Concurrency and Linear Hashing

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

TR151
March, 1985

Abstract

Hashing has long been recognized as a fast method for accessing records by key in large relatively static databases. However, when the amount of data is likely to grow significantly, traditional hashing suffers from performance degradation and may eventually require rehashing all the records into a larger space. Recently, a number of techniques for dynamic hashing have appeared. In this paper, we present a solution to allow for concurrency in linear hash files that is based on locking protocols and minor modifications in the data structure. The problem of adapting this technique for use in a distributed system is also addressed.

The preparation of this paper was supported in part by the National Science Foundation, under grant No. MCS-8203396.

1. Introduction

Hashing has long been recognized as a fast method for accessing records by key in large relatively static databases. However, when the amount of data is likely to grow significantly, traditional hashing suffers from performance degradation and may eventually require rehashing all the records into a larger space. Consequently, dynamic tree structures such as B-trees have been used as database indices. Recently, a number of techniques for dynamic hashing have appeared. These include Extendible Hashing [Fagin 79], Linear Hashing [Litwin 80], Exponential Hashing [Lomet 83], and Dynamic Hashing [Larson 78]. In this paper, we present a solution to allow for concurrency in linear hash files that is based on locking protocols and minor modifications in the data structure.

There have been many algorithms proposed to allow concurrent access to other dynamic search structures such as B-trees [Bayer 77, Ellis 80a, Lehman 81, Kwong 82, Miller 78] and binary search trees [Kung 80, Ellis 80b, Manber 82]. Concurrency in various dynamic hash structures is beginning to receive attention. We have developed solutions for concurrent operations in extendible hash files [Ellis 83]. A recent report [Shasha 84] specifies concurrent algorithms for several dynamic search structures including Lomet's exponential hashing [Lomet 83]. That solution uses techniques similar to those found in the algorithms described here. Several simple solutions for linear hashing have also been proposed [Wu 83]. These algorithms are primarily based on 2-phase locking. In the most interesting of these, an update operation consists of a non-2-phase searching phase followed by a modifying phase that does employ 2-phase locking. Changes occurring between phases that could affect the update operation result in its rollback. As an alternative to rolling back and restarting the entire operation, our solution handles the effects of concurrent updates by taking a minor detour.

2. Locking Protocols for Linear Hashing

The solution described in this paper is an adaptation of the approach presented in [Ellis 83] for concurrency in extendible hashing. Linear hashing and extendible hashing are both intended for files that grow and shrink dynamically, but there are several significant differences. Linear hashing does not require a directory component as in extendible hashing. Also, the splitting of buckets is cyclical in linear hashing rather than occurring on whatever bucket fills.

There are a couple of rules that can be used to decide when to split or merge in the linear hashing approach. One possibility is to attempt to maintain a constant storage utilization. Another is to split the next bucket in the cycle whenever any bucket overflows. These are called controlled and uncontrolled splits respectively. In developing a solution that allows concurrency, it is necessary to specify which rule is to be used because the different information requirements may call for different synchronization. The solution given below is based on the uncontrolled approach.

In linear hashing, the hash function to be applied changes as the file grows or shrinks. There is the function, $h_0 : c \rightarrow \{0, 1, \dots, N-1\}$, initially used to load the file and a sequence of functions $h_1, h_2, \dots, h_i, \dots$ such that for any c either $h_i(c) = h_{i-1}(c)$

$$\text{or } h_i(c) = h_{i-1}(c) + 2^{i-1}N.$$

When a key is to be inserted, the appropriate function is used to find the target bucket. Collisions (i.e. attempts to insert into a full bucket) are handled by creating a chain of overflow buckets. If a collision occurs, a split is performed on the bucket that is next in line to be split (a pointer, *next*, indicates which bucket this is). A variable, *level*, is used to determine the appropriate hash function using the following procedure:

```

bucket ← hlevel(key)

if bucket < next then bucket ← hlevel+1(key)

```

Splitting causes these variables to be updated as follows:

```

next ← (next + 1) mod (N*2level)

if next = 0 then level ← level + 1.

```

Deletion may result in merging buckets, moving *next* back, and re-adjusting *level*. Figure 1 shows a linear hash table before and after an insertion that triggers a splitting operation. Here $h_{\text{level}}(\text{key})$ is $\text{key} \bmod 2^{\text{level}} * N$ where ($N = 2$).

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. Locks are used to control access to the shared variables, *level* and *next*, which are denoted as the *toplevel* and to the bucket chains. The primary bucket and all of its overflow buckets are locked as a unit.

The compatibility of lock types is given by the following table.

<u>Lock request</u>		<u>Existing lock</u>		
		read-lock	selective-lock	exclusive-lock
read-lock	yes		yes	no
selective-lock	yes		no	no
exclusive-lock	no		no	no

Organization of the keys within a chain becomes important when insertion and deletion can occur in parallel with searching. With multiple disk pages making up the chain, reading the chain is not an indivisible step. If keys are kept ordered within a chain, one insert can affect every page and care must be taken that intermediate states are not visible while the chain is being rewritten. All we actually require is that a single disk write makes visible the reorganized chain resulting from inserting or deleting one record or splitting a chain. This can be implemented in

Figure 1 Sequential Linear Hashing

several ways. One approach is to build a new chain of overflow buckets on disk and then to replace the primary bucket with new contents including a pointer to the new chain. This last disk write makes the new chain available. However, the old chain cannot be immediately destroyed since a reader may still be using it. Disk pages removed from the official chain can be remembered and deallocated later in a separate phase that briefly places an exclusive lock on the chain to ensure that readers have finished using the old version. Although simpler approaches can be designed for single record updates, the substantial amount of restructuring caused by a split justifies such an effort.

Concurrency is enhanced by allowing searching processes to operate in parallel with a split operation, but there must be some means of recovery when the wrong chain is reached because of out-of-date toplevel values. In this scheme, each chain includes an additional field, *local_level*, that specifies the hash function appropriate to that chain. Note that this is entirely redundant information when no concurrent restructuring operations are in progress; however, storing it in the primary bucket ensures that the searching process can decide if it has the right chain without requiring the accuracy of the shared toplevel variables. The modified data structure is shown in Figure 2. Upon gaining access to a bucket, a process checks whether the *level* value used to calculate the address matches *local_level*, and if not, it increments its value and recalculates the address until a match is found. The chains reached in this manner are those that were created by splitting from chains at addresses already accessed during this search. Merging on the other hand does not permit concurrent use of the toplevel variables or the partner buckets of the merge. The algorithms, written in a C-like syntax [Kernighan 78], are presented in Figures 3, 4, and 5.

Showing the correctness of this solution requires demonstrating 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 record 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 a well-defined ordering on the lockable components of the structure. Specifically, the toplevel is locked first, then one bucket chain addressed $h_1(z)$ is locked possibly followed by successively higher addressed chains related to the original chain through the sequence of hash functions. Each address calculated in the rehashing loop is greater than or equal to the current bucket and a lock is requested only when the new address is greater.

Next consider potential interference with a searching process. A find operation must correctly detect the presence or absence of any key that is not the subject of a concurrent update. The first step is to find the correct bucket chain. Then it is necessary to transfer the information into private buffers to be searched.

Finding the target chain depends on the correctness of the rehashing scheme. Merging and splitting operations must be considered for possible interactions with the search. The merge operation excludes all other processes from the toplevel

Figure 2 Concurrent Linear Hashing

Shared data for the linear hashing algorithms:

```
#define toplevel lockable abstraction representing next and level
#define N some constant

struct buffer {
    int locallevel;
    int count;
    int link;
    int data[numentries]};
int level, next;

find(z)
{
    int    l, /* local idea of level */
           bucketchain, /* index of chain */
           previous; /* index of chain */

    struct buffer B, *current;
    current = &B;

    ReadLock (toplevel);
    l = level;
    bucketchain = hash(l, z);
    if (bucketchain < next) {
        l = l + 1;
        bucketchain = hash(l, z);
    }
    ReadLock (bucketchain);
    UnReadLock (toplevel);
    getchain (bucketchain, current); /* reads the
    primary bucket of bucketchain from disk into current buffer */
    while (current->locallevel != l) { /* wrong bucket */
        l = l + 1;
        previous = bucketchain;
        bucketchain = hash(l, z);
        if (bucketchain != previous) {
            ReadLock (bucketchain);
            UnReadLock (previous);
            getchain (bucketchain, current);
        }
    }
    if (search (current, z)) /* searches current buffer for z;
    reads subsequent buckets of chain as necessary */
        found (z);
    else
        notfound (z);
    UnReadLock (bucketchain);
}
```

Figure 3 Find Algorithm


```

insert(z)
{
    int l,
        bucketchain,
        previous;
    struct buffer A,
        *current;
    current = &A;
    /*-----Search Phase-----*/
    ReadLock (toplevel);
    l = level;
    bucketchain = hash(l, z);
    if ( bucketchain < next ) {
        l = l + 1;
        bucketchain = hash(l, z);
    }
    SelectiveLock (bucketchain);
    UnReadLock (toplevel);
    getchain (bucketchain, current);
    while (current -> locallevel != l) { /* wrong bucket */
        l = l + 1;
        previous = bucketchain;
        bucketchain = hash(l, z);
        if ( bucketchain != previous ) {
            SelectiveLock (bucketchain);
            UnSelectiveLock (previous);
            getchain (bucketchain, current);
        }
    }
    /*-----*/
    overflow = AddRecord (current, bucketchain, z);
    /*"atomically" update bucketchain; clean up any garbage generated;
    if chained buckets are required, return true; otherwise, return false*/
    UnSelectiveLock (bucketchain);
    if (overflow) Split();
}

Split()
{
    int last;
    list deleted;
    struct buffer A, B, C,
        *chain1, *chain2, *original;
    chain1 = &A; chain2 = &B; original = &C;

    SelectiveLock(toplevel);
    SelectiveLock(next);
    getchain (next, original);
    deleted =
        ConstructChains( chain1, chain2, original);
    /*rehash into two new chains; write overflow buckets to disk;
    return list of garbage buckets for later removal */
    putchain (next + M * 2 ** level, chain2);
    /*write primary bucket to disk */
    putchain (next, chain1); /*write primary bucket to disk */
    last = next;
    next = (next + 1) mod (M * 2 ** level);
    if (next == 0) level = level + 1;
    UnSelectiveLock(toplevel);
    UnSelectiveLock(last);
    if (deleted != nil) {
        /*ensure that readers with obsolete information have cleared
        out; then it is safe to deallocate without holding locks */
        ExclusiveLock(last);
        UnExclusiveLock(last);
        Deallocate (deleted);
    }
}

```

Figure 4 Insertion Algorithm

```

delete(z)
{
    int    l,
           bucketchain,
           previous;
    struct buffer A,
               *current;
    current = &A;
    /*-----*/
    /*          Search Phase same as in Insert          */
    /*          See figure 3                             */
    /*-----*/
    underflow = RemoveRecord (current, bucketchain, z);
    /* "atomically" update bucketchain;
    return true if chain becomes "too empty"; false otherwise */
    UnSelectiveLock (bucketchain);
    if (underflow) Merge();
}

Merge()
{
    int partner, last;
    struct buffer A, B, C,
               *chain1, *chain2,
               *newchain;

    chain1 = &A;
    chain2 = &B;
    newchain = &C;

    ExclusiveLock(toplevel);
    if (next == 0) level = level - 1;
    next = (next - 1) mod (N * 2 ** level);
    last = next;
    ExclusiveLock(next);
    partner = next + N * 2 ** level;
    ExclusiveLock(partner);
    UnExclusiveLock(toplevel);
    getchain (last, chain1);
    getchain (partner, chain2);
    MergeChains (chain1, chain2, newchain);
    /* build one chain; write any overflow buckets to disk;
    deallocate garbage buckets */
    putchain (last, newchain);
    /* write primary bucket to disk */
    UnExclusiveLock(partner);
    UnExclusiveLock(last);
}

```

Figure 5 Deletion Algorithm

variables and the chains being merged. The lock-coupling protocol employed (i.e. holding a lock on one component of the structure until the next component has been successfully locked) ensures that no attempt is made to access a deleted bucket. Thus a merge cannot interfere with any other actions. The split operation does not require exclusive access because it is possible for other processes to recover from obsolete toplevel information. The hash function initially used depends on the values of the toplevel variables seen. If this information is obsolete because of a concurrent split, the calculated address will be less than or equal to the true address. The rehash loop calculates in turn the chain that should contain the target key for each restructuring pass starting with the round indicated by the toplevel values seen and continuing through the round indicated by the value of *local_level* contained in the target bucket.

The two new chains resulting from a split appear atomically to other processes because of the order in which they are written to disk. When the primary bucket at the head of the chain that split is replaced, all the information from the original chain is still available by hashing to that chain's address and rehashing with the next hash function if necessary (as indicated by *local_level* in the bucket itself). After the reorganized chains are in place, the toplevel information is changed to allow direct calculation of the address of the new chain. Here also, the order of changing *next* and *level* matters so that seeing incorrect values gives the view of a smaller valid address space than actually exists. For example, consider a change in the split routine such that *level* is incremented before *next* gets reset to zero. It could appear to a searching process that the next round of splitting is already half done and thus invalid addresses could be generated. The order given in the algorithm allows a reader to see a new value of *next* with an old value of *level* resulting in a rehash.

Given that the desired chain has been reached, the data read must represent a valid state of the chain. Processes executing a find operation may share a chain with an update because of the lock compatibilities. Such a reader is considered correct if it sees either the old or new contents of the bucket chain. By atomically replacing a chain in the AddRecord and RemoveRecord functions, we satisfy this requirement. Searching for the key as part of a deletion or the place to insert as part of an insertion requires that the effects of previous updates (even those still active) be seen. The selective-lock placed on the chain serializes writers so that only up-to-date information is used and so there is no interference between concurrent writers.

Finally, we must consider interference among updating processes that could result in losing the effects of supposedly completed updates or producing an improperly structured hash file. Merges and splits are serialized by incompatible locks on the toplevel and all affected bucket chains for the duration of the step. Simple deletes and inserts (i.e. those involving the removal or addition of one record in a chain without triggering restructuring) may execute in parallel on different chains but are excluded from the same chain by selective-locks. Because the uncontrolled splitting strategy is being used, restructuring decisions are not affected by actions taken by other updates between the time that it is determined that restructuring is called for and the split or merge routine actually executed. Therefore the complete release of locks between the single key update (i.e. AddRecord, RemoveRecord) and the structuring operations (i.e. Split, Merge) is acceptable.

Note that there are alternative ways of handling the split and merge

Note that there are alternative ways of handling the split and merge operations. Decoupling these restructuring actions from the requested insert and delete operations and doing them as a separate background activity would be easy with these algorithms. Modifying the solution for controlled splitting would involve requiring each operation that changes the data structure to update shared utilization data atomically.

3. Linear Hashing in a Distributed System

The discussion so far has focused on allowing concurrent operations on a shared centralized hash file. Now we turn to the problem of using linear hashing in a distributed system. In order to achieve the possible benefits of distribution such as improved availability and ease of growth, the data structure itself must be spread over a number of communicating machines with the possibility that some pieces may be replicated.

For our purposes, a distributed system is viewed as a number of logical processors communicating solely through asynchronous messages. There is no memory (including secondary storage) shared among these logical processors. A logical processor may encapsulate multiple processes that execute on a single physical processor. The notion of distributing a data structure in such a system is that the data structure may be divided into disjoint portions assigned to a number of logical processors which serve as managers for the pieces they contain or some parts of the structure may be replicated in several managers for increased availability. Requests for find, insert, or delete operations may be forwarded to the appropriate data managers for service. Specifically, in linear hashing, each manager may be responsible for some number of adjacent bucket chains. In addition, the toplevel information must be maintained by at least one manager.

The absence of a directory in the linear hashing scheme might be seen as an advantage in developing a distributed solution with high availability as a major objective. However, the lack of built-in indirection requires that a naming convention be adopted to give the appearance of a network-wide address space appropriate for direct calculation of bucket locations. In addition, it is desirable for availability that replication of (at least) the toplevel variables be provided. These variables encode a global property of the distributed structure (i.e. where the next restructuring must occur) and the question arises as to how accurately copies of them must be maintained. Since, at any point in time, there is exactly one place in the data structure that is eligible for a split or merge (i.e. the two neighboring bucket chains, *next* and its predecessor with respect to the current *level*), these restructuring operations must occur serially and must correctly locate the target buckets.

A naive adaptation of the concurrent linear hashing solution just presented would use system-wide locking of the toplevel variables for serialization and propagation of the true values of *level* and *next* to all copies. This solution should be adequate when the replication factor is small, the sites holding copies are reliable, and few of the requested operations are updates.

An alternative approach would be to maintain a primary copy with accurate values of the toplevel data and allow other copies to contain out-of-date information.

Search, insert, and delete operations could use any copy by interpreting the values seen only as hints. Split and merge operations would require the primary copy to find the true *next* chain for restructuring. The primary copy would also serve as the lockable entity for serialization of these restructuring operations.

The algorithm presented in detail below is an elaboration upon the primary copy scheme in which the primary copy of the toplevel data moves to reside at the same processor as the *next* bucket chain. Under the assumption that the other copies are generally correct so that the *next* bucket and the primary copy can be located in one try, this approach involves fewer sites and less communication.

The first problem to be solved has to do with the use of toplevel hints by search, insert, and delete operations. This information takes the form of potentially old values for *level* and *next* that may not reflect recent restructuring activity. The solution in the previous section provides recovery from concurrent splitting operations. In order to avoid the need for global locking of toplevel variables in the distributed and replicated case, it becomes necessary to allow recovery when toplevel values that have been made obsolete by a merge operation are used to generate the address of a deleted chain. The key to this recovery is the ability to detect an attempt to lock a nonexistent bucket (from using the hash function h_i) and to rehash (using h_{i-1}). Each valid bucket chain is represented by an active manager. For a bucket that has been deleted, its manager may or may not have been terminated at the time of the attempted access. In the first case, we depend upon an inter-process communication mechanism that returns an error code on a send primitive directed to a nonexistent destination. In the second case, the manager can respond appropriately. Possible responses include returning a reply message indicating that the bucket has been deleted (which would then be handled like a failed send) or, if enough information comes with the request, doing the recovery calculation there. The IPC functionality described is a substitute for explicitly keeping information about deallocated chains temporarily available until all copies of toplevel variables and all affected operations in progress know that the merge occurred. Thus, we have the ability to cope with out of date toplevel data and updates to copies can be propagated asynchronously.

The next major issue is how to serialize restructuring operations. The solution is simply that split and merge operations must place a selective-lock on the primary copy of the toplevel data. In our version, there is the additional problem of finding the primary copy which can be identified by the presence of a *token* that gets passed when the designated *next* bucket chain moves. Any copy of the toplevel can serve as an initial hint for the location of the token. Since the copies are expected to be generally up-to-date, the search should be short. However, if the copy reached is not the primary copy, the values there represent a better hint and the search can proceed. The reason each try yields a better hint is that when the token is passed to make a new primary copy, the old one has been updated.

The final problem concerns the updating of toplevel copies. Each copy includes a *version number*. The manager of the current primary copy holds the up-to-date version number. Each split and merge operation increments the official version number. The manager of the old primary copy sends notifications with the new

toplevel values and version number to all copies. In addition, topLevel values can piggyback on other messages between managers. Upon receiving any topLevel information, the manager can compare the incoming version number with its own and incorporate the update if appropriate. As evident in the discussion above, the token passing procedure alone bounds how far out of date a copy can get.

Figure 6 illustrates the distributed linear hashfile structure. In this figure, sites or logical processors are represented by ellipses that contain one copy of topLevel information and some number of neighboring bucket chains. The information contained in various messages is outlined in Figure 7. The pseudocode for a manager is given in Figure 8. The procedures used to create slaves are in Figure 9. In order to simplify the presentation, each manager is responsible for only one bucket chain. A port-based asynchronous message-passing scheme (e.g. [Rashid 80]) is used. Crash tolerance and recovery have not been specifically addressed here. This is the next logical step in the development of these algorithms.

4. Summary

Linear hashing is a technique for allowing adjustment in the range of the hashing function as the amount of data stored grows and shrinks. In this paper, we have presented a solution for concurrent access to a shared centralized linear hashfile. The fundamental method for achieving greater concurrency is to provide a way to detect and recover from the effect of concurrent updates. This idea, which appears in earlier work on B-trees and extendible hashing as additional pointers in the data structure, carries over to linear hashing in a new form (i.e. recalculation).

We have also investigated the implications of using linear hashing in a distributed system. The solution sketched here depends on the provision of a network-wide address space and takes recovery from the use of incorrect information one step further. This approach is especially important in distributed environments where it is generally impractical to base algorithms on knowledge of the true instantaneous global state of the system.

Figure 6 Distributed Linear Hashing

<u>message id</u>	<u>data in message</u>	<u>message id</u>	<u>data in message</u>
user request	z (desired key) operation (find, insert, delete) user's reply port	lock	type (read, exclusive, selective) slaves' reply port operation z user's reply port l [bucket chain]
forwarded_request	z operation (find, insert, delete, merge) user's reply port l (index of hash function most recently used) next level } piggybacked version } [Bucket chain addr (only useful if manager has > 1 bucket)]	locksuccess	
die	partner's reply port [bucket chain addr]	unlock	type [bucket chain]
die reply	contents of chain	you_are_it	level version reply port [next]
new chain	next level version [bucket chain addr] contents of chain partner's reply port	you_are_it ack	reply port
new chain ack		count	count
init	master's reply port z user's reply port l next level version	copy update	version level next
		split	
		merge	
		splitdone	
		mergedone	

Figure 7 Messages

Figure 8 Pseudocode for Managers

Notation

C-like statements;

English-like pseudocode statements;

*/*comments*/*

```

/*To simplify presentation, assume each manager is responsible for
only one bucketchain and local copy of the set of toplevel variables;
comes into existence as a result of a split operation*/
initialize; /*publicize existence (make public port);
  receive "newchain" message from partner containing copy of toplevel data
  and contents of chain; putchain to disk; send "newchain ack"*/
while ( alive ) {
  messageid = ReceiveMessage (&inmsg);
  switch (messageid) {
    case userrequest: { /* from user directed to any publicly
      announced manager port*/
      l = level;
      bucketchain = hash (l, inmsg.z);
      if (bucketchain < next){
        l = l + 1;
        bucketchain = hash (l, inmsg.z);
      }
      while ( failure = ForwardRequest(bucketchain, outmsg)){
        /*Construct and send a "forwarded_request" message
        (returns true if send primitive fails; piggy back toplevel values;
        assumes desired data are not local*/
        l = l - 1;
        bucketchain = hash (l, inmsg.z);
      }
    }
    case die: { /*dies as a result of merge operation - message from partner*/
      Deallocate public port; /*so unreceived messages get failure signals*/
      For each transaction, t, on lock queue {
        do {
          state[t].l = state[t].l - 1;
          bucketchain = hash (state[t].l, state[t].z);
        } while( failure = ForwardRequest(bucketchain, outmsg);
      }
      alive = false;
    }
    case forwarded_request: { /*find, insert, delete, or merge ops*/
      update version of toplevel if needed;
      slaveport = Createprocess(inmsg.op);
      Send "init" message to slave;
    }
  }
}

```

```

case lock: {
    if (lock request satisfies compatibility rules)
        send "locksucces" message to slave's reply port;
    else enqueue transaction;
}
case unlock: {
    release lock - wake up all transactions possible
    by sending "locksucces" messages;
}
case copyupdate: {
    update toplevel values if incoming version greater;
}
case split: { /*Someone believes this mgr to be token holder*/
    if (token){
        if (selectivelock) count = count + 1;
        else {
            selectivelock = true;
            slaveport = Createprocess(split);
            Send "init" message;
        }
    }
    else Send "split" message to next's manager;
}
case merge: {
    if (token){
        if (selectivelock) count = count - 1;
        else {
            selectivelock = true;
            Construct and send a "forwarded_request"
            message to manager of  $(next - 1) \bmod (N * 2^{**} level)$ ;
        }
    }
    else Send "merge" message to next's manager;
}
case splitdone: {
    version = version + 1;
    next =  $(next + 1) \bmod (N * 2^{**} level)$ ;
    if (next == 0) level = level + 1;
    Send "you_are_it" message to new next manager;
}
case mergedone: {
    version = version + 1;
    if (next == 0) level = level - 1;
    next =  $(next - 1) \bmod (N * 2^{**} level)$ ;
    Send "you_are_it" message to new next manager;
}
case you_are_it ack: {
    broadcast "copyupdate" message;
    token = false;
    selective = false;
    Send "count" message;
}

```

```

case you_are_it: {
    Update toplevel values;
    selective = true;
    count = 0;
    token = true;
    Send "you_are_it ack" to inmsg.replyport;
}
case count: {
    count = count + inmsg.count;
    selective = false;
    if (count > 0) {
        count = count - 1;
        do same actions as in split case;
    }
    else if (count < 0) {
        count = count + 1;
        do same actions as in merge case;
    }
}
}
while (any of my slaves exist)
    Receive and process "lock" and "unlock" messages;
    getchain(bucketchain, current);
    Send contents of chain to partner's replyport as "diereply" message;
}

```

Figure 9 Slave Procedures

```

find()
{
    Receivemessage (&inmsg);
    Send to master a "lock" request for read-lock;
    receive "locksucces" reply;
    /*shorthand for this message exchange: ReadLock (bucketchain)*/
    getchain (bucketchain, current); /*reads the primary bucket of bucketchain
    from disk into current buffer. Here, bucketchain is the address of the only
    chain this manager is responsible for. If > 1, bucketchain would come from inmsg. */
    if ( current->locallevel != inmsg.l) { /* wrong bucket */
        inmsg.l = inmsg.l + 1;
        previous = bucketchain;
        bucketchain = hash(inmsg.l, inmsg.z);
        if (bucketchain != previous) {
            Send "forwarded_request" message to bucketchain's mgr;
        }
    }
    else {
        if (search (current, inmsg.z)) /* searches current buffer for z;
        reads subsequent buckets of chain as necessary */
            found ();
        else
            notfound ();
        Respond to user's reply port;
    }
    UnReadLock (bucketchain); /*Represents sending "unlock" message to master*/
}

```

```

insert()
{
    /*-----Search Phase-----*/
    ReceiveMessage (&inmsg);
    SelectiveLock (bucketchain);
    getchain (bucketchain, current);
    if (current->locallevel != inmsg.l){ /* wrong bucket */
        inmsg.l = inmsg.l + 1;
        previous = bucketchain;
        bucketchain = hash(inmsg.l, inmsg.z);
        if ( bucketchain != previous) {
            Send "forwarded_request";
        }
    }
    /*-----*/
    else {
        overflow = AddRecord (current, bucketchain, z);
        /*"atomically" update bucketchain: clean up any garbage generated;
        if chained buckets are required, return true; otherwise, return false*/
        Respond to user's reply port;
        if (overflow) Send "Split" to next's mgr;
    }
    UnSelectiveLock (bucketchain);
}

```

```

Split()
{
    ReceiveMessage (&inmsg);
    SelectiveLock(next);
    getchain (next, original);
    deleted = ConstructChains( chain1, chain2, original);
    /*rehash into two new chains; write overflow buckets to disk;
    return list of garbage buckets for later removal */
    Create manager at next + N * 2 ** level;
    /*Implemented by message to remote process manager*/
    Receive child's port;
    Send "newchain" message;
    Receive "newchain ack"; /* means chain2 is in place*/
    putchain (next, chain1); /*write primary bucket to disk */
    last = next;
    Send "Splitdone" to master;
    UnSelectiveLock(last);
    if (deleted != nil) {
        /*ensure that readers with obsolete information have cleared
        out; then it is safe to deallocate without holding locks */
        ExclusiveLock(last);
        UnExclusiveLock(last);
        Deallocate (deleted);
    }
}

delete()
{
    /*----- */
    /*          Search Phase same as in Insert          */
    /*----- */
    else {
        underflow = RemoveRecord (current, bucketchain, inmsg.z);
        /* "atomically" update bucketchain;
        return true if chain becomes "too empty"; false otherwise */
        Respond to user's reply port;
        if (underflow) Send "Merge" to next's mgr;
    }
    UnSelectiveLock (bucketchain);
}

Merge()
{
    ReceiveMessage (&inmsg);
    last = address of this mgr's bucketchain; /*(next - 1) mod (N * 2 ** level)*/
    ExclusiveLock(last);
    partner = last + N * 2 ** level;
    Send "die" to partner;
    Send "Mergedone" to manager of token; /*port was in "forwarded_request" as user's reply port*/
    getchain (last, chain1);
    Receive "diereply" containing chain2;
    MergeChains( chain1, chain2, newchain);
    /*build one chain; write any overflow buckets to disk;
    deallocate garbage buckets */
    putchain (last, newchain);
    /*write primary bucket to disk */
    UnExclusiveLock(last);
}

```

5. References

- [Bayer 77] R. Bayer and M. Schkolnick,
"Concurrency of Operations on B-trees,"
ACTA Informatica, 9, 1977, 1-21.
- [Ellis 83] C. Ellis,
"Extendible Hashing for Concurrent Operations and Distributed Data,"
Proceedings, 2nd SIGACT-SIGMOD Symp. on Principles of Database Systems, Atlanta, March 1983.
- [Ellis 80a] C. Ellis,
"Concurrent Search and Insertion in 2-3 Trees,"
ACTA Informatica 14, 1980, 63-86.
- [Ellis 80b] C. Ellis,
"Conncurrent Search and Insertion in AVL Trees,"
IEEE Trans. on Computers, Vol. C-29, No. 9, Sept. 1980, 811-817.
- [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.
- [Kernighan 78] B. Kernighan and D. Ritchie,
The C Programming Language,
Prentice-Hall, 1978.
- [Kung 80] H.T. Kung and P.L. Lehman,
"Concurrent Manipulation of Binary Search Trees,"
ACM TODS, Vol. 5, No. 3, 1980, 354-382.
- [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.
- [Larson 78] P. Larson,
"Dynamic Hashing,"
BIT, 17, 1978, 184-201.
- [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.
- [Litwin 80] W. Litwin,
"Linear Hashing: A New Tool for File and Table Addressing,"
Proceedings, 6th Conf. on Very Large Data Bases, 1980.
- [Lomet 83] D. Lomet,
"Bounded Index Exponential Hashing,"

ACM TODS, Vol. 8, No. 1, March 1983, 136-165.

- [Manber 82] U. Manber and R.E. Ladner,
"Concurrency Control in a Dynamic Search Structure,"
Proceedings, ACM SIGACT-SIGMOD Symp. on Principles of Database Systems, March 1982.
- [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.
- [Shasha 84] D. Shasha,
"How to Design a Concurrent Algorithm for your Favorite Search Structure,"
Aiken Computation Lab, Harvard University, TR-13-84, June 1984.
- [Wu 83] C.T. Wu and W.A. Burkhard,
"Concurrency in Linear and Interpolation Hashing,"
unpublished manuscript, Northwestern University, 1983.