Object and File Management in the EXODUS Extensible Database System

by

Michael J. Carey
David J. DeWitt
Joel E. Richardson
Eugene J. Shekita

Computer Sciences Technical Report #638
March 1986

Object and File Management in the EXODUS Extensible Database System

Michael J. Carey, David J. DeWitt, Joel E. Richardson, Eugene J. Shekita

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the Department of Energy under contract #DE-AC02-81ER10920, by the National Science Foundation under grants MCS82-01870 and DCR-8402818, and by an IBM Faculty Development Award.

Object and File Management in the EXODUS Extensible Database System

Michael J. Carey, David J. DeWitt, Joel E. Richardson, Eugene J. Shekita

Computer Sciences Department
University of Wisconsin
Madison, WI 53706

ABSTRACT

This paper describes the design of the object-oriented storage component of EXODUS, an extensible database management system currently under development at the University of Wisconsin. The basic abstraction in the EXODUS storage system is the storage object, an uninterpreted variable-length record of arbitrary size; higher level abstractions such as records and indices are supported via the storage object abstraction. One of the key design features described here is a scheme for managing large dynamic objects, as storage objects can occupy many disk pages and can grow or shrink at arbitrary points. The data structure and algorithms used to support such objects are described, and performance results from a preliminary prototype of the EXODUS large-object management scheme are presented. A scheme for maintaining versions of large objects is also described. We then describe the file structure used in the EXODUS storage system, which provides a mechanism for grouping and sequencing through a set of related storage objects. In addition to object and file management, we discuss the EXODUS approach to buffer management, concurrency control, and recovery, both for small and large objects.

1. INTRODUCTION

Until recently, research and development efforts in the database management systems area have focused on supporting traditional business applications. The design of database systems capable of supporting non-traditional application areas, including engineering applications for CAD/CAM and VLSI data, scientific and statistical applications, expert database systems, and image/voice applications, has emerged as an important new direction for database system research. These new applications differ from conventional applications such as transaction processing and from each other in a number of important ways. First, each requires a different set of data modeling tools. The types of entities and relationships that must be described for a VLSI circuit design are quite different from the data modeling requirements of a banking application. Second, each new application area has a specialized set of operations that must be supported by the database system. It makes little sense to talk about doing joins between satellite images. Efficient support for the specialized operations of each of these new application areas requires new types of storage structures and access methods as well. Access and manipulation of VLSI databases is facilitated by new access methods such as R-Trees [Gutt84]. Storage of image data is simplified if the database system supports large multidimensional arrays as a basic data type (a capability provided by no commercial database system at this time). Storing images as tuples in a relational database system is generally either impossible or terribly inefficient. Finally, a number of these new application areas need support for multiple versions of entities [Daya85, Katz86].

Recently, three new database system research projects have been initiated to address the needs of this emerging class of applications: EXODUS[1] at the University of Wisconsin [Care85a], PROBE at CCA [Daya85], and POSTGRES [Ston86] at Berkeley. Although the goals of these projects are similar, and each uses some of the same mechanisms to provide extensibility, the overall approach of each project is quite different. For example, POSTGRES will be a more "complete" database management system, with a query language (POSTQUEL), a predefined way of supporting complex objects (through the use of POSTQUEL and procedures as a data type), support for "active" databases via triggers and alerters, and inferencing. Extensibility will be provided via new data types, operators, access methods, and a simplified recovery mechanism. A stated goal is to "make as few changes as possible to the relational model". The objective of the PROBE project, on the other hand, is to develop an advanced DBMS with support for complex objects and operations on them, dimensional data (in both space and time dimensions), and a capability for intelligent query processing. Unlike POSTGRES, PROBE will provide a mechanism for directly representing complex objects. Like EXODUS, PROBE will use a rule-based approach to query optimization so that the query optimizer may be extended to handle new database operators, new methods for existing operators, and new data types. An extended version of DAPLEX [Ship81] is to be used as the query language for PROBE.

In contrast to these two efforts, EXODUS is being designed as a modular (and modifiable) system rather than as a "complete" database system intended to handle all new application areas. The EXODUS storage system, at the lowest level, is the kernel of the system. Since it is to be the only fixed component of the EXODUS system, its design is intended to be flexible enough to support the needs of a wide range of potential applications. Application-specific access methods, operations, and version management layers will be constructed using the primitives provided by the storage system, and higher levels of the system will in turn use the primitives supplied by these layers. As mentioned above, the EXODUS design includes a generic query optimizer that optimizes a generalized algebraic query tree based on a collection of cost and operator transformation rules that implementors of application-specific DBMS's will provide. At the top level, EXODUS will provide facilities for generating application-specific high-level query language interfaces, but applications will also be permitted to interact with the system at lower levels when necessary. Thus, the EXODUS approach might be characterized as the "DBMS generator" approach, with the overall goal of the project being to implement the storage system; the tools to support development of appropriate abstract data types, access methods, operations, and version support; the rule-based optimizer; and the flexible query interface generator. To aid application-specific DBMS developers in their task, we also expect to provide libraries of useful routines (and rules) for the extensible components of the system.

In this paper we describe object and file management in EXODUS. We feel that this component, more than any other, is the key to a successful extensible database system. If the capabilities provided at this level are not sufficiently flexible and efficient, then regardless of how the upper levels of the system are defined or implemented, the resulting database system will almost certainly fail to satisfy the requirements of a wide

[1] EXODUS: A departure, in this case from the ways of the past. Also an EXtensible Object-oriented Database System. We are offering $5 to the first person to figure out how to fit the "U" in gracefully.

variety of applications. Furthermore, without a suitably powerful storage system, it seems that the user's task of extending the system will be much more difficult. In the following paragraphs, we briefly outline the key characteristics of the EXODUS storage system; we will expand on this information in later sections of the paper.

1.1. Storage System Characteristics

Storage Objects: The basic unit of stored data in the EXODUS storage system is the *storage object*, which is an uninterpreted byte sequence of virtually unlimited size. By providing capabilities for storing and manipulating storage objects without regard for their size, a significant amount of generality is obtained. For example, an access method can be written without any knowledge of the size of the storage objects it is manipulating. Not providing this generality has severely limited the applicability of WiSS [Chou85b]. While WiSS provides a notion of long objects, one cannot build a B+ tree on a file of long objects due to the way that the system's implementation differentiates between long and short objects.

Concurrency Control and Recovery: To further simplify the user's[2] task of extending the functionality of the database system, both concurrency control and recovery mechanisms are provided in EXODUS for operations on shared storage objects. Locking is used for concurrency control, and recovery is accomplished via a combination of shadowing and logging.

Versions: As discussed in [Daya85], many new database applications require support for multiple versions of objects. In keeping with the spirit of minimizing the amount of semantics encapsulated in the storage system of EXODUS, a generalized mechanism that can be used to implement a variety of versioning schemes is provided.

Performance: An important performance issue is the amount of copying that goes on between the buffer pool and application programs. If an application is provided direct access into the buffer pool, security may become a problem. On the other hand, in a database system supporting a VLSI design system (or many other new applications), the application may require direct access to the storage objects in the buffer pool in order to obtain reasonable performance — copying large (multi-megabyte) complex objects between the database system and the application may be unacceptable. EXODUS storage system clients are thus given the option of directly accessing data in the buffer pool; clients that will almost certainly take advantage of this option are the application-specific access methods and operations layers. For applications where direct access poses a security problem, a layer that copies data from database system space to user space will be provided.

Minimal Semantics: One of our goals is to minimize the amount of information that the storage system must have in order to manipulate storage objects. In particular, in order to keep the system extensible it does not seem feasible for the storage system to know anything about the conceptual schema. On the other hand, semantics can sometimes be useful for performance reasons. For example, it was shown in [Chou85a]

[2] Internally, we speak of such "users" as "database hackers" (or DBH's for short). We do not intend to imply that EXODUS can be extended by the naive user, as we expect EXODUS to be extended for a given application once, by a DBH, and then modified only occasionally (if at all) for that application.

that buffer management performance can be improved by allowing the buffer manager to capture some semantics of the operations being performed. Our solution is to keep schema information out of the storage system, but then to allow *hints* to be provided which can help in making decisions that influence performance in important ways. For example, the buffer manager accepts hints guiding its choice of replacement policies; other examples will arise elsewhere throughout the course of this paper.

1.2. Paper Outline

In the following section we describe related work on storage systems designed to facilitate the handling of large storage objects. Section 3 contains an overview of the interface provided to the upper levels of EXODUS by the storage manager. In Section 4, which is the majority of the paper, we present a detailed description of our design for storage objects and a preliminary performance evaluation of the algorithms that operate on large storage objects. Section 4 also describes the techniques employed for versioning, concurrency control, recovery, and buffer management for such objects. Section 5 sketches the techniques used to implement files of storage objects. Finally, Section 6 summarizes our main conclusions.

2. RELATED WORK

There have been a number of earlier projects to construct file and object management services similar to those provided by EXODUS. In [Kaeh82], LOOM, a Large Object-Oriented Memory for Smalltalk-80, is described. LOOM extends the object storage capabilities of Smalltalk-80 to allow the manipulation of up to 2^{31} objects instead of 2^{15} objects. Large (multi-megabyte) objects are not supported, and the system provides no facilities for concurrency control or recovery.

The objectives of the Gemstone database system [Cope84] were similar to those of EXODUS, POSTGRES, and PROBE. Gemstone, and its query language OPAL, encapsulated a variety of ideas from the areas of knowledge representation, object-oriented programming, non-procedural programming, set-theoretic data models, and temporal data modeling. In order to simplify the implementation of the capabilities desired, each Gemstone object was decomposed into a collection of small elements. An "object manager" was responsible for clustering related elements together. While versions of elements were supported, the paper does not describe how support for large elements is provided. One serious problem with the proposed design is that it appears that there is no mechanism for implementing versions via differencing. Other object-oriented database systems [Beec83, Bato84, Lyng84, Daya85, Lyng86] have also (for the most part) ignored the issue of how objects are to be implemented, particularly very large objects.

Another related project is the file system for the iMAX-432 [Poll81]. This file system provided support for system-wide surrogates to name objects and for atomic actions on objects using a modification of Reed's versioning scheme [Reed83]. However, a majority of the design was based on the premise that objects are small (less than 500 bytes); rather than worrying about how to handle large objects, special consideration was given to clustering related objects together and to garbage-collecting deleted objects to minimize wasted file space.

The storage system of POSTGRES [Ston86] is based on the use of tuples and relations. Each tuple is identified by a unique 64-bit surrogate that never changes. Tuples are not updated in-place. Rather, the modified tuple, with the same surrogate but a new timestamp, is inserted elsewhere into the database. A "vacuuming" process moves old data to an archival disk for long-term storage. Since complex objects are implemented through the use of POSTQUEL as a data type, no explicit mechanisms for supporting the storage or manipulation of large complex objects are provided.

3. THE STORAGE MANAGER INTERFACE

Before describing the details of how large storage objects and file objects (collections of storage objects) are handled in the EXODUS storage system, we must briefly outline the nature of the interface provided for use by higher levels of EXODUS. In most cases, we expect the next level up to be the layer that provides the access methods (and perhaps version support) for a given EXODUS application. This layer is likely to change from one application to another, although we expect to provide a library of standard access methods and version management code that can be used/extended by the author(s) of an application-specific DBMS.

The EXODUS storage system provides a procedural interface. This interface includes procedures to create and destroy file objects and to open and close file objects for file scans. For scanning purposes, the storage system provides a call to get the object id of the next object within a file object. It also provides procedures for creating and destroying storage objects within a file; all storage objects must reside in some file object (residing by default in the "system" file object if no other file object is specified). For reading storage objects, the EXODUS storage system provides a call to get a pointer to a range of bytes within a given storage object; the desired byte range is read into the buffers, and a pointer to the bytes there are returned to the caller. Another call is provided to inform EXODUS that these bytes are no longer needed, which "unpins" them in the buffer pool. For writing storage objects, a call is provided to tell EXODUS that a subrange of the bytes that were read have been modified (information that is needed for recovery to take place). For shrinking/growing storage objects, calls to insert bytes into and delete bytes from a specified offset in a storage object are provided, as is a call to append bytes to the end of an object (a special case of insert). Finally, for transaction management, the EXODUS storage system provides begin, commit, and abort transaction calls. (We also anticipate the inclusion of other transaction-related hooks to aid the access methods layer in implementing concurrent and recoverable operations for new access methods efficiently.)

In addition to the functionality outlined above, the EXODUS storage system is being designed to accept a wide variety of performance-related hints. For example, the object creation routine mentioned above accepts hints about where to place a new object (i.e., "place the new object near the object with id *X*") and about how large the object is expected to be (on the average, if it varies); it is also possible to hint that an object should be alone on a disk page and the same size as the page (which will be useful for the access methods level). In regard to buffer management, hints about how many buffer blocks to use and what replacement policy to employ will be accepted by the buffer manager. These hints will be supported by allowing a *scan group* to be specified with each object access, and then having the buffer manager accept these

hints on a per-scan-group basis, which will allow buffer management policies like DBMIN to be easily supported [Chou85a]. Other storage system hints will be mentioned later as the details of the storage system design are described.

4. STORAGE OBJECTS

As described in the introduction, *storage objects* are the basic unit of data in the EXODUS storage system. Storage objects can grow and shrink in size, and their growth and shrinkage is not constrained to occur at the end of an object, as the EXODUS storage system supports insertion and deletion of new portions of a storage object anywhere within the object. This section of the paper describes the data structures and algorithms that are used to efficiently support storage objects, particularly large dynamic storage objects.

Storage objects can be either small or large, although this distinction is hidden from clients of the EXODUS storage system. Small storage objects reside on a single disk page, whereas large storage objects occupy multiple disk pages. In either case, the object identifier (OID) of a storage object is of the form (*page #, slot #*). Pages containing small storage objects are slotted pages, as in INGRES, System R, and WiSS [Astr76, Ston76, Chou85b], so the OID of a small storage object is a pointer to the object on disk. For large storage objects, the OID points to a *large object header*. This header can reside on a slotted page with other large object headers and small storage objects, and it contains pointers to other pages involved in the representation of the large object. All other pages in a large storage object are private to the object rather than being shared with other small or large storage objects (except that pages may be shared between various versions of the same object, as we will see later). When a small storage object grows to the point where it can no longer be accommodated on a single page, the EXODUS storage system will automatically convert it into a large storage object, leaving its object header in place of the original small object. We considered the alternative of using surrogates for OID's rather than physical addresses, as in other recent proposals [Cope84, Ston86], but we rejected this alternative due to what we anticipated would be its high cost — with surrogates, it would always be necessary to access objects via a surrogate index.

4.1. Large Storage Objects

The data structure used to represent large objects was inspired by the ordered relation data structure proposed for use in INGRES [Ston83], although there are a number of significant differences between our insertion and deletion algorithms and those of Stonebraker's proposal. Figure 1 shows an example of our large object data structure. Conceptually, a large object is an uninterpreted sequence of bytes; physically, it is represented on disk as a B+ tree index on byte position within the object plus a collection of leaf (data) blocks. The root of the tree (the large object header) contains a number of (*count, page #*) pairs, one for each child of the root. The count value associated with each child pointer gives the maximum byte number stored in the subtree rooted at that child; the count for the rightmost child pointer is therefore also the size of the object. Internal nodes are similar, being recursively defined as the root of another object contained within its parent node, so an absolute byte offset within a child translates to a relative offset within its parent node. The left child of the root in Figure 1 contains bytes 1-421, and the right child contains the rest of the

object (bytes 422-786). The rightmost leaf node in the figure contains 173 bytes of data. Byte 100 within this leaf node is byte 192 + 100 = 292 within the right child of the root, and it is byte 421 + 292 = 713 within the object as a whole.

The leaf blocks in a large storage object contain pure data — no control information is required since the parent of a leaf contains the byte counts for each of its children. The size of a leaf block is a parameter of the data structure, and it is an integral number of contiguous disk pages. For often-updated objects, leaf blocks will probably be one page in length so as to minimize the amount of I/O and byte-shuffling that must be done on updates; for more static objects, leaf blocks may consist of several contiguous pages to lower the I/O cost of scanning long sequences of bytes within such objects. (Leaf block size will be settable on a per-file-object basis using the hint mechanism.) As in B+ trees, leaf blocks are allowed to vary from being 1/2-full to completely full.

Each internal node of a large storage object corresponds to one disk page, and contains between n_e and $2n_e + 1$ (count, pointer) pairs. We allow a maximum of $2n_e + 1$ pairs because our deletion algorithm works in a top-down manner, and the nature of its top-down operation requires that it be possible to merge a 1/2-full node of n_e entries and a node with $n_e + 1$ entries into a single full node (as we will see shortly). Finally, the root node corresponds to at most one disk page, or possibly just a portion of a shared page, and contains between 2 and $2n_e + 1$ (count, pointer) pairs.

Table 1 shows examples of the approximate object size ranges that can be supported by trees of height two and three assuming two different leaf block sizes. The table assumes 4K-byte disk pages, 4-byte pointers, and 4-byte counts, so the internal pages will have between 255 and 511 (count, pointer) pairs. It should be obvious from the table that two or three levels should suffice for most any large object.

Associated with the large storage object data structure are algorithms to *search* for a range of bytes, to *insert* a sequence of bytes at a given point in the object, to *append* a sequence of bytes to the end of the object, and to *delete* a sequence of bytes from a given point in the object. The insert, append, and delete operations

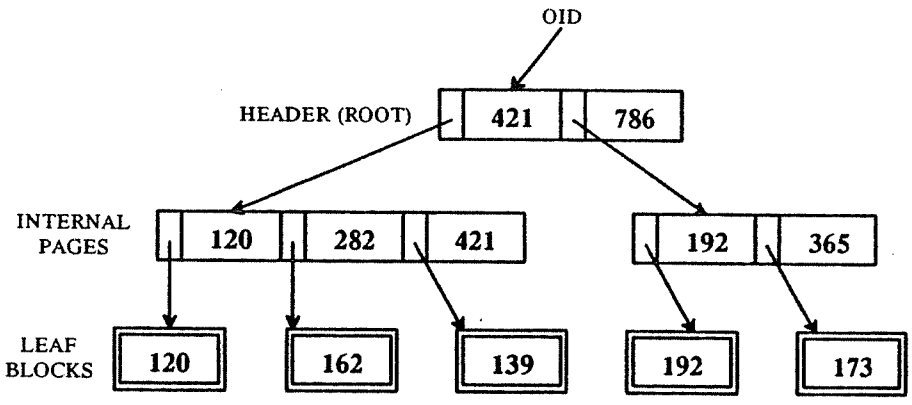

Figure 1: An example of a large storage object.

Number of Tree Levels	Leaf Block Size	Object Size Range
2	1	8KB - 2MB
	4	32KB - 8MB
3	1	2MB - 1GB
	4	8MB - 4GB

Table 1: Some examples of object sizes.

are quite different from those in the proposal of Stonebraker et al [Ston83], as the insertion or deletion of an arbitrary number of bytes into a large storage object poses some unique problems compared to inserting or deleting a single record from an ordered relation. Inserting or deleting one byte is the analogy in our case to the usual single record operations, and single byte operations would be far too inefficient for bulk inserts and deletes. As for the append operation, this is a special case of insert that we treat differently in order to achieve best-case storage utilizations for large objects that are constructed via successive appends. We consider each of these algorithms in turn.

4.1.1. Search

The search operation supports the retrieval of a sequence of N bytes starting at byte position S in a large storage object. (It can also be used to retrieve a sequence of bytes that are to be modified and rewritten, of course.) Referring to the (count, pointer) pairs using the notation $c[i]$ and $p[i]$, $1 \leq i \leq 2n_e+1$, and letting $c[0]=0$ by convention, the search algorithm can be described as follows:

(1) Let *start* = S, and read the root page and call it page P.

(2) While P is not a leaf page, do the following: Save P's address on the stack, and binary search P to find the smallest count $c[i]$ such that $start \leq c[i]$. Set $start = start - c[i-1]$, and read the page associated with $p[i]$ as the new page P.

(3) Once at a leaf, the first desired byte is on page P at location *start*.

(4) To obtain the rest of the N bytes, walk the tree using the stack of pointers maintained in (2).

Considering Figure 1 again, suppose we wish to find bytes 250-300. We set $start=250$, binary search the root, and find that $c[1]=421$ is the count that we want. We set $start = start - c[0]=250$ (since $c[0]=0$ by convention), and then we follow $p[1]$ to the left child of the root node. We binary search this node, and we find that $c[2]=282$ is the count that equals or exceeds *start*; thus, we set $start = start - c[1]=130$ and follow $p[2]$ to the leaf page with 162 bytes in it. Bytes 130-162 of this node and bytes 1-18 of its right neighbor (which is reachable by walking the stack) are the desired bytes.

4.1.2. Insert

The insert operation supports the insertion of a sequence of N bytes after the byte at position S. Since N can be arbitrarily large, an algorithm that efficiently handles bulk insertions is required; as mentioned before, the standard B-tree insertion algorithm only works for inserting a single byte, which would be too inefficient for large insertions. Our insert algorithm can be described as follows:

(1) Traverse the large 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.

(2) Call the leaf 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 (1).

(3) If an overflow does occur, allocate as many leaves as necessary to hold the overflow from L, and evenly distribute L's bytes and the bytes being inserted among L and the newly allocated leaves.

(4) Propagate the counts and pointers for the new leaves upward in the tree using the stack built in (1). If an internal node overflows, handle it in the same way that leaf overflows are handled.

This algorithm appears attractive because it touches the smallest possible number of internal pages and leaf blocks, thus minimizing the I/O cost for insertion. Unfortunately, experience with a prototype implementation of the large storage object data structure showed that this insert algorithm leads to relatively poor storage utilization (as we will see shortly). In order to improve the storage utilization, step (3) can be changed to:

(3') 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 B be the number of bytes per leaf block. If L and M together have a sufficient amount of free space to accommodate N *modulo* B bytes of data (i.e., the overflow that would remain after filling as many leaves with new data as possible), then evenly distribute the new data plus the old contents of L and M evenly between these two nodes and $\lfloor N/B \rfloor$ newly allocated nodes. Otherwise, proceed as in step (3) above.

The motivation for this modification is to avoid creating an additional node in cases where the overflow can instead be accommodated by a neighboring node. This is reasonable here because it is not necessary to access a neighboring leaf until it is known (from the parent) that redistribution of data between L and M will indeed succeed, whereas the neighbors would have to be read from disk before this could be known in the case of a standard B+ tree. Note that this modification does increase the I/O cost for insertion in cases where such redistribution is possible — instead of reading L and then writing back L and a new node created by splitting L (along with $\lfloor N/B \rfloor$ other new nodes), L and M are both read and written. However, as we will see, the I/O cost increase in this one case is probably worth it, as the modification leads to a significant improvement in storage utilization. Also, it might be argued that the additional cost for reading M is not the whole picture — by redistributing the data in this way, we avoid having the system go through the process of allocating an additional node from the free list to handle the overflow.

4.1.3. Append

The append operation supports the addition of N bytes to the end of a large object. Appending N bytes differs from inserting N bytes in the way in which data is redistributed among leaf pages when an overflow occurs. The append algorithm is as follows:

(1) Make a rightmost traversal of the large object tree. As the tree is being traversed, update the counts in the internal nodes to reflect the effect of the append. As always, save the search path on the stack.

(2) Call the rightmost leaf R. If R has enough free space to hold the new bytes, then append the bytes to R. The append operation is now complete in this case.

(3) Otherwise, call R's left neighbor (if it exists) L. Allocate as many leaves as necessary to hold L's bytes, R's bytes, plus the new bytes being appended to the object. Fill L, R, and the newly allocated leaves in such a way that all but the two rightmost leaves of the tree are completely full. Balance the remaining data between the two rightmost leaves, leaving each leaf at least 1/2-full. (If L has no free space, we can ignore L during this step.)

(4) Propagate the counts and pointers for the new leaves upward in the tree using the stack built in (1), and handle node overflow as in the insertion algorithm.

The key point of this algorithm is that it guarantees that a large object which is constructed via successive append operations will have maximal leaf utilization (i.e., all but the last two leaves will be completely full). This is particularly useful because it allows large objects to be created in steps, something which may be necessary if the object being created is extremely large. While this algorithm could be improved to yield higher internal node utilization by treating the internal nodes the same way that leaves are treated, we decided not to do this — it would increase the I/O cost of the algorithm, and internal node utilization is not as critical as leaf node utilization because of the large fanout of internal nodes.

4.1.4. Delete

The delete operation supports the deletion of N bytes starting at a specified byte position. In a B+ tree, the analogous problem would be that of range deletion, i.e., deleting all keys between some lower and upper bounds. Again, since the traditional B+ tree deletion algorithm removes only one record at a time, it would be unacceptably slow for large deletions. Instead, our bulk delete algorithm proceeds in two phases. In the first phase, it deletes the specified range of bytes, possibly leaving the tree in an unbalanced state. The second phase makes one pass down the tree structure to rebalance the tree.

Deletion of an arbitrary range of bytes from the leaves of a large object will, in general, imply the deletion of a number of entire subtrees, leaving a "raw edge" of damaged nodes. These nodes form the *cut-path* of the deletion. In general, the *left* and *right cut-paths* will start at the root, include some number of common nodes, and then split off and proceed down the tree to two different leaves. The node at which the left and right cut-paths diverge is called the *lowest common ancestor* or *lca* for the delete. Figure 2 illustrates the relationship between the deleted portion of the tree, the left and right cut-paths, and their *lca*. Note that if any of the nodes remaining in the tree have underflowed, they must necessarily occur along the cut-path. The rebalancing algorithm therefore traces the cut-path in a top-down fashion, attempting to "zipper up" the split in the tree.

In order to minimize the I/O cost of the deletion algorithm, we use a small data structure in memory, *path*, which describes the cut-path. The *path* data structure is built during the delete phase of the algorithm, and it stores the disk address of each cut-path node plus the number of children that it has (including nodes from both the left and right cut-paths). The information stored in *path* is sufficient to determine if a node is *in danger* of underflowing (as defined shortly). The rebalancing algorithm then examines *path* in a top-down fashion — for each *path* node, if it is in danger of underflowing, its corresponding tree node is merged or reshuffled with a neighboring node until it is safe.

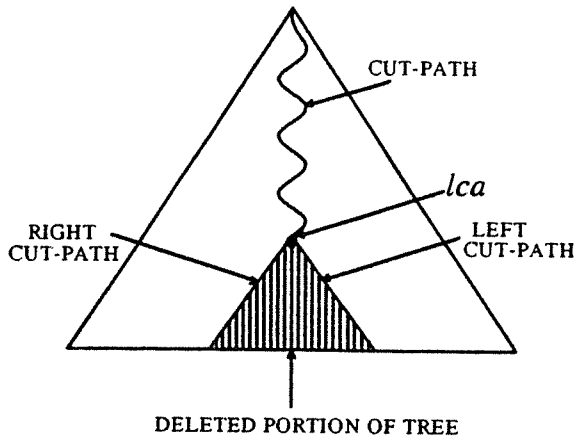

Figure 2: Terminology for deletion algorithm.

The notion of a node being in danger of underflowing (possibly without actually having underflowed) is what allows the algorithm to operate in one downward pass through the tree. A node is in this situation if it cannot afford to have a pair of its child nodes merged into a single child node, as this would cause the node itself to underflow. To prevent this possibility, all potential underflows are instead handled on the way down the tree by merging endangered nodes with neighboring nodes, or else by borrowing entries from neighboring nodes if such merging is impossible (i.e., if both neighbors have more than n_e entries). A node is said to have *underflowed* if either of the following conditions holds for the node:

(1) The node is a leaf and it is less than 1/2-full.

(2) The node is an internal node and it has fewer than n_e entries (or fewer than two entries if it is the root node).

We say that a node is *in danger* of underflowing if any of the following three conditions holds:

(1) The node has actually underflowed.

(2) The node is an internal node with exactly n_e entries (2 entries if it is the root), and one of its children along the cut path is in danger.

(3) The node is the *lca*, and it has exactly $n_e + 1$ entries (3 entries if it is the root), and both of its children along the cut path are in danger.

Given this background and our definitions of underflowed and endangered nodes, we can now describe each phase of the deletion algorithm as follows:

Deletion Phase:

(1) Traverse the object to the left and right limits of the deletion. All subtrees completely enclosed by the traversal are deleted, and the counts in all nodes along the cut-path are updated to show the results of the deletion. Also, for each node along the cut-path (as the tree is traversed), create a representative node in the main memory data structure *path* which records the address of the node and the number of children that it has left.

(2) Traverse the *path* data structure bottom-up, marking each node that is in danger (as defined above).

Rebalancing Phase:

(1) If the root is not in danger, go to step (2). If the root has only one child, make this child the new root and go to (1). Otherwise, merge/reshuffle[3] those children of the root that are in danger and go to (1).

(2) Go down to the next node along the cut-path. If no nodes remain, then the tree is now rebalanced.

(3) While the current node is in danger, merge/reshuffle it with a sibling. (For a given node along the cut-path, this will require either 0, 1, or 2 iterations of the while loop.)

(4) Go to (2).

One additional note is in order with regard to the I/O cost of the deletion phase of the algorithm — in this phase, only one leaf block ever has to be touched. Entirely deleted nodes can simply be handed back to the free space manager directly, as their addresses are available in their parent node(s); furthermore, deletion can be accomplished for the partially deleted leaf block on the left cut-path by simply decrementing the byte count in its parent node. Thus, only the partially deleted leaf block on the right cut-path needs to be read and written during the deletion phase.

4.1.5. Preliminary Performance Results

In order to verify that the algorithms indeed work as claimed, to investigate certain algorithm design decisions and tradeoffs, and to find out what sort of performance can be expected using the scheme, we implemented a prototype of our large object design in the C programming language. Each of the operations (search, insert, append, and delete) operates as though the prototype was a real implementation of the design, but these routines are then interfaced to a simulated CLOCK (LRU approximation) buffer manager that counts disk accesses (rather than really accessing pages on disk). Thus, our prototype implementation actually runs in main memory, simulating the I/O costs involved in the operations. Reads are counted by keeping track of the number of buffer pool misses, and writes are counted by noting the number of times pages are dirtied (for the first time) in the buffer pool. This subsection of the paper describes the results of the tests that we conducted using the prototype, including an investigation of the level of storage utilization provided by the scheme for two variants of the insert algorithm, a study of the tradeoff between search/scan costs and update costs as the leaf block size is varied, and a study of the average costs of the various tree operations.

In our study, we assumed a 4K-byte page size, and we experimented with both 1-page and 4-page leaf blocks. Our experiments consisted of using the append routine to construct an object of some initial size with approximately 100% storage utilization, and then running a mix of randomly generated searches, inserts, and deletes on the object. We experimented with object sizes of 10 megabytes and 100 megabytes, running a query mix consisting of 40% searches, 30% inserts, and 30% deletes. (Only the 10 megabyte results will be presented, as the 100 megabyte experiments produced similar trends except for minor buffer-related differences.) Equal percentages of inserts and deletes were used in order to ensure that the object size remained stable, and these update operations were uniformly distributed throughout the body of the object. (This

[3] The merge/reshuffle step decides whether nodes can be merged or whether bytes must be reshuffled with a neighbor, does it, and then updates *path* to maintain a consistent view of the cut-path.

uniform distribution assumption is a pessimistic assumption, as it produces worst-case average storage utilizations.) Two variants of the insert algorithm were tested, one that always splits a leaf block on overflow and another that tries to avoid a split by moving data to the less full of the leaf block's neighbors if doing so will help. Several mean operation sizes were tested, where the operation size is the number of bytes to be searched for, inserted, or deleted; mean sizes of of 100 bytes and 10K bytes were used. (We also ran experiments with 1-byte operations, but the results were basically the same as those of the 100-byte runs.) Operation sizes were drawn from a discrete uniform distribution varying plus or minus 50% from the mean. Pointer and count values were assumed to require four bytes each in computing the capacity of internal nodes. In our experiments, we assumed the availability of 12 buffer pages for buffering data for the operations. Finally, we used disk time estimates of 33 milliseconds for disk arm movement plus rotational latency (for random I/O) and 1 millisecond of transfer time for each 1K bytes transferred; thus, the cost to read a 4K byte page was taken to be 37 milliseconds, and the cost for a 16K byte block was 49 milliseconds.

The storage utilization results from our experiments are shown in Figures 3 and 4 for average operation sizes of 100 bytes and 10K bytes, respectively. The horizontal axis shows the number of operations executed from the mix, so the figures illustrate how storage utilization degrades from the initial near-100% figure as random insertions and deletions break up the initially full leaf blocks. In both figures, it is clear that the insertion algorithm that tries to avoid splitting provides significant storage utilization improvements — for example, in Figure 3, the basic insert algorithm gives utilizations in the high 60% range, whereas the improved insert algorithm provides utilizations in the low 80% range. It appears in Figure 3 that 1-page leaf blocks provide slightly better storage utilization for the small operations when insertion does not look at neighboring nodes; this is because 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). This difference disappears for the better insert algorithm, as data is redistributed to avoid splits when possible. In Figure 4, however, 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 10K bytes, or 2.5 pages of data. This data is distributed over as few newly allocated 1-page leaf blocks as possible (as well as one or two existing, partially-filled leaf blocks), leading to 3-4 nearly full leaf blocks. With 4-page leaf blocks, however, the average insert is sure to split a leaf block, creating two relatively empty blocks as a result. If the operation size were increased further, this difference would diminish, as much larger operations would create a number of nearly full leaves at either leaf size.

The I/O cost for search operations is presented in Figures 5 and 6. In Figure 5, the search cost is basically independent of the leaf block size and the insert algorithm, with the 1-page leaves having a tiny advantage over 4-page leaves since 100 byte searches can almost always be satisfied with data from a single disk page (i.e., without the additional transfer time for another 3K bytes of data). The average I/O cost in this case is in the neighborhood of two disk accesses, meaning that the root of the object is being successfully buffered, and the other two levels are each being read. (The tree height for a 10MB object turns out to be 3 — the root, one other internal level, and then the leaf blocks.) Turning to Figure 6, which shows the search

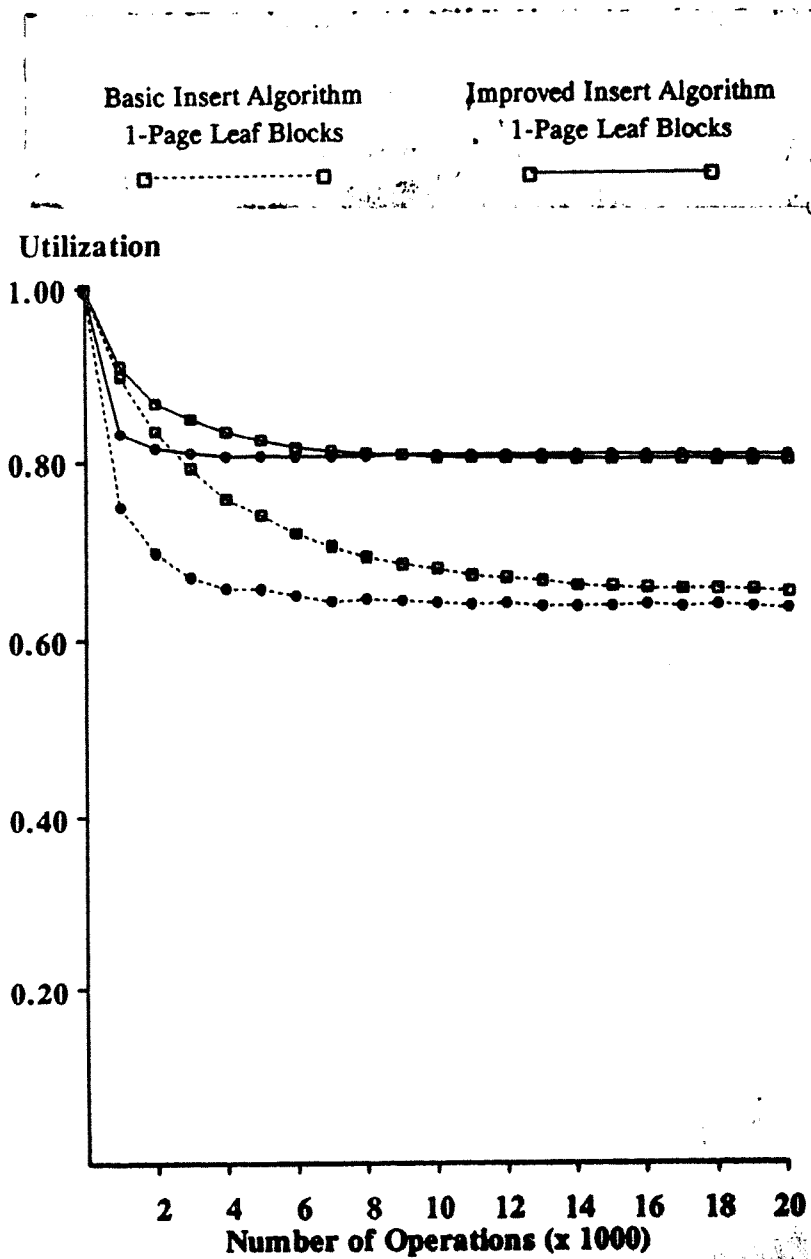

Figure 3: Storage utilization (100-byte operations).

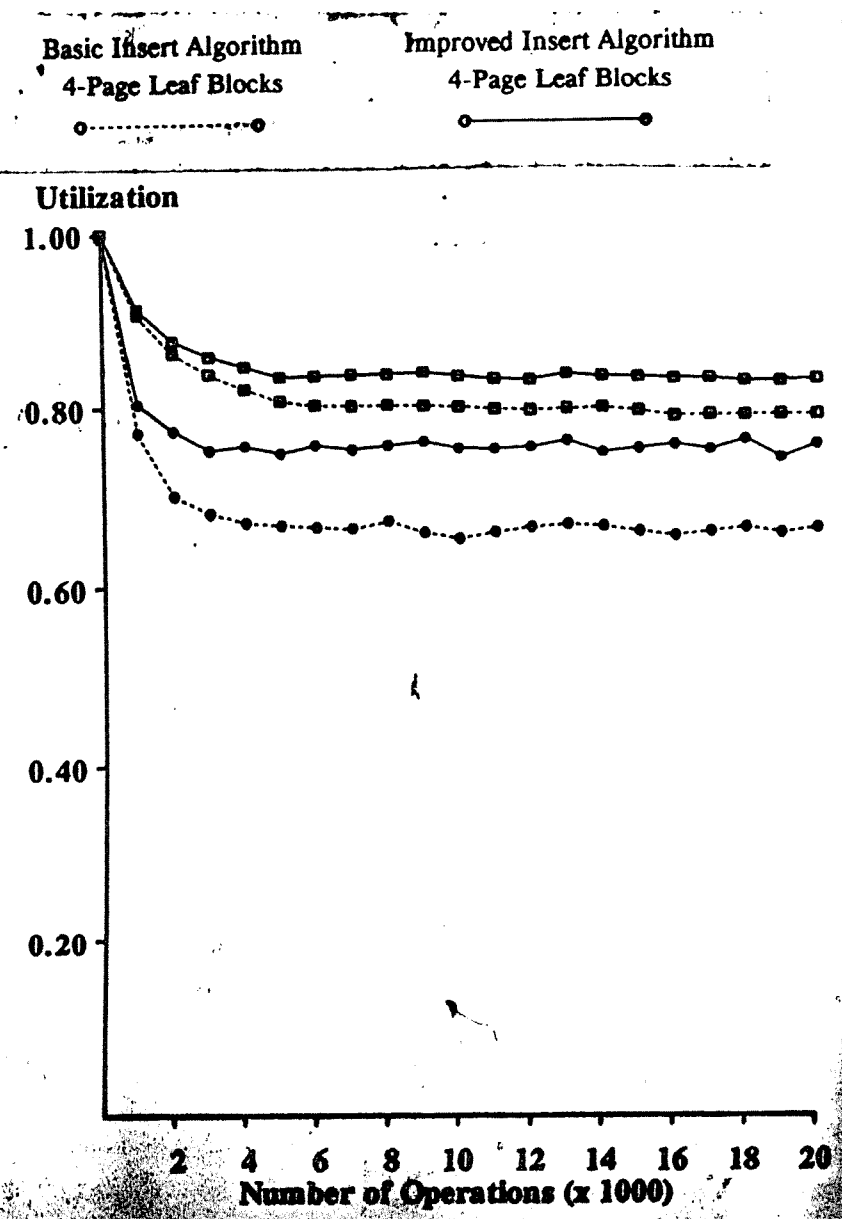

Figure 4: Storage utilization (10K-byte operations).

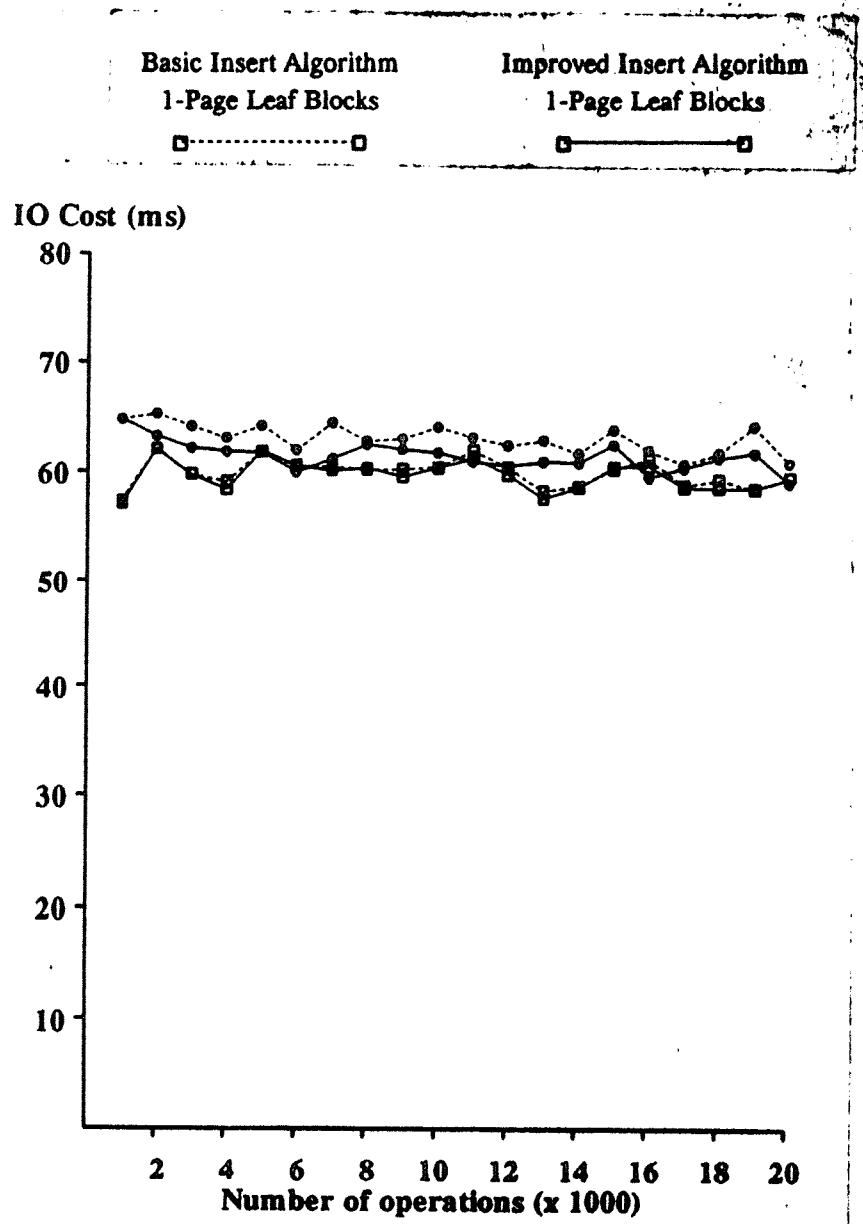

Figure 5: Search cost (100-byte operations).

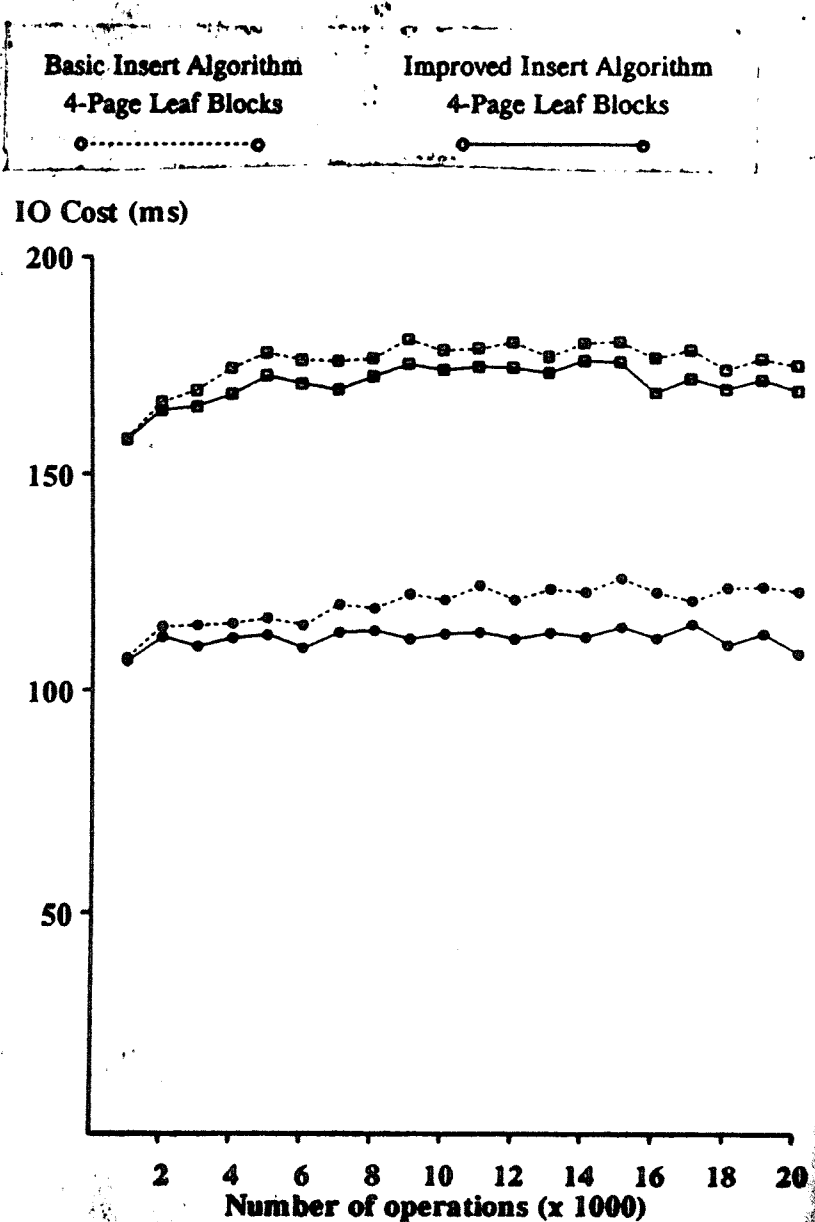

Figure 6: Search cost (10K-byte operations).

cost for 10K bytes of data, it is evident that the 4-page leaves have a definite advantage here. 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 2-4 sequential pages worth of data. Figure 6 also shows that the better insert algorithm has a slight advantage here — by providing better storage utilization, it leads to a slight decrease in the average number of leaf blocks that have to be read to obtain the desired data. Comparing Figures 5 and 6, of course, the cost of 10K byte retrieves is higher than the cost of 100 byte searches because of the extra leaf block I/O for reading the entire 10K bytes.

Figures 7 and 8 present the I/O cost results for insert operations. In both figures, the improved insert algorithm is seen to be somewhat more expensive than the basic algorithm. This is expected, as the improved algorithm sometimes reads and writes a neighboring leaf block rather than simply writing a new leaf block when doing so avoids the creation of a new leaf. (Recall that it prefers redistribution of data over leaf block splitting.) Figure 8 also shows that 4-page leaves have a slight insert performance advantage over 1-page leaves. This is due to the fact that, for large inserts, fewer leaves have to be touched on the average when the leaf block size is 4 pages. This decrease in the amount of random I/O more than offsets the increase in sequential I/O for insertion. This is not the case for small inserts, of course, as Figure 7 shows the basic insert algorithm with 1-page leaves as being the cheapest alternative (since it avoids the cost of reading the neighboring nodes, and also avoids the additional sequential read/write costs associated with 4-page leaves). Finally, the cost of inserting 10K bytes is significantly higher than the cost for 100 bytes, as several additional page I/O's are required to write out the data in the 10K case.

Figures 9 and 10 present the I/O cost results for delete operations. The improved insert algorithm leads to slightly lower deletion costs in both cases, which is due to a lesser probability of having to borrow data from a leaf page to fill in gaps in partially deleted leaves; this lesser probability comes from the increased storage utilization provided by the improved insert algorithm. The figures also show that the 1-page leaves have a cost advantage over the 4-page leaves for deletion; this is due to the additional sequential I/O cost associated with the 4-page leaves, both during the deletion and rebalancing phases. (Recall that the deletion phase never accesses more than a single leaf block, regardless of the amount of data being deleted.)

To summarize these results, the EXODUS large storage object mechanism has been shown to provide operations on very large dynamic objects at relatively low cost, and at a reasonable level of storage utilization (80% or higher). With respect to the appropriate choice of leaf block size, there are clearly tradeoffs — larger leaf blocks have a definite advantage for multi-page searches, but they also increase the cost somewhat for updates and lead to somewhat lower storage utilizations. We expect multi-page leaves to offer the greatest advantages for large, relatively static objects, where the storage utilization will be close to 100% (because such objects will be built via appends and not subjected to mixes of frequent and randomly distributed updates). Finally, we should mention that the results presented here are actually pessimistic in some ways. Our storage utilization results are pessimistic because, for more static objects, or objects where updates tend to be clustered in just a few regions of the object, storage utilizations in the 90-100% range would really be the norm. Also, our I/O cost results are a bit pessimistic because our prototype does not handle leaf blocks

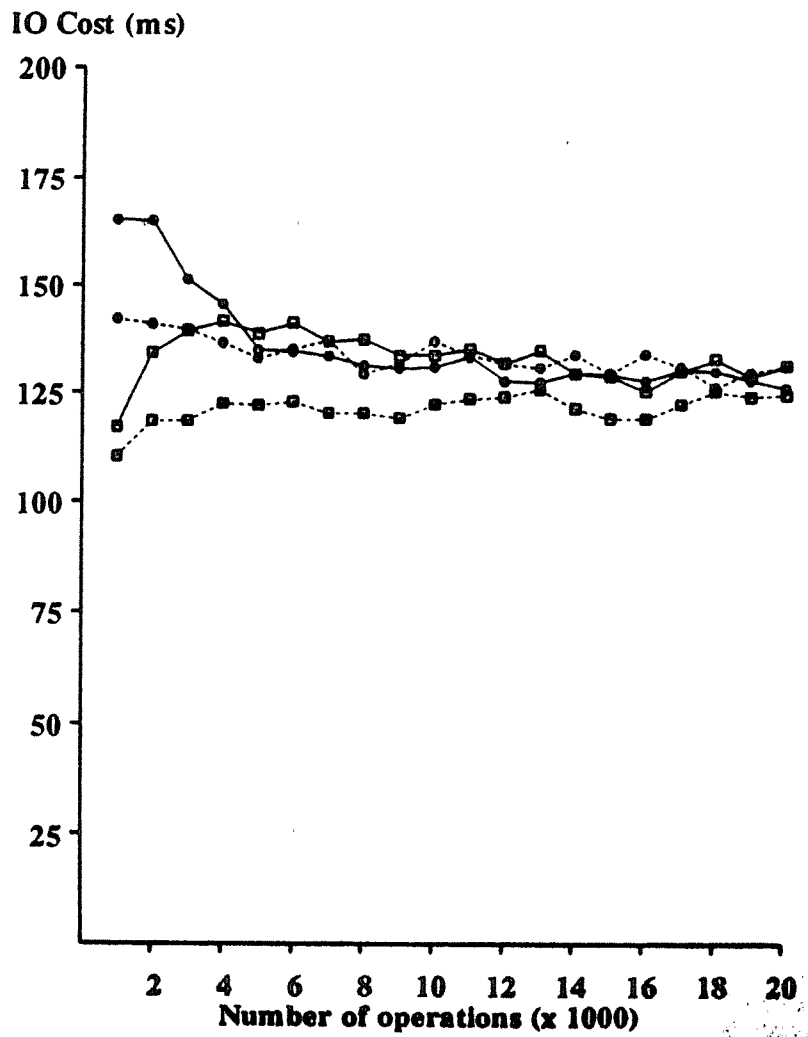

Figure 7: Insert cost (100-byte operations).

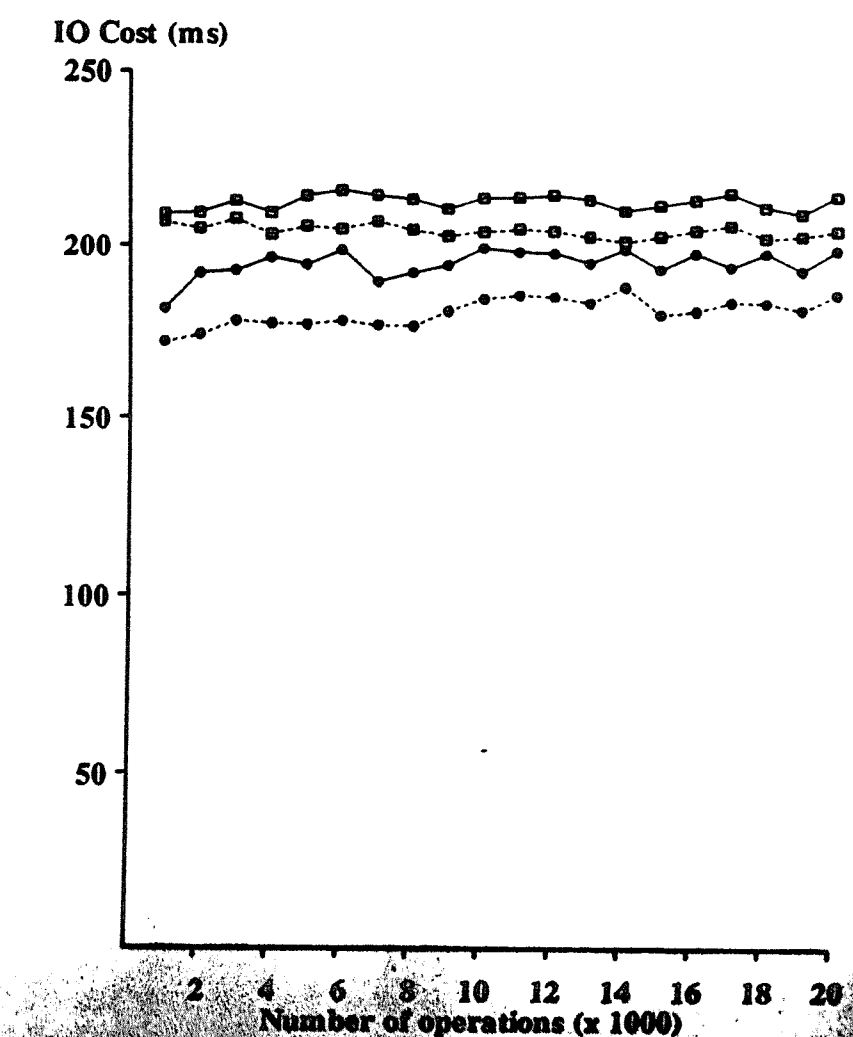

Figure 8: Insert cost (10K-byte operations).

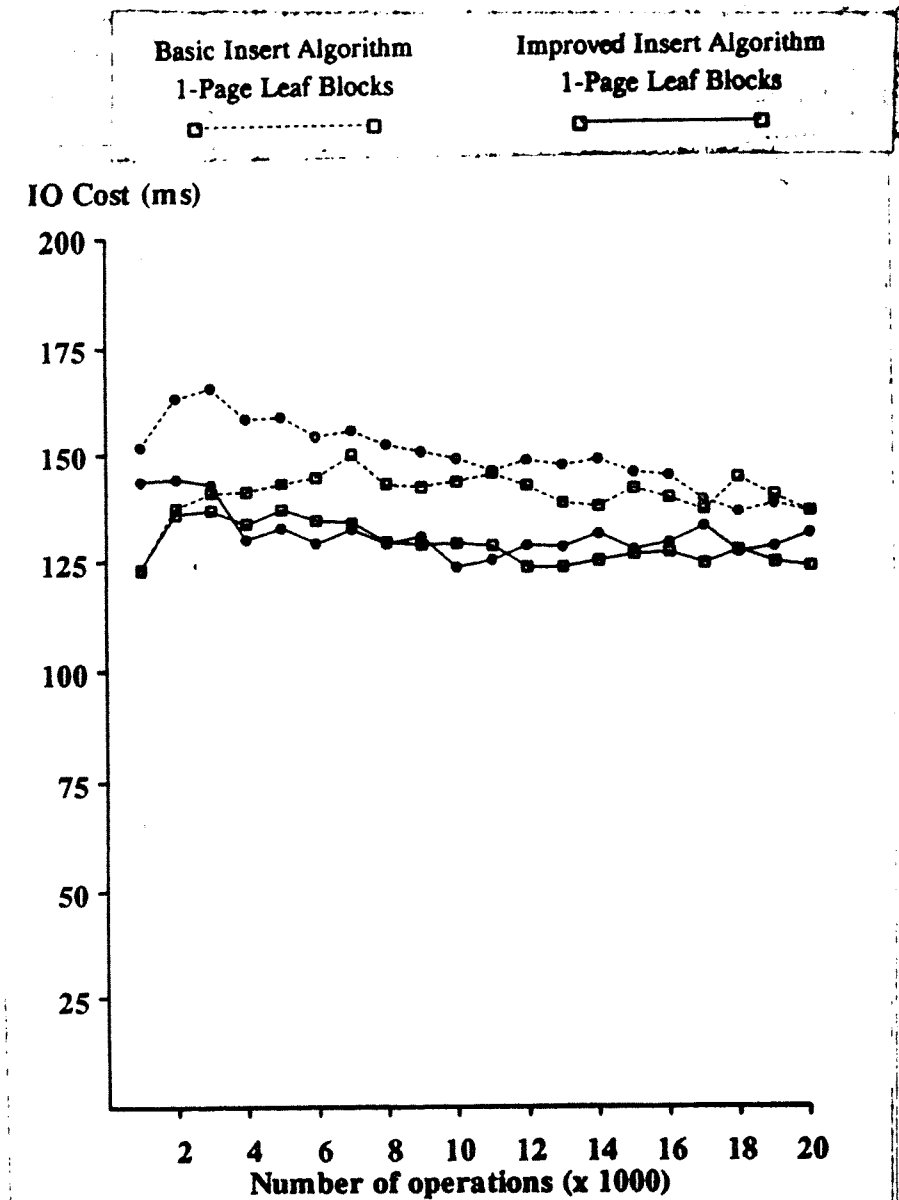

Figure 9: Delete cost (100-byte operations).

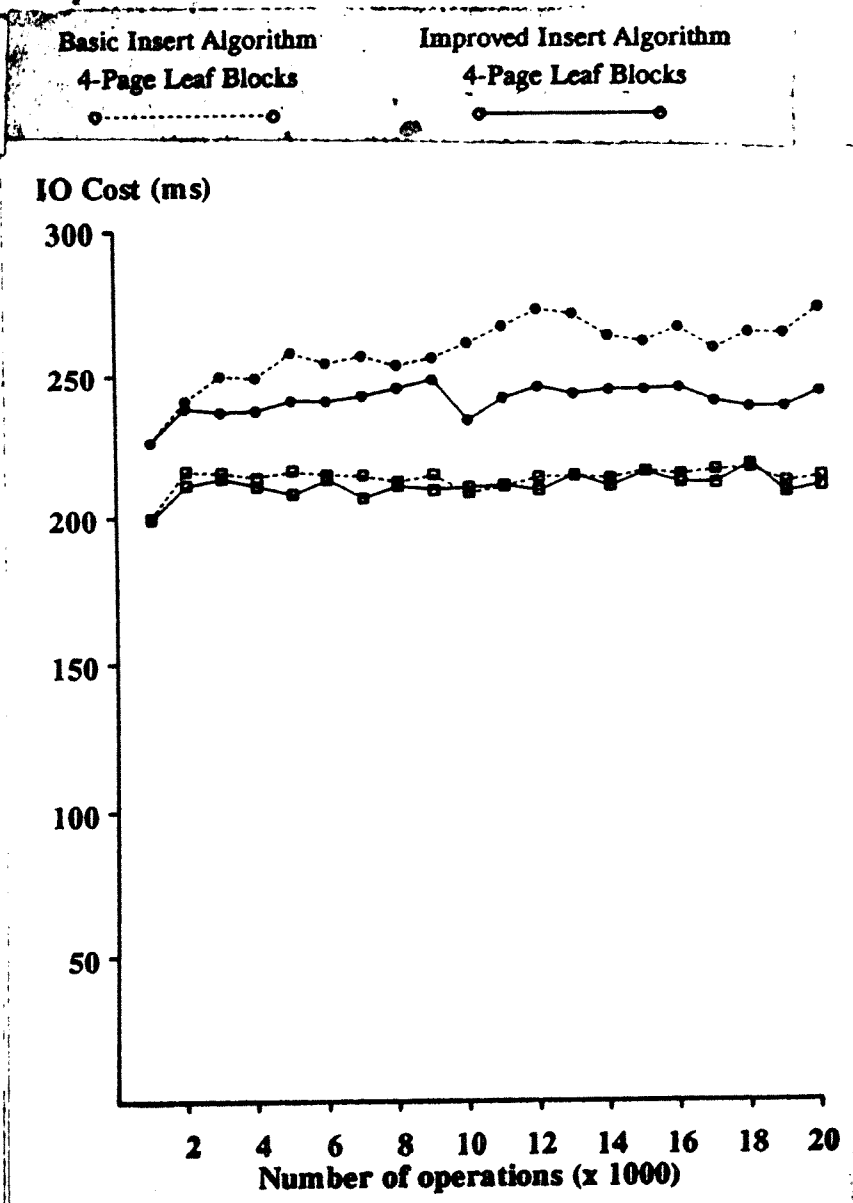

Figure 10: Delete cost (10K-byte operations).

as efficiently as it might — entire leaf blocks are read and written (rather than partial blocks) even when only the last page or two of a block is affected by an operation.

4.2. Versions of Storage Objects

As described earlier, the EXODUS storage system also provides support for versions of storage objects. The support provided is quite primitive — one version of each storage object is retained as the current version, and all of the preceding versions are simply marked (in their object headers) as being old versions. When a storage object is updated with the versioning option on, the old object header (or the entire object, in the case of a small storage object) is copied to a new location on disk as an old version of the object. The old version of the object header is then overwritten (in place) by the new version of the header. The OID of the old version is returned to the updater, and the OID of the new version is the OID that was originally passed to the update routine (since OID's are basically physical addresses). To ensure that the cost of copying the old version elsewhere is not as prohibitive as it might otherwise be [Care85b], the old version is placed on the same page of the file object as the new version, or else on a nearby page, if possible. (Note that we do not plan on using versions as our recovery mechanism, or this would be unreasonable.)

The reason for such a primitive level of version support is that different EXODUS applications may have widely different notions of how versions should be supported, as evidenced by the wide range of version-related proposals in the recent literature [Ston81, Dada84, Katz84, Bato85, Clif85, Klah85, Snod85, Katz86]. Therefore, we leave the maintenance of data structures such as graphs of the versions and alteratives of objects to a higher level of the system, a level that will undoubtedly vary from application to application (unlike the EXODUS storage system). The reason that we do not leave version management out of the EXODUS storage system altogether is one of efficiency — it would be prohibitively expensive, both in terms of storage space and I/O cost, to maintain versions of large objects by maintaining entire copies of objects.

Versions of large storage objects are maintained by copying and updating the pages that differ from version to version. Figure 11 illustrates this by an example. The figure shows two versions of the large storage

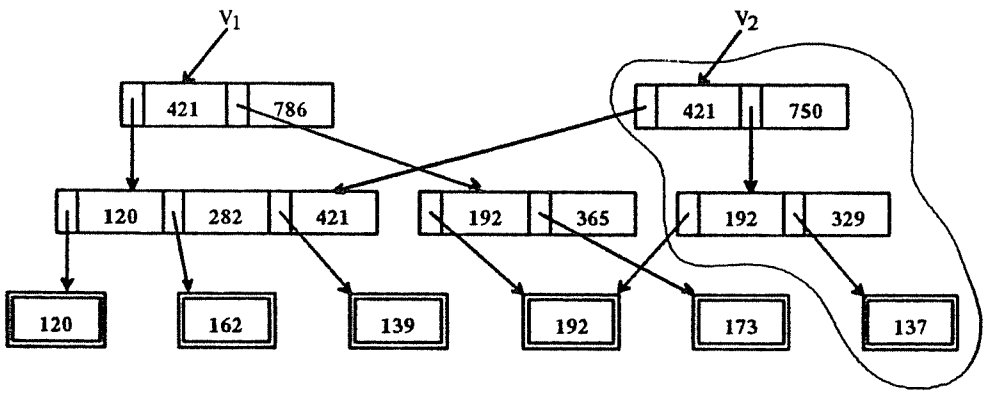

Figure 11: Two versions of a large storage object.

object of Figure 1, the original version, V_1, and a newer version, V_2. In this example, V_2 was created by deleting the last 36 bytes from V_1. Note that V_2 shares all nodes of V_1 that are unchanged, and it has its own copies of each modified node. A new version of a large storage object will always contain a new copy of the path from the root to the new leaf (or leaves); it may also contain copies of other internal nodes if the change affects a very large fraction of the object. Since the length of the path will usually be two or three, however, and the number of internal pages is small relative to the number of pages of actual data (due to high fanout for internal nodes), the overhead for versioning large objects in this scheme is small — for a given tree height, it is basically proportional to the difference between adjacent versions, and not to the size of the objects.

Besides allowing for the creation of new versions of large storage objects, which is supported by allowing the insert, append, delete, and write (i.e., read and modify a byte range) operations to be invoked with the versioning option turned on, the EXODUS storage system also supports the deletion of versions. Again, this is necessary from an efficiency standpoint; it is also necessary if the storage system is to successfully hide the physical representation of storage objects from its clients. The problem is that, when deleting a version of a large object, we must avoid discarding any of the object's pages that are shared (and thus needed) by other versions of the same object. In general, we will have a situation like the one pictured in Figure 12, where we wish to delete a version V which has a direct ancestor V_a (from which V was created) and descendents V_{d_1} through V_{d_n} (which were created from V).

A naive way to insure that no shared pages are discarded would be to traverse all other versions of V, marking each page as having been visited, and then traverse V, discarding each unmarked page. The problem with this approach is that there may be many versions of V, and consequently the number of pages visited could be quite large. One way to cut down on the number of pages visited is to observe that, if an ancestor of version V_a shares a page with a page with V, then V_a itself must also share that same page with V. Likewise, if a descendent of V_{d_1} shares a page with V, V_{d_1} itself must also share that page with V. Thus, it suffices to just visit the pages of the direct ancestor and the direct descendents of an object, i.e., of the

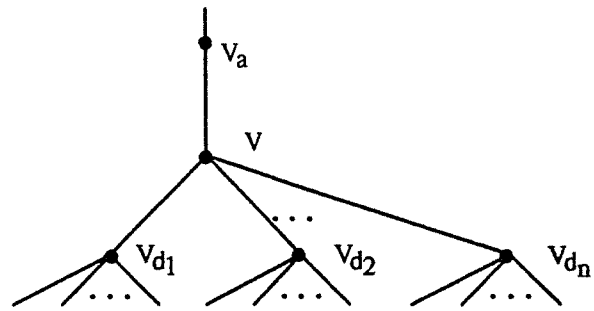

Figure 12: An example version history.

adjacent versions of an object (the version from which the object was directly created, or versions which were themselves directly created from the object).

We can further reduce the number of pages visited by observing two things. First, if a page is shared by two versions of a large object, then the entire subtree rooted at that page must be shared by the two versions. (An example is the leftmost child of the two version root pages in Figure 11.) Second, if a subtree is shared by two versions, then the root of that subtree must have the same height (i.e., distance above the leaf level) in both versions of the object. (Again, see Figure 11.) The first observation means that we only need to visit a shared subtree's root; there is no need to visit its descendent pages since they will necessarily be shared. The second observation means that if we scan versions of equal height level by level, then we will be able to detect the roots of shared subtrees as the level is scanned; further, for versions of unequal height, we need not check for shared pages until we get down to the appropriate level in the taller of the two versions.

Suppose for the moment that we wish to delete version V of an object, and that V has just one direct descendent, V_d. Further, suppose that V and V_d are the same height. Then, based on these two observations, the deletion algorithm can be described as follows:

(1) For each internal level l of V, do the following (working top-down from the root level):
   (a) Scan the index nodes at level l in V, tentatively marking all of the page numbers encountered in the nodes at this level for deletion. (Note that these page numbers are for pages at level $l+1$.)
   (b) Now scan level l in V_d. If a marked page number is encountered, unmark it and avoid scanning that page (and the subtree rooted at that page) in subsequent iterations.
   (c) Discard the pages from level $l+1$ of V that are still marked for deletion after step (b).

(2) Finish by discarding the root of V as well.

This algorithm is easily generalized to handle the case where the heights of versions V and V_d are unequal as well. If the height of V is greater, then we delay scanning V_d until we are scanning the level in V with the same height as the root of V_d; the case where the height of V_d is greater is handled similarly. It should also be clear that the algorithm can be generalized for the case where there are several versions adjacent to V (i.e., an ancestor and several descendent versions). In this latter case, step (b) must be performed for level l of each adjacent version, as a page of V cannot be discarded unless no adjacent version shares that page with V. As input, then, the version deletion operation takes the OID of the version to be deleted and the set of OID's of its adjacent versions; it deletes the specified version while leaving all of the pages that it shares with adjacent versions intact. As described earlier, we leave the problem of maintaining information about adjacent versions, like those in the example of Figure 12, to a higher level of the system.

A reasonable implementation of this algorithm would use a breadth-first search to scan the objects and a main-memory hash table to store the page numbers of the marked pages. Note that it is *never* necessary to actually read any leaf pages from the disk with this algorithm — in the worst case, where there is no sharing of pages between versions, the algorithm will simply end up visiting every non-leaf page of every version, which is much better than also visiting the leaves. (Leaf blocks comprise the vast majority of each version — with internal node fanouts of several hundred, non-leaf pages will represent less than 1% of the large object

storage requirements). In typical cases, however, the algorithm will visit relatively few pages, as adjacent versions are likely to share the bulk of their pages.

4.3. Concurrency Control and Recovery

The EXODUS storage system will provide both concurrency control and recovery services for storage objects. Two-phase locking [Gray79] of byte ranges within storage objects will be used for concurrency control, with a "lock the entire object" option being provided for cases where object (OID) level locking will suffice. For small storage objects, object level locking will probably be the norm. For large storage objects, however, byte range locking may be useful in some applications: For updates that change the contents of a range of bytes without changing the size of that range (i.e., updates that read and then rewrite a byte range), searches and updates in disjoint regions of the object will still be able to proceed. Updates that insert, append, or delete bytes will lock the byte range from where the operation begins to the end of the object, as the offsets of the remaining bytes cannot be known until the updater either commits or aborts. To ensure the integrity of the internal pages of large storage objects while insert, append, and delete operations are operating on them (e.g., changing their counts and pointers), non-two-phase B+ tree locking protocols [Baye77] will be employed. Searches and byte range updates will descend the tree structure by chaining their way down with read locks, read-locking a node at level $i+1$ and then immediately releasing the level i read-lock, holding only byte range read or write locks in a two-phase manner. Since inserts, appends, and deletes will normally affect an entire root-to-leaf path[4], the root and internal pages along the path for this type of update will be write-locked for the duration of the operation (e.g., the insert, delete, or append); again, though, only byte range write locks will be held in a two-phase manner once the operation has completed.

For recovery, small storage objects will be handled using before/after-image logging and in-place updating at the object level [Gray79]. Recovery for large storage objects will be handled using a combination of shadows and logging — updated internal pages and leaf blocks will be shadowed up to the root level, with updates being installed atomically by overwriting the old object header with the new header [Verh78]. Prior to the installation of the update at the root level, the other updated pages will be forced to disk; also, the name and parameters of the operation that caused the update will be logged, with the log sequence number (ala [Gray79]) of the log record for the update being placed on the root page of the object. This will ensure that operations on large storage objects can be undone (by performing the inverse operation) or redone (by re-performing the operation) as necessary in an idempotent manner. For versioned objects, the same recovery scheme will be used. In this case, however, the before-image of the updated large object header (or the entire small object) will first be copied elsewhere to be maintained as the version from before the updates.

---

[4] Recall that inserting, appending, or deleting bytes will cause counts to change all the way up to the root, unlike a record insertion or deletion in a standard B+ tree.

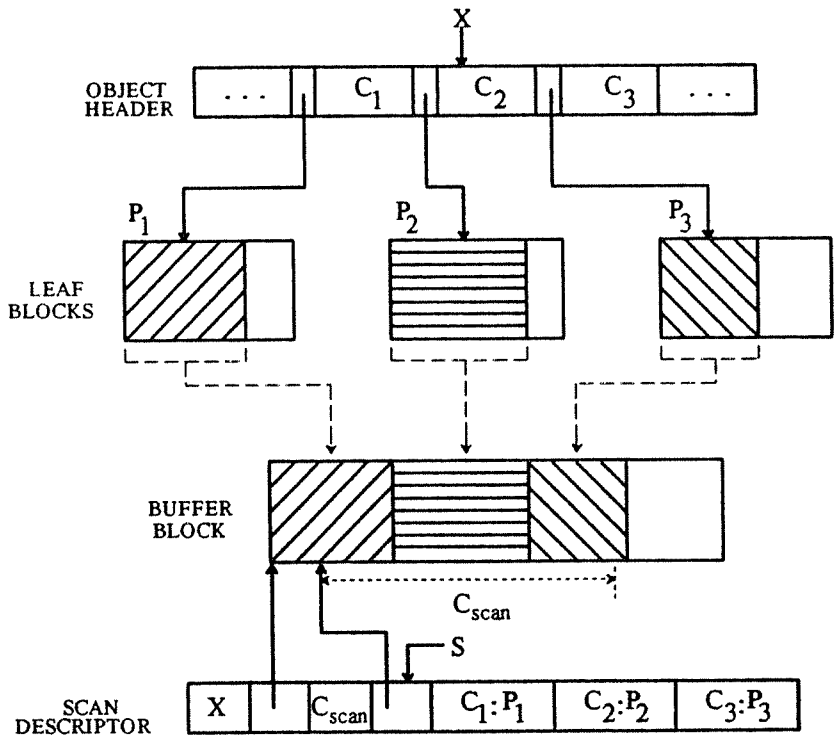

Figure 13: Contiguous buffering in EXODUS.

4.4. Buffer Management for Storage Objects

As described in the introduction, one objective of the EXODUS storage system design was to minimize the amount of copying from buffer space that is required. A second (related) objective is to allow sizable portions of large storage objects to be scanned directly in the buffer pool if this is desired by higher levels of software. To accommodate these needs, we plan to allocate buffer space in variable-length *buffer blocks*, which are integral numbers of pages, rather than only in single-page units. This will simplify things for higher-level software, making it possible to read and then scan a multi-page sequence of bytes without concern for page boundaries.

Figure 13 sketches the key aspects of the EXODUS buffering scheme for large storage objects. When an EXODUS client requests that a sequence of C_{scan} bytes be read from object X, the non-empty portions of the leaf blocks of X containing the desired range of bytes (leaf blocks P_1, P_2, and P_3 in Figure 13) will be read into one contiguous buffer block. This can be accomplished by obtaining a buffer block of the appropriate size from the buffer space manager, and then reading P_1, P_2, and lastly P_3 into the block — in that order, and so that P_2 begins right after the end of the non-empty portion of P_1, with P_3 being placed similarly. (While this constrains the order in which the leaf blocks of a large object can be read into the buffer pool, we do not expect this to be a serious limitation.) A scan descriptor will be maintained for the current region of X being scanned, including such information as the OID of X, a pointer to its buffer block, the

length of the actual portion of the buffer block containing the bytes requested by the client, a pointer to the first such byte, and information about where the contents of the buffer block came from (for replacement purposes). The client will receive an indirect pointer[5] to the buffer contents (*S* in Figure 13) through which the buffer contents may be accessed. Free space for the buffer pool will be managed using one of the standard dynamic storage allocation techniques (with the smallest unit of allocation being one disk page.) Finally, buffer space allocation and replacement policy selection will be guided by the hint mechanism mentioned in Section 3.

5. FILE OBJECTS

File objects in the EXODUS storage system are collections of storage objects (sets of storage objects, roughly speaking). File objects are useful for grouping objects together for several purposes. First, the EXODUS storage system provides a mechanism for sequencing through all of the objects in a file, so related objects can be placed in a common file for sequential scanning purposes. Second, objects within a given file are placed on disk pages allocated to the file, so file objects provide support for objects that need to be co-located on disk.

5.1. File Representation

The representation of file objects in EXODUS is similar in some respects to the representation of large storage objects. A file object is identified by its OID, and the OID for a file object is a pointer to the root page (i.e., the header) of the file object. (Storage objects and file objects are distinguished by a bit in their object headers.) Like large storage objects, file objects are represented by an index structure similar to a B+ tree, but the key for the index is different in this case — a file object index uses *disk page number* as its key. Each leaf page of the file object index contains a collection of page numbers for the slotted pages contained in the file. (The actual slotted pages themselves are managed separately, using standard techniques for page allocation and free space management.) The file object index thus serves as a mechanism to gather the pages of a file together, but it also has several other nice properties — it facilitates the scanning of all of the objects within a given file object *in physical order* for efficiency, and it also allows fast deletion of an object with a given OID from a file object (as we will see momentarily). We considered several other file designs, including the possibility of representing files as large storage objects containing a sequence of OID's, but none supported fast object deletion as well as this scheme does. Note that since all of the objects in a file are directly accessible via their OIDs, file objects are *not* comparable to surrogate indices — indices on the objects within a given file object will contain entries that point directly to the objects being indexed.

Creation of a file object allocates the file object header. Later, when objects are to be created within the file object, the object creation routine will be called with an optional hint of the form "place the new object near *X*", where *X* is the OID of an existing object within the file. If this hint is present, the new object will

[5] We are still debating whether this should be a direct pointer or an indirect pointer, so this aspect of the design may change slightly. A level of indirection allows buffer blocks to be relocated (in a critical section) for storage management purposes without interfering with client accesses.

be inserted on the same page as X if possible. (Recall that X's OID identifies the desired page.) If there is not enough space on X's page, then a new page near X's page on the disk will be allocated for the newly inserted object and its page number will be inserted into the file object B+ tree; the OID of the file object will be recorded on the newly allocated page. If no hint is present, X will be appended to the file (i.e., placed on the last page listed in the file object index, with overflows handled in the manner just described). Object deletion will lead to the removal of the object from the page where it resides. If this page becomes empty as a result of deleting the object, its page number will be deleted from the file object index and the page itself will be returned to the free space manager. Lastly, file deletion will lead to the deletion of all of the objects residing on pages listed in the file object index, the return of those pages to the free list, and then the removal of the index itself. If a file page contains one or more large object headers or file object headers, then these will of course have to be recursively deleted; otherwise, the page can be freed immediately.

5.2. Other File Object Issues

Concurrency control and recovery for file objects will be handled via mechanisms similar to those used for large storage objects. Concurrency control (for page number insertions and deletions) will be provided using B+ tree locking protocols. Recovery will be accomplished by shadowing changes up to the highest affected level of the file object index, logging the before- and after-images of the highest affected node, and then finally overwriting this node to atomically install the update. It is important to note that these concurrency control and recovery protocols will only be exercised when the file index is modified via the insertion or deletion of leaf page entries, as the storage object concurrency control and recovery protocols will handle slotted page changes that do not cause the file object index to be modified. Also, file object index changes can be deferred until commit time, allowing them to be grouped for efficiency.

The final non-trivial issue related to file objects is the question of how one can sort a file object. Since schema information has been carefully kept out of the EXODUS storage system, the storage system does not have sufficient information to do this on its own — it has no idea what fields the storage objects in a given file object have, nor does it know what the data types for the fields are. Since sorting is likely to be important for performance in some applications, and we do not wish the applications to be aware of the way that storage objects are laid out on disk, the EXODUS storage system will provide a generic file object sorting routine. One of the arguments to this sort routine will be a procedure parameter for an object comparison routine; the sort utility will call this routine to compare storage objects as it sorts the file. Sorting will necessarily move objects from page to page, so their OID's will be invalidated when sorting is performed. (This is really the only way in which OID's differ from surrogates, as other storage system operations preserve the integrity of OID's by leaving a forwarding address at an object's original location when the object must be relocated.)

6. CONCLUSIONS

In this paper we described the design of the object-oriented storage component of EXODUS, an extensible database management system under development at the University of Wisconsin. The basic abstraction in the EXODUS storage system is the storage object, an uninterpreted variable-length record of arbitrary

size. File objects are used to group together and sequence through collections of storage objects. The data structure and algorithms used to support large storage objects were described, and performance results from a preliminary prototype of the EXODUS large object management scheme were presented. It was shown that the proposed scheme indeed provides efficient support for large dynamic storage objects, both in terms of their storage utilization and performance. A scheme was described for maintaining versions of large objects by sharing common pages between versions, and an efficient version deletion algorithm was then presented. Also described in the paper, albeit briefly, were our design for file objects and our approaches to the problems of buffer management, concurrency control, and recovery. We are now working on a detailed design document for the EXODUS storage system based on the algorithms and data structures described here, and we expect implementation to commence in several months.

ACKNOWLEDGEMENTS

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the Department of Energy under contract #DE-AC02-81ER10920, by the National Science Foundation under grants MCS82-01870 and DCR-8402818, and by an IBM Faculty Development Award. Also, the authors wish to acknowledge the contributions of the other members of the EXODUS project, including Dan Frank, Goetz Graefe, and M. Muralikrishna.

REFERENCES

[Astr76] Astrahan, M., et. al., "System R: Relational Approach to Database Management", *ACM Transactions on Data Systems* 1, 2, June 1976.

[Bato84] Batory, D., and A. Buchmann, "Molecular Objects, Abstract Data Types, and Data Models: A Framework", *Proceedings of the 1984 VLDB Conference*, Singapore, August 1984.

[Bato85] Batory, D., and W. Kim, *Support for Versions of VLSI CAD Objects*, M.C.C. Working Paper, March 1985.

[Baye77] Bayer, R., and Schkolnick, M., "Concurrency of Operations on B-trees", *Acta Informatica* 9, 1977.

[Beec83] Beech, D., *Introducing the Integrated Data Model*, Hewlett-Packard Computer Science Laboratory Technical Note CSL-15, January 1983.

[Care85a] Carey, M. and D. DeWitt, "Extensible Database Systems", *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, Febuary 1985.

[Care85b] Carey, M., and W. Muhanna, "The Performance of Multiversion Concurrency Control Algorithms", *ACM Transactions on Computer Systems*, to appear.

[Chou85a] Chou, H-T., and D. DeWitt, "An Evaluation of Buffer Management Strategies for Relational Database Systems", *Proceedings of the 1985 VLDB Conference*, Stockholm, Sweden, August 1985.

[Chou85b] Chou, H-T., D. DeWitt, R. Katz, and A. Klug, "Design and Implementation of the Wisconsin Storage System", *Software Practice and Experience* 15, 10, October 1985.

[Clif85] Clifford, J., and A. Tansel, "On An Algebra for Historical Relational Databases: Two Views", *Proceedings of the 1985 SIGMOD Conference*, Austin, Texas, May 1985.

[Cope84] Copeland, G. and D. Maier, "Making Smalltalk a Database System", *Proceedings of the 1984 SIGMOD Conference*, Boston, MA, May 1984.

[Dada84] Dadam, P., V. Lum, and H-D. Werner, "Integration of Time Versions into a Relational Database System", *Proceedings of the 1984 VLDB Conference*, Singapore, August 1984.

[Daya85] Dayal, U. and J. Smith, "PROBE: A Knowledge-Oriented Database Management System", *Proceedings of the Islamorada Workshop on Large Scale Knowledge Base and Reasoning Systems*, February 1985.

[Gray79] Gray, J., "Notes On Database Operating Systems", in *Operating Systems: An Advanced Course*, R. Bayer, R. Graham, and G. Seegmuller, eds., Springer-Verlag, 1979.

[Gutt84] Guttman, T., "R-Trees: A Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 SIGMOD Conference*, Boston, MA, May 1984.

[Kaeh83] Kaehler, T. and G. Krasner, "LOOM — Large Object-Oriented Memory for Smalltalk-80 Systems", in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, ed. Addison-Wesley, 1983.

[Katz84] Katz, R. and T. Lehman, "Database Support for Versions and Alternatives of Large Design Files", *IEEE Transactions on Software Engineering* SE-10, 2, March 1984.

[Katz86] Katz, R., E. Chang, and R. Bhateja, "Version Modeling Concepts for Computer-Aided Design Databases", *Proceedings of the 1986 SIGMOD Conference*, Washington, DC, May 1986, to appear.

[Klah85] Klahold, P., G. Schlageter, R. Unland, and W. Wilkes, "A Transaction Model Supporting Complex Applications in Integrated Information Systems", *Proceedings of the 1985 SIGMOD Conference*, Austin, TX, May 1985.

[Lyng84] Lyngbaek, P. and D. McLeod, "Object Management in Distributed Information Systems", *ACM Transactions on Office Information Systems* 2, 2, April 1984.

[Lyng86] Lyngbaek, P. and W. Kent, "A Data Modeling Methodology for the Design and Implementation of Information Systems", *Proceedings of the 1986 SIGMOD Conference*, Washington, DC, May 1986, to appear.

[Poll81] Pollack, F., K. Kahn, and R. Wilkinson, "The iMAX-432 Object Filing System", *Proceedings of the 8th Symposium on Operating Systems Principles*, Pacific Grove, CA, December 1981.

[Reed83] Reed, D., "Implementing Atomic Actions on Decentralized Data", *ACM Transactions on Computer Systems* 1, 1, March 1983.

[Ship81] Shipman, D., "The Functional Data Model and the Data Language DAPLEX", *ACM Transactions on Database Systems* 6, 1, March 1981.

[Snod85] Snodgrass, R., and I. Ahn, "A Taxonomy of Time in Databases", *Proceedings of the 1985 SIGMOD Conference*, Austin, TX, May 1985.

[Ston76] Stonebraker, M., G. Wong, P. Kreps, and G. Held, "The Design and Implementation of INGRES", *ACM Transactions on Database Systems* 1, 3, September 1976.

[Ston81] Stonebraker, M., "Hypothetical Data Bases as Views", *Proceedings of the 1981 SIGMOD Conference*, Boston, MA, May 1981.

[Ston83] Stonebraker, M., H. Stettner, N. Lynn, J. Kalash, and A. Guttman, "Document Processing in a Relational Database System", *ACM Transactions on Office Information Systems* 1, 2, April 1983.

[Ston86] Stonebraker, M., and L. Rowe, "The Design of POSTGRES", *Proceedings of the 1986 SIGMOD Conference*, Washington, DC, May 1986, to appear.

[Verh78] Verhofstad, J., "Recovery Techniques for Database Systems", *ACM Computing Surveys* 10, 2, June 1978.