Concurrent Operations in Extendible Hashing

Meichun Hsu
Wei-Pang Yang

Harvard University
Cambridge MA 02138

Abstract.

An algorithm for synchronizing concurrent operations on extendible hash files is presented. The algorithm is deadlock free and allows the search operations to proceed concurrently with insertion operations without having to acquire locks on the directory entries or the data pages. It also allows concurrent insertion/deletion operations to proceed without having to acquire locks on the directory entries. The algorithm is also unique in that it combines the notion of verification, fundamental to the optimistic concurrency control algorithm, and the special and known semantics of the operations in extendible hash files. A proof of correctness for the proposed algorithm is also presented.

1. Introduction

The concurrency control algorithm in a conventional database management system enforces serializability of transactions [Papadimitriou79]. Each transaction is normally modeled as a sequence of read and write steps, and the concurrency control algorithm enforces serializability without assuming much knowledge of the semantics of the read and write steps of the transactions. While this level of generality enables the concurrency control algorithm to be applicable to any transaction system, it does not take advantage of the structures inherent in the applications to optimize for higher level of concurrency and lower synchronization overhead.

In recent years specialized concurrency control algorithms that take advantage of the knowledge of the structure and/or the semantics of transactions have appeared [e.g., SK80, KS83, KP79, HM83, HC85, O'Neil85]. In particular, much attention has been paid to the optimization of algorithms that synchronize concurrent operations on B-trees [e.g., BS77, LY81, MR85].

In this paper we present an algorithm that synchronize concurrent operations on a file structured using extendible hashing [FNPS79]. Extendible hashing is a form of dynamic hashing which adaptively updates a directory of pointers to data bucket, or data pages. Since the directory entries are subject to update at any moment, a search operation would normally be required to obtain a lock on the directory entry it reads to prevent the directory entry from being inadvertently changed. However, by exploiting the known semantics of the accesses to the directory entries, it is conceivable that one can devise concurrency control algorithms that minimize such overhead.

We present a concurrency control algorithm that allows the search operation in an extendible hash file to proceed without having to set locks on the directory entries. We also allow concurrent insertions to be synchronized with a mechanism which is simpler and potentially able to offer a higher degree of concurrency.

The algorithm is also unique in that it utilizes the general mechanism behind the optimistic concurrency control algorithms [KR81]. By making use of ***verification*** at the right moment, operations are guaranteed a consistent view of the data structures required to ensure their correctness while minimizing the locking overhead.

The structure of the paper is as follows. In the next section, the general mechanism of the extendible hashing scheme is reviewed. In Section three, we present our concurrent search and insertion algorithms, followed by a proof of correctness in Section four. Section five concludes the paper and presents a discussion of future extensions.

2. Review of Extendible Hashing

Extendible hashing [FNPS79] is a file structuring and searching technique in which the user is guaranteed no more than two page accesses to locate the data associated with a given key. Unlike conventional hashing, extendible hashing has a dynamic structure that grows and shrinks gracefully as the database grows and shrinks.

The file consists of a directory (D) and data pages. The directory is characterized by a *global depth* g, and contains 2^g entries, each of which points to a data page. The hash function, h, transforms the keys of the key set into a "pseudo key" of a bit form; the first g bits of the pseudo key determine the directory entry corresponding to a key. Each data page is characterized by a *local depth* $l \leq g$, and a bit pattern bp of length l. A data page with an l-bit bit pattern bp contains all keys the first l bits of whose pseudo keys conform to the bit pattern bp. When a data page overflows, its local depth is incremented by 1 and the page is split in two: one page is now characterized by a bit pattern which is the old bit pattern concatenated with an additional bit of '0' and the other, with the bit of '1'.

Example. Consider the state of an extendible hash file as shown in Figure 2.1. Currently there are very few records with pseudo keys that begin at '1'. All such records are collected into a single data page whose local depth is 1 and whose 1-bit bit pattern is '1'. When the page becomes full, as shown in Figure 2.2, it splits into two data pages, each with local depth of 2: one data page now has a bit pattern of '10' and the other '11'. All keys whose pseudo keys begin at '10' appear in the first of these data pages, and all keys whose pseudo keys begin at '11' appear in the other.

When the data page whose local depth is equal to the global depth of the directory overflows, the directory size is doubled, i.e., the global depth is incremented by 1, and the overflowing data page is again allowed to split. For example, if we start with the situation as shown in Figure 2.2, and if the data page pointed to by the "010" pointer is already full, then the directory is doubled and the page splits, as shown in Figure 2.3. (Figures 2.1 to 2.3 are taken from Figures 8 to 10 in [FNPS79].)

The extendible hashing scheme uses a contiguously allocated directory whose size changes by factors of two. It enables direct access to the right data page (or bucket). No overflow area is used. In [FNPS79], it is shown that, in the case where the bucket (page) size is 400 and the size of the key set is 40,000, the storage utilization, on the average, is about 69%.

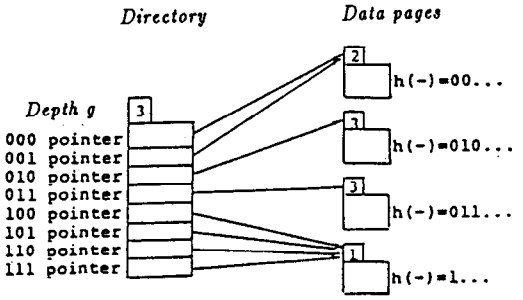

Fig. 2.1. A directory with g=3.

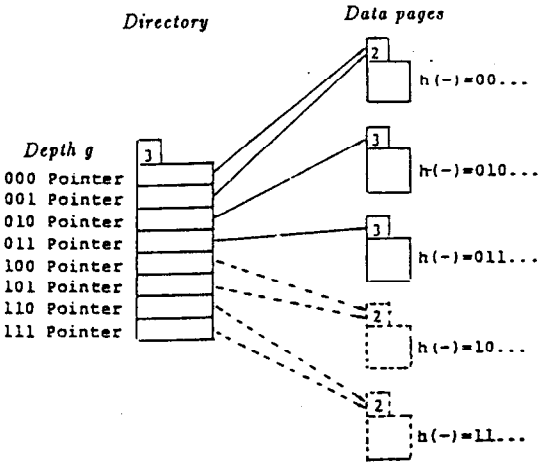

Fig. 2.2. A page splits into two data pages.

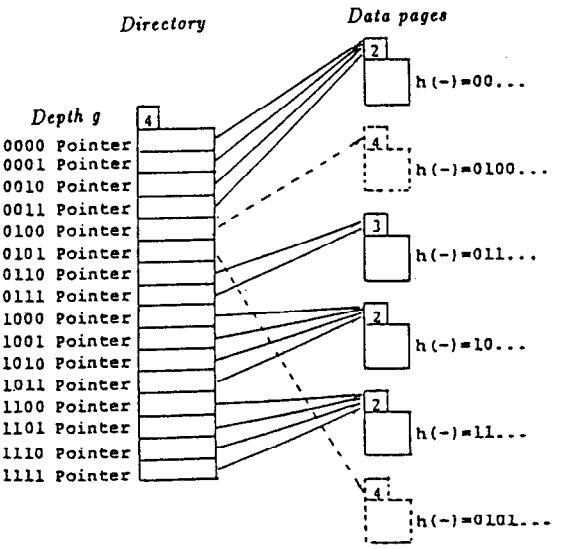

Fig. 2.3. Directory doubled with g=4.

3. Concurrent Operations in Extendible Hashing

In this section we describe the algorithm of our concurrent operations in extendible hash files. Throughout we will ignore the issue of underflow and compaction. In other words, the number of pages of the file only grows and never shrinks. The compaction issue was also ignored in [LY81] and is generally justified by the observation that databases tend to grow and the utility of the storage recovered from on-line real-time compaction may not be worth the trouble. Compaction can be handled by taking the database offline for a reorganization.

3.1. Search Algorithm

The search operation on an extendible hash file consists of (1) applying the hash function to obtain a pseudo key, (2) examining the first g bits of the pseudo key to determine the directory entry to be read, (3) reading the directory entry to find a pointer to the data page to be searched, and (4) searching in the data page to find the key desired.

What the search operation is vulnerable to is the concurrent insertion operation that splits a data page and relocates a range of the keys that include the key desired by the search operation. This type of interference can be eliminated by requiring the search and the insertion operations to obtain a lock on the directory entry and hold it until the operation ends. In our search algorithm, however, this type of interference is avoided by re-reading the directory entry when a search operation could *not* find the key in the data page it has just read, without having to hold any lock on the directory. This form of re-reading, or verification, continues until either the key is found, or the value of the directory entry does not change between two consecutive readings. The algorithm is formally defined shortly.

Intuitively, the search algorithm attempts to *verify* the directory entry it has previously read before it would conclude a search failure. If the content of the directory entry has changed in the mean time, the search operation automatically retries with the new pointer obtained. A formal proof of correctness of the algorithm is presented in Section 4.

Definition of the Search Algorithm.

```
Algorithm Search(given key k);
begin
  initialization:
    xold:=0;
  hashing:
    calculate k' = h(k)= b_0 b_1 ··· b_{n-1};
  getpointer:
    read d, base ; /* the global depth and base address of the directory D */
    t := b_0 b_1 ··· b_{d-1}; /* take the initial d bits of k' */
    x := get(D[t]); /* D[t] is the t-th entry in D */
  probe:
    do while x ≠ xold;
      A := get(x); /* read a data page */
      if key k in A then 'success', return(x); /* ends search */
      xold := x;
      x := get(D[t]); /* re-read directory */
      end;
    return ('search fails');
end;
```

3.2. Insertion Algorithm

The insertion operation in an extendible hash file consists of (1) applying the hashing function to the key to obtain the pseudo key, (2) examine the first g bits of the pseudo key to determine the directory entry to be read, (3) reading the directory entry to obtain a pointer to a data page, (4) reading the data page to search for the existence of the same key, and (5) inserting the key in the data page, if the key does not already exist. When inserting the new key, if the data page is full, then a split is performed, resulting in a new data page to be created and at least one directory entry to be updated. For now we will ignore the issue of directory expansion (i.e., doubling in size). We will revisit this issue briefly in the final section of this paper.

Two insertion operations may interfere even when they are inserting different keys. Undesirable interference may be eliminated by requiring the insertion operation to hold locks on both the directory entries and the data page that it updates till the end of the operation. In our algorithm, however, the need to hold locks on the directory entries is avoided by requiring the insertion operation to perform *verification* of the content of the directory entry it has previously read *after* locking the data page and *before* performing updates on the data page. If verification fails, the operation would unlock the page and lock a different one, and perform another verification. The insertion operation never blocks once its first lock is granted, therefore deadlock is eliminated.

In handling splitting, our algorithm requires that the newly allocated page be locked until the affected directory entry(entries) is(are) updated. Inherent in the dynamic hashing algorithm, however, is the complication that when a key k is to be inserted into a page which is already full, one split may not be enough. When splitting occurs, the local depth of the splitting page is incremented by one and a new page is allocated in the database. The original key range in the splitting page is divided in half, with the higher half distributed into the new page and the lower half retained in the splitting page. One of these two pages, say p, now contains the key range that includes k. It is noted that in extreme cases p may be full again before k is inserted. This occurs when all the existing records in the splitting page are all hashed into the halved-key-range that contains k. When this occurs, p needs to be split again before k can be inserted. This process must continue until k finally falls in a page which is not full. However, the

number of splits required, and therefore the number of new pages need to be allocated to allow k to be inserted, can be determined from the contents of the splitting page when it is first examined. We will denote this number to be n. In general, n ranges from 0 to $\log_2(2^{d-l.d})-1$, where d is the global depth and $l.d$ is the local depth of the splitting page before splitting.

The way our concurrent insertion algorithm deals with the above complication is to (1) have the splitting page as well as *all* the newly allocated pages in the database locked, (2) rearrange contents of these pages in private work space and allowing k to be inserted, (3) write the newly allocated pages back to the database, (4) update all the affected directory entries, (5) unlock all new pages, (6) write the splitting page back to the database, and finally (7) unlock the splitting page. One may choose to combine steps (5) and (7) together as the last step, but that is not strictly necessary. Note that during the entire operation no directory entries are locked and all search operations proceed without being blocked. In particular, in step (4) above, when multiple directory entries are updated, they are updated one by one without having to be updated all in one atomic action. It is assumed, however, that updating any single directory entry is atomic, as well as writing any single data page to the database.

We provide the definition of our insertion algorithm below, and the formal proof of correctness is presented in Section 4.

3.3. Deletion Algorithm

A deletion operation in an extendible hash file consists roughly of the same set of steps as the insertion operation, except that it needs not to deal with the issue of overflow and page splitting. For our purpose, as mentioned in the beginning of this section, we will ignore the issue of underflow and compaction. Therefore syntactically a deletion operation is just like an insertion operation that does not encounter overflow. For brevity, we do not include a formal definition of its algorithm.

Definition of the Insertion Algorithm.

```
Algorithm Insert(given key k);
begin
 hashing:
   Calculate k' = h(k)= b_0 b_1 ... b_{n-1};
 getpointer:
   read d, base; /* the global depth and base address of the directory D*/
   t := b_0 b_1 ... b_{d-1}; /* take the initial d bits of k' */
   x := get(D[t]);  /* D[t] is the t-th entry in D */
 lock_and_verify:
   xold := x;
   lock (x);
   x := get(D[t]);  /* re-read directory entry */
   do while xold ≠ x;  /* verification loop */
     unlock(x);
     xold := x;
     x := get(D[t]); /* re-read */
     lock (x);
     end;
 probe:
   A := get(x→p);  /* read data page p pointed to by x */
   if key k in A then 'error duplication', return;
 insertion:
   case 1. |A| < c  /* no need to split, where c is the capacity of a page */
     A := page.insert (A,k);
   case 2. |A| = c  /* split required; assume no directory doubling */
     n := number of new pages required;
     y_1,y_2,...,y_n := allocate n new pages in database;
     lock (y_1,y_2,...,y_n);  /* keep new pages locked */
     A, B_1,B_2,..,B_n := rearrange old A and B's, adjust l.d, insert k;
     for i = 1 to n do;
        put (B_i,y_i→p);  /* write B's into database */
        end;
     directory.modify(D,y_1,..,y_n);
     unlock (y_1,..,y_n);
   put(A, x→p);
   unlock(x);
end;

The function of directory.modify is

Procedure directory.modify(D,y_1,..,y_n);
begin
 for all directory entries j affected by split do;
   i := subscript of newly allocated page containing key range of entry j;
   put (y_i, D[j]);
   end;
end;
```

3.4. Discussion of Performance

In this subsection we briefly discuss how our proposed algorithm compares with "standard techniques". To our best knowledge, there has been little discussion of concurrent operations in extendible hashing in the literature. Therefore we will assume the "standard technique" in this case to be two-phase locking (2PL). Using 2PL, a search operation must (1) obtain a shared-lock on the directory entry, (2) obtain a shared-lock on the data page pointed to by the directory entry, (3) perform search and then release both locks. An insertion/deletion operation must (1) obtain an exclusive-lock on the directory entry, (2) obtain an exclusive-lock on the data page pointed to by the directory entry, and (3) perform updates and release both locks. If the insertion encounters the need to split the data page, it must additionally acquire exclusive locks on all directory entries affected by the split before updating these entries and before releasing *any* lock that it has acquired.

We first show that the standard technique is prone to deadlocks. Consider two adjacent directory entries d_1 and d_2 pointing to the same data page p where p currently has a local depth which is 1 less than the global depth. Two insertion operations I_1 and I_2 are run, one with a pseudo key mapped to d_1 and the other to d_2. Consider the following interleaved execution sequence using the standard technique:

I_1 locks d_1;
I_2 locks d_2;
I_1 locks p;
I_1 reads p and encounters overflow;
I_1 attempts to lock d_2;
I_2 attempts to lock p;

The two operations are now deadlocked.

Also, using the standard technique, while a search operation is never blocked by another search operation, it may be blocked by an insertion operation, and vice versa. In our algorithm, a search operation is never blocked by an insertion operation. Furthermore, in our algorithm, insertion operations do not have to acquire a lock on the directory entry before reading it, resulting in savings in locking overhead. The exact nature of the performance of the algorithm as compared to the standard technique would require additional analysis.

While the proposed algorithm offers freedom from deadlocks, potentially higher level of concurrency and savings in locking overhead, it is conceptually simple and should be just as easy, if not easier, to implement. The only additional cost in the proposed algorithm is the cost of verification. The search operation is potentially required to perform verification of the content of the directory entry previously read. This verification is needed only when the key desired is not found. The insertion algorithm is always required to perform verification. However, it can be argued that, when a verification is performed on a directory entry, the likelihood that the latter is memory-resident (i.e., in the buffer pool) is very high. This is true even if one does not in general keep the entire directory in memory. Therefore the cost of verification due to re-reading the directory entries is but a few memory accesses, and can be largely ignored.

4. Proof of Correctness

To show that the above algorithm is correct, we use the following steps:

(1) Show that all operations are deadlock-free and will terminate.

(2) Show that the search operation is correct.

(3) Show that the insertion/deletion operation is correct.

Assumptions:

(1) The database is finite in size. In other words, there exists a bound on the global depth.

(2) Each search/insertion/deletion operation consists of a sequence of read and write steps. Each read/write step involves a *data granule* which is either a *directory entry* or a *data page*. We assume that *each read and write step on such data granule is guaranteed to be atomic* by the underlying system, on top of which the current algorithms are implemented. In other words, we assume that the *get* and *put* steps in the definition of the algorithm are atomic steps. Note that this assumption can be supported by a synchronization mechanism at a lower level if necessary.

In order to provide a proof of correctness the criterion of correctness must first be articulated. We first give the following definitions before we discuss the criterion of correctness.

Definition. A *schedule* is a sequence of steps, each of which is in the form of $A_x(OP)$. The action A can be read (R) or write (W). The data granule is x, which can either be a directory entry, denoted as d, or a data page, denoted as p. OP is an operation, which may either be a search operation, denoted as S, which consists of two steps R_d and R_p, or an insertion/deletion operation, denoted as I, which consists of at least three steps, R_d, R_p and W_p. (Additional W_p and W_d may also appear in an insertion operation.) An operation can also be denoted, together with the key k of the record to be operated on, as S(k) or I(k).

Example. An example of a schedule is

$$<R_d(S),R_d(I),R_p(S),R_d(I'),R_p(I),W_d(I),W_p(I),R_p(I'),W_p(I')>,$$

in which three operations S,I and I' are involved.

Definition. Let A and A' be two steps in a schedule. We say that A $<$ A' if A occurs before A' in the schedule.

Example. In the above example schedule, $R_d(I') < W_d(I)$.

Criterion of Correctness. The unit of atomicity used for the purpose of defining correctness is the operation. In other words, the algorithm is correct if any interleaved schedule C that the algorithm allows is *equivalent* (i.e., *having the same net effect*) to *some* serialized execution SE of the same set of operations, subject to an additional restriction to be described in the next paragraph. The notion of "having the same net effect" is defined as follows: if a search operation fails(succeeds with record r) in C it also fails(succeeds with record r) in SE, and if an insertion/deletion operation succeeds(fails) in C it also succeeds(fails) in SE.

We first motivate the additional restriction, followed by the formal definition of the criterion of correctness. It is spurious to consider an interleaved schedule C correct if it results in a failure of a search operation (i.e., the search operation does not find the key it is looking for) while the search operation starts in C after an insertion operation that inserts that key has finished its last step. For example, consider an interleaved schedule C = $<...,W_p(I),...,R_p(S),..>$ and assume that I inserts key k in page p and $W_p(I)$ is its last step, S searches for key k and fails, and no deletion operation is involved in this schedule. While one may find the net result of schedule C equivalent to that of a serialized execution where S is run before I, it is meaningless to consider C correct. Therefore we define a more meaningful and more intuitive criterion of correctness, while retaining the basic notion of atomicity at the operation level, as follows:

> A schedule C of an interleaved execution of a set of search and insertion/deletion operations is correct if the net effect of C is equivalent to some serialized execution SE of the same set of operations s.t. if the last step of OP_1 is before the first step of OP_2 in C then OP_1 is before OP_2 in SE.

4.1. Proof of Termination

Lemma 1. All operations terminate.

Proof. Since no operation would hold any lock while *waiting* for another, no circular wait-for is possible, therefore no deadlock is possible. Therefore the termination proof amounts to proving that the potential loop in the operation will terminate. All operations potentially involve a loop of re-reading a directory entry. Given an operation O that involves such a loop, the loop in O terminates when the content of the last directory entry read is the same as that of the previous directory entry read. The content of any directory entry would change only when a split occurs in the data page that the directory entry points to. Since the number of times that any data page can split is bounded by the $\log_2(N)$, where N is the maximum number of pages allowed in this system, i.e., it is bounded by the maximum global depth of the system, the number of times the value of a directory entry will change is bounded by $\log_2(N)$. Therefore the loop of re-reading the directory entry in O will terminate.

4.2. The Search Operation is Correct

Lemma 2. The search operation is correct.

Proof. To prove that search operations are correct, we investigate what could possibly be the cause for it to be incorrect. Since all search operations terminate, they either succeed or fail. We consider each of these two cases separately.

(i) If a search operation S succeeds, i.e., if it finds the key it is looking for, then it must be correct. This can be shown as follows. Suppose the record it finds is r. Then there must exist an insertion operation I that inserts r. We can construct an equivalent serialized execution in which I is before S. If there also exists a deletion operation I_d which deletes r, then in the equivalent serialized execution we must let I_d be after S. This equivalent serialized execution is legal (according to the definition of correctness) as long as the last step of I_d did not come before the first step of S in our interleaved schedule. Suppose the last step of I_d did come before the first step of S. Then the only way for r to still linger in the database when S starts is for it to be in some data page p from out of which r was relocated (i.e., via page split) to a different page p', from which I_d deleted r, and p is still in a transient state containing r. However, if I_d is finished by the time S starts, the directory entry corresponding to r would have already be pointing to p'. S therefore could not possibly get access to p. Therefore the last step of I_d could not come before the first step of S in our interleaved schedule. Therefore the equivalent serialized execution is legal. Therefore the search operation is correct.

(ii) If a search operation S fails, it could fail incorrectly only when concurrent relocation exists. In other words, we want to show that if a search operation S(k) fails, and the *last* data page read by S(k) is p, then there exists no insertion operation I such that I relocates the key range containing k from p to p' $\neq$ p before S(k) reads p.

Suppose that there exists such an insertion operation I. Let d be the directory entry corresponding to the key k. Then I, before finishing, would first write the directory entry d and then writes p. We denote these steps as $W_d(I)$ and $W_p(I)$. We also denote the final steps of S(k) in reading directory d, reading page p, then re-reading (i.e., verifying) directory d as $R_d(S)$,$R_p(S)$ and $V_d(S)$. By definition of the failed search operation, the value read in $R_d(S)$ would be equal to that of $V_d(S)$. There are four cases of possible interleaving:

(1) $W_d(I)<R_d(S)$ and $W_p(I)<R_p(S)$. In this case, since I relocates k from p to p', the directory entry read by S(k) should not contain a pointer to p, therefore S(k) would not have read p, contradictory.

(2) $W_d(I)<R_d(S)$ and $R_p(I)<W_p(S)$. In this case, similar argument as above, S should not have read p, also contradictory.

(3) $R_d(S)<W_d(I)$ and $W_p(I)<R_p(S)$. In this case, the V_d step of S(k) would have read the new pointer (i.e., to p') which is not equal to the old pointer (i.e.,to p) read in the R_d step, contradictory.

(4) $R_d(S)<Wd(I)$ and $R_p(S)<W_p(I)$. In this case, S(k) would read the old content of page p before I relocates k out of p, contradictory to definition of I.

Therefore we conclude that there exists no insertion operation I that could have relocated k out of p before S(k) reads p as its last data page to read before termination. Therefore the search operation is correct.

Combining (i) and (ii) above we conclude that the search operation is correct.

4.3. Insertion/Deletion is Correct

Since search operations do not update the database, they would not affect the correctness of an insertion operation. Therefore to prove that insertion operations are correct we need only to take into account interferences among insertion operations themselves, and between insertion and deletion.

We introduce some notations to refer to specific steps of an insertion/deletion operations. We are interested in the tailing end of the steps in these operations, i.e., those in the final round of the verification loop and those at the very end. The sequence of the read/write steps of the *last* round of the verification loop of an insertion/deletion consists of $<R_d,L_p,V_d>$, where L_p stands for exclusive lock of p, V_d stands for the step of verifying the content of the directory entry read in R_d. We denote R_d and V_d in this last round of verification as R^*_d and V^*_d. Note that by definition, the content of the directory entry read in R^*_d and V^*_d must be identical. After the last round of verification, the page pointed to by the value read in R^*_d is read. We denote this step as R^*_p. The final sequence of steps of an insertion/deletion operation that does not involve a split is $<W_p,U_p>$, where W_p and U_p stand for write and unlock of the page p which was locked between R^*_d and V^*_d during the last round of verification. The sequence for one involving a split is $<W_d,U,W_p,U_p>$. where U unlocks all new pages, and W_d is the last directory entry update. We will denote these last steps of directory update and page write as W^*_d and W^*_p. Note that the directory entry written in W^*_d may not be the same entry read in R^*_d or V^*_d.

Definition. Let I be an insertion/deletion operation. We define the range of the keys relocated by I as the *migration set* of I, denoted as *migration*(I).

Since the deletion operation never relocates any record, its migration set is obviously empty.

Lemma 3. Any two concurrent insertion/deletion operations I_1 and I_2 always interleave correctly.

Proof. Let the key to be operated by I_1 be k_1 and that by I_2 is k_2. Assume without loss of generality $R^*_p(I_1)<R^*_p(I_2)$. We consider the following cases, and for each case we show that they interleave correctly.

(1) *migration*(I_1) contains k_2. Then I_1 must update the directory entry for k_2, denoted as d_{k_2} that I_2 needs to read. Two subcases are considered. (i) I_2 reads d_{k_2} in the final round *after* I_1 updates it. (i.e., $W_{d_{k_2}}(I_1)<R^*_d(I_2)$ where $W_{d_{k_2}}(I_1)$ is the step in which I_1 updates d_{k_2}.) Then I_2 cannot read the page pointed to by d_{k_2} it read until I_1 releases the lock on it, by which time I_1 would have finished all its operations on directories. Therefore the only dependency that the directory entry operations can possibly induce between I_1 and I_2 are I_1 giving to I_2. Since I_1 will not read or write any data pages after I_2 writes them, the only dependency that the data page operations can induce are also I_1 giving to I_2. Therefore any interleaving between I_1 and I_2 is equivalent to

serializing I_1 before I_2, therefore they are correct. (ii) $W_{d_{k_2}}(I_1) > R^*_d(I_2)$, i.e., I_2 reads d_{k_2} before I_1 updates it. In this case I_2 will be forced to wait till I_1 releases its lock on the page it is splitting, by which time $W_{d_{k_2}}(I_1)$ would have already occurred, which means $V^*_d(I_2)$ would have failed, contradictory.

(2) *migration*(I_1) does not contain k_2. There are also two subcases. (i) *migration*(I_2) contains k_1. Let the page read in $R^*_p(I_1)$ be p. I_1 holds a lock on p till finish. Since $R^*_p(I_1) < R^*_p(I_2)$, I_2 can read p (if it ever does) only after I_1 is finished. Therefore the only possible dependency is I_1 giving to I_2, therefore the interleaving is correct. (ii) *migration*(I_2) does not contain k_1. In this case no conflict can occur between I_1 and I_2 on directory entries. And since data pages are two-phase locked, the interleaving must be correct.

From the above three lemmas, one concludes that our algorithms for concurrent search/insertion/deletion operations are correct. *Q.E.D.*

5. Conclusion

We have presented an algorithm for synchronizing concurrent operations in extendible hash files. The algorithm allows the search operations to proceed concurrently with insertion operations without having to acquire locks on the directory entries or the data pages. It also allows concurrent insertion/deletion operations to proceed without having to acquire locks on the directory entries. Moreover, because at most a single lock is required at any time for each of these operations, the algorithm is deadlock free. The algorithm combines the method of verification used in the optimistic concurrency control algorithm and the special structures of operations in extendible hash files together to yield a higher level of concurrency as well as a lower synchronization overhead.

In this paper we ignore the issues of underflow and compaction. We also did not discuss the issue of directory expansion (i.e., doubling) extensively. However, the latter can be handled by a straightforward extension of the current algorithm, to require that (1) every time a verification (i.e., re-read) of the content of the directory entry is performed, the global depth and the base address of the directory are also re-read, and that (2) the old version of the directory is carried around in memory for a specified period of time. (Incidentally, (2) can be relaxed if the bits in the pseudo key used to index into the directory are the suffix rather than the prefix of the pseudo key.) If these prove to be practical to implement, directory expansion can be allowed to proceed concurrently with search operations. In any case, database quiescence can always be resorted to as the method for handling directory expansion.

The algorithm can also be applied to handle dynamic perfect hash files [YD84]. The dynamic perfect hash file structure employs a method that optimizes the space requirement of the directory used in an extendible hash file, thus rendering it more practical to consider the directory being memory resident. However, the structure of the directory in a dynamic perfect hash file is more complicated than that of an ordinary extendible hash file, and extensions of the current algorithm must be sought for.

Acknowledgement: Work reported herein has been supported, in part, by the Naval Electronic Systems Command through contract N00039-85-C-0571.

Wei-Pang Yang is currently on leave from National Chiao-Tung University, Taiwan, R.O.C., on a grant from Ministry of Education, R.O.C.

References

[BS77]
Bayer, R. and Schkolnick, M. Concurrency of operations on B-trees. Acta Inf. 9, 1977.

[FNPS79]
Fagin, R., Nievergelt, J., Pippenger, N. and Strong, H.R., Extendible Hashing - a fast access method for dynamic files," ACM Transactions on Database Systems, 4, 3, September 1979.

[HC85]
Hsu, M. and Chan, A. Partitioned two-phase locking. First International Workshop on High-Performance Transaction Systems, September 1985.

[HM83]
Hsu, M and Madnick, S.E. Hierarchical database decomposition: a technique for database concurrency control. Proceedings of 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 1983.

[KP79]
Kung, H.T. and Papadimitriou, C.H. An optimality theory of concurrency control for databases. ACM SIGMOD Conference Proceedings, 1979.

[KR81]
Kung, H.T. and Robinson, J.T. Optimitic methods for concurrency control. ACM Trans. on Database Systems, 6, 2. June 1981.

[KS83]
kedem, Z. and Silberschatz, A. Locking protocols: from exclusive to shared locks. Journal of ACM, 30, 4, October 1983.

[LY81]
Lehman, P.L. and Yao, S.B. Efficient locking for concurrent operations on B-trees. ACM Trans. on Database Systems, 6, 4, December 1981.

[MR85]
Mond, Y. and Raz, Y. Concurrency control in B+ trees databases using preparatory operations. Proc. of VLDB 85, Stockholm.

[O'Neil85]
O'Neil, P. Escrow transactions permitting concurrent record updates. First International Workshop on High-Performance Transaction Systems, September 1985.

[Papadimitriou79]
Papadimitriou, C.H. The serializability of concurrent database updates. Journal of ACM 26, 4, October 1979.

[SK80]
Silberschatz, A. and Kedem, Z. Consistency in hierarchical database systems. Journal of ACM, 27, 1, January 1980.

[YD84]
Yang, W.P. and Du, M.W. A dynamic perfect hash function defined by an extended hash indicator table. Proc. of VLDB 84, Singapore, August 1984.