Concurrency in Linear Hashing

CARLA SCHLATTER ELLIS
University of Rochester

Concurrent access to complex shared data structures, particularly structures useful as database indices, has long been of interest in the database community. In dynamic databases, tree structures such as B-trees have been used as indices because of their ability to handle growth; whereas hashing has been used for fast access in relatively static databases. Recently, a number of techniques for dynamic hashing have appeared. They address the major deficiency of traditional hashing when applied to databases that experience significant change in the amount of data being stored. This paper presents a solution that allows concurrency in one of these dynamic hashing data structures, namely linear hashfiles. The solution is based on locking protocols and minor modifications in the data structures.

Categories and Subject Descriptors: D.4.1 [**Operating Systems**]: Process Management—*concurrency*; *deadlocks*; *multiprocessing*; *synchronization*; E.1 [**Data Structures**]: H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*

General Terms: Algorithms

Additional Key Words and Phrases: data structures, dynamic hashing, concurrency control, concurrent algorithms, locking protocols

1. INTRODUCTION

Concurrent access to complex shared data structures, particularly structures useful as database indices, has long been of interest in the database community. The need to provide multiple users with acceptable response times has motivated such work even in the more traditional computing environments. With the increasing availability of truly parallel architectures, the techniques developed for specialized concurrency control in database structures may be transferred to a wider range of data structures in a variety of parallel application areas.

In dynamic databases, tree structures, such as B-trees, have been used as indices because of their ability to handle growth. Recently, a number of techniques for dynamic hashing have appeared. They include Extendible Hashing [7], Linear Hashing [12], Exponential Hashing [13], and Dynamic Hashing [10]. These methods address a major deficiency of traditional hashing when it is applied to

The preparation of this paper was supported in part by the National Science Foundation under grant number IST-8025761. A preliminary report on this research appears in the *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems* (Portland, Ore., March 1985), pp. 1–7.

Author's current address: Department of Computer Science, Duke University, Durham, NC 27706.

databases that experience significant change in the amount of data being stored. In this paper, we present a solution, based on locking protocols and minor modifications in the data structure, that allows concurrency in linear hashfiles.

There have been many algorithms proposed for the allowance of concurrent access to other dynamic search structures such as B-trees [1, 2, 9, 11, 15, 16] and binary search trees [3, 8, 14]. Concurrency in various dynamic hash structures is beginning to receive attention. A preliminary version of the solution being described here was presented in [5]. Solutions for concurrent access in extendible hashfiles have also been developed [4, 6]. A recent report [17] specifies concurrent algorithms for several dynamic search structures including Lomet's exponential hashing [13]. That solution uses techniques similar to those found in our algorithms. Several simple solutions for linear hashing have also been proposed [18]. These algorithms are primarily based on two-phase locking.

The solutions described here apply several techniques that have been used previously in various tree structures. The linear hashfile represents a different type of data structure from those of the earlier concurrency studies. In particular, it is not a linked structure. The solution for concurrent linear hashing requires a rethinking and generalization of those familiar techniques.

In the next section, the sequential version of linear hashing is described. The concurrent algorithms are developed in Section 3. In Section 4, correctness arguments are given. Finally, Section 5 summarizes the work.

2. SEQUENTIAL ALGORITHM

The basic sequential algorithm assumes a contiguous logical address space of *primary buckets* each capable of holding some number b of records. Collisions (i.e., attempts to insert into a full primary bucket) are handled by creating a chain of *overflow buckets* associated with that particular bucket address.

The hash function to be applied changes as the file grows or shrinks. Each new hash function assigns new bucket addresses to some of the records previously placed using the old function. This new hash function is applied to one bucket chain at a time in the linear ordering. The resulting modification in the data structure is called a *split* and moves some records from the original bucket to a new primary bucket that is appended at the current end of the hashfile. Splitting serves to reduce the accumulation of overflow chains, which degrade performance, by linearly increasing the address space of primary buckets. The split operation is applied cyclically.

To be specific, there is the function h_0: $k \rightarrow \{0, 1, \ldots, N-1\}$, initially used to load the file and a sequence of functions $h_1, h_2, \ldots, h_i, \ldots$, such that for any key value k, either $h_i(k) = h_{i-1}(k)$ or $h_i(k) = h_{i-1}(k) + 2^{i-1}N$. If a split is called for, it is performed on the bucket that is next in line to be split (a pointer *next* indicates which bucket this is). For each pass with a new hash function h_j, the next pointer travels from bucket 0 to $2^{j-1}N$. A variable *level* is used to determine the appropriate hash function for find, insert, or delete operations using the following procedure:

```
bucket ← $h_{\text{level}}$(key)
**if** bucket < next **then** bucket ← $h_{\text{level}+1}$(key)
```

Splitting causes these variables to be updated as follows:

next $\leftarrow$ (next + 1) mod(N*2^{level})
if next = 0 **then** level $\leftarrow$ level + 1.

There are a couple of rules that can be used to decide when to split in the linear hashing approach. One possibility is to attempt to maintain an approximately constant storage utilization. In this approach, a split is done only when the load factor exceeds some threshold. The other policy 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 because it is simpler. Later, we indicate what is required for the more popular controlled splitting.

Deletion of records may result in merging buckets, moving next back, and readjusting level. In the controlled approach, merging is performed when overall space utilization falls below some threshold. In the uncontrolled approach, emptying an individual primary bucket by a delete operation triggers a merge.

Figure 1 shows a linear hash table before and after an insertion that triggers a splitting operation. Here h_{level} (key) is key mod $2^{\text{level}}N$ where $N = 2$. In this example, the key to be inserted is 42. To find the target bucket, one applies the hash function indicated by the value of level ($h_1(42) = 2$) and compares the result with the next pointer to determine whether the hash function h_2 should be used instead. The value of next is 0, indicating that h_1 is the appropriate function. The record for key 42 is added to the bucket chain at bucket address 2 after checking that it is not already there. This causes the creation of an overflow bucket. Under the uncontrolled splitting policy, the overflow triggers a split operation. All the records in the bucket pointed to by the next pointer, bucket 0, are rehashed using h_2. This eliminates the overflow bucket associated with bucket 0 and creates a new primary bucket at address 4. The next pointer is advanced to bucket 1. Suppose now that a find operation is requested for key 20. Calculating h_{level} yields $h_1(20) = 0$, but now 0 is less than next, and the target bucket address is recalculated using h_2 (20 mod 8). The desired record is found when bucket 4 is searched.

3. CONCURRENT SOLUTION

Our goal is to allow a high degree of concurrency among processes executing *find*, *insert*, and *delete* operations on a shared linear hashfile. We discuss the parallel behavior of this solution in terms of its five major procedures. The user's requests to search or modify the set of keys stored in the hashfile correspond to the routines **find**, **insert**, and **delete**. The procedures **Split** and **Merge** are concerned with the restructuring of the hashfile. Since 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 is actually executed. Owing to this independence, the restructuring operations can be viewed as separate operations in spite of the fact that they are called from the procedures **insert** and **delete**.

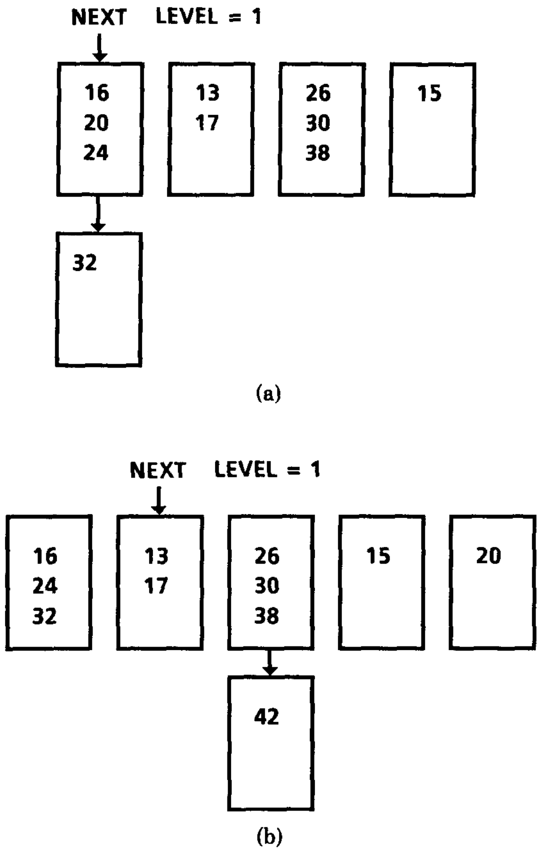

Fig. 1. Sequential linear hashing (a) before inserting 42 into bucket 2; (b) after inserting into bucket 2 and splitting bucket 0.

In our solution, the find operation can be performed concurrently with other processes executing the procedures **find, insert, delete,** and **Split.** Processes executing the **insert** and **delete** procedures may operate in parallel if they are working on different bucket chains. A split may be performed in parallel with insert and delete operations that are not accessing the particular chain being split. The interaction between a merge and processes doing find, insert, or delete operations is more complicated. Those processes may not access the two buckets being merged and may not read the values of level and next while the merging process is using them. At most, one restructuring operation can be executing at any time.

The most interesting interaction described above involves the concurrency allowed between the split operation and that phase of find, insert, and delete in which the target bucket is being located. The basic strategy is to permit the use of potentially obsolete information to access a bucket initially and then, if it turns out to be the wrong bucket, to follow a remedial path through related buckets.

Table I

	Existing lock		
Lock Request	Read lock	Selective lock	Exclusive lock
Read lock	yes	yes	no
Selective lock	yes	no	no
Exclusive lock	no	no	no

Locks are used to control access to the shared variables level and next, which are referred to as *root* variables, and to the bucket chains. The primary bucket and all its overflow buckets are locked as a unit. The compatibility of lock types is given by Table I.

The find algorithm calls for the use of *lock-coupled* read locks. Lock-coupling is a particular flow of locking in which the next component (assuming some ordering) is locked before releasing the lock on the current component. The procedures **insert** and **delete** read-lock root and selective-lock buckets with lock coupling. The split operation uses selective locks. Exclusive locks are used for merging chains and deallocating old overflow buckets.

In our model of computation, a number of actions are assumed to be inherently atomic. These include the lock and unlock operations. Reading or writing a single shared variable, such as level or next, is also considered an indivisible step. Each bucket of a chain occupies a disk page, and the data are transferred into private buffers for processing. The operations of reading or writing a single physical disk page are also assumed to be atomic.

Organization of the keys within a chain becomes important when insertion and deletion can occur in parallel with searching. When multiple disk pages make up the chain, reading the chain is not an indivisible step. If keys are kept ordered within a chain, one insertion 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 procedure can be implemented in 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 it is possible for a reader to still be using it. Disk pages removed from the official chain can be remembered and deallocated later (e.g., in a separate phase). In our presentation, the removal of old overflow buckets is done in a procedure called **GarbageCollect**. This procedure involves briefly acquiring an exclusive lock on the affected chain to ensure that readers have finished using the old version before the buckets are subject to deallocation. In the **Merge** routine, garbage buckets are directly deallocated (embedded in a procedure, **MergeChains**), since the process doing the merge already holds exclusive locks on the chains.

Concurrency is enhanced by allowing a searching process to operate in parallel with a split operation, but there must be some means for it to reorient itself when the wrong chain is reached because of out-of-date root values. In this scheme,

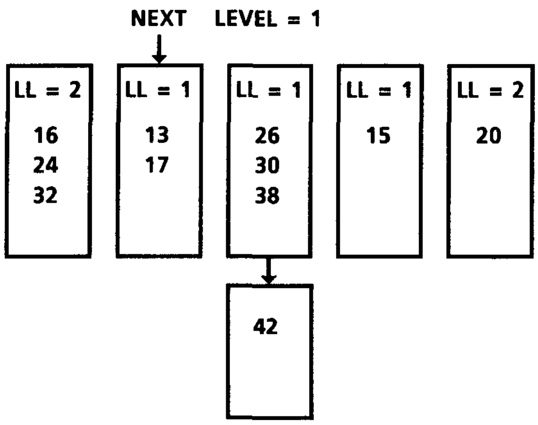

Fig. 2. Concurrent linear hashing.

each chain includes an additional field *locallevel* that specifies the hash function appropriate to that bucket chain. This field captures the fact that in the most recent split affecting this bucket (not inverted by a subsequent merge), it was the hash function $h_{\text{locallevel}}$ that was used to divide up the keys. Basically, the locallevel value characterizes the set of key values that can belong in this chain. Note that locallevel is entirely redundant information (with level and next), when no concurrent restructuring operations are in progress. In that case, all bucket chains with addresses less than the value of next have locallevel = level + 1. Buckets with addresses greater than or equal to $2^{\text{level}}N$ also have values of locallevel = level + 1. All of the buckets in between have locallevel = level, which indicates that they have not effectively been split in the current round of restructuring. If there is a process actively doing a split operation, the two reorganized buckets may already be in place while the old root values are still being used. Storing locallevel in the primary bucket ensures that the searching process can decide if it has the right chain without requiring the accuracy of the shared root variables. The modified data structure is shown in Figure 2.

The intuition behind the search phase of the find, insert, and delete algorithms (in a procedure called **LocateAndLockChain**) is the key to understanding this solution. A process executing in its locate phase behaves as follows: The root variables are read, and the values seen determine which hash function is to be used initially. Let the private variable *lev* record the hash function applied, namely h_{lev}. So, the first value of lev is the value of level at the time the root variables were read. Upon gaining access to a bucket, the process checks whether lev matches that bucket's locallevel, and if not, it increments its lev value and recalculates the address $h_{\text{lev}}(k)$ until a match is found. This is referred to as the rehashing loop. It essentially reenacts the history of splits that have affected the placement of the target key. In other words, the rehashing loop calculates in turn each bucket chain that should have contained the target key in each restructuring pass. This may mean revisiting the same bucket chain in successive iterations. The rehashing loop starts with the round indicated by the root values seen and continues through the round indicated by the value of locallevel contained in the target bucket. Note that this approach is somewhat analogous to the use of the next links in [4] and [11], but in a very different form.

The calculated address at each iteration of the rehashing loop will always be less than or equal to the address of the eventual destination. This is a direct consequence of the definition of the sequence of hash functions (i.e., keys moved from a bucket b during a split using h_i go to the higher addressed chain $b + 2^{i-1}N$). Thus, the bucket chain in which the desired key belongs should be reachable using this rehashing strategy as long as each address calculated is within the valid address space at the time of access.

The first problem is to ensure that a process does not prematurely try to access a new chain being created by a concurrent split operation. The two new chains resulting from a split appear atomically to other processes because of the order in which they are written to disk. Specifically, the chain at the new bucket address is written before the new version replaces the chain at the target bucket address. At this point, no information contained in the hashfile points to the existence of this new bucket. Once the primary bucket at the head of the chain at the target address has been written, its locallevel value indicates that the bucket has split and a new bucket has been incorporated into the data structure. All the information from the original chain is still available through the rehashing scheme. After the reorganized chains are safely in place, the root information is changed to allow direct calculation of the address of the new chain. Here also the order of changing next and level matters because seeing incorrect values must give the view of a smaller valid address space than actually exists. The order given in the algorithm allows a reader to see a new value of next with an old value of level, possibly resulting in a rehash. By contrast, 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 second aspect to ensuring valid addressing is that a concurrent merge operation should not make a calculated address invalid. The lock-coupling protocol used in the locate phase prevents interference by merging processes. While reading level and next for the initial address calculation, searching processes place a read lock on root and hold the lock until a lock (read lock for find, selective lock for insert or delete) is placed on that bucket chain. If rehashing is called for, a lock is placed on the subsequent bucket before the lock is released on the current wrong bucket. A process responsible for merging two buckets holds exclusive locks on root and both partners of the merge, while it makes its changes. The read lock on root held by the searching process prevents a merge from decreasing the size of the address space (by updating level and next) during the initial bucket access. If it appears that another iteration of the rehashing loop is necessary, the next address calculated must be valid, because a lock is still being held on the bucket that would be its partner in any merge operation that could cause the new destination to disappear. In such a situation, the merging process would need the incompatible exclusive lock on the partner.

During the locate phase of **find, insert,** and **delete,** locks are placed according to a well-defined ordering. Merges and splits also respect the ordering in requesting their locks. Thus, deadlock cannot occur.

Searching for the key as part of a deletion, or for the place to insert as part of an insertion, requires that the effects of previous updates (even those still active) be seen. Selective locks are placed on the chains during the locate phase to serialize writers of the same individual buckets so that only up-to-date information is seen. This guarantees that two changes being performed in private buffers are not based on the same version of the bucket chain, and that two successive put operations to the same primary bucket will not lose the effect of an update. So, there is no interference between concurrent executions of **insert** and **delete.**

Merges and splits are completely serialized with respect to one another by incompatible locks on the root. All affected bucket chains are also locked by a restructuring process for the duration of the step. The **Split** procedure allows, because of its selective lock, only processes executing the **find** routine to concurrently access the chain being split. The exclusive locks held during a merging process do not permit any concurrent use of the partner buckets of the merge or the root variables until they have been changed. After the root variables have been changed to reflect the smaller hashfile that will result from the merge, the lock on root is converted to a selective lock, and processes entering their locate phase may then concurrently access root.

The pseudocode procedures are presented in Figures 3, 4, and 5.

An example of a parallel computation involving three requests is given in Figure 6. The vertical columns of text give the steps executed by each of the three processes. The horizontal alignment of these steps indicates when concurrent execution is assumed. Gaps show when delays occur (possibly caused by blocking on lock requests). The hashfile shown in Figure 2 is assumed to be the initial state. Figure 7 shows the updated data structure at designated points in this computation. The notation SL_I means that Process I (the inserter) holds a selective lock on that bucket. Similarly, RL_F is a read lock held by Process F.

This example illustrates several of the important interactions between processes. At the beginning of the computation, the process attempting to delete 15 and the process inserting 40 can concurrently locate and modify the data in their respective target bucket chains. Each process places a selective lock, but since they are working with different parts of the data structure, they do not conflict. Both of these updates trigger a restructuring. In the second phase of the computation, the merge and the split operations are serialized by the incompatible locks required on root. Process I manages to acquire its selective lock on root first and performs the split while process D is blocked. Meanwhile, process F starts executing the find operation in parallel with the split and involving the same buckets. This demonstrates the rehashing strategy that is fundamental in the locate phase. The first address calculated using the value of level (1) read by process F is for bucket chain 1. Process F reads the primary bucket (i.e., **getchain(1, current)**) after the newly split version has been written by process I. Seeing a locallevel value of 2, process F rehashes and calculates the address 5. It places a read lock on bucket chain 5 and releases the lock on bucket chain 1 (lock-coupling). In the final part of the computation, Process I has finished, allowing process D to get its lock on root. Now, it is process F with its read locks, first on bucket 1 and then on 5, that delays process D in acquiring exclusive locks on these two partners of the merge.

```
Shared data for the linear hashing algorithms:
root is a lockable abstraction representing next and level
const N = some constant
type buffer = record
  local level,
  count : integer;
  link : AddrofOverflow;
  data[numentries] : integer
end;
var level, next : integer;

     procedure find(k : integer);
       var bucketchain : integer;   /* index of chain */
         B : buffer;
         current : ↑buffer;
     begin
       current := addressof (B);
       LocateAndLockChain (bucketchain, current, k, Read);
       /* last getchain is decisive action for find */
       if (search (current, k) is successful)
       /* searches current buffer for k;
       reads subsequent buckets of chain as necessary */
      then found (k);
      else notfound (k);
      UnLock (Read, bucketchain)
    end

    procedure LocateAndLockChain (var bucketchain : integer,
      var current : ↑buffer, k : integer, locktype : Read or Selective);
      var lev, previous : integer;
    begin
      Lock (Read, root);
      lev := level;
      bucketchain := hash(lev, k);
      if (bucketchain < next) then
      begin
        lev := lev + 1;
        bucketchain := hash(lev, k)
      end;
      Lock (locktype, bucketchain);
      UnLock (Read, root);
      getchain (bucketchain, current);
      /* Getchain reads the primary bucket of bucketchain from disk into current buffer */
      while (current↑ . locallevel not = lev) do/* wrong bucket */
      begin
        lev := lev + 1;
        previous := bucketchain;
        bucketchain := hash(lev, k);
        if (bucketchain not = previous) then
        begin
          Lock (locktype, bucketchain);
          UnLock (locktype, previous);
          getchain (bucketchain, current)
        end
      end
    end
```

Fig. 3. Searching: Algorithm for the find operation and procedure for locate phase.

```
     procedure insert (k : integer);
       var bucketchain : integer;
         A : buffer;
         current : ↑buffer;
         overflow : Boolean;
         deleted : list;  /* of removed overflow buckets */
1    begin
2      current := addressof(A);
3      LocateAndLockChain (bucketchain, current, k, Selective);
4      overflow := AddRecord (current, bucketchain, k, deleted);
       /* AddRecord "atomically" updates bucketchain :
       it reorganizes the data in current and existing overflow buckets,
       writes out any new overflow buckets needed,
       builds a list (deleted) of old overflow buckets to be removed,
       and then writes the primary bucket of bucketchain
       (Case a of structural consistency proof and the decisive action for insert).
       If chained buckets are required, AddRecord returns true; otherwise, returns false. */
5      UnLock (Selective, bucketchain);
6      if (overflow) then Split;
7      GarbageCollect (bucketchain, deleted)   /* Case b */
8    end

     procedure delete (k : integer);
       var bucketchain : integer;
         A : buffer;
         current : ↑buffer;
         deleted : list;
         underflow : Boolean;
1    begin
2      current := addressof (A);
3      LocateAndLockChain (bucketchain, current, k, Selective);
4      underflow := RemoveRecord (current, bucketchain, k, deleted);
       /* RemoveRecord "atomically" updates bucketchain:
       It reorganizes the data in current and overflow buckets,
       writes any overflow buckets needed,
       builds a list (deleted) of old overflow buckets to be removed,
       writes the primary bucket of bucketchain
       (Case a of structural consistency proof and decisive action for delete),
       and returns true if chain becomes "too empty;" false otherwise   */
5      UnLock (Selective, bucketchain);
6      if (underflow) then Merge;
7      GarbageCollect (bucketchain, deleted)   /* Case b */
8    end

     procedure GarbageCollect (chain : integer, deleted : list);
     begin
       if (deleted is not an empty list) then
       begin
         /* ensure that readers with obsolete information have cleared out; then it is safe to deallocate
         without holding locks   */
         Lock(Exclusive, chain);
         UnLock(Exclusive, chain);
         Deallocate(deleted);
       end
     end
```

Fig. 4. Algorithms for changing the data in the hashfile: insert and delete procedures.

```
    procedure Split;
      var last : integer;
        deleted : list;
        A, B, C : buffer;
        chain1, chain2, original : ↑buffer;
1   begin
2     chain1 := addressof (A);
3     chain2 := addressof (B);
4     original := addressof (C);

5     Lock(Selective, root);
6     Lock(Selective, next);
7     getchain (next, original);
8     deleted := ConstructChains(chain1, chain2, original);
      /* rehash the data in the original buffer and overflow buckets into two new chains;
      increment locallevel value for new chains;
      write overflow buckets to disk;
      the primary buckets of the new chains are returned in buffers, chain1 and chain2;
      return list of garbage buckets for later removal  */
9     putchain (next + N * 2 ** level, chain2);  /* Case c */
      /* write primary bucket to disk */
10    putchain (next, chain1);  /* Case d */
      /* write primary bucket to disk */
11    last := next;
12    next := (next + 1) mod (N * 2 ** level);  /* Case e */
13    if (next = 0) then level := level + 1;  /* Case f */
14    UnLock (Selective, root);
15    UnLock (Selective, last);
16    GarbageCollect (last, deleted)
17  end

    procedure Merge;
      var partner : integer;
        A, B, C : buffer;
        chain1, chain2, newchain : ↑buffer;
1   begin
2     chain1 := addressof (A);
3     chain2 := addressof (B);
4     newchain := addressof (C);

5     Lock(Exclusive, root);
6     if (next = 0) then level := level − 1;
7     next := (next − 1) mod (N * 2 ** level);
8     Lock(Exclusive, next);
9     partner := next + N * 2 ** level;
10    Lock(Exclusive, partner);
11    DowngradeLock(root, ExclusiveToSelective);
      /* Case g—atomically converts exclusive lock to selective lock */
12    getchain (next, chain1);
13    getchain (partner, chain2);
14    MergeChains(chain1, chain2, newchain);
      /* build one chain out of the data accessible from buffers chain1 and chain2;
      decrement locallevel value for newchain;
      write any overflow buckets to disk;
      the primary bucket of the merged chain is in buffer newchain;
      deallocate garbage buckets */
15    putchain (next, newchain);
      /* write primary bucket to disk */
16    UnLock(Exclusive, partner);
17    UnLock(Exclusive, next)  /* Case h */
18    UnLock(Selective, root);
19  end
```

Fig. 5. Restructuring operations: Split and merge procedures.

Process D: delete (15)	Process I: insert (40)	Process F: find (13)
LocateAndLockChain:	*LocateAndLockChain*:	
Lock(Read, root)	Lock(Read, root)	
read level : lev = 1	*read level* : lev = 1	
hash : bucketchain = 3	*hash* : bucketchain = 0	
read next to compare	*read next to compare*	
Lock(Sel, 3)	*rehash* : bucketchain = 0	
Unlock(Read, root)	Lock(Sel, 0)	
getchain(3, current)	Unlock(Read, root)	
locallevel = lev, quit	getchain(0, current)	
RemoveRecord:	locallevel = lev, quit	
write primary bucket 3	*AddRecord*:	
	write out overflow bucket	
	write primary bucket 0	
	(See Fig. 7a)	
Unlock(Sel, 3)	Unlock(Sel, 0)	*LocateAndLockChain*:
Merge:	*Split*:	Lock(Read, root)
	Lock(Sel, root)	*read level* : lev = 1
Lock(Ex, root)—*waits*	Lock(Sel, 1)	*hash* : bucketchain = 1
	getchain(1, original)	*read next to compare*
	ConstructChains	Lock(Read, 1)
	write new partner, 5	Unlock(Read, root)
	rewrite 1	
	Increment next	
	(See Fig. 7b)	
Lock(Ex, root)—*succeeds*	Unlock(Sel, root)	getchain(1, current)
	Unlock(Sel, 1)	locallevel $\neq$ lev
Decrement next	*GarbageCollect* : *empty*	*rehash with lev* = 2
		Lock(Read, 5)
		Unlock(Read, 1)
Lock(Ex, 1)		getchain(5, current)
Lock(Ex, 5)—*waits*		locallevel = lev, quit
		Search : found (13)
		Unlock(Read, 5)
Lock(Ex, 5)—*succeeds*		
DowngradeLock on root		
getchains 1 & 5		
MergeChains : deallocate 5		
putchain (1, newchain)		
Unlock(Ex, 5)		
Unlock(Ex, 1)		
Unlock(Sel, root)		
	(See Fig. 7c)	

Fig. 6. Example computation.

It is worthwhile to mention some possible variations on this solution. In particular, there are alternative ways of handling the split and merge operations. Decoupling the restructuring actions from the requested insert and delete operations and doing them as a separate background activity is easy with these algorithms. One way of doing this is to define a new shared integer that records

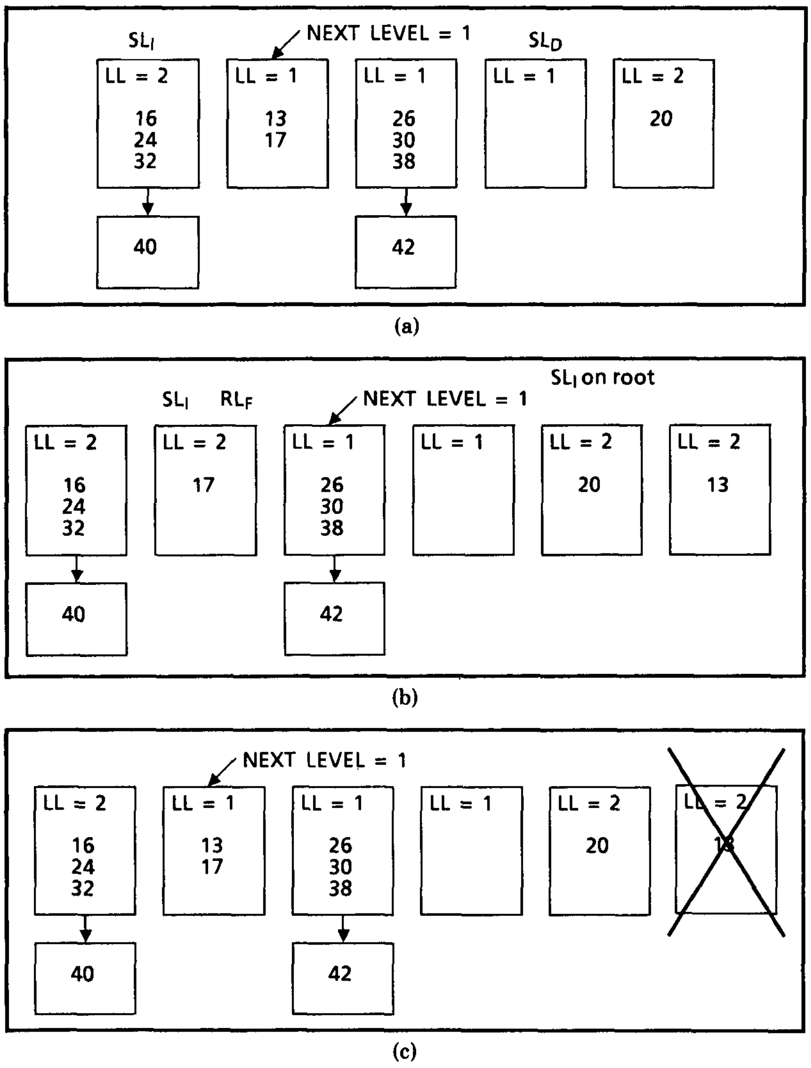

Fig. 7. Progressive states of the hashfile.

the balance between the number of overflows and underflows that have occurred. A positive value would signify the need for splitting, and a negative value, the need for merging. Instead of calling **Split**, the **insert** algorithm would atomically increment this variable. Similarly, the **delete** procedure would atomically decrement it rather than calling **Merge**. Then a single maintenance process could wait for a nonzero value, update it, and call the **Split** or **Merge** procedure, as appropriate. Modifying the solution for controlled splitting would involve requiring each operation that changes the data structure to update shared utilization data atomically. Using this approach, the separate maintenance process would monitor the utilization data. Finally, if a translation table is used for noncontiguous allocation of buckets in physical storage, locks can simply be associated with the table entries rather than the chains themselves.

4. CORRECTNESS ARGUMENTS

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, we must prove the following properties:

P1. The system is deadlock free.

P2. The data structure is always consistent.

A consistent linear hashfile is defined by the following four statements:

S1. The contents of each bucket chain are characterized by the locallevel value of that chain. Specifically, for all keys k contained in bucket i, $h_{i.\mathrm{locallevel}}(k) = i$.

S2. The buckets fall into N equivalence classes related to the N original buckets established by hash function h_0. If $h_0(k) = i$, then k belongs in one of the buckets from the ith equivalence class. The relationship between the contents of any two buckets from the same equivalence class can be represented by a binary tree, as in Figure 8. Nodes at depth d of this tree correspond to the buckets from this class accessible using h_d. The two children of a node indicate how the contents of the parent are divided by the next hash function. From this tree formulation, it is clear that buckets from disjoint subtrees contain disjoint sets of keys.

S3. The locallevel values of any two buckets present in the hashfile differ by at most one. In particular, level $\leq$ locallevel $\leq$ level + 1 holds for all bucket chains. The data structure can be described by a pattern of locallevel values across the sequence of buckets. Let a be the symbol representing the value of level and b be the symbol for level + 1. Define x, y, and z such that $x \leq n/2$, $y = 2^{\mathrm{level}}N - x$, and $x \leq z \leq x + 1$. Then, considering the sequence of locallevel values as a string over the alphabet $\{a, b\}$, the pattern is given by the regular expression $b^x a^y b^z$.

S4. Let n denote the number of buckets. Then the size of the hashfile can be expressed as $2^{\mathrm{level}}N + \mathrm{next} \leq n \leq 2^{\mathrm{level}+1}N$ with $\mathrm{next} \leq 2^{\mathrm{level}}N - 1$.

Note that stronger statements of consistency can be made initially and when no restructuring operation (split or merge) is in progress:

S2′. The contents of all bucket chains are disjoint.

S3′. The pattern $b^x a^y b^z$is observed with $x = z = \mathrm{next}$ and $y \neq 0$.

S4′. The size of the hashfile is $n = 2^{\mathrm{level}}N + \mathrm{next}$.

It can be seen that upon initially loading the hashfile with function h_0, the data structure is consistent.

P3. When the search phase of the find, insert, and delete operations (i.e., in the procedure **LocateAndLockChain**) stops, an appropriate target bucket has been locked (called *bucketchain* in each algorithm) and its contents read into a private buffer (called *current*). Intuitively, the key k specified in the request belongs among the contents of bucketchain. At the time of the last call to **getchain**, $h_{\mathrm{bucketchain.locallevel}}(k) = \mathrm{bucketchain}$.

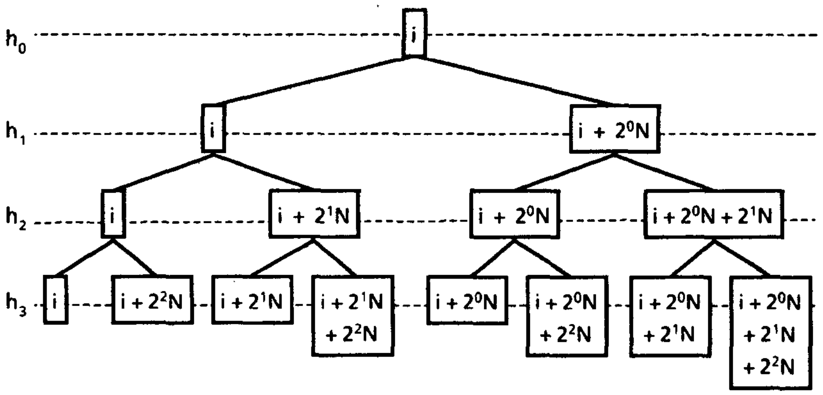

Fig. 8. Evolution of equivalence class i (through h_3).

P4. Parallel computations possible in this solution are semantically serializable (e.g., [17]) on the basis of observed results of the requested operation). The set of records in the hashfile at the completion of the *decisive step* of an update is the set existing prior to the operation plus (minus) the record inserted (deleted). The term decisive step is taken from [17] and distinguishes one particular step in the execution of a requested operation. For insert and delete operations, the decisive step is the writing of the primary bucket in **AddRecord** and **RemoveRecord**, respectively. The decisive step for the find operation is the final **getchain** in the call to **LocateAndLockChain**. The subsequent call to **search** acts upon those contents read into current.

Note that starvation is theoretically (but not practically) possible.

In the following discussion, the emphasis is on the problems that arise because of concurrency. The algorithms are assumed to correctly perform the find, insert, and delete functions when executed in the absence of parallel operations.

Several of our arguments require that the concurrent execution of a set of requested operations can be viewed as an interleaved computation of the steps within those operations. The program executed in response to a request imposes a total ordering on its own steps. Within a parallel computation involving multiple requests, steps of different processes may be ordered by locking constraints or because the steps are inherently atomic. Thus, there is a partial order defined on the steps of a computation. If all pairs of actions that do not fall under this partial order commute with the same results, then there should exist a new computation in which there is a total ordering of steps that is consistent with the partial order. This interleaved computation yields the same results as the original concurrent computation. One can see by inspecting pairs of operations, allowed to execute in parallel, that they do commute.

4.1 Freedom from Deadlock

The freedom from deadlock argument depends on the fact that locks are requested according to an ordering on the lockable components of the structure. We define an ordering relation (<) as follows: For any bucket chain b root < b. The ordering between any two buckets is based on the integer values of the logical bucket addresses. Specifically, let b_i denote the bucket with address i. Then, $b_i < b_j$ if the integer value i is less than j.

All of the locks needed in the find operation and in the first phase of the insert and delete operations (i.e., when the target record is being added or removed) are acquired in the **LocateAndLockChain** procedure. The root is locked first, then one bucket chain is locked, possibly followed by the locking of successively higher addressed chains related to the initial chain through the sequence of hash functions. Each address calculated in the rehashing loop is generated by the successive hash function and is guaranteed to be greater than or equal to the current bucket address, by definition. A lock is requested only when the new address is greater. Thus, this locking protocol respects the ordering relation.

Before checking the conditions that call for a split or merge operation, the updating process releases all locks that it holds. The **Split** procedure places selective locks first on root and then on the chain designated as the next one to be split (indicated by the variable, next). This clearly follows the ordering as defined. After releasing the selective locks, deleted overflow buckets, if any, are deallocated using a single exclusive lock which obviously cannot cause a deadlock problem. In the **Merge** procedure, three exclusive locks are held simultaneously. Root is locked first, followed by the bucket chain into which the merged records will be written (next), and finally by the bucket chain at address next + $2^{\text{level}}N$. Again, this obeys the ordering.

4.2 Structural Consistency

We present an inductive argument to show that the data structure remains consistent as defined. Initial consistency is assumed. For induction, we assume that the hashfile is consistent at time t. Time is viewed as proceeding in discrete intervals defined in terms of atomic changes in the data structure. Thus, between time t and $t + 1$ exactly one such modification is done. For each type of atomic change, the resulting structure must be shown to also be consistent.

Actions that can be considered atomic modifications include the inherently atomic operations of assigning a new value to next, assignment to level, and writing a bucket to disk. Since new overflow buckets do not become accessible through the structure until the primary bucket is rewritten, only **putchain** calls on primary buckets are important. Exclusive locks are used in **Merge** to create a critical section involving multiple modifications. Then, releasing the exclusive lock makes a sequence of changes appear as an atomic change to other processes. Thus, there are eight cases to consider: (a) replacement of the primary bucket after reorganizing the target bucket chain to add or delete a single record (in **AddRecord** and **RemoveRecord**); (b) deallocation of garbage overflow buckets; (c) writing the new bucket chain created in the **Split** routine; (d) rewriting the target bucket chain of a split operation; (e) moving next forward in **Split**; (f) conditionally incrementing level in **Split**; (g) downgrading the exclusive lock on updated values of root during a merge operation; and (h) release of the lock on a merged chain in the **Merge** procedure. These cases cover all actions that cause visible changes in the data structure (i.e., all assignments to root variables and all **putchain** calls on primary buckets). Note that only Case a changes the set of keys stored in the hashfile.

Case a. (Figure 4, line 4 in procedures **insert** and **delete**, within calls to **AddRecord** and **RemoveRecord**) At time t prior to the replacement of the primary bucket, the updating process holds a selective lock on the correct target bucket chain, bucketchain. In the next section, we show that given a consistent hashfile, the search phase finds the correct target bucket. Here, we assume consistency holds at all times up to and including time t, so the correct target bucket has been found and locked.

This selective lock prevents other updating processes from locking the chain and then reading the contents of its buckets. All bucket modifications require an incompatible lock (either selective or exclusive) on the bucket before it is accessed. Thus, modifications to individual bucket chains are serialized.

The contents of the resulting bucket chain that is made accessible by atomically writing the primary bucket back to disk are exactly the contents of the original chain plus or minus the single key used to locate this chain. The contents of other buckets are not affected. All other records remain assigned to buckets in exactly the same way as they were before. Thus, statements S1 and S2 hold at time $t + 1$.

This update has no effect on the size or organization of the hashfile or on the locallevel values (statements S3–S4).

Case b. (Figure 4, line 7 in procedures **insert** and **delete**) Deallocating garbage buckets that have been removed from a chain and are not reachable through the hashfile structure cannot affect its consistency.

Case c. (Figure 5, line 9 in procedure **Split**) The first change made to the hashfile during a split operation is the writing of the new primary bucket $2^{\text{level}}N$ + next. Merge and split operations execute sequentially because of incompatible locks on root; so at time t, the stronger definition of consistency applies to the hashfile. The process performing the split holds selective locks on root and the next bucket chain, next.

The bucket chain at location next is the one with the lowest numbered index such that its locallevel = level. This bucket is represented by the first a symbol in $b^x a^y b^z$. The conditions that x = next and $y \neq 0$ imply that such an a exists.

The records in the target bucket chain have been divided up in private buffers according to hash function $h_{\text{level}+1}$, and the overflow buckets of the new chains have been written to disk, but are not yet part of the hashfile. The second partner of the split (i.e., the chain addressed $2^{\text{level}}N$ + next) is officially incorporated into the hashfile upon writing the primary bucket (from buffer, chain2). Thus, at time $t + 1$, all records with keys k, such that $h_{\text{level}+1}(k) = 2^{\text{level}}N$ + next, are available at that address (statement S1).

The contents of this new bucket chain are contained in the contents of the current version of the target bucket chain that is still present at location next. In the tree formulation, bucket $2^{\text{level}}N$ + next is the right child of bucket next, and disjointness of bucket contents is not required in this situation. The contents of all other buckets remain distinct from those in this subtree (S2).

The locallevel of the new bucket is set to the value level + 1 and in the pattern $b^x a^y b^z$, $z = x + 1 =$ next + 1 (S3). The size of the structure is $n = 2^{\text{level}}N +$ next + 1. Since next $\leq 2^{\text{level}}N - 1$, $n \leq 2^{\text{level}+1}N$ (S4).

Case d. (Figure 5, line 10 in procedure **Split**) The consistent hashfile present at time t has the same size and structure as the hashfile resulting from the writing of the new partner, considered above. The actual bucket contents may have been affected by insertions and deletions. The restructuring process continues to hold the selective locks.

At time $t + 1$, a subset of the records available in the original target bucket are present in the new version of that chain. This subset consists of all records in the bucket chain at time t with keys k such that $h_{\text{level}+1}(k) =$ next (S1).

The contents of all bucket chains are disjoint in the resulting hashfile. Basically, writing this bucket supplies a left son to the node that corresponds to next at depth level of the tree formulation. All buckets are represented by leaves in the tree that describes the state of the data structure (S2′).

The locallevel of the new version is set to level + 1. The effect is to change the leading a in the string $b^x a^y b^z$ at time t to a b, incrementing x and decrementing y. Since $z = x + 1 =$ next + 1 at time t, $x = z$ at time $t + 1$. At time t, $y \neq 0$, so next $= x < 2^{\text{level}}N$. After incrementing x (time $t + 1$), it is still true that $x \leq n/2$ (S3).

This action has no effect on the size of the hashfile $n = 2^{\text{level}}N +$ next + 1 (S4).

Case e. (Figure 5, line 12 in procedure **Split**) Advancing next (i.e., the assignment *next* $\leftarrow$ (*next* + 1) *mod* $(2^{\mathit{level}}N)$) has no effect on the statements regarding bucket contents (S1 and S2′). The remaining issues are the size (S4) and organization (S3) of the hashfile.

At time t, $n = 2^{\text{level}}N +$ next + 1. If the new value of next is not zero, then $n = 2^{\text{level}}N +$ next at time $t + 1$. Since $x = z =$ next + 1 in the string at time t, $x =$ next after this action.

If the new value of next is zero, the old value of next $= 2^{\text{level}}N - 1$ and $n = 2^{\text{level}+1}N$ at time $t + 1$. Then, $x = n/2 = 2^{\text{level}}N$, and $y = 0$ in the string $b^x a^y b^z$ at time $t + 1$.

Case f. (Figure 5, line 13 in **Split**) Following the step of advancing next in the split operation, the value of level is conditionally adjusted. The consistent data structure at time t inherits the size, pattern of locallevel values, and disjointness characteristics of the hashfile resulting from the previous action. Concurrent inserts and deletes may have changed the particular bucket contents.

This change has no effect on bucket contents (S1 and S2′).

The value of level is incremented if the value of next has just become zero. Since $n = 2^{\text{level}+1}N$ at time t, $n = 2^{\text{level}}N +$ next with the new value of level at time $t + 1$ (S4′). All locallevel values equal level + 1 at time t, so increasing level has the effect of making all locallevel values equal level. In $b^x a^y b^z$, $y = 2^{\text{level}}N$, and $x = z = 0 =$ next (S3′). Note that the strong definition of consistency is again in effect.

Case g. (Figure 5, line 11 in procedure **Merge**) Now consider updating the root values in the merge operation. Since merge and split operations execute sequentially, the stronger definition of consistency holds at time t. The process performing the merge has acquired exclusive locks on root and both partners. The root values are guaranteed to be accurate when the partners are located because of the exclusive lock.

Downgrading the exclusive lock on root to a selective lock makes the updated values of level and next visible in a single step to processes executing find, insert, or delete operations. Note that completely releasing the exclusive lock on root also works and allows some overlap of merge operations, but this severely complicates the correctness proof. The new values are the result of the following statements:

```
if next = 0 then level ← level − 1;
next ← (next − 1) mod $2^{\text{level}}N$;
```

These updates have no effect on the bucket contents or disjointness constraint (S1 and S2′).

If next is not zero at time t, then the value of level remains unchanged, and the relationships between locallevel values and the value of level are unaffected. The pattern $b^x a^y b^z$ also remains the same. Since x = next prior to modifying next, then x = next + 1 at time $t + 1$ (S3). The size of the hashfile n does not change, but the expression of the lower bound on n in terms of next does. At time t, $n = 2^{\text{level}}N + \text{next}$. Decrementing next makes $n = 2^{\text{level}}N + \text{next} + 1 \leq 2^{\text{level}+1}N$ at time $t + 1$ (S4). The resulting hashfile satisfies the conditions for consistency.

If next does equal zero, both level and next are modified. Since $x = z =$ next = 0 in $b^x a^y b^z$, all locallevel values equal the value of level at time t, and decrementing level makes all locallevel values equal level + 1. This meets the condition about the relationship between locallevel values and level. In $b^x a^y b^z$, $y = 0$, and $x = z = n/2 = \text{next} + 1$ at time $t + 1$ (S3). At time t, $n = 2^{\text{level}}N$. Thus, after decreasing the value of level and without changing the size of the hashfile, $n = 2^{\text{level}+1}N$ (S4). Since next $= 2^{\text{level}}N - 1$, the size can also be expressed as $n = 2^{\text{level}}N + \text{next} + 1$.

Case h. (Figure 5, line 17 in **Merge**) Release of the exclusive lock on the target bucket next makes the merged bucket chain appear atomically. The hashfile at time t inherits size, the locallevel pattern, and the tree representation from the previous step. The merging process retains the exclusive locks on next and its partner.

The fact that $x = z = \text{next} + 1$ indicates that the target bucket chain corresponds to the last b symbol in the b^x prefix of $b^x a^y b^z$. Thus, its locallevel = level + 1, and its partner exists in the data structure. Merging those two buckets and setting the locallevel value of the resulting bucket to level has the effect of decreasing z and x each by one and incrementing y. Merging also decreases the size by one; thus, $n = 2^{\text{level}}N + \text{next}$. Thus, statements S3′ and S4′ hold.

The contents of the two buckets being merged are characterized by the function $h_{\text{level}+1}$, since locallevel = level + 1 in both buckets. For all keys k in these two buckets, $h_{\text{level}}(k) = \text{next}$. Thus, the contents of the merged bucket at time $t + 1$ is characterized by $h_{\text{locallevel}}$ (S1). The effect is to prune off the two children of the node that corresponds to next at depth level of the tree formulation. As a result, the bucket at location next is represented by a leaf, and its contents are therefore disjoint from the contents of all other buckets (S2′).

4.3 Correctness of Searches

For split and merge operations, locating the appropriate target bucket chain is trivially correct because the lock placed on root by the restructuring process guarantees accuracy of the value of next.

Finding the target chain in the search phase of find, insert, or delete depends on the functioning of the rehashing scheme in the procedure **LocateAndLockChain**. The correctness of this search phase relies on the consistency of the data structure, the atomicity of individual bucket reads, and the lock-coupling protocol. We need to show that (a) the bucket address initially calculated using the values of level and next seen by the searching process is in the currently valid address space; (b) subsequent probes in the rehashing loop also fall within the valid address space; (c) the bucket chain that should contain the target key k is reachable by rehashing from the current bucket; and (d) it is possible to decide when the right chain has been reached. So, if the rehashing loop terminates, k belongs in the returned bucketchain.

(a) At the time that the root values are read by the searching process, it holds a read lock on them. Therefore, no merge can be underway, and the size of the data structure cannot decrease while the initial address is being calculated. Let *lev* denote the value of level and *nx* denote the value of next seen by the process. The initial calculation generates an address between 0 and $2^{\text{lev}}N + \text{nx} - 1$. The size of the hashfile is given by statement S4 of the consistency property P2 as $n \geq 2^{\text{level}}N + \text{next}$. Throughout this step, level $\geq$ lev. If level = lev, either next $\geq$ nx, or a split operation is at the stage in which $n = 2^{\text{level}+1}N$ (i.e., between setting next to zero and incrementing level). If level > lev or next $\geq$ nx, $2^{\text{lev}}N + \text{nx} \leq 2^{\text{level}}N + \text{next}$. In the special case of the split, $2^{\text{lev}}N + \text{nx} \leq 2^{\text{level}+1}N$. Thus, the initial probe is within range.

(b) Starting with the hash function h_{lev}, the rehashing loop iterates through successive functions (by incrementing lev) until the hash function h_{lev}, used to reach the current chain, bucketchain, matches the function characterizing the contents of that chain, $h_{\text{bucketchain.locallevel}}$. In other words, the loop terminates when lev = bucketchain.locallevel at the time the bucket was read. Because of the lock coupling done in this search and the fact that merging requires exclusive locks on both partners of the merge, lev is guaranteed to be less than or equal to the locallevel value of the bucket reached using h_{lev}. Statement S3 of property P2 says that level $\leq$ locallevel $\leq$ level + 1 holds for all bucket chains. If lev $\leq$ level, then $2^{\text{lev}}N \leq 2^{\text{level}}N \leq n$. The function h_{lev} generates addresses within the range 0 to $2^{\text{lev}}N - 1$. If lev = level + 1, then the previous hash function h_{level} accessed a bucket represented within the b^x prefix of the string $b^x a^y b^z$, since the locallevel of that bucket equals level + 1, h_{level} generates addresses in the range 0 to $2^{\text{level}}N - 1$, and $x + y = 2^{\text{level}}N$ by S5 of P2. In this case, the function h_{lev} calculates a value between 0 and $2^{\text{level}}N + x - 1$. Except in the special case during a split (when next = 0, and level has not yet been incremented, and $n = 2^{\text{level}+1}N$), $2^{\text{level}}N + x \leq 2^{\text{level}}N + \text{next} \leq n$.

(c) In order to make this argument somewhat more formal, we adapt some terminology from [17]. The set of possible key values is called *KeySpace*. The hash function h_0 partitions KeySpace into N subsets associated with the N equivalence classes described in S2 of P2. We define *keyset* (b) to be those values of KeySpace that are in or could be in the bucket chain b. More precisely, *keyset* $(b) = \{k \mid h_{b.locallevel}(k) = b\}$. Let *reach* (l, b) be the set of keys reachable from b, given that h_l was used to access b. Formally, $reach(l, b) = \{k \mid h_l(k) = b\}$. Applying the successive hash function h_{i+1} determines two sets, $reach(i + 1, b)$ and $reach\ (i + 1, b + 2^iN)$, where $reach(i, b) = reach(i + 1, b) \cup reach(i + 1, b + 2^iN)$. This partitioning of reach(i, b) corresponds to the split of b that took place when level equalled i and divided keyset(b) into keyset(b) and keyset$(b + 2^{\text{level}}N)$. The split operation (using $h_{\text{level}+1}$)has no effect on reach(i, b), $i \leq$ level.

The rehashing loop of **LocateAndLockChain** visits the sequence of buckets found by applying each hash function in sequence to the desired key k. From the definition of reach(l, b) given above it can be seen that with each iteration of the rehashing loop, $k \in$ reach(lev, bucketchain), lev $\leq$ bucketchain.locallevel.

Another way of looking at this is in terms of the tree formulation of the hashfile in Figure 8. Each address calculated in the rehashing loop is in the same equivalence class as the initial probe. A node of this tree at depth *dep* with label (i.e., bucket address) *lab* also represents the set *reach*(*dep*, *lab*). Then, the rehashing scheme can be viewed as a search of the tree in which lev gives the depth of the current probe, and the correct target bucket is reachable from the node representing the current bucket address at that depth. The lock-coupling protocol guarantees that the target bucket remains reachable from the current point along the searching process's path (i.e., preventing interference by a merge). Splitting serves to lengthen some paths, but reachability is not threatened.

(d) The ability to decide whether the right bucket has been reached follows from statements S1 and S2 of the consistency property. Formally, $k \in$ reach (lev, bucketchain) and lev = bucketchain.locallevel at the final **getchain** call imply $k \in$ keyset(bucketchain).

4.4 Correctness of Updates

The first step is to show that the target chain remains appropriate throughout the lifetime of the operation. For insertions and deletions, the locate phase ends with a selective lock placed on the target bucket. This lock guarantees no interference by concurrent updates with the contents of the version of the bucket in the private buffer, current, during the subsequent decisive action. The affected buckets of splits and merges are also locked for the duration of the step.

The locate phase of the find operation ends with a read lock on the target bucket chain and the contents of the primary bucket atomically read into a private buffer (the decisive step). The results of the subsequent search are meaningful. The atomicity of writing a reorganized bucket chain means that the version of the chain in current is valid (i.e., pure readers may see either the old or new version of concurrent updates, but not intermediate states). The old overflow buckets reachable through the pointer in the old version of the primary bucket (if that is what the reader possesses) are protected from deallocation by the read lock held by the reader on the chain.

In order to make the argument that updates are serializable, it must be possible to construct an equivalent serial schedule of user-observable events. It is easy to identify the atomic changes to the data structure that affect the set of records contained in the hashfile. For each update request, there is exactly one such event (the decisive action), and that event can be considered the point of completion of the operation. It is at this point that the user could be notified of success.

For insert and delete operations, the atomic rewrite of the primary bucket in the **AddRecord** or **RemoveRecord** procedures represents the completion point. The selective lock on the target bucket after the final search phase excludes other updaters from sharing the version of the bucket being manipulated and thus interfering with the update.

Split and merge operations preserve the set of records in the hashfile and are essentially invisible to users. Therefore, it is not necessary to argue, using our criteria for serializability, that they fit into the serial schedule.

5. SUMMARY

Linear hashing is one of the techniques recently proposed to allow adjustment in the range of the hashing function as the amount of data stored grows and shrinks. The existence of a highly concurrent dynamic indexing structure can be extremely important for databases designed to support a large number of users and variability in the size of the database. In this paper, we have presented a solution for concurrent access to a shared centralized linear hashfile.

This solution appears to deliver a very high degree of concurrency. The locking scope (i.e., the maximum number of locks held at one time by any process) is constant and small (≤ 3). The use of the most restrictive lock type, exclusive locks, is minimal and most exclusive locks are held for a very short time (e.g., during access to root for a merge, momentary locking of a chain for garbage collection). Since this is a hashing approach, requests are expected to be spread out evenly over the address space. Assuming a sufficient number of requests, this solution should be able to support essentially n (the number of buckets) concurrent updaters and an unlimited number of readers.

As in proposals for concurrency in other data structures, making modifications to the data structure has proved to be a useful technique for achieving greater concurrency. In this solution, the modifications are relatively minor (i.e., the addition of a locallevel field to each bucket chain), yet sufficient to detect the effects of concurrent updates and allow the search to resume from that point along an alternate path.

This solution also demonstrates that lock-coupling protocols, found to be useful in B-trees and other linked structures, carry over to a fundamentally different type of data structure. The linear hashfile is basically not a linked structure (ignoring the overflow chains), and there is no explicit notion of reachability built into it. Recovering from the use of obsolete information is done by rehashing with a sequence of functions rather than by following a detour through physical pointers, and it may require repeatedly visiting the same bucket chain. Thus, this reorientation technique, although inspired by the earlier algorithms based on links, has a significantly different flavor in this solution.

Although traditionally viewed as a database problem, concurrent access to structured data is an issue that will become increasingly more important as applications are developed for multicomputer architectures. This paper illustrates a set of techniques applied to one particular data structure. We hope that some of these ideas can transfer to new problem domains and different architectural models. For instance, we are investigating using these techniques in various data structures of a multiprocessor operating system, and the solution presented here has been used as the basis for a version of linear hashing intended for a distributed system.

REFERENCES

1. BAYER, R., AND SCHKOLNICK, M. Concurrency of operations on B-trees. *Acta Inf. 9* (1977), 1–21.
2. ELLIS, C. Concurrent search and insertion in 2–3 trees. *Acta Inf. 14* (1980), 63–86.
3. ELLIS, C. Concurrent search and insertion in AVL trees. *IEEE Trans. Comput. C-29*, 9 (Sept. 1980), 811–817.
4. ELLIS, C. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Atlanta, Ga., Mar. 21–23). ACM, New York, 1983, pp. 106–116.
5. ELLIS, C. Concurrency and linear hashing. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Portland, Ore., Mar. 25–27). ACM, New York, 1985, 1–7.
6. ELLIS, C. Distributed data structures: A case study. *IEEE Trans. Comput. C-34*, 12 (Dec. 1985), pp. 1178–1185.
7. FAGIN, R., NIEVERGELT, J., PIPPENGER, N., AND STRONG, H. R. Extendible hashing-a fast access method for dynamic files. *ACM Trans. Database Syst. 4*, 3 (Sept. 1979), 315–355.
8. KUNG, H. T., AND LEHMAN, P. L. Concurrent manipulation of binary search trees. *ACM Trans. Database Syst. 5*, 3 (Sept. 1980) 354–382.
9. KWONG, Y. S., AND WOOD, D. New method for concurrency in B-trees. *IEEE Trans. Softw. Eng. SE-8*, 3 (May 1982) 211–222.
10. LARSON, P. Dynamic hashing. *BIT 17* (1978), 184–201.
11. LEHMAN, P., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst. 6*,4 (Dec. 1981), 650–670.
12. LITWIN, W. Linear hashing: A new tool for file and table addressing. In *Proceedings of the 6th Conference on Very Large Data Bases* (Montreal, 1980), 212–223.
13. LOMET, D. Bounded index exponential hashing. *ACM Trans. Database Syst. 8*, 1 (Mar. 1983), 136–165.
14. MANBER, U., AND LADNER, R. E. Concurrency control in a dynamic search structure.) *ACM Trans. Database Syst. 9*, 3 (Sept. 1984), 439–455.
15. MILLER, R., AND SNYDER, L. Multiple access to B-trees. In *Proceedings of Conference on Information Sciences & Systems* (Baltimore, Md., Mar. 1978), (preliminary report).
16. SAGIV, Y. Concurrent operations on B-trees with overtaking. In *Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Portland, Ore., Mar. 25–27). ACM, New York, 1985, 28–37.
17. SHASHA, D., AND GOODMAN, N. Concurrent search structure algorithms. To appear in *ACM Trans. Database Syst.*
18. WU, C. T., AND BURKHARD, W. A. Concurrency in linear and interpolation hashing. Unpublished manuscript. Northwestern University, Evanston, Ill., (1983).

Received May 1986; revised August 1986; accepted October 1986