Chapter 14

Storage Management for Objects in EXODUS

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

Introduction

In the 1970's, the relational data model was the focus of much of the research in the database area. At this point, relational database technology is well-understood, a number of relational systems are commercially available, and they support the majority of business applications relatively well. One of the foremost database problems of the 1980's is how to support classes of applications that are not well-served by relational systems. For example, computer-aided design systems, scientific and statistical applications, image and voice applications, and large, data-intensive AI applications all place demands on database systems that exceed the capabilities of relational systems. Such application classes differ from business applications in a variety of ways, including their data modeling needs, the types of operations of interest, and the storage structures and access methods required for their operations to be efficient.

A number of database research efforts have recently begun to address the problem of building database systems to accommodate a wide range of potential applications via some form of extensibility. Such projects include EXODUS at the University of Wisconsin [Carey and DeWitt 1985, Carey, et al. 1986b, Carey and DeWitt 1987], PROBE at CCA [Dayal and Smith 1985, Manola and Dayal 1986], POSTGRES at Berkeley [Stonebraker and Rowe 1986, Rowe and Stonebraker 1987], STARBURST at IBM Almaden [Schwarz, et al. 1986, Lindsay, et al. 1987], and GENESIS at UT-Austin [Batory, et al. 1986]. 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. STARBURST, POSTGRES, and PROBE are complete database systems, each with a (different) well-defined data model and query language. Each system aims to provide the capability for users to add extensions such as new abstract data types and access methods within the framework provided by their data model. STARBURST is based on the relational model; POSTGRES extends the relational model with the notion of a procedure data type, triggers and inferencing capabilities, and a type hierarchy; PROBE is based on an extension of the DAPLEX functional data model, and includes support for spatial data and a class of recursive queries.

The EXODUS project is distinguished from all but GENESIS by virtue of being a "database generator" effort as opposed to an attempt to build a single (although extensible) DBMS for use by all applications. The EXODUS and GENESIS efforts differ significantly in philosophy and in the technical details of their approaches to DBMS software generation. GENESIS has a stricter framework (being based on a "building block" plus "pluggable module" approach), whereas EXODUS has certain powerful fixed components plus a collection of tools for a database implementor (DBI) to use in building the desired system based around these components.

The EXODUS group is addressing the challenges posed by emerging database applications by developing facilities to enable the rapid implementation of high-performance, application-specific database systems. EXODUS provides a set of kernel facilities for use across all applications, such as a versatile storage manager (the focus of this chapter) and a general-purpose manager for type-related dependency information. In addition, EXODUS provides a set of tools to help the DBI to develop new database system software. The implementation of some DBMS components is supported by tools which actually generate the components from specifications; for example, tools are provided to generate a query optimizer from a rule-based description of a data model, its operators, and their implementations [Graefe and DeWitt 1987]. Other components, such as new abstract data types, access methods, and database operations, must be explicitly coded by the DBI due to their more widely varying and highly algorithmic nature.[1] EXODUS attempts to simplify this aspect of the DBI's job by providing a programming language with constructs designed specifically for use in writing code for the components of a DBMS [Richardson and Carey 1987]. Finally, we are currently developing a data model and an associated query language to serve as starting points for subsequent EXODUS DBMS implementation efforts.

In this chapter we describe the object and file management facilities of EXODUS. The initial design of the EXODUS storage manager was outlined in [Carey, et al. 1986a]; this is an updated description, reflecting changes in the design that occurred as we developed the (now operational) first version of the system. The chapter is broken down as follows: The next section describes related work on "next genera-

[1] Actually, EXODUS will provide a library of generally useful components, such as widely-applicable access methods including B+ trees and some form of dynamic hashing, but the DBI must implement components that are not available in the library.

tion" storage systems. In Section 2, an overview of the EXODUS storage system is presented and the key characteristics of the EXODUS storage manager are described. Section 3 describes the interface that is provided by the EXODUS storage manager to the higher levels of EXODUS. In Section 4, which makes up the majority of the chapter, we present a detailed description of our data structures for object storage and summarize results from an early 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, the last section summarizes the contents of this chapter.

1. Related Work

There have been a number of other projects to construct file and object management services related to those provided by EXODUS. In [Kaehler and Krasner 1983], LOOM, a Large Object-Oriented Memory for Smalltalk-80, is described. LOOM extended the object storage capabilities of Smalltalk-80 to allow the manipulation of up to 2^{31} objects instead of 2^{15} objects. Problems associated with large objects were not addressed, and the system provided no facilities for sharing (e.g., shared buffers, concurrency control or recovery).

Another related system is the file system for the iMAX-432 [Pollack, et al. 1981]. 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 [Reed 1983]. The design was based on the premise that most objects are small (less than 500 bytes); special consideration was given to clustering related objects together and to garbage-collecting deleted objects to minimize wasted file space.

The GemStone database system [Maier, et al. 1986] is an object-oriented DBMS based on a Smalltalk-like data model and interface. In terms of storage management, GemStone objects are decomposed into collections of small elements (ala Smalltalk objects); the system's object manager is responsible for clustering related elements together on disk via segments. The Gemstone architects have investigated indexing issues that arise in an object-oriented DBMS environment. Large objects and collections of objects are represented on disk via mechanisms similar to those employed in EXODUS (and described in this chapter).

The Darmstadt Database Kernel System [Depp86, Paul, et al. 1987] is also related to our work, having been motivated by similar application-related considerations. However, the emphasis of their kernel architecture is on support for complex records (i.e., nested relations), and in particular on clustering their components and providing an efficient means to relocate entire complex objects (e.g., from a host to a workstation). Components of complex objects are viewed as records with bytestring attributes. Objects are passed between the kernel and its clients by copying data between a page-oriented kernel buffer and a client-provided object buffer.

The storage system of POSTGRES [Stonebraker and Rowe 1986, Stonebraker 1987] 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. Instead, new versions of modified tuples are 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.

Finally, the object server developed at Brown University [Skarra, et al. 1986] is also relevant. The object server provides a notion of objects and files similar to that of EXODUS. However, the main focus of their work has been on issues arising from a workstation/server environment; for example, they provide a set of "notify" lock modes to support efficient object sharing in such an environment.

2. Overview of the EXODUS Storage System

This section presents a brief overview of the EXODUS storage system. The following paragraphs highlight the key characteristics of the EXODUS storage system that will be expanded upon in the remainder of the chapter.

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 such as a B+ tree 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 [Chou, et al. 1985], which is another storage system that was developed at the University of

Wisconsin. While WiSS provides a notion of long records, one cannot build a B+ tree on a file of long records because of the way the system's implementation differentiates between long and short records.

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 [Dayal and Smith 1985], 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 given 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 can easily 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 seems infeasible for the storage system to know anything about the conceptual schema. (By conceptual schema, we mean the way in which data is interpreted by higher levels of the system.) On the other hand, semantics can sometimes be useful for performance reasons. For example, it was shown in [Chou

[2] Internally, we speak of such "users" as "database implementors" (or DBI'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 domain once, by a DBI, and then modified only occasionally (if at all) for applications within that domain.

and DeWitt 1985] 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, and the storage manager also supports clustering hints which guide its placement of objects on disk.

3. The Storage Manager Interface

Before describing the details of how large storage objects and file objects (collections of storage objects) are represented in the EXODUS storage system, we 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 authors of an application-specific DBMS. The intended implementation language for such a layer is E [Richardson and Carey 1987], which shields clients from some of the details of the storage manager interface, although we also expect to support some clients who wish to use only the storage manager of EXODUS.

The EXODUS storage system provides a procedural interface. This interface includes procedures to create and destroy file objects and to scan through the objects that they contain. For scanning purposes, the storage system provides a call to get the object id of the next storage 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. 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 is 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 is to be replaced with an indicated new bytestring of the same size. 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 transaction-related hooks, such as a call to release certain locks early, to aid the in the efficient implementation of concurrent and recoverable operations on new access methods.

In addition to the functionality outlined above, the EXODUS storage system accepts a number of performance-related hints. For example, the object creation routine accepts hints about where to place a new object (e.g., "place the new object near the object with id *X*"); such hints can be used to achieve clustering for related or complex objects. It is also possible to request that an object be alone on a disk page and the same size as the page, which is very useful when implementing access methods. In regard to buffer management, information about how many buffer page frames to use and what replacement policy to employ are accepted by the buffer manager. This is supported by allowing a *buffer group* to be specified with each object access; the buffer manager accepts this information on a per-buffer-group basis, which allows variable-partition buffer management policies such as DBMIN [Chou and DeWitt 1985] to be easily supported.

4. Storage Objects

As described earlier, *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 anywhere within a storage object. This section of the chapter 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 [Astrahan, et al. 1976, Stonebraker, et al. 1976, Chou, et al. 1985]; as such, 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 resides 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 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 automatically converts it into a large storage object, leaving its object header in place of the original small object[3]. We considered the alternative of using surrogates for OID's rather than physical addresses, as in other recent proposals [Pollack, et al. 1981, Copeland and Maier 1984, Skarra, et al. 1986, Stonebraker and Rowe 1986, Stonebraker 1987], 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. (Besides, a surrogate index can be implemented above the storage manager level for applications where surrogate support is required.)

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 [Stonebraker, et al. 1983], 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; thus, an absolute byte offset within a child translates to a relative offset within its parent node. The left child of the root in Fig. 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.

[3]Note that for performance reasons the inverse operation is not done; once a small object is converted to a large object, it is never again converted back to a small object.

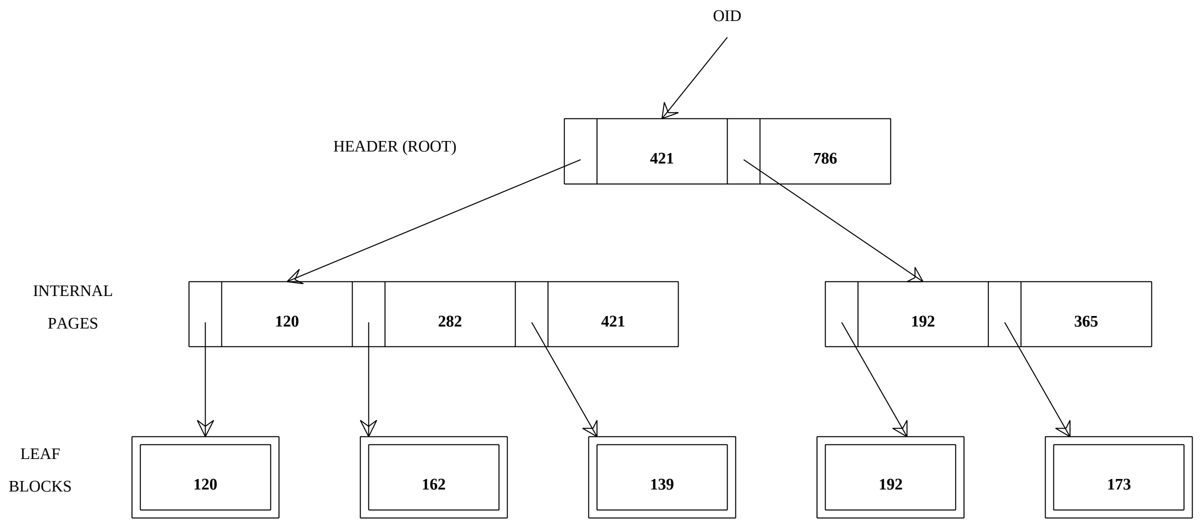

Figure 1: An Example of a Large Storage Object

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 can be made 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 can consist of several contiguous pages to lower the I/O cost of scanning long sequences of bytes within such objects. (The leaf block size can be set on a per-volume basis.) 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).

No. 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.

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 contain between 255 and 511 (count, pointer) pairs. It should be obvious from the table that two or three levels will suffice for most large objects.

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 are quite different from those in the proposal of Stonebraker, et al [Stonebraker, et al. 1983], 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. The append operation 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 $p[i]$, $1 \le i \le 2n_e+1$, 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 $start \le c[i]$. 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 Fig. 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 occurs, proceed as follows: 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, simply 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 (but without the neighbor check).

The motivation for the neighbor checking portion of step 3. is to avoid allocating an additional leaf node in cases where the overflow can instead be accommodated by a neighboring node. This adds only a small cost to the overall expense of insertion, as it is unnecessary to access a neighboring leaf unless it is determined (based on examining the parent's count values) that the 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, the I/O cost increase is worth it in this case, as the modification leads to a significant improvement in storage utilization [Carey, et al. 1986a]. 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 that 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 internal 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 in the same way that leaves are treated, we decided not to do this; doing so 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.

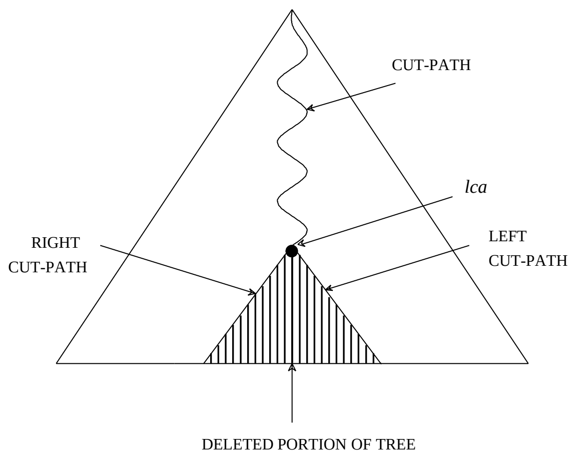

Figure 2: Terminology for Deletion Algorithm

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, that is, 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. (Note that the second phase is less expensive than it may sound, as most of the relevant nodes will be buffer-resident after the first phase.)

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.

The notion of a node being in danger of underflowing (possibly without actually having underflowed) is what allows the rebalancing 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[4] 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; 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 rewritten during the deletion phase.

4.1.5. Performance Characteristics

Before we actually implemented our large object data structure and algorithms, we constructed a main-memory prototype in order to investigate various design alternatives and performance tradeoffs. In our study, we assumed a 4K-byte page size, and we tried using both 1-page and 4-page leaf blocks. Our experiments consisted of using the append operation to construct objects of several different initial sizes (with an initial storage utilization of approximately 100%), and then running a mix of randomly generated read (search), insert, and delete operations on the objects. We experimented with 10MB and 100MB

[4] 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.

objects, and we ran tests with 1-byte, 100-byte, and 10K-byte search, insert, and delete operations. The details of this study are described in [Carey, et al. 1986a]. We summarize the major results here, and refer the interested reader to the original paper for further details.

Basically, we found that the EXODUS large storage object mechanism can operate on very large dynamic objects at relatively low cost, and at a very reasonable level of storage utilization. In particular, with the insert algorithm described in this chapter, which avoids allocating new leaf pages unless both neighbors are unable to accommodate the overflow, we obtained average storage utilizations in the range of 80-85 percent. These results were obtained in the presence of random byte insertions and deletions; for large static objects, utilizations of close to 100% would be the norm, and utilizations in the 90-100% range would be expected for objects where updates are more localized. With respect to the choice of leaf block size, our experiments highlighted the expected tradeoffs: 4-page leaf blocks were definitely advantageous for large, multi-page reads, leading to a 30-35 percent performance improvement compared to 1-page leaf blocks. However, they increased the cost somewhat for updates, and led to a 5-10% decrease in storage utilization in the presence of random insertions and deletions. Multi-page leaf blocks thus offer the greatest advantages for large, relatively static objects (e.g., raster images), where storage utilization will be close to 100% (because such objects are built via appends and are not subjected to frequent, random updates).

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, with updates to a versioned object leading to the creation of a new version of the object. An object is designated as being either a versioned or non-versioned object when it is first created. When a versioned storage object is updated, its object header (or the entire object, in the case of a small storage object) is copied to a new location on disk and updated there as a new version of the object. The OID of the new version is returned to the updater, and the OID of the old version remains the same (i.e., it is the OID that was originally passed to the update routine, since OID's are physical addresses). To ensure that the cost of copying the new version elsewhere is not as prohibitive as it might otherwise be [Carey and Muhanna 1986], the new version is placed on the same page of the file object, or else on a nearby page, if possible. (Note that we do not use 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 [Stonebraker 1981, Dadam, et al. 1984, Katz and Lehman 1984, Batory and Kim 1985, Clifford and Tansel 1985, Klahold, et al. 1985, Snodgrass and Ahn 1985, Katz, et al. 1986]. Therefore, we leave the maintenance of data structures such as graphs of the versions and alternatives of objects to a higher level of the system, a level that will vary from application to application (unlike the storage system). The reason that we do not leave version management out of the EXODUS storage system altogether is one of efficiency — it could be prohibitively expensive, both in terms of storage space and I/O cost, to maintain versions of very 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 3 illustrates this by an example. The figure shows two versions of the large storage object of Fig. 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

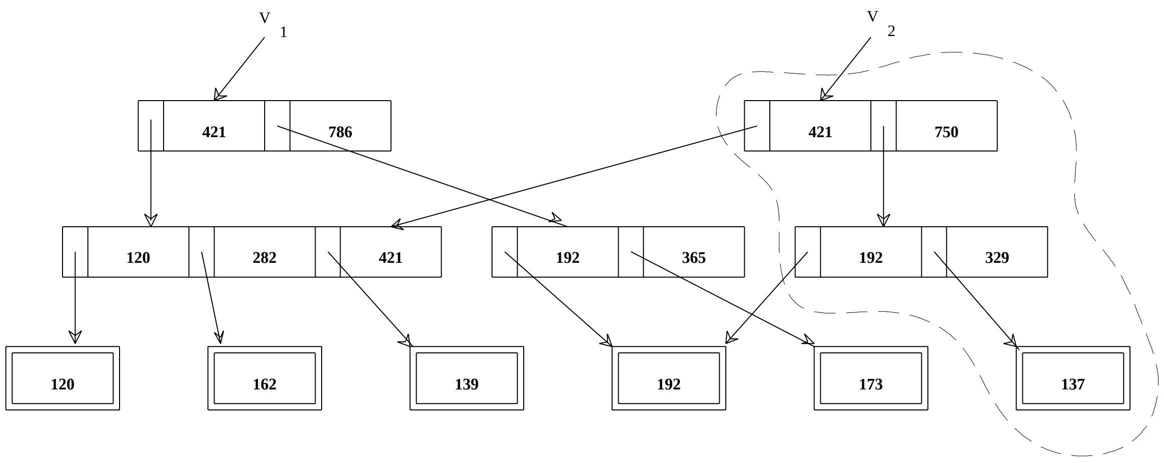

Figure 3: Two Versions of a Large Storage Object

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, 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 Fig. 4, where we wish to delete a version V which has a direct ancestor V_a (from which V was derived) and descendants V_{d_1} through V_{d_n} (which were derived 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 descendant 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 descendants of an object, that is,

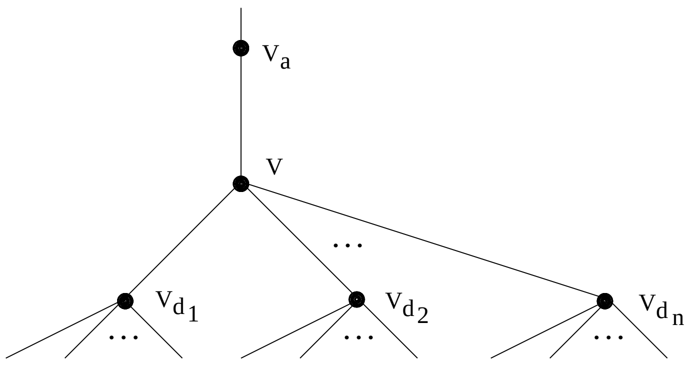

Figure 4: An Example Version History

the *adjacent* versions of an object (the version from which the object was directly derived, or versions which were themselves directly derived 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 Fig. 3.) 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 Fig. 3.) The first observation means that we only need to visit a shared subtree's root; there is no need to visit its descendant pages since they are necessarily shared. The second observation means that if we scan versions of equal height level by level, then we can 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 descendant, 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, where $l=0$):
   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 descendant 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 Fig. 4, to a higher level of the system.

To implement this algorithm efficiently, one can 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 simply ends 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 represent less than 1% of the storage requirements for very large objects). In typical cases, however, the algorithm will visit relatively few pages, as adjacent versions are likely to share the bulk of their pages. Thus, despite its apparent complexity, this approach is likely to be cheaper in terms of I/O cost than an approach based on reference counts, as a reference counting scheme would almost certainly require reference counts to be inspected on all deleted leaf pages (requiring these pages to be read).

4.3. Concurrency Control and Recovery

The EXODUS storage system provides concurrency control and recovery services for storage objects. Initially, concurrency control is being provided via hierarchical two-phase locking [Gray 1979] at the levels of file objects and storage objects. Our eventual plans involve locking byte ranges of storage objects, with a "lock entire object" option being provided for cases where object level locking is sufficient. While we expect object level locking to be the norm for small storage objects, byte range locking may be useful for large storage objects in some applications: For updates that change the contents of a byte range without changing the size of the range (i.e., updates that read and then rewrite bytes in some range), concurrent searches and updates in disjoint regions of the object will be permitted. However, 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 are indeterminate 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 [Bayer and Scholnick 1977] 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[5], 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. We have opted to use locking techniques because they are likely to perform at least as well as other concurrency control techniques in most environments [Agrawal, et al. 1987].

For recovery, small storage objects are being handled via classical logging techniques and in-place updating of objects [Gray 1979, Lindsay, et al. 1979]. Log entries are operation-oriented, recording byte-range update operations and their arguments. Recovery for large storage objects is handled using a combination of shadows and operation logging — updated internal pages and leaf blocks are shadowed up to the root level, with updates being installed atomically by overwriting the old object header with the new header [Verhofstad 1978]. Prior to the installation of the update at the root, the other updated pages must be written to disk; also, the name and parameters of the update operation will be logged, with the log sequence number (ala [Gray 1979, Lindsay, et al. 1979]) of the update log record being placed on the root page of the object. This ensures 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. The same recovery scheme will be employed for versioned objects as well.

4.4. Buffer Management for Storage Objects

As described earlier, one objective of the EXODUS storage system design was to minimize the amount of copying that takes place between higher-level software and the system buffer. A second (related) objective is to allow sizable portions of large storage objects to be scanned directly in the buffer pool. To accommodate these needs, we allocate buffer space in either single-page units or variable-length

[5] Recall that inserting, appending, or deleting bytes causes counts to change all the way up to the root. We may decide to treat write operations similarly to simplify the recovery code.

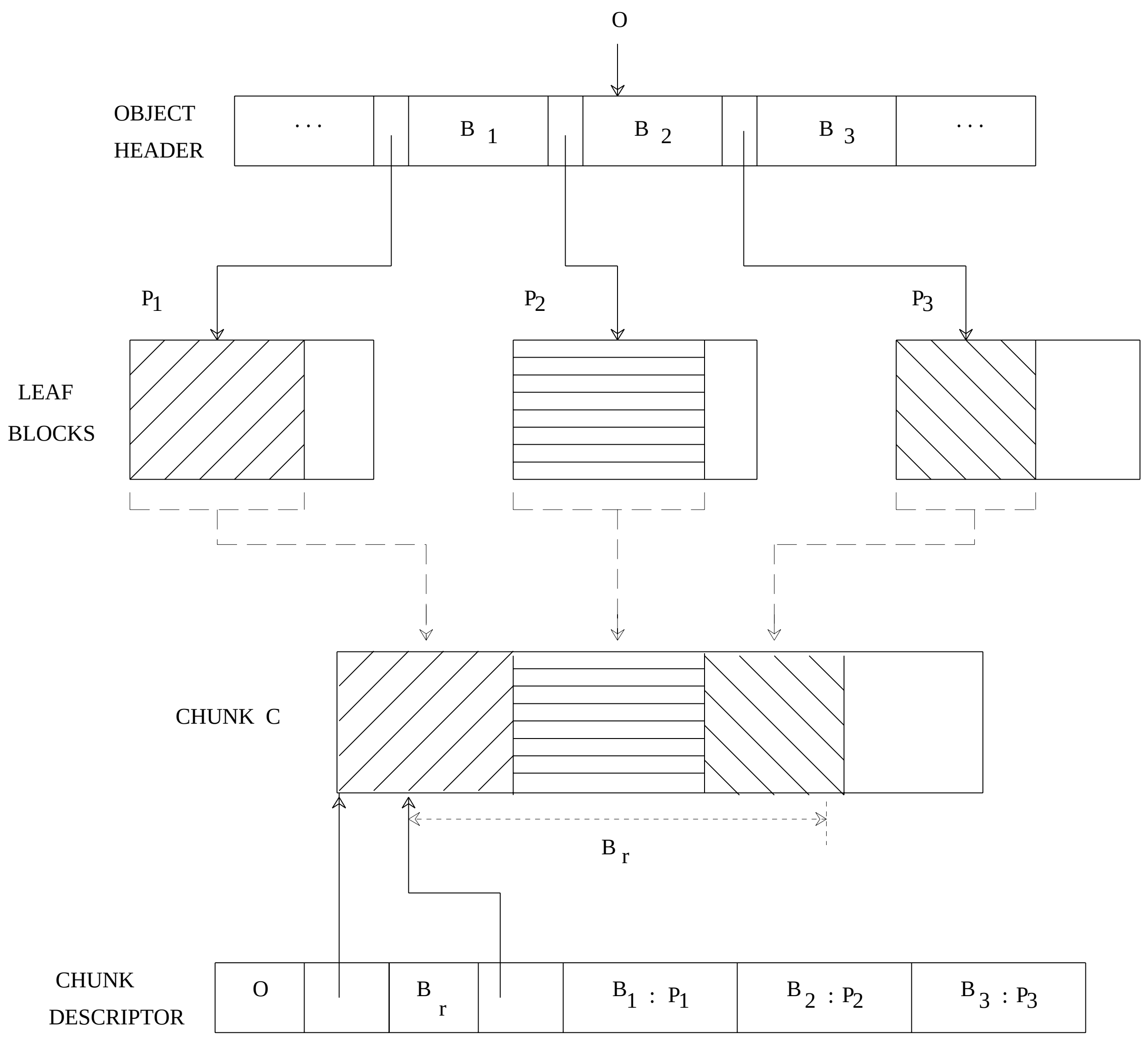

Figure 5: Contiguous Buffering in EXODUS

chunks, where each chunk consists of an integral number of contiguous pages. Single-page units are used for data structures that occupy one page or less of disk space, such as small objects. Requests for single-page units are handled separately to improve system performance, as we expect the majority of buffer requests to be associated with small objects. Chunks are used exclusively for buffering the data of large storage objects. The ability to allocate variable-length chunks simplifies higher-level software by making it possible to read then scan a multi-page sequence of bytes in a large storage object without concern for page boundaries.

Figure 5 sketches the key aspects of the EXODUS buffering scheme for large storage objects. Suppose an EXODUS client requests that a sequence of B_r bytes be read from object *O*. If these bytes are not already resident in the buffer pool, the non-empty portions of the leaf blocks of *O* containing the desired range of bytes (leaf blocks P_1, P_2, and P_3 in Fig. 5) are read into one contiguous chunk *C* of buffer pages. Leaf blocks are supported by our I/O routines as a unit of transfer between the disk and the buffers, so this can be accomplished by obtaining a chunk 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 view this as a serious limitation. Also, chained I/O could potentially be used to advantage here, as in [Paul, et al. 1987]). A *chunk descriptor* is maintained by the buffer manager for the current region of *O* being scanned, including information such as the OID of *O*, a pointer to its first page frame in the buffer pool, the length of the portion of the chunk containing the bytes requested by clients, a pointer to the first such byte, and information about where the contents of the chunk came from (for buffer replacement purposes). In the rare event that another client has already read a portion of B_r into some other chunk C', then *C* is still allocated. This time, however, only the portion of B_r that does not appear in C' is read from disk; the remaining portion of B_r that appears in C' is then replicated in C via a memory-to-memory copy. To avoid inconsistencies, measures are taken to ensure that replicated data is never updated. In particular, any replicated portions of a chunk are deleted from the buffer pool before the chunk is updated. (Concurrency control ensures that this is always possible, as a write lock must be obtained prior to the update.)

Since chunks can be variable-length runs of page frames, the EXODUS buffer manager employs more sophisticated free space management techniques than other buffer managers that we have encountered. To satisfy requests for various sizes of chunks, we employ a data structure that maintains lists of free chunks for each currently available size. When a new chunk is requested, the data structure is examined to find the "best" free chunk that satisfies the request. The notion of "best" that we currently use favors chunks of the size requested, breaking up larger free chunks only when necessary; within a given free chunk size, the chunk with the fewest dirty pages is taken. When there is no free chunk large enough to satisfy the request, an attempt is made to coalesce free chunks that are adjacent to each other. If coalescing fails to construct a large enough chunk, then the the request is rejected. Although this sounds expensive, preliminary test results have shown that most chunk requests can be satisfied without coalescing.

Finally, as mentioned in the previous section, buffer space allocation and replacement are performed using the notion of a *buffer group*. When a client opens a buffer group, it specifies the number of page frames for the group (*group-size*) and the replacement policy (*group-policy*) that should be employed when replacing its pages. A buffer group descriptor is returned to the client, and it is passed to the storage manager by the client with requests for subsequent operations on storage objects. These operations are then performed with respect to the group, meaning that if the client has *group-size* pages in use, replacement is performed from within the group's pages using *group-policy* as the replacement strategy. A client is permitted to have multiple buffer groups open at once, so buffering schemes such as DBMIN [Chou and DeWitt 1985], which allocates a buffer partition to each active index and relation instance in a relational query, can be supported. Simple schemes such as global LRU are also easily supported via the buffer group mechanism.

5. File Objects

File objects in the EXODUS storage system are collections of storage objects (i.e., sets of storage objects, with the restriction that an object resides in exactly one set). 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, which 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 of slotted pages contained in the file; the

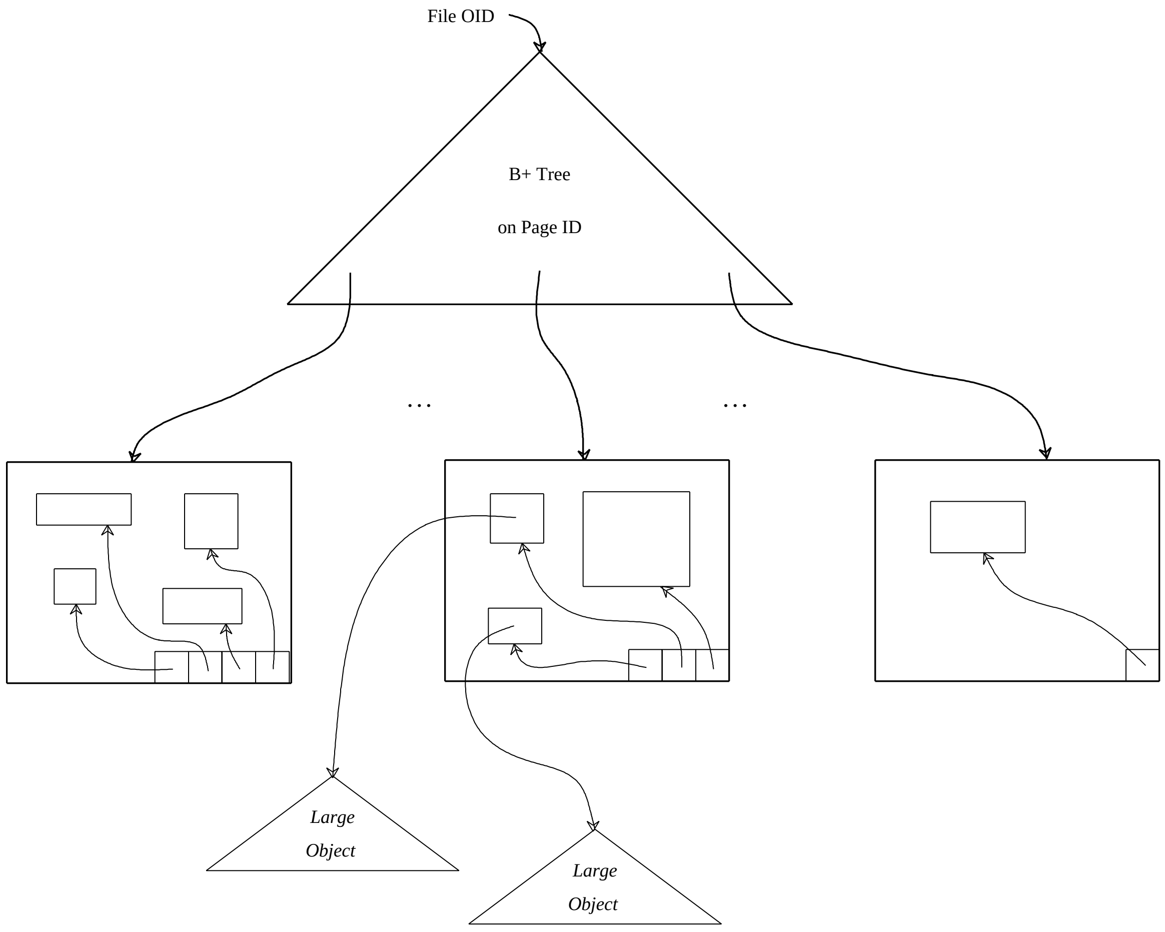

Figure 6: EXODUS File Object Example

actual slotted pages themselves are managed separately, using standard techniques for page allocation and free space management. (Figure 6 shows a file object and how its index relates to the slotted pages on which its objects reside.) The file object index thus serves as a mechanism to gather the pages of a file together, but it also has several additional properties of interest: It facilitates the scanning of all of the objects within a given file object *in physical order* for efficiency, and it supports the fast deletion of an object with a given OID from a file object (as we will see in a moment). We considered several other file designs before settling on this one, 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 also that since all of the objects in a file are directly accessible via their OIDs, file objects are *not* to be confused with surrogate indices — external references to an object within a given file object are OID values, and these point *directly* to the object.

Creation of a file object allocates the file object header. Later, when an object is to be created within the file object, the object creation routine can 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. When this hint is present, the new object is inserted on the same slotted 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 the new object is inserted on a neighboring page of the file if possible. If a neighboring page is inspected and it is also found to be full, then a new slotted page near *X*'s page is allocated for the newly inserted object and its page number is inserted into the file object B+ tree; the OID of the file object is recorded on the newly allocated page. If no co-location hint is present, *X* is simply appended to the file by placing it on the last page listed in the file object index, with overflows again being handled in the manner just described. To speed the location of a neighboring page for overflow handling, each slotted page has a "neighbor hint" field containing the page number of the most recent neighboring page on which one of its overflows was placed.

Deletion of an object within a file is accomplished by simply removing the object from the page where it resides. If the page becomes empty as a result, its page number must be deleted from the file object index, and the page itself must be returned to the free space manager. Lastly, the deletion of an entire file object must lead to the deletion of all of the objects residing on slotted 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 slotted page contains one or more large object headers or file object headers, then these must of course be recursively deleted; otherwise, the page can be freed immediately. Each leaf page entry in a file object index has a bit that indicates whether or not the corresponding slotted page contains any large object headers (or file object headers), permitting the necessary check to be performed when the file is deleted without reading slotted pages that contain only small objects.

5.2. Other File Object Issues

Concurrency control and recovery for file objects is handled through mechanisms similar to those used for large storage objects. Concurrency control (for page number insertions and deletions) is provided using B+ tree locking protocols. Recovery is accomplished by shadowing changes up to the highest affected level of the file object index, logging the insert or delete operation, and finally overwriting the highest affected node to atomically install the update. Note that these concurrency control and recovery protocols are only exercised when the file index is modified via the insertion or deletion of page number entries, which only occurs when a slotted page is added or removed from the file object; the storage object concurrency control and recovery mechanisms handle slotted page changes that do not involve updates to the file object index.

The final non-trivial issue related to file objects is the question of how one might sort a file object, given that it is not a sequential file. In particular, 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 might have, nor does it know what the data types for the fields are. Such sorting can be accomplished at the client level by exchanging the contents of storage objects according to the client's notion of how the objects should be reordered. Note that sorting necessarily moves the contents of objects from page to page, so their OID's are no longer valid when sorting has been performed. (This is the main way in which OID's differ from surrogates, as all 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.)

Summary

In this chapter, we described the design and implementation of the storage management 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 for grouping and sequencing through collections of storage objects. The data structures and algorithms used to support large storage objects were described, and the results of a study of their performance were summarized; they support large dynamic storage objects efficiently, both in terms of storage utilization and in terms of access and update performance. An approach was described for maintaining versions of large objects by sharing common pages between versions, and an efficient version deletion algorithm was presented. Also described in this chapter were the EXODUS notion of file objects and our novel approach to buffer management. Concurrency control and recovery mechanisms were briefly covered as well.

As of this writing, the design of the EXODUS storage system is complete, and an initial version of the system (based on the ideas presented in this chapter) is fully operational. Included in this version of the system are support for small and large objects, file objects, buffering of variable-length portions of objects, and shadow-based atomic updates for large objects and file objects. With the exception of concurrency control, most multi-user aspects of the design have been accounted for in this implementation. We are now actively building the next version of the system, which will include support for versions and full multi-user transaction management (including concurrency control and operation logging).

Acknowledgements

This research was partially supported by the Defense Advanced Research Projects Agency under contract N00014-85-K-0788, by the National Science Foundation under grant DCR-8402818, by IBM through a Fellowship and a Faculty Development Award, by DEC through its Initiatives for Excellence program, and by a grant from the Microelectronics and Computer Technology Corporation (MCC). Also, the authors wish to acknowledge the contributions of several other people affiliated with the EXODUS project, including Goetz Graefe, Mohamed Isa Hashim, Dan Frank, M. Muralikrishna, and David Haight.

References

[Agrawal, et al. 1987]
Agrawal, R. Carey, M., and Livny, M., "Concurrency Control Performance Modeling: Alternatives and Implications", *ACM Transactions on Database Systems*, 12, 4, December 1987.

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

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

[Batory, et al. 1986]
Batory, D., et al, "GENESIS: A Reconfigurable Database Management System," Technical Report No. TR-86-07, Department of Computer Sciences, University of Texas at Austin, March 1986.

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

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

[Carey, et al. 1986a]
Carey, M., et al, "Object and File Management in the EXODUS Extensible Database System," *Proceedings of the 1986 VLDB Conference*, Kyoto, Japan, August 1986.

[Carey, et al. 1986b]
Carey, M., et al, "The Architecture of the EXODUS Extensible DBMS," *Proceedings of the International Workshop on Object-Oriented Database Systems*, Asilomar, CA, September 1986

[Carey and Muhanna 1986]
Carey, M., and Muhanna, W., "The Performance of Multiversion Concurrency Control Algorithms", *ACM Transactions on Computer Systems*, 4, 4, November 1986.

[Carey and DeWitt 1987]
Carey, M. and D. DeWitt, "An Overview of the EXODUS Project", *Database Engineering* 10, 2, June 1987.

[Chou and DeWitt 1985]
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.

[Chou, et al. 1985]
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.

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

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

[Dadam, et al. 1984]
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.

[Dayal and Smith 1985]
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.

[Deppish, et al. 1987]
Deppisch, U., H-B. Paul, and H-J. Schek, "A Storage System for Complex Objects," *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Graefe and DeWitt 1987]
Graefe, G. and D. DeWitt, "The EXODUS Optimizer Generator," *Proceedings of the 1987 SIGMOD Conference*, San Francisco, CA, May 1987.

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

[Kaehler and Krasner 1983]
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.

[Katz and Lehman 1984]
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.

[Katz, et al. 1986]
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.

[Klahold, et al. 1985]
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.

[Lindsay, et al. 1979]
Lindsay, B., et al, *Notes on Distributed Databases*, IBM Research Report No. RJ2571, IBM San Jose Research Center, July 1979.

[Lindsay, et al. 1987]
Lindsay, B., et al, "A Data Management Extension Architecture," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Francisco, CA, 1987.

[Maier, et al. 1986]
Maier, D., et al, "Development of an Object-Oriented DBMS," *Proceedings of the 1st Annual ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, Portland, OR, 1986.

[Manola and Dayal 1986]
Manola, F., and U. Dayal, "PDM: An Object-Oriented Data Model," *Proceedings of the International Workshop on Object-Oriented Database Systems,* Pacific Grove, CA, September 1986.

[Paul, et al. 1987]
Paul, H.-B., et al, "Architecture and Implementation of the Darmstadt Database Kernel System," *Proceedings of the ACM-SIGMOD International Conference on Management of Data*, San Francisco, CA, 1987.

[Pollack, et al. 1981]
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.

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

[Richardson and Carey 1987]
Richardson, J., and M. Carey, "Programming Constructs for Database System Implementation in EXODUS," *Proceedings of the 1987 SIGMOD Conference,* San Francisco, CA, May 1987.

[Rowe and Stonebraker 1987]
Rowe, L., and Stonebraker, M., "The POSTGRES Data Model," *Proceedings of the 13th International Conference on Very Large Data Bases,* Brighton, England, 1987.

[Schwarz, et al. 1986]
Schwarz, P., et al, "Extensibility in the Starburst Database System," *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

[Skarra, et al. 1986]
Skarra, A., S. Zdonik, and S. Reiss, "A Object Server for an Object-Oriented Database System," *Proceedings of the International Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, September 1986.

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

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

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

[Stonebraker, et al. 1983]
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.

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

[Stonebraker 1987]
Stonebraker, M., "The POSTGRES Storage Manager," *Proceedings of the 13th International Conference on Very Large Data Bases*, Brighton, England, 1987.

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