Extendible Hashing for Concurrent Operations and Distributed Data

Carla Schlatter Ellis
Computer Science Department
The University of Rochester
Rochester, NY 14627

TR110
October 1982

Abstract

The extendible hash file is a dynamic data structure that is an alternative to B-trees for use as a database index. While there have been many algorithms proposed to allow concurrent access to B-trees, similar solutions for extendible hash files have not appeared. In this paper, we present solutions to allow for concurrency that are based on locking protocols and minor modifications in the data structure.

Another question that deserves consideration is whether these indexing structures can be adapted for use in a distributed database. Among the motivations for distributing data are increased availability and ease of growth; however, unless data structures in the access path are designed to support those goals, they may not be realized. We describe some first attempts at adapting extendible hash files for distributed data.

The preparation of this paper was supported in part by the National Science Foundation, under Grant No. IST-8025761.

1. Introduction

The extendible hash file [Fagin 79] is a dynamic data structure that is an alternative to B-trees for use as a database index. While there have been many algorithms proposed to allow concurrent access to B-trees [Bayer 77, Ellis 80, Lehman 81, Kwong 82, Miller 78], similar solutions for extendible hash files have not yet appeared. In this paper, we present solutions to allow for concurrency that are based on locking protocols and minor modifications in the data structure. In addition to developing new algorithms, this work aims to provide a better understanding of techniques for adapting data structures to allow concurrent access. Thus, we investigate what happens when one tries to apply some of the techniques used in B-tree solutions to extendible hash files.

The sequential algorithms for extendible hashing are described in [Fagin 79]. The basic ideas and terminology are summarized below. The data structure consists of two parts: a set of *buckets* and the *directory*. The buckets reside on secondary storage and contain keys and associated information. The order of the data within buckets is not important for this discussion. The directory is an array of pointers to buckets. A hash function is used that generates a very long *pseudokey* when applied to a key. The number of bits of the pseudokey actually used to index into the directory is called the *depth* of the directory and changes as the file grows or shrinks. In our work, the least significant bits are used in order to simplify manipulations of the directory. Suppose that the directory's depth is currently three. This means that at the moment, there are eight valid directory entries. The i^{th} entry, $0 \leq i \leq 7$, points to the bucket that holds all the records whose pseudokeys end in the three bit binary representation of *i*. Each bucket includes a *localdepth* ($\leq$ depth) indicating that the pseudokeys of the records it contains agree in only that number of bits. Thus multiple directory entries will point to the same bucket if its localdepth is less than the directory's depth. Figure 1 gives an example of an extendible hash file for sequential access. To perform a find operation for a key, *k*, one would apply the hash function to *k* to obtain the pseudokey (imagine it is '...101'), determine the current depth of the directory (2 in this example), and use the appropriate bits ('01') as an index. Following the pointer in the directory entry, one would search the third bucket for *k*. As insertions occur, a bucket may become full (indicated by the *count* field) and split into two buckets. If the old localdepth equals depth, the directory doubles in size and depth increases by one. Similarly, deletions may result in two buckets merging and possibly reducing the depth of the directory. One way of detecting the condition that allows halving the size of the directory is to keep a count (named *depthcount*) of the number of buckets whose localdepth equals depth. Figure 2 shows how a sequence of updating operations would affect the structure given in Figure 1 where $x < y = z =$ maximum number of keys per bucket. This data structure is our point of departure for introducing concurrency in Section 2.

The next step is to consider the question of whether these indexing structures can be adapted for use in a distributed database. Among the motivations for distributing data are increased availability and ease of growth; however, unless data structures in the access path are designed to support those goals, they may not be realized. In Section 3 we describe some first attempts at adapting extendible hash files for distributed data. Our thesis is that locking patterns and other aspects of the solutions for concurrency in shared data structures can lead to insights into how to

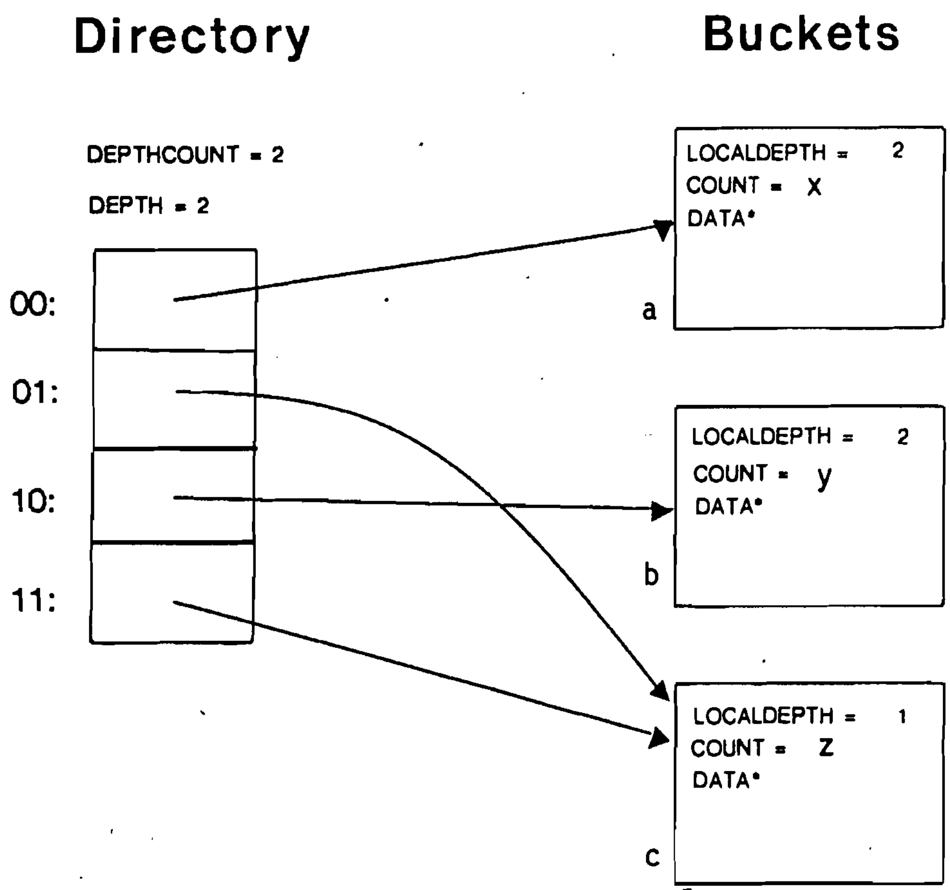

Figure 1

Sequential Access Extendible Hash File

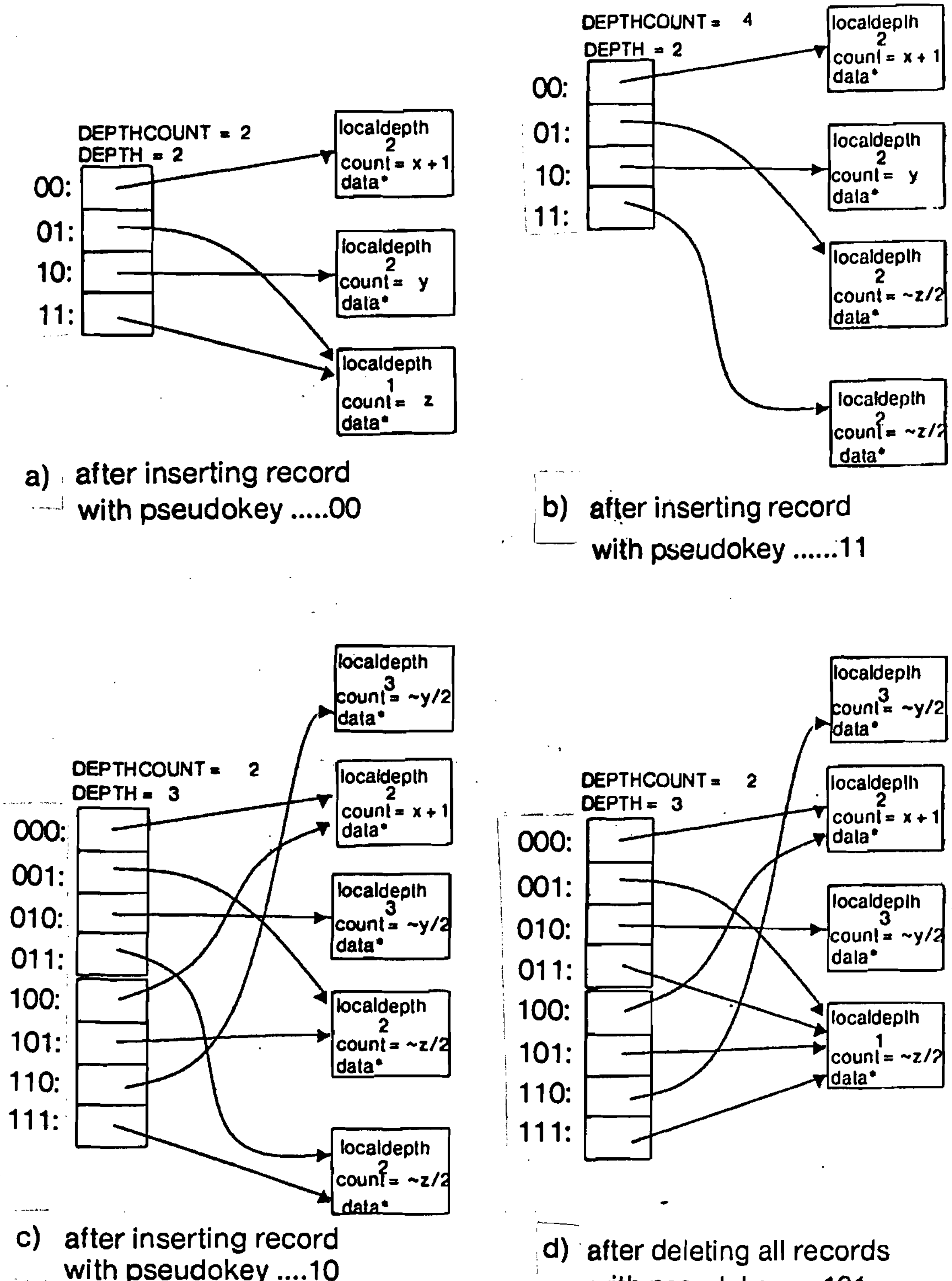

Figure 2
Sequence of Updates

partition the data among processes in a distributed environment. This suggests a methodology for developing distributed solutions.

2. Locking Protocols for Extendible Hash Files

2.1 Common Aspects

In this section, we present two solutions which have evolved from the sequential algorithms for concurrent manipulation of a centralized extendible hash file. Figure 3 shows the modified structure used in these solutions. The fundamental change is that the buckets are linked through a *next* field to allow recovery from concurrent restructuring operations. This provides an alternate path to the desired data that can be used by a searching process when the information is involved in a split or merge operation. Thus when a bucket splits, the next link of the original bucket is reassigned to point to the newly created bucket. The new bucket gets the original bucket's old next pointer. Merging does the reverse. Figure 4 shows what happens when the second bucket in Figure 3 splits. The approach is similar to the use of link pointers in Lehman and Yao's B^{link}-tree solution [Lehman 81]. In addition, there must be a way for a process to tell if it has the wrong bucket. We chose to include a field (*commonbits*) containing the common bit pattern that characterizes the pseudokeys that belong in the bucket. Alternatively, one could reapply the hash function to any key stored in the bucket and use this for comparison with the target pseudokey as long as the possibility of an empty bucket is taken care of.

The goal is to allow a number of processes to be in various stages of *find*, *insert*, or *delete* operations at the same time. Each process can manipulate the data after locking appropriate portions of the shared structure and transferring the information into private buffers. The buckets are assumed to occupy physical pages on disk which are read and written as single operations. The locking protocol uses various types of locks placed on the directory (as a whole) and on individual buckets. The compatibility of lock types is given by the following table.

Lock request	Existing lock		
	ρ	α	ξ
ρ (read-lock)	yes	yes	no
α (selective lock)	yes	no	no
ξ (exclusive lock)	no	no	no

2.2 First Solution

The following set of algorithms is similar to top-down locking protocols for B-tree variants (cf. [Bayer 77], [Ellis 80]), in that a lock is placed on each level of the structure (in this case there are only two levels, the directory then a bucket) and held until it is found to be no longer needed. The procedures are given in Figures 5, 6, and 7 for find, insert, and delete respectively.

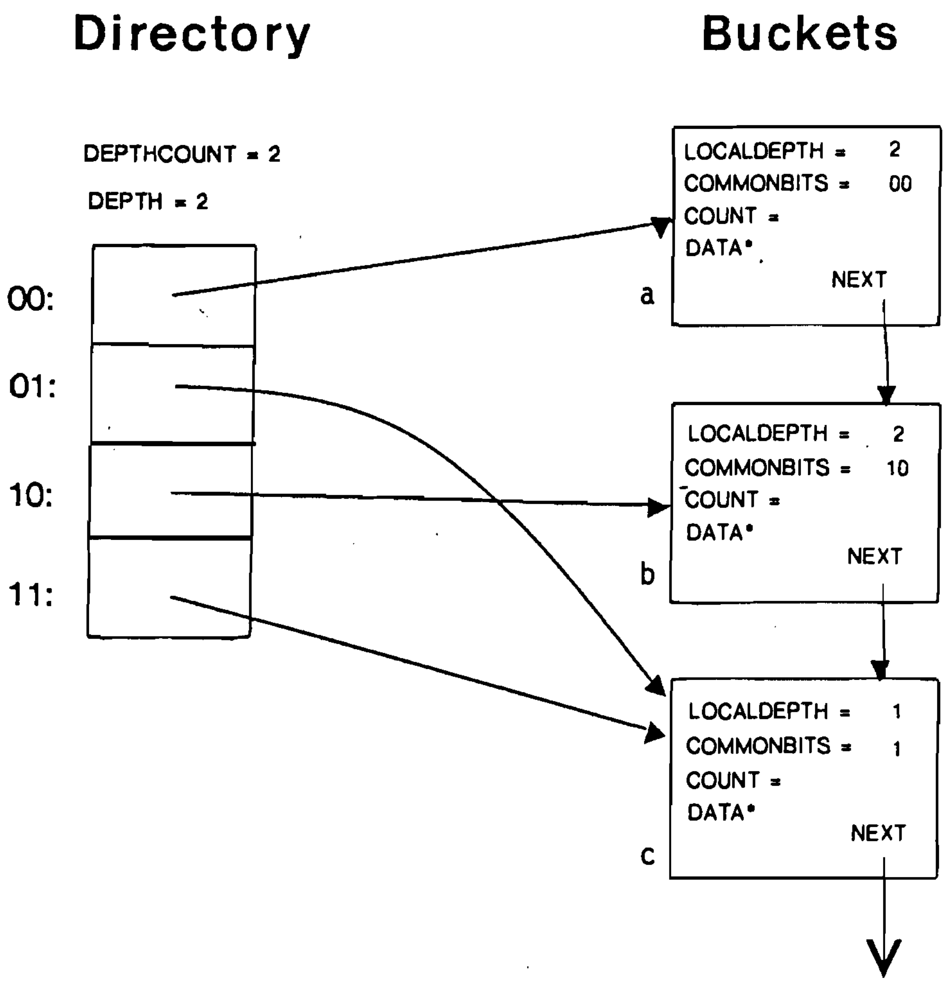

Figure 3

Centralized Concurrent Extendible Hash File

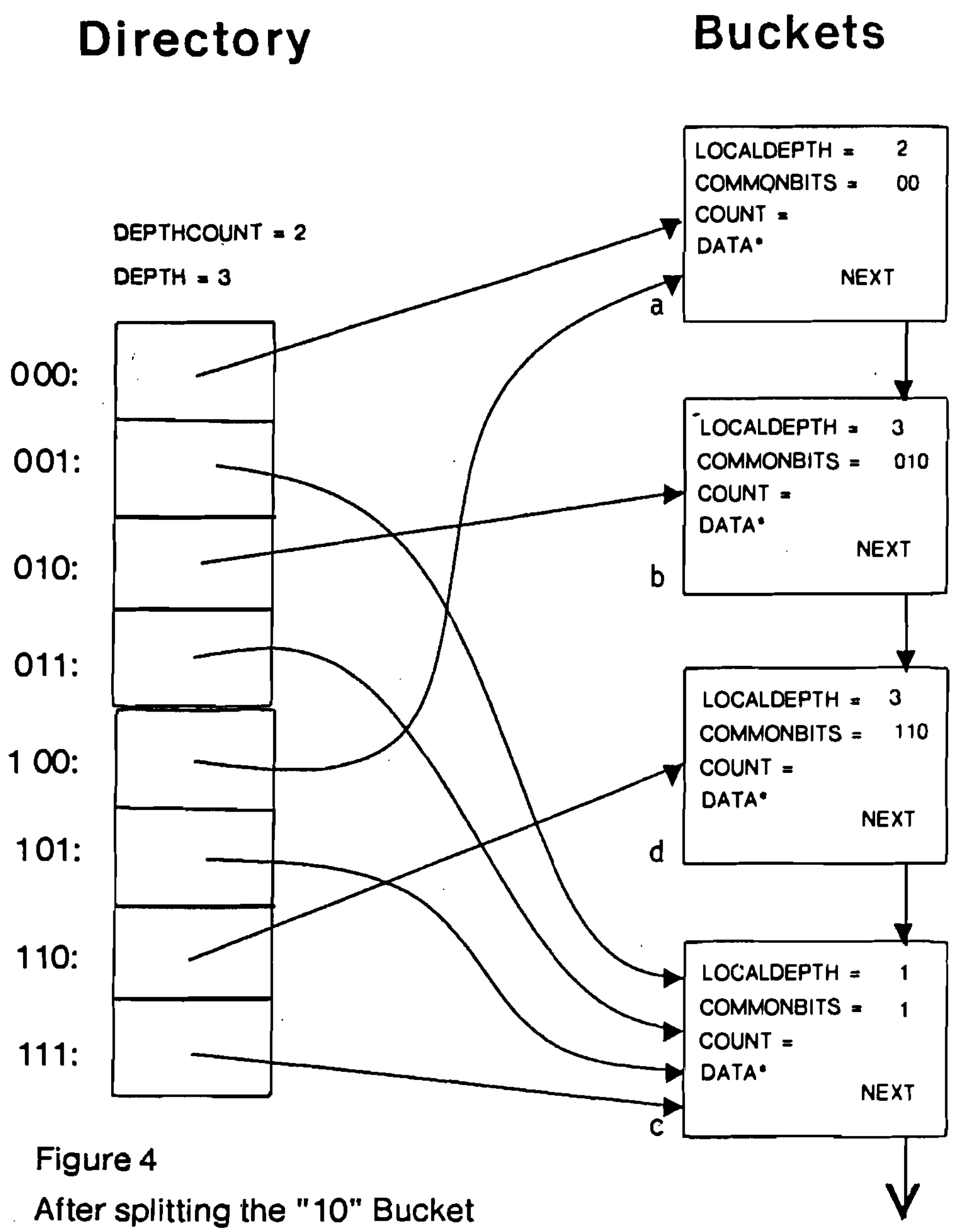

Figure 4
After splitting the "10" Bucket

A process executing the find operation must ρ-lock the directory before reading the depth and extracting the apparent bucket pointer. The ρ-lock is necessary to prevent interference from a deleting process. If the deleter did not exclude the reader and was in the process of halving the directory, the reader might try to access an invalid directory entry based on the old value of depth. A similar interference could occur between readers and deleters with regard to newly deallocated buckets. Therefore, deleting processes must place incompatible ξ-locks. Readers can safely execute in parallel with inserting processes because of the next links and the fact that no portion of the structure is lost during bucket splitting or directory doubling actions. After determining which bucket is to be searched, the reader places a ρ-lock on it, releases the lock on the directory, and transfers the contents into a private buffer. The reader may then discover that it has the wrong bucket. This means that a split occurred after the directory was read and before the data was retrieved. Now the localdepth low order bits of the target pseudokey do not match the commonbits of this bucket. By following the next pointer, the right bucket will eventually be found. The next bucket is always ρ-locked prior to releasing the lock on the current bucket. This flow of locks prevents processes from leapfrogging each other.

Updating operations are serialized with respect to each other by α-α, α-ξ, and ξ-ξ lock incompatibilities. To insert, an α-lock is placed on the directory and held until there is no need for further directory manipulation due to this insertion. Readers can still proceed because of lock compatibility. Changes made by inserters to the official shared structure appear to readers as atomic actions. Splitting a bucket appears as an atomic action because of the order in which the new bucket pair is written back to disk. Doubling the directory appears atomic because of the choice to use the least significant bits of the pseudokey. Deleting processes use ξ-locks because of the previously mentioned problems with readers. If the target bucket is too empty, the deleter will try to merge with the partner bucket. The simplest interpretation for "too empty" is that the only record contained in the bucket is the one to be deleted. Here "partner" refers to the partner with respect to the target bucket's localdepth. Merging is then possible if the partners have the same localdepth (and it is not 1). Two buckets are defined as partners with respect to bit position d if their commonbits match in bits d-1 to 1 and differ at bit d (where the least significant bit is numbered 1). Suppose we want to merge bucket B with its partner bucket C. ξ-locking the partner is straightforward if C follows B in the linked ordering of buckets. Otherwise this action actually involves temporarily releasing the lock on B and requesting ξ-locks on C and B in order. This avoids deadlock with a reader following next links from C to B. Alternatively, the reader could have held the ρ-lock on the directory until it had the right bucket, but this would be a more pessimistic approach and would have to be abandoned in the next solution anyway. Detecting the condition necessary for halving the directory could be done in several ways. Here, a depthcount field containing the number of buckets with localdepth = depth is maintained by structure modifying operations (e.g. splitting a bucket of localdepth = depth-1 would add two, merging two buckets of localdepth = depth would subtract two, halving the directory would involve a scan of directory contents to determine depthcount for the new depth by comparing corresponding entries in the top and bottom halves for pointers which differ, and doubling the directory would set it to zero).

Figure 5 Find Algorithm

Shared data for all of the centralized algorithms:

```
struct buffer {
   int localdepth;
   int commonbits;
   int count;
   int next;
   int data[numentries]};
int depth, depthcount;
int directory[1<<maxdepth];
```

```
find(z)
{
    int     pseudokey,
            oldpage, /*disk page address*/
            newpage; /*disk page address*/
    struct buffer   B,
                *current; /* pointer to buffer */
    unsigned    m;
    current = &B;

    pseudokey = hash (z);
    RhoLock (directory);
    oldpage = indexdirectory (pseudokey & mask (depth));
    RhoLock (oldpage);
    UnRhoLock (directory);
    getbucket (oldpage, current);
    m = mask (current -> localdepth);
    while ((m & pseudokey) !=
            current -> commonbits) {/* wrong bucket */
        RhoLock (newpage = current -> next);
        getbucket (newpage, current);
        m = mask (current -> localdepth);
        UnRhoLock (oldpage);
        oldpage = newpage;
    }

    if (search (current, z))/* is z there? */
        found (z);
    else
        notfound (z);
    UnRhoLock (oldpage);
}
```

Figure 6 Insertion Algorithm

```
insert(z)
{
    int      pseudokey,
             oldpage,
             newpage,
    int      done;
    struct buffer   A,
                    B,
                    C,
                   *current,
                   *half1,
                   *half2;
    current = &A;
    half1 = &B;
    half2 = &C;

    pseudokey = hash (z);
    AlphaLock (directory);
    oldpage = indexdirectory (pseudokey & mask (depth));
    AlphaLock (oldpage);
    getbucket (oldpage, current);
    if (search (current, z)) { /* z is already there */
        UnAlphaLock (directory);
        UnAlphaLock (oldpage);
    }
    else
        if (current -> count != numentries) {
                                /* current bucket not full */
            UnAlphaLock (directory);
            add (current, z);
            putbucket (oldpage, current);
            UnAlphaLock (oldpage);
        }
        else {  /* current is full */
            if (current -> localdepth == depth)
                    doubledirectory ();
            newpage = allocbucket ();
            done = split (current, half1, half2, z, newpage);
            putbucket (newpage, half2);
            putbucket (oldpage, half1);
            UnAlphaLock (oldpage);
            updatedirectory (newpage, half1 -> localdepth, pseudokey);
            UnAlphaLock (directory);
            if (!done)
                insert (z);
        }
}
```

Figure 7 Deletion Algorithm

```
delete(z)
{
    int     pseudokey,
            selectedbits,
            oldpage, /* disk address */
            newpage, /* disk address */
            merged,  /* disk address */
            garbage; /* disk address */
    struct buffer   B,
                    C,
                   *brother,
                   *current;
    unsigned    m;
    current = &B;
    brother = &C;

    pseudokey = hash (z);
    XiLock (directory);
    selectedbits = pseudokey & mask (depth);
    oldpage = indexdirectory (selectedbits);
    XiLock (oldpage);
    getbucket (oldpage, current);
    if ((current -> count > 1) || (current -> localdepth == 1)) {
                                    /* current not too empty */
        UnXiLock (directory);
        if (remove (z, current)) /* successful */ putbucket (oldpage, current);
        UnXiLock (oldpage);
    }
    else {
        if (search (current, z)) { /* z is there */
            m =leftshift( 1, current -> localdepth - 1);
            if ((pseudokey & m) != m) {/* z goes in first of pair */
                newpage = current -> next;
                XiLock (newpage);
                getbucket (newpage, brother);
                merged = oldpage;
                garbage = newpage;
            }
            else {                  /* z goes in second of pair */
                newpage = indexdirectory (selectedbits & ~m);
                UnXiLock (oldpage);
                XiLock (newpage);
                XiLock (oldpage);
                getbucket (newpage, brother);
                merged = newpage;
                garbage = oldpage;
                brother -> next = current -> next;
            }
            if (current -> localdepth != brother -> localdepth) {
                                    /* not possible to merge these two */
                if (remove (z, current)) putbucket (oldpage, current);
            }
            else {                  /* mergable */
                if ((brother -> localdepth--) == depth)
                    depthcount = depthcount - 2;
                brother -> commonbits = brother -> commonbits & mask (brother -> localdepth);
                putbucket (merged, brother);
                if (depthcount == 0)
                    halvedirectory ();
                else
                    updatedirectory (merged, brother -> localdepth + 1, pseudokey);
                deallocbucket (garbage);
            }
            UnXiLock (newpage);
        }
        UnXiLock (oldpage);
        UnXiLock (directory);
    }
}
```

2.3 Correctness of the First Solution

Showing the correctness of this solution requires a proof that it is deadlock free and that requested operations perform correctly both with respect to the target key and the integrity of the data structure. Specifically, a key to be inserted (deleted) should be present (absent) when the update terminates. If the desired data for a find operation is in the file and not the subject of a concurrent update operation, it should be found.

The freedom from deadlock argument depends on the fact that locks are requested according to an ordering on the lockable components of the structure. The directory is always locked first, followed by one of the buckets. While a bucket is locked, additional locks are requested only on buckets reachable from it via next links. The only processes that ever attempt to lock more than one bucket are those executing find or delete operations. Readers follow next links from buckets they have locked. Deleters attempt to lock both partners of a potential merge. For as long as any two buckets remain in the hashfile, the ordering imposed on them by reachability through next links remains the same and between any two partner buckets, there is a path from the "0" partner to the "1" partner. Thus a process trying to delete from the "1" partner will have to release its lock on that bucket in order to get both partners locked according to the ordering. In addition, it is impossible for a process to read a pointer for a bucket that will be deallocated before it can make its lock request since a deleter excludes other processes from parts of the data structure that contain pointers to the buckets being removed. This point is important to ensure that lock requests can eventually be satisfied.

It is almost trivial to show the correctness of update operations in this solution since they are essentially sequential. Removing or adding a key to the hash file depends first of all on the updating process getting to the right bucket. Since a lock is held on the directory while an updater initially reads the bucket pointer and kept until the directory reflects all changes in the structure resulting from its update, the information seen by updaters when they read the directory is the same as it would be if updates were completely serial. Arriving at the right bucket, the updater must also see the right version of it. Again a lock which excludes other updaters is required in order to read the bucket contents into private storage and is held until the bucket is rewritten (or it is discovered that no change is needed). Thus previous updaters have made their modifications known by the time a new updater gains its lock. Since updates do not interfere with each other, the data structure should be correct when no update operations are in progress.

Finally, we must consider interactions between readers and updaters. The locking protocol ensures that a reader and a deleter are serialized according to the order in which they lock the directory. A deleter exclusively locks the directory, the target bucket, and its partner (when necessary) while modifications are taking place. No intermediate stages of the deletion operaton will be visible to other precesses. A deleter could potentially interfere with a reader if the effects of the deletion appeared after the reader gained some information from the file and before that information was acted upon (e.g. the reader gets a bucket pointer from the directory, the deleter merges that bucket into its partner, then the reader tries to follow the pointer). However, this is impossible since the source of the reader's information remains ρlocked until the next lock is granted. This is also true when the reader is following next links. Whenever a bucket, A, can be merged into its partner, B, then B's next link will point to A.

By contrast, a reader may see intermediate stages of an insertion operation but this does not prevent it from ascertaining the presence or absence of any key other than the one being added. The possible changes in the data structure caused by an inserting process are as follows: If the inserter's target bucket is not full, it is replaced in a single put operation with the original contents plus the new record. A reader will see either the old or the new bucket and the only difference is the key being added. If the inserter's bucket is full, it will be replaced by a pair of buckets in which the old contents are distributed between the two according to pseudokey. The new record will be included in the appropriate partner if there is room. The second half of the pair is written first in a newly allocated disk page and then the old bucket is replaced by the first half of the pair. Immediately after the first put, the new bucket is still not reachable through pointers in the hash file. Thus writing the pair is equivalent to the single operation of writing the first partner. A reader which sees a directory entry before it is updated to point to the new bucket will get either the old bucket or the first half of the pair. If the reader's desired data has moved to the second half, it will detect this and follow the next link. Finally, the inserter may need to double the directory. This appears to readers as a single operation. The directory space is extended and the old contents copied prior to incrementing depth and it is the act of incrementing depth that makes the new directory entries visible.

Even assuming fairness in the granting of lock requests (e.g. FIFO subject to the compatibility relationship), lockout of readers is possible if their target buckets are constantly changing due to a steady stream of updates.

2.4 Second Solution

The recognized problem with top-down protocols is the need to hold a lock on the bottleneck of the structure while determining if restructuring will be required. This is avoided in the next protocol. The idea is for updating processes to act like readers during their search for the right bucket. The procedure for the find operation is the same as before. The algorithms for insert and delete are found in Figures 8 and 9.

For the insert operation, a ρ-lock is placed on the directory that will be converted to an α-lock if the directory actually will be modified. Other insert or delete operations can also be active. The next pointer is again used for recovery but now deleted, but not yet deallocated, buckets also provide a recovery path. Because of the additional concurrency, updaters may also find themselves with the wrong bucket and must follow the recovery path. "Wrong bucket" now includes the case where this bucket has been merged into a preceeding bucket. The bucket is marked as "deleted." Since there are no circular paths through the next pointers that are not protected with the deleting process's ξ-locks, this protocol can be shown to be deadlock free.

In addition to setting up the merged bucket, merging now involves marking the old partner as "deleted" (we use the commonbits field for this), setting its next field

Figure 8 Insertion Algorithm

```
insert(z)
{
    int     pseudokey,
            oldpage,
            newpage;
    int     done;
    struct buffer   A,
                    B,
                    C,
                   *current,
                   *half1,
                   *half2;
    unsigned    m;
    current = &A;
    half1 = &B;
    half2 = &C;

    pseudokey = hash (z);
    RhoLock (directory);
    oldpage = indexdirectory (pseudokey & mask (depth));
    AlphaLock (oldpage);
    getbucket (oldpage, current);
    m = mask (current -> localdepth);
    while ((m & pseudokey) != current -> commonbits) {/* WRONG BUCKET */
        AlphaLock (newpage = current -> next);
        getbucket (newpage, current);
        m = mask (current -> localdepth);
        UnAlphaLock (oldpage);
        oldpage = newpage;
    }
    if ( search (current, z)) { /* IS Z ALREADY THERE? */
        UnRhoLock (directory);
        UnAlphaLock (oldpage);
    }
    else
        if (current -> count != numentries) {
                                 /* CURRENT BUCKET NOT FULL */
            UnRhoLock (directory);
            add (current,  z);
            putbucket (oldpage, current);
            UnAlphaLock (oldpage);
        }
        else {                   /* CURRENT IS FULL - DIRECTORY WILL BE AFFECTED */
            AlphaLock (directory);
            if (current -> localdepth == depth)
                    doubledirectory ();
            newpage = allocbucket ();
            done = split (current, half1, half2, z, newpage);
            putbucket (newpage, half2);
            putbucket (oldpage, half1);
            updatedirectory (newpage, half1 -> localdepth, pseudokey);
            UnAlphaLock (oldpage);
            UnAlphaLock (directory);
            UnRhoLock (directory);
            if (!done)
                insert (z);
        }
}
```

Figure 9 Deletion Algorithm

```
delete(z)
{
    int     pseudokey,
            selectedbits,
            oldpage,
            newpage,
            garbage,
            merged;
    struct buffer   B,
                    C,
                   *brother,
                   *current;
    unsigned    m;
    current = &B;
    brother = &C;

    pseudokey = hash (z);
    RhoLock (directory);
    selectedbits = pseudokey & mask (depth);
    oldpage = indexdirectory (selectedbits);
    XiLock (oldpage);
    getbucket (oldpage, current);
    m = mask (current -> localdepth);
    while ((m & pseudokey) != current -> commonbits) {/* WRONG BUCKET */
        XiLock (newpage = current -> next);
        getbucket (newpage, current);
        m = mask (current -> localdepth);
        UnXiLock (oldpage);
        oldpage = newpage;
    }
    if ((current -> count > 1) || (current -> localdepth == 1)) {
                                    /* CURRENT NOT TOO EMPTY */
        UnRhoLock (directory);
        if (remove (z, current)) putbucket (oldpage, current);
        UnXiLock (oldpage);
    }
    else {
    /* IF EVERYTHING STAYS THE SAME - TRY TO MERGE */
        if (!search (current, z)) { /*  Z NOT THERE */
            UnXiLock (oldpage);
            UnRhoLock (directory);
            return;
        }
        else {
            m = leftshift(1, current -> localdepth - 1);
            if ((pseudokey & m) != m) {/* Z IN FIRST OF PAIR */
                newpage = current -> next;
                XiLock (newpage);
                getbucket (newpage, brother);
                garbage = newpage;
                merged = oldpage;
            }
            else { /* Z IN SECOND OF PAIR */
                newpage = indexdirectory (selectedbits & ~m);
                UnXiLock (oldpage);
                XiLock (newpage);
                getbucket (newpage, brother);
/* A */         if (brother -> next != oldpage) {
                        /* OLDPAGE AND NEWPAGE ARE NOT MERGABLE PARTNERS */
                    UnXiLock (newpage);
                    UnRhoLock (directory);
                    delete (z);
                    return;
                }
                else {
                    XiLock (oldpage);
                    getbucket (oldpage, current);
                    garbage = oldpage;
                    merged = newpage;
                    brother -> next = current -> next;
                    if ((mask (current -> localdepth) & pseudokey) != current->commonbits) {
                                /* Z no longer belongs in oldpage - while
```

```
                               waiting to re-lock oldpage it may have
                               filled up and split, moving z */
                        UnXiLock (oldpage);
                        UnXiLock (newpage);
                        UnRhoLock (directory);
                        delete (z);
                        return;
                    }
                }
            }
            if (current -> localdepth != brother -> localdepth ||
                    current -> count > 1 || (current -> count == 1 &&
                        !search (current, z))) {
                                /* Either it is not possible to merge
                                   because of localdepths or something
                                   happened while waiting to re-lock
                                   oldpage - more data inserted into
                                   oldpage so it is no longer empty and
                                   maybe then z deleted */
                UnXiLock (newpage);
                UnRhoLock (directory);
                if (remove (z, current)) putbucket (oldpage, current);
                UnXiLock (oldpage);
                return;
            }
        /* MERGE */
            AlphaLock (directory);
            if ((brother -> localdepth--) == depth)
                depthcount = depthcount - 2;
            brother -> commonbits = brother -> commonbits & mask (brother -> localdepth);
            current -> commonbits = deleted;
            current -> next = merged;
            putbucket (merged, brother);
            putbucket (garbage, current);
            updatedirectory (merged, current -> localdepth + 1, pseudokey);
            UnXiLock (oldpage);
            UnXiLock (newpage);
            UnAlphaLock (directory);
            UnRhoLock (directory);
            XiLock (directory);
            XiLock (garbage);
            if (depthcount == 0)
                halvedirectory ();
            deallocbucket (garbage);
            UnXiLock (directory);
            UnXiLock (garbage);
        }
    }
}
```

to point to the merged bucket, updating the next field of the merged bucket, and writing both buckets back to secondary storage. If it is necessary to release the lock on the target bucket so that ξ-locks may be requested in order on the pair to be merged, then a number of conditions must be checked after gaining the locks. These will be elaborated in the proof. Deleted buckets and discarded halves of the directory are actually deallocated only after ensuring that no process needs them anymore.

2.5 Correctness of Second Solution

The freedom from deadlock issue has been complicated by the presence of deleted buckets and the delayed α-locking of the directory. The key observation to be made with regard to the α-locking is that a process requesting an α-lock on the directory already holds a ρ-lock on it (essentially doing lock conversion) and has all the necessary locks on buckets. This lock request will be refused if there already is an incompatible lock on the directory. If this lock is an α-lock held by another updater, that process will make no further lock requests. The lock cannot be a ξ-lock because of the existing ρ-lock. Therefore, there is no possibility of deadlock due to α-locking. Given the way deleted buckets are handled in this solution, it is not true that the ordering between two buckets stays the same. Thus, bucket B may be reachable from bucket A but if they are partners this relationship may be reversed as B is merged into A. However, it is not possible for processes following the old ordering to coexist with processes following the new ordering because the deleter uses ξ-locks to ensure that all the processes with old information have cleared out of the vicinity of the merge. Extra precautions must be taken by deleters to check that the locking of partners is consistent with reachability (line labeled A in figure 9).

This solution allows more concurrency among updaters than the first solution because of the delay in α-locking for updating the directory and in ξ-locking the directory for garbage collection. Updaters in their searching phase are like readers, so arguments for getting to the right bucket hold for each type of process. With this locking scheme, processes are allowed to read out of date directory entries including pointers to deleted buckets. Imagine a searching process that indexes into the directory and finds a pointer to bucket A as that directory entry is about to be changed to reflect a split or merge. If A has recently been split, A's next link will lead to the new bucket which contains the records moved from A. If A has just been merged into its partner, it will be marked as deleted, making it the "wrong bucket" for any searching process and the next link again will provide recovery. The important observation is that obsolete directory entries that are still visible always point to a bucket from which the correct bucket is reachable via next links. Doubling the directory appears atomic. Finally, searching processes do not access the directory while it is being shrunk. Discarding deleted components is done in a separate phase which is truly serialized with respect to other actions by ξ-locking.

Once an updater arrives at the right bucket and gains the locks it requires, the actual modifications are essentially serialized as in the first solution. Thus updaters work with the most recent version of that bucket. However, for a deleter to get to the point where it has all the locks its needs can be somewhat involved if the target bucket is the "1" partner of a potential merge. The deleter must release its lock on the target bucket, place a lock on the "0" partner, and then re-lock the "1" partner.

While this is taking place, other update operations may be affecting these buckets. In particular, a concurrent insertion could add new records to the target bucket once the deleter's lock is released so that it is not longer empty enough to allow merging. It is even theoretically possible for a stream of inserters to fill up the target bucket and cause a split, thereby moving the key that is to be deleted. In addition, another deleter might get the two partners locked and merged before the deleter we are focusing on does. Each of these conditions is checked for and the pitfalls avoided. After gaining the lock on the "0" partner, the deleter checks whether merging might be possible (the partner's next link points to the target bucket), and if this check fails, it goes back to simply trying to remove its key. If the two buckets are not linked in this way, it may mean the localdepths do not match or that the target bucket has been deleted. Attempting to lock the target bucket under these circumstances would carry with it the danger of deadlock. Upon finding the two buckets directly linked and re-locking the "1" partner, the deleter checks the emptiness of the bucket, whether the desired key is still there, and whether localdepths still match before going ahead with the merge. Unless the key has moved, the deleter at this point would have the needed locks and no further interference could occur at the bucket level.

Processes executing the find operation may legitimately see either an old or the new version of the target bucket. No intermediate states are visible (i.e. adding or removing a key is a single put operation, splitting is equivalent to a single put, and merging is protected with ξ-locks) Differences between old and new only involve records that are moved to a reachable bucket or that are the subject of a concurrent update operation. Note that lockout is possible for all processes while they are trying to get the right set of buckets locked.

3. Use with Distributed Data

We have presented two approaches to solving the problem of allowing concurrency within a shared extendible hash file. Now we turn to the problem of distributing this information. Developing a distributed solution raises a number of issues; although some are unique to this particular model of computation, the aspect of achieving a degree of concurrency is common to both distributed and shared data systems. Thus a *correct* centralized solution may prove to be a good starting point in determining how to partition structured data. We can assess the previous algorithms on the basis of their potential for distribution.

First it must be clear what is meant by the phrase "distributing the data structure" and what our model of a distributed system is. We assume there are a number of processes each encapsulating some portion of the data structure (i.e. the entire directory or whole buckets) and acting as a manager for it. Certain pieces of the data structure may be replicated in several processes. Processes do not share storage (including secondary storage) and they communicate through asynchronous messages. The style of message-passing used in our protocol depends on reliable delivery, buffering, and possible anonymity of senders (e.g. port-based communication as in [Rashid 80].) These assumptions allow the processes to reside on different machines connected by a network, and since this is possible, interactions between processes are potentially costly. Requests for find, insert, or delete operations may be forwarded to the appropriate data managers for service.

There are a couple of principles influencing this particular design. First of all, if distributing the data is actually going to achieve an increased level of availability, the directory should be highly accessible. This suggests the need to replicate the directory information and maintain consistency to the extent that a request can be made to any of the copies and eventually it will reach the desired data. We assume that each copy of the directory is managed as a whole (i.e. it is not partitioned). Given the decision to replicate this component of the data structure, the consistency issue becomes important. If α- or ξ-locking the directory in the centralized solutions is straightforwardly translated into some action involving all copies simultaneously, it will be an expensive operation and require some strategies for avoiding deadlock and dealing with temporarily missing copies. Thus, the analogue to global α-locking should be avoided as much as possible; implying that the second of the two previous solutions is more compatible with replication. Although a number of general purpose mutual consistency algorithms are available [Gifford 79, Stonebraker 79, Thomas 79], it may be possible to exploit certain properties of this problem to arrive at a less synchronized method. A second goal is to minimize message traffic. Whenever possible, the information needed for decision-making should be available locally. Additional modifications in the data structure may be desirable. For example, in the centralized algorithms it was acceptable to locate a partner bucket using the directory. In the distributed case, this would involve a bucket manager sending an inquiry message to a directory manager. Finally, there are no constraints to be put on the placement of data. One can imagine policies that would try to group certain buckets within one server. This is reasonable for a static data structure. However, ease of growth is a major goal both for extendible hash files and for distributing data. The problem of allocating buckets to servers on any basis other than availability of space is a hard problem for a dynamic data structure such as this and is not considered here.

As indicated above, this distributed solution is derived fron the second set of procedures for the centralized hash file. The replication of the directory is the main justification for choosing this approach. The data structure would now appear as in figure 10. Two copies of the directory are shown. A *prev* link has been added to each bucket that leads to the bucket from which this bucket originally split off. This is used to find the "0" partner of a possible merge with information local to this manager process. Each link represents a pair consisting of a long-lived identifier for a manager port and a bucket address that is meaningful to that manager. A *version* field introduced into each bucket and each directory entry is used in updating directory copies asynchronously.

There are two types of processes, namely directory managers and bucket managers. Each bucket manager is responsible for a disjoint subset of the buckets. Figure 11 shows the message types that flow between the various processes. The information contained in these messages is outlined in figure 12.

The procedure for the directory manager processes (see figure 13) is described in terms of actions taken in response to messages received. The directory manager is designed as a server which can keep track of several user requests. The locking of the directory in the centralized solution is embodied in the manager's explicit scheduling of requests for its attention. Upon receiving a *request* message, state is saved in a context table and the request is forwarded to the appropriate bucket

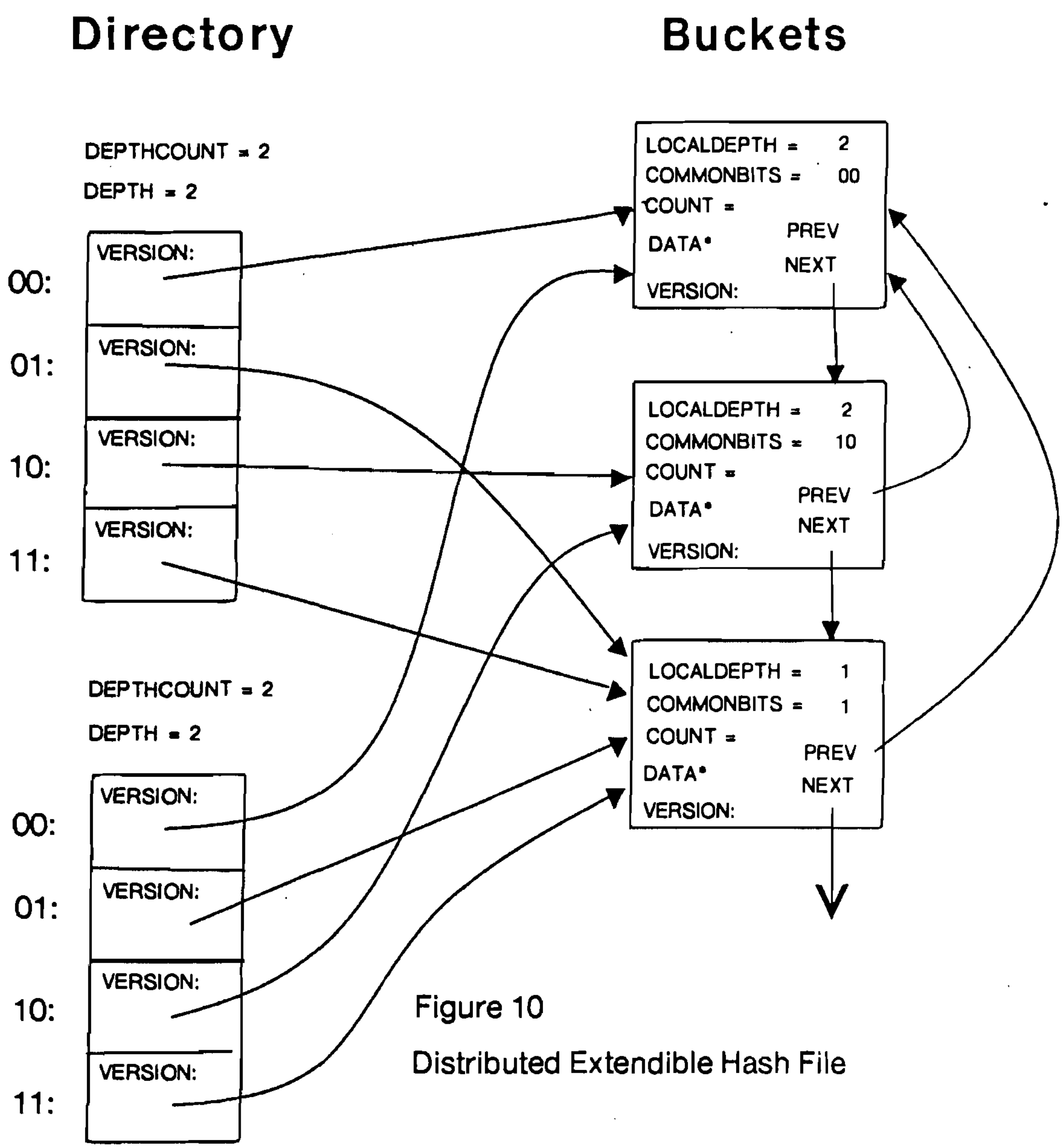

Figure 10

Distributed Extendible Hash File

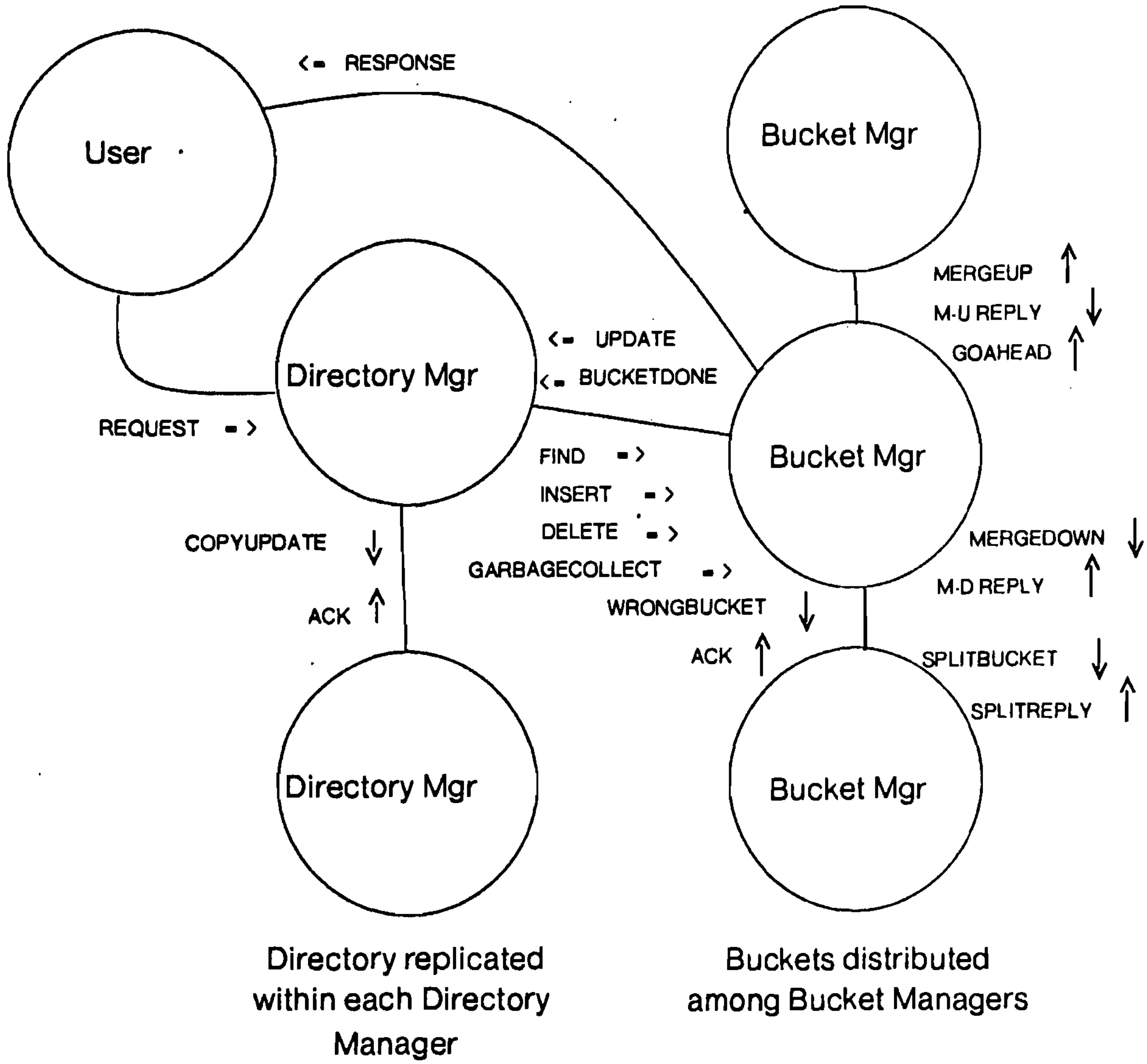

Figure 11

Protocols for the Distributed Hashing Algorithms

message id	data in message
Request	desired key op: (find|insert|delete) user's port
Bucketdone	transaction # success: (true|false)
Update	transaction # old local depth version # of "0" partner version # of "1" partner new page address id of bucket manager success: (true|false)
Copy update	op: (insert|delete) pseudo key old local depth version # of "0" partner version # of "1" partner new page address id of bucket manager acknowledgement port
Ack for Copy update	
Find, Insert, Delete	desired key transaction # page address user's port directory manager's reply port pseudo key
Garbage Collect	list of page addresses

message id	data in message
Wrongbucket	op: (find|insert|delete) desired key transaction # page address user's port directory manager's reply port pseudo key bucket manager's reply port
Ack for Wrongbucket	
Splitbucket	manager's reply port buffer contents of new half
Splitreply	new page address id of bucket manager
Mergedown	partner's address local depth bucket manager's reply port
M.D. Reply	buffer contents success: (true|false)
Mergeup	partner's address bucket manager's reply port target bucket's address bucket manager's id
M.U. Reply	local depth version # bucket manager's reply port success: (true|false)
Go ahead	next link next bucket manager id version # success: (true|false)

Figure 12 Messages

manager. Two possible responses may come from a bucket manager, either *bucketdone* or *update.* Bucketdone will generally signify that no directory modifications are needed and the directory manager may now forget about this request. An update message schedules an update on the local copy according to version number and notifies all other directory managers by broadcasting a *copyupdate* message. For each outstanding unacknowledged remote directory modification, a counter is incremented that serves one of the purposes of an α-lock (i.e. preventing garbage collection). A bucket may not be deallocated until all directories send an *acknowledge* message. Upon receiving a copyupdate message, a directory manager schedules the update on its local copy and when the changes have been applied (and in the case of delete operations, when the equivalent of ξ-locking occurs), acknowledgements are sent.

Because obsolete directory information is usable, the multiple copy update does not have to be strictly synchronized (in the sense of an atomic transaction). However, the ordering of different directory modifications due to operations on the *same bucket* should be the same across all copies and determined by the order in which the bucket operations are performed. Each bucket contains a version number that increases with each update that causes a directory update. The version number in each directory entry should match the version of the bucket it points to when the directory is completely up to date. The following example illustrates why this ordering approach is adopted. Suppose first a split operation is performed almost immediately followed by a merge involving those two buckets. Imagine a directory manager that hears about these updates in the opposite order and applies them. The directory update related to the merge would essentially have no effect since the split had not yet been processed. The subsequent update related to the split would result in directory entries leading to a deleted bucket. At this point the directory is usable since next links provide recovery. However, since it appears that both messages have been serviced, the deleted bucket could then be deallocated. This would leave that copy of the directory in a truly incorrect state from which recovery would be impossible.

For simplicity, the bucket manager is presented here as a front end process and a set of associated processes that are assumed to reside at the same site and share secondary memory. These processes taken together perform the duties of the bucket manager and preserve the specified interface with other processes. The code for these processes is given in figure 14. The front end process serves as the initial contact for its set of buckets. The auxiliary processes operate much like processes in the centralized solution until they require pieces of the data structure that are outside this manager's domain. We have already discussed the directory update messages. Protocols are also available for off-site searching (*wrongbucket* message), merging (*mergeup* and *mergedown* messages), and splitting (*splitbucket* message). Taking off-site actions and the need to exchange messages into account, the procedures are not radically different from those in the centralized solution.

In this report, we just suggest what the proof of correctness would require. Given the correctness of the centralized algorithm, one approach is to show that the distributed implementation is in some sense equivalent. By following an execution of a user's request through the various processes that become involved and comparing this with the steps taken by the one process handling that request in a centralized system, the correspondence between execution sequences can be seen. This needs to be formalized. In addition, it is necessary to show that the multiple copy update strategy applied to the replicated directories is correct. We must also demonstrate that the multiplexing of servers and the message flows between them do not introduce deadlock. Crash tolerance has not been specifically addressed but our solution does not appear to present major obstacles to incorporating it. These issues will be elaborated upon in a future paper.

4. Summary

Extendible hash files have been proposed as a data structure for sequential find, insert, and delete operations. In this report, we have presented two solutions that allow concurrent operations on a slightly modified structure. As in proposals for concurrency in B-tree variants, making modifications to the data structure to provide alternate pathways to the desired data is a fundamental technique. In a future paper, we will evaluate the performance of these algorithms and comparable B-tree solutions.

Starting from one of these solutions for concurrency in a centralized hash file, we developed a distributed version. The important point is that concurrent algorithms involving shared storage may often provide insights into how to partition and/or replicate data. This suggests a methodology in which the problems of correctly introducing concurrency and of distributing the computation are addresses as distinct issues.

Acknowledgments

I would like to thank Jurg Nievergelt for stimulating this work.

Figure 13 Pseudocode for Directory Manager

```
while ( true )  {
     messageid = GetMessage (&msg);  /* Either receives a message or takes a
             message off the list of delayed but now ready directory updates. */
     switch (messageid)  {
         case request: {
                pseudokey = hash (msg.key);
                transaction# = SaveState (msg, pseudokey); /* Multiplexing requests-
                           remember data related to this one - context of a request
                           consists of operation, key, userport, and pseudokey */
                rho = rho + 1;
                indexdirectory (pseudokey & mask (depth), &oldpage, &bucketmgr);
                bucketport = namelookup (bucketmgr);
                messageid = msg.op;
                ContactBucket(bucketport, messageid,  msg.key, transaction#,
                             oldpage, msg.userport, myreplyport, pseudokey);
                             /* construct a Find, Insert, or Delete message and
                                send it to the appropriate bucket manager */
         }
         case bucketdone: {
                RestoreState (msg.transaction#); /* Recall context for this request */
                if (!msg.success &&  operation = delete) {
                     indexdirectory (pseudokey & mask (depth), &oldpage, &bucketmgr);
                     bucketport = namelookup (bucketmgr);
                     messageid =  operation;
                     ContactBucket (bucketport, messageid, key, msg.transaction#,
                                    oldpage,  userport, myreplyport,  pseudokey);
                }
                else {
                     rho = rho - 1;
                     CleanState (msg.transaction#); /* forget about this request */
                }
         }
         case update:  {
                RestoreState (msg.transaction#);
                broadcast(operation, pseudokey, msg.oldlocaldepth, msg.version#1, msg.version#2,
                          msg.newpage, msg.bucketmgr, myackport);
                            /*  sends a copyupdate message to all other directory managers and
                                increments alpha for each outstanding directory update  */
                if (VersionsDoNotMatch()) save (msg); /* Delay this directory update
                              until its time */
                else {
                     if ( operation == insert) {
                            if (msg.oldlocaldepth == depth) doubledirectory;
                            updatedirectory ( msg.newpage, msg.bucketmgr, ++msg.version#1,
                                      ++msg.oldlocaldepth, pseudokey);
                            if (!msg.success) {
                                   indexdirectory (pseudokey & mask (depth), &oldpage, &bucketmgr);
                                   bucketport = namelookup (bucketmgr);
                                   messageid = context -> op;
                                   ContactBucket (bucketport, messageid, key, msg.transaction#,
                                             oldpage, userport, myreplyport,  pseudokey);
                            }
                            else {
                                  rho = rho - 1;
                                  CleanState (msg.transaction#);
                            }
                     }
                     else { /* op = delete */
                            if (msg.oldlocaldepth == depth) depthcount = depthcount -  2;
                            RememberDeleted(); /* Keep track of deleted buckets for
                                   the eventual garbage collection phase */
                            if depthcount == 0) halvedirectory();
                            else updatedirectory ( msg.newpage, msg.bucketmgr, msg.oldlocaldepth,
                                   max(msg.version#1, msg.version#2) + 1, pseudokey);
                     }
                     ReleaseSaved(); /* If finishing this directory update enables
                            previously delayed ones, make them accessible to GetMessage */
                }
         }
```

Figure 13 Pseudocode for Directory Manager.

```
        case copyupdate: {
              if (VersionsDoNotMatch()) save (msg);
              else {
                   if (msg.op == insert) {
                         if (msg.oldlocaldepth == depth) doubledirectory;
                         updatedirectory ( msg.newpage, msg.bucketmgr, ++msg.version#1,
                                 ++msg.oldlocaldepth, msg.pseudokey);
                         SendAck(msg.ackport); /* respond to directory manager
                                 who initiated this update */
                   }
                   else { /* op = delete */
                         if (msg.oldlocaldepth == depth) depthcount = depthcount -  2;
                         if depthcount == 0) halvedirectory();
                         else updatedirectory ( msg.newpage, msg.bucketmgr, msg.oldlocaldepth,
                                 max(msg.version#1, msg.version#2) + 1, pseudokey);
                         RememberAck(msg.ackport); /* save up acks until the
                                 equivalent of xi-locking occurs */
                   }
                   ReleaseSaved();
              }
        }
        case ack : alpha = alpha - 1;
    }
    if (!rho) SendRememberedAcks();
    if (!rho && !alpha) GarbageCollect();
}
```

5. References

[Bayer 77] R. Bayer and M. Schkolnick
"Concurrency of Operations on B-trees",
ACTA Informatica, 9, 1977, 1-21.

[Ellis 80] C. Ellis
"Concurrent Search and Insertion in 2-3 Trees,"
ACTA Informatica 14, 1980, 63-86.

[Fagin 79] R. Fagin, J. Nievergelt, N. Pippenger, and H.R. Strong,
"Extendible Hashing - A Fast Access Method for Dynamic Files,"
ACM TODS, Vol. 4, No. 3, September, 1979, 315-355.

[Gifford 79] D. Gifford,
"Weighted Voting for Replicated Data",
Proceedings, 7th Symposium on OS Principles, December 1979.

[Kwong 79] Y.S. Kwong and D. Wood,
"New Method for Concurrency in B-trees",
IEEE Transactions on Software Engineering, Vol. SE-8, No. 3, May 1982.

[Lehman 81] P. Lehman and S.B. Yao
"Efficient Locking for Concurrent Operations on B-Trees",
ACM TODS, Vol. 6, No. 4, December 1981, 650-670.

[Miller 78] R. Miller and L. Snyder,
"Multiple Access to B-trees",
Proc. Conf. Information Sciences & Systems
(preliminary report) March 1978.

[Rashid 80] R. Rashid,
"An Interprocess Communication Facility for UNIX,"
CMU-CS-80-124, Carnegie-Mellon University, June 1980.

[Stonebraker 79] M. Stonebraker,
"Concurrency Control and Consistency of Multiple Copies of Data in Distributed INGRES",
IEEE Transactions on Software Engineering, Vol. SE-5, No. 3, May 1979.

[Thomas 79] R. H. Thomas,
"A Majority Consensus Approach to Concurrency Control for Multiple Copy Databases",
ACM TODS, Vol. 4, No. 2, July 1979, 180-209.

Figure 14 Pseudocode for Bucket Managers

```
Bucket Manager Front End Process:

while (true) {
    messageid = receivemessage (&msg);
    if (messageid == splitbucket) {
        newpage = allocbucket(); /* assumes available page */
        putbucket (newpage, msg.half2);
        SendSplitReply (msg.replyport, newpage, myid);
    }
    else {
        p = createprocess (bucketslave);
        forward (msg, p);
    }
}



Bucket Slave Process:

messageid = receivemessage (&msg);
if (messageid == wrongbucket) sw = msg.op;
else sw = messageid;
switch (sw) {
    case find: {
        onmachine = true;
        oldpage = msg.page;
        RhoLock (oldpage);
        if (messageid == wrongbucket) SendAck (msg.buckmgrport);
        else SendBucketdone (msg.dirmgrport, msg.transaction#, success = true);
        getbucket (oldpage, current);
        m = mask (current -> localdepth);
        while ((m & msg.pseudokey) !=
            current -> commonbits && onmachine) {/* wrong bucket */
            newpage = current -> next;
            machine = current -> nextmgr;
            if (machine != me) {
                SendWrongbucket (namelookup (machine), op = find, msg.key, msg.transaction#,
                          newpage, msg.userport, msg.dirmgrport, msg.pseudokey, myreplyport);
                onmachine = false;
            }
            else {
                RhoLock (newpage);
                getbucket (newpage, current);
                m = mask (current -> localdepth);
                UnRhoLock (oldpage);
                oldpage = newpage;
            }
        }
        if (onmachine) {
            if (search (current, msg.key))/* is key there? */
                found (msg.key);
            else
                notfound (msg.key);
        }
        else receivemessage (&msg); /* Wrongbucket reply */
        UnRhoLock (oldpage);
    }
    case insert: {
        onmachine = true;
        oldpage = msg.page;
        AlphaLock (oldpage);
        if (messageid == wrongbucket) SendAck(msg.buckmgrport);
        getbucket (oldpage, current);
        m = mask (current -> localdepth);
        while ((m & msg.pseudokey) != current -> commonbits && onmachine) {/* WRONG BUCKET */
            newpage = current -> next;
            machine = current ->nextmgr;
```

Figure 14 Pseudocode for Bucket Managers

```
        if (machine != me) {
            SendWrongbucket (namelookup (machine), op = insert, msg.key, msg.transaction#
                    newpage, msg.userport, msg.dirmgrport, msg.pseudokey, myreplyport);
            onmachine = false;
        }
        else {
            AlphaLock (newpage);
            getbucket (newpage, current);
            m = mask (current -> localdepth);
            UnAlphaLock (oldpage);
            oldpage = newpage;
        }
    }
    if (!onmachine) {
        receivemessage (&msg); /* Wrongbucket reply */
        UnAlphaLock (oldpage);
    }
    else {
        if ( search (current, msg.key)) { /* IS KEY ALREADY THERE? */
            SendBucketdone (msg.dirmgrport, msg.transaction#, success = true);
            UnAlphaLock (oldpage);
        }
        else if (current -> count != numentries) {
                        /* CURRENT BUCKET NOT FULL */
            SendBucketdone (msg.dirmgrport, msg.transaction#, success = true);
            add (current,  msg.key);
            putbucket (oldpage, current);
            UnAlphaLock (oldpage);
        }
        else {                      /* CURRENT IS FULL - DIRECTORY WILL BE AFFECTED */
            directorymgr = msg.dirmgrport;
            trans# = msg.transaction#;
            oldlocaldepth = current -> localdepth;
            success = split (current, half1, half2, msg.key);
            if (AvailablePages()) {
                newpage = allocbucket ();
                machine = myid;
                putbucket (newpage, half2);
            }
            else {
                SendSplitbucket(MgrWithSpace(), myreplyport, half2);
                receivemessage (&msg); /* split bucket reply */
                machine = msg.bucketmgr;
                newpage = msg.page;
            }
            half1 -> next = newpage;
            half1 -> nextmgr = machine;
            putbucket (oldpage, half1);
            UnAlphaLock (oldpage);
            SendUpdate (directorymgr, oldlocaldepth, trans#, newpage, machine, success,
                    half1 -> version#, half2 -> version#);
        }
    }
}
case delete: {
    /* Find the right bucket as in the beginning of insert except place XiLocks */
    if (!onmachine) {
        receivemessage (&msg); /* Wrongbucket ack */
        UnXiLock (oldpage);
    }
    else {
        if ((current -> count > 1) || (current -> localdepth == 1) ||
            (!search(current, msg.key)) { /* CURRENT NOT TOO EMPTY */
            SendBucketdone(msg.dirmgrport, msg.transaction#, success = true);
            if (remove (msg.key, current)) putbucket (oldpage, current);
            UnXiLock (oldpage);
        }
```

Figure 14 Pseudocode for Bucket Managers

```
else {
    m = leftshift(1, current -> localdepth - 1);
    if ((msg.pseudokey & m) != m) {/* MSG.KEY IN FIRST OF PAIR */
        newpage = current -> next;
        machine = current -> nextmgr;
        if (machine = me) {
            /* Try to merge on site as in figure 9 */
        }
        else {
            directorymgr = msg.dirmgrport;
            trans# = msg.transaction#;
            version#1 = current -> version#;
            z = msg.key;
            SendMergedown (namelookup (machine), newpage, current -> localdepth,
                          myreplyport);
            receivemessage (&msg); /* MD Reply */
            if (msg.success) {
                current = msg.buffer;
                oldlocaldepth = current -> localdepth;
                version#2 = current -> version#;
                current -> version# = max(version#1, version#2) + 1;
                current -> commonbits = current -> commonbits &
                                        mask (--current -> localdepth);
                putbucket (oldpage, current);
                SendUpdate(directorymgr, trans#, oldlocaldepth, version#1, version#2,
                          oldpage, myid, success = true);
            }
            else {
                SendBucketdone (directorymgr, trans#, success = true);
                if (remove (z, current)) putbucket (oldpage, current);
            }
            UnXiLock (oldpage);
        }
    }
    else { /* MSG.KEY IN SECOND OF PAIR */
        newpage = current -> prev;
        machine = current -> prevmgr;
        UnXiLock (oldpage);
        if (machine == me) {
            /* Try to merge on site as in figure 9 */
        }
        else {
            directorymgr = msg.dirmgrport;
            z = msg.key;
            trans# = msg.transaction#;
            pseudokey = msg.pseudokey;
            SendMergeup(namelookup (machine), newpage, oldpage, myid, myreplyport);
            receivemessage (&msg); /* MU Reply */
            if (!msg.success) {
                SendBucketdone(directorymgr, trans#, success = true);
                if (remove (z, current)) putbucket (oldpage, current);
            }
            else {
                XiLock (oldpage);
                getbucket (oldpage, current);
                if ((mask (current -> localdepth) & pseudokey) != current -> commonbits) {
                    UnXiLock (oldpage);
                    success = false;
                    SendGoahead(msg.replyport, success);
                    SendBucketdone (directorymgr, trans#, success);
                }
                else if (current -> localdepth != msg.localdepth ||
                    current -> count > 1 || (current -> count == 1 &&
                    !search (current, z))){
                        SendBucketdone(directorymgr, trans#, success = true);
                        if (remove(z, current)) putbucket (oldpage, current);
                        UnXiLock (oldpage);
                        success = false;
                        SendGoahead(msg.replyport, success);
                }
```

Figure 14 Pseudocode for Bucket Managers

```
                                        else {
                                            version#1 = msg.version#;
                                            version#2 = current -> version#;
                                            current->commonbits = deleted;
                                            success = true;
                                            SendGoahead(msg.replyport, current -> next, current -> nextmgr,
                                                    max (version#1, version#2) + 1, success);
                                            current -> next = current -> prev;
                                            current -> nextmgr = current -> prevmgr;
                                            putbucket (oldpage, current);
                                            SendUpdate(directorymgr, current -> localdepth, current -> next,
                                              current -> nextmgr, success, version#1, version#2);

                                        }
                                  }
                              }
                          }
                      }
                  }
    }
    case mergedown: {
        newpage = msg.partner;
        XiLock (newpage);
        getbucket (newpage, brother);
        success = brother -> localdepth == msg.localdepth;
        SendMDReply(msg.buckmgrport, brother, success);
        if (success) {
           brother -> commonbits = deleted;
           brother -> next = brother -> prev;
           brother -> nextmgr = brother -> prevmgr;
           putbucket (newpage, brother);
        }
        UnXiLock (newpage);
    }
    case mergeup: {
        newpage = msg.partner;
        XiLock (newpage);
        getbucket (newpage, brother);
        success = (brother -> next == msg.target) && (brother -> nextmgr == msg.managerid);
        SendMUReply(msg.buckmgrport, brother -> localdepth, myreplyport, success, brother -> version#);
        if (success) {
            receivemessage (&msg); /* GoAhead */
            if (msg.success) {
                brother -> commonbits = brother -> commonbits & mask (--brother -> localdepth);
                brother ->next = msg.next;
                brother -> nextmgr = msg.nextmgr;
                brother -> version# = msg.version#;
                putbucket ( newpage, brother);
            }
        }
        UnXiLock (newpage)
    }
    case garbagecollect: {
        for each page in msg.list {
            XiLock (page);
            deallocate (page);
            UnXiLock (page);
        }
    }
}
```