit of 448 megabytes.

3: New storage manager for large objects

In this section, we present a new storage manager, Buddy-size Segment Storage System (BSSS), to handle large dynamic objects. Also, the search, insertion, append, and deletion algorithms for the proposed storage manager are described in this section.

3.1: Structure of a large object

In this section, we propose a new data type called *Buddy_Object* to manage large objects of any size. The proposed *Buddy_Object* data is represented on disk as a B^+-tree index and a collection of leaf blocks with binary buddy-size leaf segments. An example of a *Buddy_Object* data is shown in *Fig. 1*. In *Fig. 1*, the size of a page is assumed to be 100 bytes. The root of the tree contains a number of (count, page number) pairs, one for each child of the root. The count value associated with each child pointer gives the maximum number of bytes stored in the subtree rooted at that child. Therefore, the count for the rightmost child pointer is the size of the object. Internal nodes are recursively defined as the root of another object contained within its parent node; thus, an absolute offset within a child is translated to a relative offset within its parent node. The leaf segments in a *Buddy_Object* data contain pure data. The left child of the root in *Fig. 1* contains bytes 1-471, and the right child contains the rest of the object (bytes 472-736). The rightmost leaf segment in *Fig. 1* contains 144 bytes of data. Byte 100 within this leaf node is 121+100=221 within the right child of the root, and it is 471+221=692 within the object as a whole.

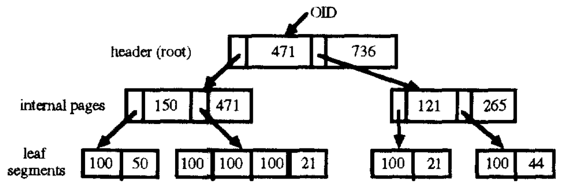

Fig. 1. An example of a *Buddy_Object* data

3.2: Search

The search algorithm supports the retrieval of a sequence of N bytes starting at byte position S in a *Buddy_Object* data. Only minor modifications need to be made to the algorithm proposed in Exodus so that buddy size leaf segments can be supported.

Search (S: byte position, N: number of bytes)

```
S1. [Search the Root Page]
    Let $start = S$, and read the root page and call it page $R$. The $count$[i] and $pid$[i] represent (count, pointer) pairs, and let $count[-1] = 0$ by convention.
S2. [Search Internal Pages]
    While $R$ is not a leaf page, do the following: Save the address of $R$ on the stack and binary search $R$ to find the smallest $count$[i] such that $start \le count[i]$. Set $start = start - count[i - 1]$, and read the page associated with $pid$[i] as the new page $R$.
S3. [Search Leaf Pages]
    Let a page size be $P\_S$. $R$ is now a leaf segment. Once at a leaf, the first desired byte is on the leaf segment $R$ at location $start$. Byte $start$ within $R$ is in page $P = R + \lfloor start/P\_S \rfloor$ at byte $B = start \bmod P\_S$ within $P$. If $N$ is less than the number of bytes in $R$ after $B$, read all pages from $P$ up to $R + \lfloor (start + N)/P\_S \rfloor$. If $N$ is greater than the number of bytes in $R$ after $B$, read $P$ and all pages of $R$ on the right of $P$. Use the stack to obtain the rest of the bytes.
```

Suppose we want to read 360 bytes starting from byte 170 of the *Buddy_Object* data shown in *Fig. 1*. To locate byte *start* = 170, we find that *count*[0] = 471 is the smallest count of the root that is greater than 170. Now we have to locate byte *start* = 170 - *count*[-1] = 170 in the child node pointed by *pid*[0]. We repeat the same process in the child. That is, we find that *count*[1] = 471 is the smallest count greater than 170, and thus, we set R = *pid*[1], and *start* = 170 - *count*[0] = 20. Byte 20 is in page $R + \lfloor 20/100 \rfloor = R + 0$, at byte 20 within that page. We read pages R through $R + 3$ to retrieve the first (*count*[1] - *count*[0]) - 19 = (471-150) - 19 = 302 of the desired bytes. Then, the next leaf segment needs to be retrieved for the remaining 58 bytes. 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.

3.3: Insertion

The insertion algorithm supports the insertion of a sequence of N bytes after the byte at position S. An example of N bytes insertion in segment L is shown in *Fig. 2*. In *Fig. 2*, B represents the byte position within P where the first byte of N should be placed. Let M be the left or right neighbor of L with the most free space which can be determined by examining the count information in L's parent node, and let L_S be the number of bytes of the largest buddy segment that is smaller than or equal to $\lfloor N / P\_S \rfloor$.

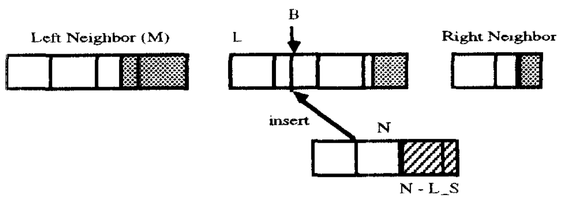

Fig. 2. An example of N bytes insertion in segment L

Insert (S: byte position, N: number of bytes)

I1. [Search the *Buddy_Object* Tree]
Traverse the *Buddy_Object* tree until the leaf containing byte S is reached, as in the **Search** Algorithm. As the tree is traversed, update the counts in the nodes to reflect the number of bytes to be inserted, and save the search path on the stack.

I2. [Insert N Bytes into the Leaf Node]
Call the leaf segment into which bytes are being inserted L. When L is reached, try to insert the N bytes there. If no overflow occurs, then the insert is done, as the internal node counts will have been updated in step I1.

I3. [Process Overflows]
If an overflow occurs, proceed as follows: If L and M together have a sufficient amount of free space to accommodate N - L_S bytes of data, then evenly distribute the new data plus the old contents of L and M between these two nodes and the largest buddy segment that is smaller than or equal to $\lfloor N / P\_S \rfloor$. Otherwise, simply allocate the smallest buddy segment that is larger than or equal to $\lceil N / P\_S \rceil$, and evenly distribute L's bytes and the bytes being inserted among L and the newly allocated leaf segment.

I4. [Propagate Upward]
Propagate the counts and pointers for the new leaf segment upward in the tree, using the stack built in step I1. If an internal node overflows, handle it in the same way that leaf overflows are handled.

3.4: Append

The append algorithm supports the addition of N bytes to the end of a *Buddy_Object* data. In most case, the last segment of a *Buddy_Object* data will be trimmed to some fraction of its original size to reduce international fragmentation.

Append (N: number of bytes)

A1. [Traverse Rightmost]
Make a rightmost traversal of the *Buddy_Object* tree. As the tree is being traversed, update the counts in the internal nodes to reflect the effect of the append. Save the search path on the stack.

A2. [Append to the Rightmost Leaf Segment]
The final size of a *Buddy_Object* data may or may not be known in advance. When the size of a *Buddy_Object* data is known a priori, a segment just large enough to hold the entire object is allocated. Then, each chunk of bytes is appended at the end of the previous one with no holes in between them. This is shown in *Fig. 3*. When the eventual size of a *Buddy_Object* data is not known a priori, we follow the growth scheme used in [11]; 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

Fig. 4 is created by successively appending each chunk of bytes with binary buddy-size leaf segments.

A3. [Trim the Last Allocated Segment]

To reduce *internal fragmentation*, the last allocated segment is trimmed. Consider how a segment of 16 pages would be trimmed to hold a string of bytes that occupy 11 pages. The sequence of segment sizes needed is the binary representation of the number of pages needed to hold the *Buddy_Object* data. For instance, 11_{10} is 1011_2, thus there is a segment of size eight, a segment of size two, and a segment of size one. The remaining segments (size one and four) are given back to the free space.

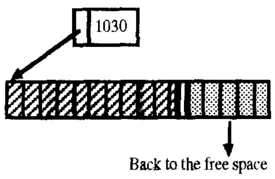

Fig. 3. *N* bytes append when object size is known a priori

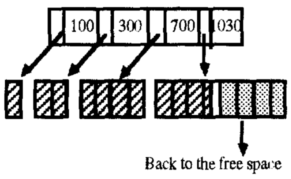

Fig. 4. *N* bytes append when object size is not known a priori

3.5: Deletion

The deletion algorithm supports the deletion of *N* bytes starting at a specified byte position *S*. This operation can result in either deletion of entire subtrees or partial deletion 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. Then, the deletion algorithm proceeds to the second phase to perform partial deletion. For a segment *A*, A_t is used to denote the total number of bytes kept in *A* and A_l is used to denote the number of bytes in the last page of *A*. An example of *N* bytes deletion is shown in *Fig. 5*.

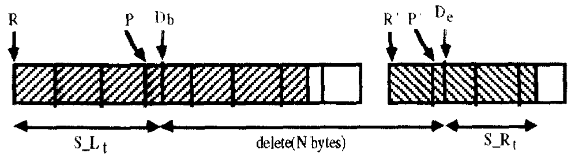

Fig. 5. An example of *N* bytes deletion

In *Fig. 5*, D_b in page *P* of segment *R* and D_e in page *P'* of segment *R'* are used to denote the first byte and the last byte to be deleted, respectively. Let *L_B* and *R_B* be the size of the smallest binary buddy segment larger than or equal to S_L_t and S_R_t, respectively. To delete all bytes of *R* on the right of D_b, we simply decrement the counts in the parent of *R* and free all pages of *R* on the right of *P*, which are *R* - *L_B*. Then, we proceed by freeing all pages of *R'* on the left of *P'*, which are *R'* - *R_B*. Now, the bytes of *P'* on the right of D_e must be shifted to the left. The deletion algorithm is as follows.

Delete (*S*: byte position, *N*: number of bytes)

D1. [Delete *N* Bytes]

Traverse the *Buddy_Object* tree to the left and right limits of the deletion, and save each path to a stack. All subtrees completely enclosed by the traversal are deleted.

D2. [Prepare Deletion Operation]

Compute the number of bytes P_t and the byte D_b of page *P* within *R* where deletion starts. Similarly, compute P'_t and D_e of page *P'* within *R'* where deletion ends. Set $S\_L = R$ and $S\_L_t = P \times P\_S + D_b - 1$. Set $S\_R = R' + P'$ and $S\_R_t = R'_t - (P' \times P\_S + D_e - 1)$.

D3. [Trim the Partially Deleted Segments]

When *L_B* is the size of the smallest binary buddy segment larger than or equal to S_L_t, free all pages of *R* on the right of *P*, which are *R* - *L_B*. When *R_B* is the size of the smallest binary buddy segment larger than or equal to S_R_t, free all pages of *R'* on the right of *P'*, which are *R'* - *R_B*.

D4. [Propagate Upward]

Propagate the new counts and pointers up to the root of the tree, using the stacks built in step D1. If an internal node has less than the allowed number of pairs, merge or reshuffle it with a sibling.

D5. [Fix Root]

If the root has only one child, make this child the new root and go to step D5.

4: Performance of storage managers

We present the performance evaluation results for two managers which use a B-tree like directory structure – Exodus [3] and our storage manager, BSSS. The objective of our experiments is to demonstrate the effectiveness of clustering in our storage system which is based on the binary buddy system to manage disk space. We choose a simulation modeling as the approach of the performance evaluation since an analytical modeling can not consider a dynamic behavior of the storage managers.

4.1: Environment

The experiments are performed on 4 Kbyte 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 1 Kbyte 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 (12 Kbytes) segment is 33+4x3=45 milliseconds; the cost of reading the same number of blocks with 3 I/O calls is (33+4)x3=111 milliseconds. The size of the buffer pool was set to 12 pages.

The simulations run on a 10 Mbyte object. The root of the object is placed in a page with no other objects in it. When pointer and count values of an index page require 4 bytes each, we may store up to 507 pairs in the root and 511 pairs in internal pages with 4 Kbyte pages. For BSSS, we used maximum segment size of 64 pages. For Exodus, leaf segment size of 1, 4, 16, and 64 pages were used.

4.2: Object creation time

This section presents the time needed to built a 10 Mbyte object by successively appending fixed-size chunks of bytes. We start with 1 Kbyte and go up to 512 Kbyte chunks.

In Exodus, with 1 page leaves, a 10 Mbyte object turns out to be of level 2 – the root, the one level of 6 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 BSSS, the tree level is always 1. *Fig. 6* shows the time required to build a 10 Mbyte object. 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, 48, 64, 96, 128, 192, 256, and 512.

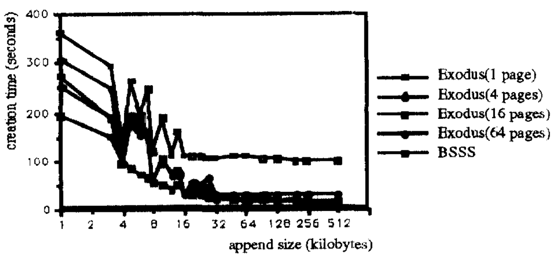

Fig. 6. 10 Mbyte object creation time

In two algorithms of *Fig. 6*, object creation time depends on the append size by a few kilobytes. For Exodus, the object build cost for 1 page leaves and for 3 Kbyte appends is approximately 292 seconds; it drops to 134 seconds for 4 Kbyte appends and rises up again to 265 seconds for 5 Kbyte appends. This is due to the change of the number of disk accesses by means of the mismatch of the block boundaries and append size. For instance, the number of disk accesses for 3 Kbyte appends is 9387; it drops to 3628 for 4 Kbyte appends and rises up again to 7902 for 5 Kbyte appends. Furthermore, the object creation time depends on 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 distributed 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 BSSS algorithm but to a lesser degree. In BSSS, the new 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 BSSS, to come up with a tree of level greater than 1, the size of the object being created must be larger than 16 Gigabytes.)

For Exodus, we can not select a particular leaf block size as the winner. Referring to *Fig. 6*, for appends of 4, 16, 64, and 256 Kbytes, the best performance is achieved when the 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 256 Kbytes, larger leaf blocks have better performance.

4.3: Sequential scan time

This section shows the time required to sequentially retrieve the entire large object from the database. After the 10 Mbyte object was built in the experiment of Section 4.2, 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 since the resulting structure of BSSS depends on the size of the first append. For sequential scan time of BSSS, the best performance that can be achieved is approximately 12 seconds with a transfer rate of 1 Kbytes per millisecond. *Fig. 7* shows the time required to scan a 10 Mbyte object sequentially.

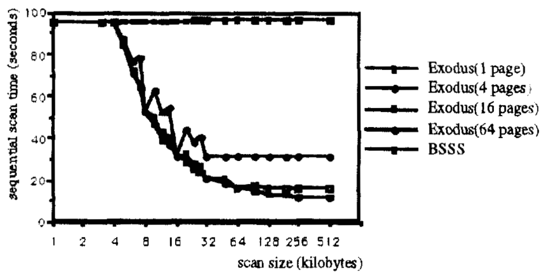

Fig. 7. 10 Mbyte object sequential scan time

As shown in *Fig. 7*, for scans shorter than the page size two 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 Exodus, 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 gives the best result when the scan size exceeds the segment size. The performance of BSSS follows the expected pattern; larger scans produce better response time.

4.4: Storage utilization cost

Storage utilization compares the object size with the actual space required to store the object including possible index pages. The average operation sizes are 100 bytes and 10 Kbytes.

In Exodus of 100 byte operations, 1 page leaf blocks provide slightly better storage utilization over 4 page leaf blocks for the small operations since a larger fraction of the leaf blocks are split for a given number of random update operations in the 4 page case and each one leaves more empty space as a fraction of the overall object size. It stabilizes at the same level of the low 80 percent. However, as larger operations (10 kbytes) are performed on the object, 1 page leaf blocks have a large storage utilization advantage over 4 page leaf blocks. This is due to the average operation size being large – the average insertion adds 10 Kbytes, or 2.5 pages of data. The inserted data is distributed over as few newly allocated 1 page leaf blocks as well as one or two existing partially filled leaf blocks, leading to 3-4 nearly full leaf blocks. However, with 4 page leaf blocks, the average insert is sure to split a leaf block, creating two relatively empty blocks as a result. In comparing Exodus and BSSS, we can see that their performance is approximately the same.

4.5: Random search cost

As random insertions and deletions degrade the large object structure, the search I/O cost for Exodus and BSSS is shown in *Fig. 8*. Each mark in the graph represents the average cost of the search operations performed since the previous mark. For example, the mark at the 10,000 operations indicates the average cost of the searches performed within the last 2,000 operations.

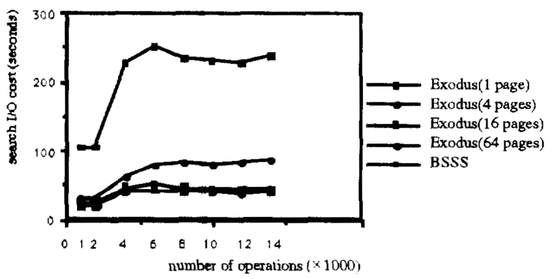

Fig. 8. Search I/O cost for 10 Kbyte operations

In Exodus of *Fig. 8*, which shows the cost of 10 Kbyte searches, it is evident that the 4 page leaves have a definite advantage over 1 page leaves. This is due to the fact that much less random I/O is needed to read 2.5 pages of data when each leaf block contains 2 to 4 sequential pages of data.

4. 6: Insert I/O cost

In Exodus of 10 Kbyte operations, 4 page leaves have about 10-13 percent performance advantage over 1 page leaves since fewer leaves need to be touched on the average when leaf blocks are 4 page leaves. Thus, the resulting decrease in random I/O outweighs the increase in sequential I/O. For 100 byte inserts, the performance of the 4 page and 1 page cases converge, with the 16 page case being slightly more costly. The 64 page case gives the most expensive cost for insertion 1 to 3 pages of data since 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.

In comparing BSSS and Exodus, the performance of BSSS is the same as or worse than the corresponding performance of Exodus since large portions of the segment must be written to disk.

5: Conclusions

In this paper, we have presented a new database storage structure, called BSSS, to manage large dynamic objects of any size. BSSS has binary buddy-size leaf segments, whereas Exodus has fixed-size leaf segments. The new storage manager has been compared with Exodus which has the same directory structure as our storage manager, through the performance results from simulation approach. For Exodus, large leaf blocks have a definite advantage for multi-page searches, but they also increase the cost for updates and lead to lower storage utilizations. Thus, in general, storage utilization and search time can not be optimized at the same time. BSSS performs the same as or better than Exodus for object creation time and sequential scan time. For the same search size, BSSS performs the better random search performance than Exodus.

We experimented with object sizes of 10 Mbytes and 100 Mbytes, running a query mix consisting of 40 percent searches, 30 percent inserts, and 30 percent deletes. This is a small percentage for searches of real application. However, the results do not depend on the query mix rather on the operation size. A large search percentage will simply require more runs to stabilize the performance curves. Equal percentages of inserts and deletes were used in order to ensure that the object size remained stable. Furthermore, these update operations were uniformly distributed throughout the body of the object. This uniform distribution assumption is a pessimistic assumption, as it produces worst-case average storage utilizations. For more static objects, or objects where updates tend to be clustered in just a few regions of the object, storage efficiency will improve up to 90-99 percent. Also, our insert I/O cost results are slightly pessimistic because our prototype does not handle that entire leaf blocks are read and written, or only the last page of a block is affected by an operation.

The Starburst long field manager uses the same binary buddy system as BSSS. However, Starburst has a sequential directory structure, whereas BSSS has a B-tree like structure. Therefore, Starburst does not nicely handle byte inserts and deletes on large objects since these operations require all segments to the right of the segment on which the update is performed to be reorganized. The original bytes together with the new ones are placed into a new set of segments. Thus, for insertion and deletion, Starburst produces the worse results than BSSS. However, the (sequential and random) searches, appends, and byte range replace results of Starburst are similar to those of BSSS.

References

[1] M. M. Astrahan, et. al, "System R: Relational Approach to Database Management," ACM Trans. on Database Systems, Vol. 1, No. 2, 1976, 97-137.

[2] A. Biliris, "An Efficient Database Storage Structure for Large Dynamic Objects," Proc. of 8th Int. Conf. on Data Engineering, 1992, 301-308.

[3] M. J. Carey, D. J. Dewitt, J. E. Richardson, and E. J. Shekita, "Object and File Management in the EXODUS Extensible Database System," Proc. of 12th Int. Conf. on Very Large Data Bases, Aug. 1986, 91-100.

[4] H. T. Chou, D. J. Dewitt, R. H. Katz, and A. C. Klug, "Design and Implementation of the Wisconsin Storage System," Software Practice and Experience, Vol. 15, No. 10, Oct. 1985, 943-962.

[5] S. K. Chowdhury and P. K. Srimani, "Worst Case Performance of Weighted Buddy Systems," Acta Informatica, Vol. 24, 1987, 555-564.

[6] D. Comer, "The Ubiquitous B-Tree," ACM Computing Surveys, Vol. 11, No. 2, Jun. 1979, 121-137.

[7] O. Deux, et. al, "The Story of O_2," IEEE Trans. on Knowledge and Data Engineering, Vol. 2, No. 1, 1990, 91-108.

[8] M. J. Egenhofer, "What's Special about Spatial? Database Requirements for Vehicle Navigation in Geographic Space," Proc. of ACM SIGMOD, 1993, 398-402.

[9] S. Khoshafian, M. J. Franklin, and M. J. Carey, "Storage Management for Persistent Complex Objects," Information Systems, Vol. 15, No. 3, 1990, 303-320.

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

[11] T. J. Lehman and B. G. Lindsay, "The Starburst Long Field Manager," Proc. of 15th Int. Conf. on Very Large Data Bases, 1989, 375-383.

[12] G. M. Lohman, B. Lindsay, H. Pirahesh, and K. B. Schiefer, "Extensions to Starburst: Objects, Types, Functions, and Rules," Communications of the ACM, Vol. 34, No. 10, 1991, 94-109.

[13] H. Lu, B. C. Ooi, A. Souza, and C. C. Low, "Storage Management in Geographic Information Systems," Proc. of 2nd Symposium on Large Spatial Databases, 1991, 451-470.

[14] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith, "The Sequoia 2000 Storage Benchmark," Proc. of ACM SIGMOD, 1993, 2-11.

[15] M. Sullivan and M. Olson, "An Index Implementation Supporting Fast Recovery for the POSTGRES Storage System," Proc. of 8th Int. Conf. on Data Engineering, 1992, 293-300.

[16] F. Velez, G. Bernard, and V. Darnis, "The O_2 Object Manager: An Overview," Proc. of 15th Int. Conf. on Very Large Data Bases, 1989, 357-366.