Concurrent Operations in Extendible Hashing

Meichun Hau
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, HIC85, 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 synchronizes concurrent operations on a file structured using extendible hashing [FNPS79]. Extendible hashing is a form of dynamic hashing 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 into 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.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment.

Proceedings of the Twelfth International Conference on Very Large Data Bases

Kyoto, August, 1986

-241-
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%.
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.

Definition of the Search Algorithm.

Algorithm Search(given key k);
begin
initialization:
   xold=0;
hashing:
calculate k' = h(k) = b0b1...bn-1;
getpointer:
   read d, base ; /* the global depth and base address of the directory D */
   t := b0b1...bn-1 ; /* take the initial d bits of k' */
   x := get(D[t]); /* D[t] is the t-th entry in D */
   xold := x;
   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(ies) 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 \( \log_2(2^d-1) \), 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 \( n \)'s 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 \( k' = h(k) = b_0b_1 \cdots b_{d-1} \);

- **getpointer:**
  read \( d \), base; /* the global depth and base address of the directory \( D^* \)*/
  \( t := b_0b_1 \cdots b_{d-1} \); /* 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 \( xold \neq x \); /* verification loop */
  unlock \( x \);
  xold := 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 */
  \( A := pageinsert(A,k); \)
  case 2. \( |A| = c \) /* split required; assume no directory doubling */
  \( n := number of new pages required; \)
  \( y_1, y_2, \ldots, y_n := allocate \( n \) new pages in database; \)
  lock \( (y_1, y_2, \ldots, y_n); /* keep new pages locked */
  \( A, B_1, B_2, \ldots, B_n := rearrange old A and B's, adjust l.d, insert k; \)
  for \( i = 1 \) to \( n \) do;
    put \( (y_i, y_{i+p}); /* write B's into database */
  end;
  directory.modify(D,y_1, \ldots, y_n);
  unlock \( (y_1, \ldots, y_n); \)
  put\( (A, x=p); \)
  unlock \( x \);
  end;

The function of directory.modify is

Procedure directory.modify(D,y_1, \ldots, 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:

\[
\begin{align*}
I_1 & \text{ locks } d_1; \\
I_2 & \text{ locks } d_2; \\
I_1 & \text{ locks } p; \\
I_1 & \text{ reads } p \text{ and encounters overflow; } \\
I_1 & \text{ attempts to lock } d_2; \\
I_2 & \text{ attempts to lock } p; \\
\end{align*}
\]

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 will 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.
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 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 for another, no circular wait-for is possible, therefore no deadlock is possible, therefore no deadlock is possible.

**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 \( I' \) which deletes \( r \), then in the equivalent serialized execution we must let \( I' \) be after \( S \). This equivalent serialized execution is legal (according to the definition of correctness) as long as the last step of \( I' \) did not come before the first step of \( S \) in our interleaved schedule. Suppose the last step of \( I' \) 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 \( I' \) deleted \( r \), and \( p' \) is still in a transient state containing \( r \). However, if \( I' \) is finished by the time \( S \) starts, the directory entry corresponding to \( r \) would have already be pointed to \( p' \). Since \( r \) therefore could not possibly get access to \( p' \). Therefore the last step of \( I' \) 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 \( p' \) 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 write \( p \). 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 \( R_p(S) < W_p(I) \). 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. \( R_d(S) < W_d(I) \) 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 \( <R_d,I_d,V_d> \), where \( I_d \) stands for the 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 \( R_d'' \).

The final sequence of steps of an insertion/deletion operation that does not involve a split is \( <W_p,U_p,V_p> \), 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 \( <W_p,U_p,V_p,W_p'> \), where \( U_p \) unlocks all new pages, and \( W_p' \) 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 \( W_p' \) 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 \( I_2 \) be \( k_2 \). Assume without loss of generality \( k_1 < k_2 \). 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) \( I_1 \) 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 \( d_{k_2} \) right 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 \( I_2 \) is \( I_1 \) 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. (i) $W_{d,k}(i_1) > R_{d,k}(i_1)$, i.e., $I_2$ reads $d_k$ 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 $W_{d,k}(i_1)$ would have already occurred, which means $V_{d,k}(i_2)$ would have failed, contradictorily.

(2) migration($i_1$) does not contain $k_2$. There are two subcases. (i) migration($i_2$) contains $k_1$. Let the page read in $R_{d,k}(i_1)$ be $p$. $I_1$ holds a lock on $p$ till finish. Since $R_{d,k}(i_1) < R_{d,k}(i_2)$, $I_2$ can read $p$ (if it ever does) only after $I_1$ is finished. Therefore the only possible dependency in $I_1$ 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.