ARIES/LHS: A Concurrency Control and Recovery Method Using Write-Ahead Logging for Linear Hashing with Separators

C. MOHAN

Data Base Technology Institute, IBM Almaden Research Center, San Jose, CA 95120, USA
mohan@almaden.ibm.com

Abstract Larson has proposed a dynamic hashing algorithm called *Linear Hashing with Separators* (LHS) that, given a unique primary key value, uses a table in memory to allow the retrieval of the corresponding record in the file in one page access to secondary storage. Larson considers LHS to be *the first practical method offering one-access retrieval for large dynamic files.* He did not discuss the impact of concurrent operations by different users, some of whom are reading the file while others are performing operations like inserts, deletes, updates, file expansions or file contractions which can cause relocations of records. We present a method, called ARIES/LHS (*Algorithm for Recovery and Isolation Exploiting Semantics for Linear Hashing with Separators*), for controlling such concurrent operations with fine-granularity (e.g., record) *locking*, while guaranteeing serializability. ARIES/LHS prevents rolling back transactions from getting involved in deadlocks. It also includes recovery techniques for handling transaction and system failures, while allowing multiple operations in each transaction.

1. Introduction

Even though extendible hashing has been studied for a long time (see [EnDu88, Lars88] and the references in them), very little has been reported in the literature on the concurrency control of multiple transactions simultaneously accessing such structures. Whatever little has appeared [Elli83, Elli87, HsTY90, HsYa86, OuAb89, ShGo88] is usually based on a very simplified notion of a transaction. Generally, each transaction is assumed to consist of only one action (insert, delete, or retrieval) against the search structure. The problems associated with guaranteeing serializability become much more complicated when one considers transactions consisting of multiple actions (for examples of these in the context of B^+-trees, the reader is referred to [Moha90, MoLe92]). An exception is the work reported in [KeRu88]. Unfortunately, that paper does not give enough details about the locking protocols used. Even though [Kuma89] also supports multiaction transactions, the smallest granularity of locking is a page. [Elli83, HsYa86, KeRu88, Kuma89] deal with only extendible hashing, rather than the more complicated linear hashing. In any case, none of the papers deals with the problem of providing recovery from transaction and system failures for a general model of transactions with fine-granularity locking. Some of the concurrent activities permitted by the algorithms in the literature will cause inconsistencies when one considers failures and recovery.

This paper concentrates on the concurrency control and recovery (***CCR***) aspects of multiaction transactions accessing dynamic hashing-based storage structures and introduces the ***ARIES/LHS (ARIES for Linear Hashing with Separators)*** method. Due to space constraints, most of the information regarding recovery is not presented here. Recovery is discussed further in [Moha92]. ARIES/LHS is an adaptation of the ARIES (*Algorithm for Recovery and Isolation Exploiting Semantics*) method which was introduced in [MHLPS92] and which has been implemented, to varying degrees, in the IBM products OS/2 Extended Edition Database Manager™, Workstation Data Save Facility/VM and DB2™ V2, in the IBM Research prototypes Starburst and QuickSilver, in Transarc's Encina™ Product Suite, and in the University of Wisconsin's EXODUS extensible DBMS and Gamma data base machine. Extensions to the original ARIES algorithm are described in [MoPi91, RoMo89].

The rest of the paper is organized as follows. In the next subsection, we give an introduction to Larson's hashing method without getting into details that are irrelevant for CCR. This includes a listing of the operations that are supported by the hashing method. In the following subsection, we list the problems relating to CCR that need to be addressed. In section 2, we present the locking and latching protocols of ARIES/LHS. Section 3 presents our conclusions.

1.1. Linear Hashing with Separators

A dynamic hashing method called *Linear Hashing with Separators* (LHS) for managing *fixed* length records was presented by Larson in [Lars88]. Unlike Larson, we deal with varying length records and updates of records also. Larson considers LHS to be *the first practical method offering one-access retrieval for large dynamic files.* It is a ***dynamic hashing method*** because it allows the file size to grow or shrink dynamically (i.e., not all the records in the file are relocated when the file size changes). It is a ***linear hashing method*** because there is a linear relationship between the amount of storage *used* and the amount of storage *required* for storing the records that exist in the file at anytime. Unlike in the earlier methods, in LHS, storage utilization is controllable by the user.

In LHS, a file's address space (i.e., the space into which records are hashed) always consists of a set of groups of pages. Initially, when the file is loaded with the first set of records, all the **N** groups have the same number of pages, namely **n0**. As more records are added to (respectively, deleted from) the file, based on the desired maximum (minimum) storage utilization α (β), the file's address space gets expanded (contracted) by adding (removing), at (from) the end of the file, a page to (from) one of the groups at a certain point in time. All groups are expanded (contracted) in turn. Each time all the groups have been ex-

™DB2, IBM and OS/2 are trademarks of the International Business Machines Corp. Transarc is a registered trademark of Transarc Corp. Encina is a trademark of Transarc Corp.

1063-6382/93 $03.00 © 1993 IEEE

panded (contracted) by one page, a ***partial expansion (contraction)*** of all the groups is said to have been completed. During each partial expansion (contraction) of a group, some of the records belonging to the group may be relocated to (from) the new (removed) page from (to) the existing (remaining) pages of the same group. During an expansion, a hashing function which takes as its input the primary key of the records of the file is used to determine which existing records of the expanding group should be relocated. Once enough (this number should be equal to n0) expansions (respectively, contractions) have taken place for the group size to have doubled (halved), a ***full expansion (contraction)*** is said to have been completed. At this point, the number of groups is assumed to have doubled (halved) and the state is again one in which each group has only n0 pages.

Whenever a record needs to be inserted, the primary key of the record, which is required to be unique, is hashed, using the function *home_addr*, to obtain the ***home page (address)*** in which that record belongs. Since the home page may not have enough space for the record and hence an alternate page may have to be found, LHS assumes that, given the primary key K of a record, the ***probe sequence*** $H(K) = (h_1(K), h_2(K), \ldots, h_m(K))$ may be computed, where $h_1(K)$ is the home page of the record. The probe sequence is uniquely determined by the key, and defines the order in which pages will be searched when inserting or locating the record. In fact, LHS uses ***linear probing*** which means that the probe sequence is h, h+1, h+2, ... where h is $h_1(K)$. Computing the home address is not trivial. A hashing function is used to determine the home page for the *initial* size (N*n0) of the file. Then, the complete sequence of partial expansions that have taken place to bring the file's size to its current value must be considered. During each of those partial expansions, LHS must determine whether the record would have been relocated had it existed and if so to which page. This computation determines the home address of the record given the *current* size of the file. Due to overflows caused by space unavailability, the record may not exist currently in its home address. While the exact algorithm used for the home page computation is not of interest here, the important point to note with respect to CCR is that we cannot permit a file expansion or contraction to occur when home page computations and the subsequent probings are going on since it could lead to erroneous results.

In order to handle overflows due to space unavailability in the home page or the pages following the home page of a record, the primary key is also used to compute the ***signature sequence***, $S(K) = (s_1(K), s_2(K), \ldots, s_m(K))$, where the signature $s_i(K)$ is used when probing the page $h_i(K)$. Thus, a new signature is computed for each probe using the function *sign*. A random number generator which uses the key as its seed may be used for this purpose. Let us denote by ***MaxSig*** the maximum possible signature value.

Associated with every page that is currently part of the file is a ***separator value***. The latter is a *signature value* such that all the records that are currently stored in the corresponding file page have signatures which are *less than* it. The separator values for all the pages of the file are stored in a main-storage-resident data structure called the ***separator table*** (***ST***). Each ST entry (***STE***) consists of a separator value. For a page from which no records have been overflowed to the following pages, the signature value will be MaxSig + 1. This is the value that the entries of ST are initialized with. In an STE, there is no need for an explicit pointer to the corresponding data page, since the ith STE corresponds to the ith page in the file. Typically, only one byte is needed to store the separator of each page and hence, for a file of size 4GB (1M 4K-sized pages), the table will require only 1MB of memory. As the file is expanded or contracted, the number of entries in ST will also have to vary accordingly. If a record to be inserted or retrieved for read, update, or deletion probes a page having a separator *greater than* the signature of the record for the probed page, then that record belongs on that page. Otherwise, because of the use of linear probing, the separator for the next entry in ST is compared with the signature of the record for the next page. This process is continued until a page is found whose separator is *greater than* the signature of the record for that page. Such a page is called the ***current page (address)*** of the record. Note that the current page of a record may be different from the home page of the record only if all pages starting from the home page until (but excluding) the current page have overflowed. Thus, the current address of a record depends on its home address, its signature sequence, and the separators in the separator table.

The operations that we have to consider while designing the latching, locking, and logging protocols are described next.

DNR - Delete a record with No record Relocation (i.e., earlier, no records had overflowed from current page - page's current signature must be *equal to* MaxSig + 1).

DR - Delete a record that possibly causes record Relocation (i.e., earlier, records had overflowed from the current page and some of them may now be able to fit on this page - page's current signature must be *less than* MaxSig + 1). This may cause some records which are not currently in their home pages to be moved up (i.e., towards the beginning of the file) to occupy the space freed up due to the record deletion. Since we are dealing with varying length records, even though there are overflowed records, none of them may be small enough to fit into the space made available by the deleted record. The moved records must be the ones with the *lowest* signatures (amongst the moveable records) for the pages into which they are moved. These upward movements in turn might allow other records, if any, which are not presently in their home pages to be moved up. Thus, DR might affect a set of *contiguous* pages starting from the page in which the user-initiated deletion is performed (i.e., the *current page*). This moving up of records is performed in two passes: starting from the current page and ending with the page with no overflows, the first pass identifies and copies into a work area those records that are not in their home pages; the second pass is the one which actually modifies the data pages in a top-down fashion (lowest numbered to highest numbered pages) to make as many of those records with the *lowest* signatures as possible get back to their home pages and at least get the other records closer to their home pages. Separators of those pages into/from which records were relocated will be appropriately modified to reflect their new contents. In ARIES/LHS, we take advantage of this top-down, two-pass strategy to improve concurrency.

INR - Insert a record with No record Relocation (i.e., current page has enough free space).

IR - Insert a record that causes record Relocation (i.e., not enough space on the *current* page of the new record). The records which are forced out of the page must be the ones with the *highest* signatures for that page. This will generally cause separators to be changed starting from the current page of the record being inserted until the closest following page whose signature is MaxSig + 1. In the case of varying length records, it may stop earlier or continue beyond that point. In ARIES/LHS, we take advantage of the fact that the separators are changed in a top-down fashion. Note that IR does a single pass relocation, whereas DR does a two pass relocation.

R - Retrieve a record, given the primary key value. Only the current page of the record needs to be accessed.

UR - Update a record with record Relocation (i.e., record size changed in such a way that record relocation was necessary - if the record size decreased then it is like DR; otherwise, it is like IR).

UNR - Update a record with No record Relocation (i.e., record size did not change or it changed without requiring any records to be relocated).

FC - File Contraction. This results in the address space for hashing being contracted. Any existing records for which the home page is the page being removed from the file must be moved over to one of the other pages of the group being contracted. The effects of changing the home pages of records are similar to what happens during record insertions, as in IR and INR. *Multiple* sets of contiguous pages may be affected during the consequent record relocations since the new home pages for the affected records may not be contiguous.

FE - File Expansion. This results in the address space for hashing being expanded. As explained before, some existing records of the group being expanded may be relocated to the newly added page which in turn might allow other records that are not in their home pages to be moved up, as in DR. In FE, unlike in DR, more than one set of contiguous pages may be involved in the moving up of records since the records being moved to the new page may come from many discontiguous pages of the group being expanded.

Note that DR, IR and UR may only change the *current* pages of some records, but FC and FE, in addition to changing the *current* pages of some records, may also change their *home* pages. INR, DNR and UNR affect only one page. IR, DR and UR affect one set of contiguous pages. FC and FE, which should be very infrequent operations, may affect more than one set of contiguous pages.

1.2. Problems

Some of the problems to be dealt with in providing concurrency control and recovery for LHS are:

1. How to support fine-granularity locking (e.g., record locking), as is done for B^+-trees in [Moha90, MoLe92]?
2. When one transaction is performing record relocations, due to DR, IR or UR, which may involve waiting for many I/Os to take place, how to permit concurrent record relocations, reads and modifications by other transactions?
3. How to guarantee serializability (Repeatable Read (RR) or level 3 consistency [Gray78]) - i.e., how to ensure that even if a record with a certain primary key value is *not* found by transaction T1, no other transaction is allowed to insert such a record before T1 terminates?
4. How to perform logging to support recovery using write-ahead logging (WAL) [Gray 78, MHLPS92] so that, in the interest of high concurrency, even *uncommitted* changes of one transaction may be migrated to a different page by another transaction during record relocations and the system will still be able to rollback the first transaction correctly, as is done for B^+-trees in [MoLe92]?
5. How to ensure that the relocations performed by one transaction while other transactions are simultaneously accessing the ST and the data pages for reading, modifications, and relocations of their own do not confuse the other transactions and do not cause inconsistencies between the contents of the ST and those of the file pages?
6. How to ensure that any synchronization techniques (e.g., *latches*)[1] that are used to provide physical consistency do not cause deadlocks, especially during rollbacks?

2. ARIES/LHS

In this section, we discuss the ARIES/LHS latching and locking protocols which are followed during the forward processing of transactions for the different LHS operations (i.e., 'R', 'I', 'D' and 'U'). But, before doing that, we discuss how the LHS operations are supported. We use 5 procedures for this purpose: *Find_Current_Page_No_Reloc* (see Figure 3), given a primary key and the type of operation, assumes that record relocations will not be necessary and returns the identifier of the *current* page of the corresponding record. For the delete operation alone, it may determine that record relocations may be necessary and return *perform DR*. *Access_Current_Page* (see Figure 4), given a primary key, the type of operation, the current page ID, and, possibly, a record, does the corresponding operation on the current page if definitely no record relocations are needed. For the insert operation, it may determine that already another record with the same primary key exists and return *unique_key_violation*. For the insert and update operations, it may determine that relocations may be necessary and hence, without performing the desired operation, return *perform IR* or *perform UR*. *Find_Current_Page_With_Reloc* (see Figure 5) does essentially the same work as *Find_Current_Page_No_Reloc* except that it anticipates the possibility of record relocations

[1] We assume that the reader is familiar with the concept of a latch, the different degrees of consistency (e.g., *repeatable read*, *cursor stability*), the different durations (e.g., *instant*, *commit*) and modes (e.g., *S*, *X*, *IS*, *IX*, *SIX*) of locking and latching, and the differences between locks and latches, as described in [Gray78, MHLPS92, Moha90].

```
PROCEDURE Read (Key,Rec)
Find_Current_Page_No_Reloc(Key,'R',Pg)
Access_Current_Page(Key,'R',Pg,Rec)
RETURN(Return_Code)

PROCEDURE Update (Key,Rec)
Find_Current_Page_No_Reloc(Key,'U',Pg)
Access_Current_Page(Key,'U',Pg,Rec)
IF Return_Code='Perform UR' THEN
  Find_Current_Page_With_Reloc(Key,'U',Pg)
  IF update causes record expansion THEN
    Do_UR_With_Rec_Expand_OR_IR(Key,'U',Pg,Rec)
  ELSE
    Do_UR_With_Rec_Contract_OR_DR(Key,'U',Pg,Rec)
RETURN(Return_Code)
```

```
PROCEDURE Delete (Key)
Find_Current_Page_No_Reloc(Key,'D',Pg)
IF Return_Code='Perform DR' THEN
  Find_Current_Page_With_Reloc(Key,'D',Pg)
  IF Return_Code='success' THEN
    Do_UR_With_Rec_Contract_OR_DR(Key,'D',Pg,Rec)
ELSE Access_Current_Page(Key,'D',Pg,Rec)
RETURN(Return_Code)

PROCEDURE Insert (Key,Rec)
Find_Current_Page_No_Reloc(Key,'I',Pg)
Access_Current_Page(Key,'I',Pg,Rec)
IF Return_Code='Perform IR' THEN
  Find_Current_Page_With_Reloc(Key,'I',Pg)
  Do_UR_With_Rec_Expand_OR_IR(Key,'I',Pg,Rec)
RETURN(Return_Code)
```

Figure 1: Psuedo-Code for Implementing Different Record Operations

being needed and takes different preparatory synchronization steps which will be explained later. *Do_UR_With_Rec_Contract_OR_DR* (see Figure 6) performs a record update operation which causes record contraction or a record deletion operation when either operation is possibly accompanied by relocations of records. *Do_UR_With_Rec_Expand_OR_IR* (see Figure 7) performs a record update operation which causes record expansion or a record insertion operation when either operation is possibly accompanied by relocations of records. For brevity, no pseudo-code is given for implementing the FC and FE operations which, as explained before, are very similar to DR and IR.

For the benefit of the reader who is interested in the detailed workings of the above procedures, their pseudo-code is presented in the figures identified above. In order to understand the following discussions, the reader should keep in mind the descriptions given for the different operations in the section "1.1. Linear Hashing with Separators". The pseudo-code fragments in Figure 1 show the implementations of the following record operations using the previously-described 5 procedures: (1) a read operation, which, given a primary key, returns the corresponding record, if it exists; (2) a delete operation, which, given a primary key, deletes the corresponding record, if it exists; (3) an insert operation, which, given a primary key and a record, inserts the record when no record already exists with the same primary key; and (4) an update operation, which, given a primary key and the new version of the corresponding record, updates the existing version of that record. Record insert, delete, and update operations are initiated to start with as INR, DNR and UNR, respectively. Later, if it is suspected that relocations may be necessary, then these operations are retried as IR, DR and UR, respectively.

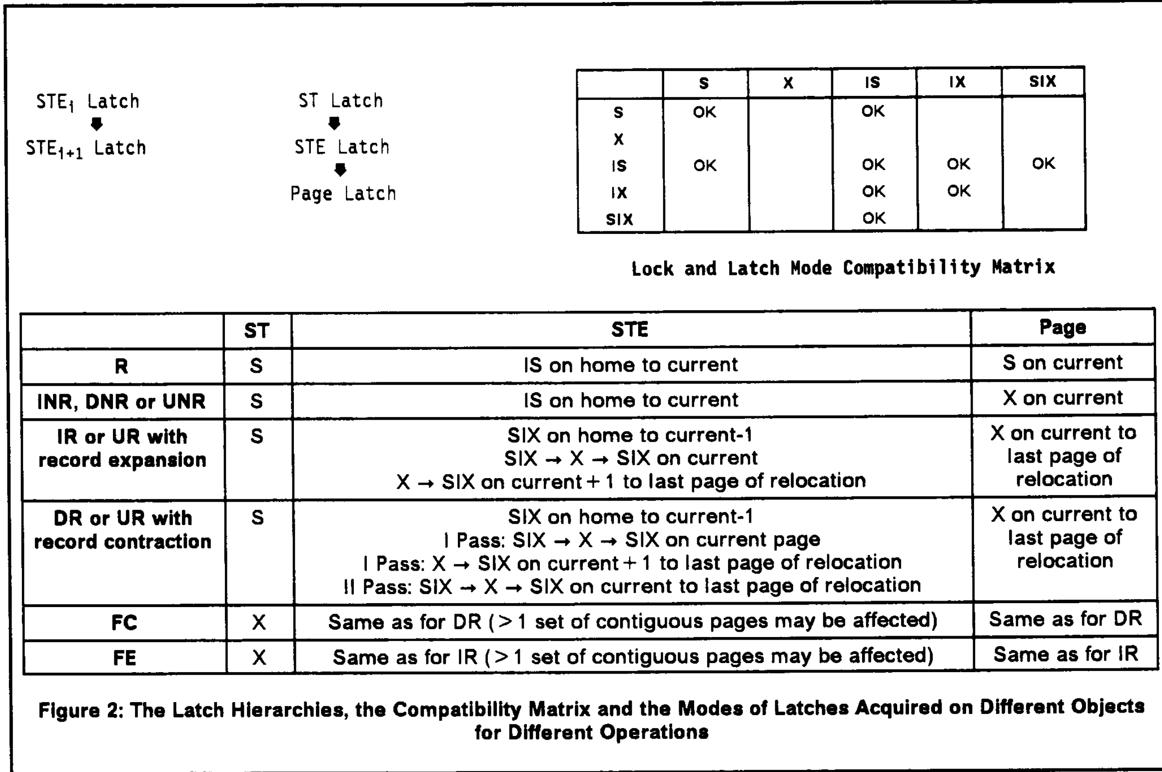
2.1. Locking

Since every record in an LHS file has to have a unique key, the locking protocol turns out to be fairly straightforward. Locks need to be acquired on records only by transactions which actually insert, delete, read, or update records. *Before* accessing the ST and the file, the R operation gets the S mode lock on its target, while the DNR, DR, INR, IR, UNR and UR operations get the X mode lock on their target records. These locks are retained for transaction duration to guarantee serializability. This locking will also ensure that the primary key uniqueness constraint can be enforced correctly and efficiently. Transactions do **not lock** those records which they only relocate from one page to another during operations like DR, FC, FE, IR and UR. This supports very high concurrency since even *uncommitted* changes of one transaction may be relocated to another page by another transaction. It also reduces the overhead of performing those relocations. The logging and recovery protocols should be prepared to handle these situations to ensure correctness in the face of concurrent activities and various types of failures. This is accomplished by making use of the *nested top actions* feature of ARIES (for more explanations, see [Moha92]).

If a transaction, say T1, were to try to retrieve a record with key K and it is noticed that no such record exists, then ARIES/LHS has to guarantee that no other transaction will be allowed to insert such a record (and commit) until T1 terminates. It should also be guaranteed that K is not currently in the uncommitted deleted state. These can be accomplished by making T1 obtain a commit duration S lock on K and requiring that an inserter or deleter X lock for commit duration the key value *before* the insert or delete is allowed to proceed.

2.2. Latching

The trickiness with respect to concurrency control in ARIES/LHS comes in the design of the protocols needed to synchronize accesses to the entries in ST and the data file pages by different transactions so that erroneous answers are not produced, the transaction *ACID* (atomicity, consistency, isolation and durability) properties are guaranteed, and the ST entries' separator values and the file



	S	X	IS	IX	SIX
S	OK		OK		
X					
IS	OK		OK	OK	OK
IX			OK	OK	
SIX			OK		

	ST	STE	Page
R	S	IS on home to current	S on current
INR, DNR or UNR	S	IS on home to current	X on current
IR or UR with record expansion	S	SIX on home to current-1 SIX → X → SIX on current X → SIX on current + 1 to last page of relocation	X on current to last page of relocation
DR or UR with record contraction	S	SIX on home to current-1 I Pass: SIX → X → SIX on current page I Pass: X → SIX on current + 1 to last page of relocation II Pass: SIX → X → SIX on current to last page of relocation	X on current to last page of relocation
FC	X	Same as for DR (> 1 set of contiguous pages may be affected)	Same as for DR
FE	X	Same as for IR (> 1 set of contiguous pages may be affected)	Same as for IR

Figure 2: The Latch Hierarchies, the Compatibility Matrix and the Modes of Latches Acquired on Different Objects for Different Operations

pages' contents are kept mutually consistent. We use *latches* for performing such synchronizations. The rest of this section describes the ARIES/LHS latching protocols as they are implemented by the previously-introduced 5 procedures.

We associate with each data page that has been brought into the buffer pool in virtual storage a ***data page latch***. We associate with each STE a latch called the ***STE latch***. There is also a latch associated with the ST as a whole called the ***ST latch***. The ST latch is used to synchronize FE and FC with the other operations.

Since latches would have to be acquired during rollbacks, we do not wish to let deadlocks be caused by waits involving latches. Dealing with deadlocks during rollbacks has been a problem in the past in System R and R* (see [MoLO86] for more discussions). In the ARIES family of algorithms, we always avoid rolling back transactions from ever getting involved in deadlocks (see [MHLPS92, Moha90, MoLe92]). We need to ensure that latch usage is such that no deadlocks involving waits on them ever arise. In order to avoid such deadlocks, a hierarchical ordering is imposed amongst the latches. The ordering is as follows: ST latch, STE latches and page latches (see Figure 2). That is, when the ST latch is held, an STE latch may be requested *unconditionally* and once that is obtained, the corresponding data page latch may be requested *unconditionally*. Amongst the STE latches themselves, the ordering is from top to bottom. That is, a latch may be held on the ith STE while waiting *unconditionally* for the latch on the $(i+1)^{th}$ STE (see the pseudo-code in Figure 3, Figure 5, Figure 6 and Figure 7). In addition to avoiding deadlocks, we also increase concurrency by avoiding waiting for locks *unconditionally* while holding latches.

In order to detect that a page that is about to be accessed for a *modification* is a participant in an ongoing record relocation operation, we introduce a bit, called the ***relocation bit*** (***R_Bit***), which is set to '1' **in the STE** for the page when that page is involved in a relocation operation (DR, IR or UR). It will be reset to '0' once **all** the changes relating to the relocation operation have been completed. See the pseudo-code in Figure 6 and Figure 7. The R_Bit is used to delay the updating of a page by a second transaction when the page is involved in an on-going relocation operation by a first transaction until the relocation is completed. This is necessary since otherwise the second transaction may commit, the first transaction may not complete the relocation before a system failure happens and recovery will restore the page to the pre-relocation state. The latter will wipe out the second transaction's committed change. This would be unacceptable. Even though the R_Bit relates to recovery (see [Moha92]), we had to introduce it here since it has some implications on the latching logic, as we will discuss below. The R_Bit is similar to the SM_Bit used for dealing with structure modifications (like page split and page delete) which affect multiple pages in B^+-trees [MoLe92].

```
PROCEDURE Find_Current_Page_No_Reloc (Key,Op,Pg)
Lock(Key) in S or X mode depending on operation
                 /* S for R; X for D, I and U */
S_latch(ST)      /* ensure no FC/FE in progress */
pg0 := home_addr(Key) /* note home addr of rec */
RESTART: pg := pg0 /*start with rec's home addr*/
probe# := 1     /* first separator to be probed */
found := false /* not found rec's current page */
IS_latch(STE(pg))/* ensure nobody is changing
                 separator value to be probed  */
WHILE ¬found DO
| IF sign(Key, probe#) < STE(pg).separator THEN
| | found := true /* found rec's current page  */
| ELSE /* must continue search for current page*/
| | pg := pg+1
| | probe# := probe#+1
| | IS_latch(STE(pg))     /* latch next STE before
| | unlatch(STE(pg-1))     unlatching previous so
| |      that nobody that is actually relocating
| |       overtakes and misleads current tran */
END /* of WHILE finding current page of record */
IF op = 'R' THEN          /* read-only operation */
| unlatch ST     /* can let FC & FE occur since we
|                            have desired page */
| locate and S_latch(pg)  /* do I/O, if needed */
ELSE              /* some change op - D, I or U  */
| IF STE(pg).R_Bit = '1' THEN /* reloc is on.
| |                 can't perform page updates  */
| | unlatch(STE(pg))/* release IS latch before
| |   requesting in S mode to avoid a deadlock */
| | S_latch(STE(pg))/* let record relocation be
| |   over; relocator will be having SIX latch */
| | IF pg <> pg0 THEN  /* not the home page.
| | | since we did not hold any STE latch for a
| | | while we have to restart from home page.
| | | To avoid starvation, we can restart op as
| | |              a DR, IR or UR, if necessary */
| | | unlatch(STE(pg))
| | | GOTO RESTART
| IF op = 'D' & STE(pg).separator < MaxSig+1 THEN
| | unlatch(STE(pg)) /* may cause record reloc */
| | unlatch(ST)          /* release all latches */
| | RETURN(perform DR)     /* retry call as DR */
| unlatch ST          /* can let FE & FC occur */
| locate and X_latch(pg)
unlatch(STE(pg))
RETURN(success) /* S/X latch held on curr page */
END Find_Current_Page_No_Reloc
```

Figure 3: Procedure for Locating Current Page When No Record Relocation is Expected

Note that ARIES/LHS permits the *reading* of even data pages which are participants in still-incomplete relocations, thereby improving concurrency. Even FC and FE must get the latches on STE and data pages, since some concurrent operations by other transactions are being permitted while FC or FE is being executed by one transaction. As mentioned before in the section "1.1. Linear Hashing with Separators", this latching for FC (respectively, FE) should be similar to that for IR (DR).

```
PROCEDURE Access_Current_Page (Key,Op,Pg,Rec)
/* Appropriate latch must be held on current
   page (pg) before this procedure is called */
search pg for record with Key
IF record not found and op <> 'I' THEN
| unlatch(pg)
| RETURN(¬found)
IF record found & op = 'I' THEN
| unlatch(pg)
| RETURN(unique_key_violation)
IF op = 'R' THEN
| Rec := found record
| unlatch(pg)
| RETURN(success)
Attempt insert/delete/update operation
IF op = 'I'/'U' & need record relocation THEN
| unlatch(pg)  /* need for DR checked before */
| RETURN(perform IR/UR)  /* no latches held  */
ELSE        /* no need for record relocation */
| perform op and log it
| unlatch(pg)
| RETURN(success)
END Access_Current_Page
```

Figure 4: Procedure for Performing Desired Operation on Current Page Without Record Relocation

Figure 2 presents the hierarchy imposed on the different latches and the compatibility relationship amongst the different modes of latching. It also summarizes the latching logic which we discuss in detail next.

On **data pages**, the S latch is used if the data is only being read; otherwise, the X latch is used.

The **ST latch** is acquired in the ***S mode*** by DNR, DR, INR, IR, R, UNR and UR, and in the ***X mode*** by FE and FC. The intent here is to enable the former set of operations to perform computations to determine *home* pages and to search the ST for *current* pages of records of interest, without any file expansion or contraction (FE or FC) being in progress. FE and FC could change the results of those computations and searches. They modify state variables (see [Lars88] for details) which are used to compute home addresses. Once the *current* page of a record has been determined and the operation is DNR, INR, R or UNR, then the ST latch can be released even before the I/O is done to bring into virtual memory the corresponding data page. So, an FE or FC can go on in parallel with page accesses. This is possible, since we latch the *STE* of the current page *before* we unlatch the *ST*, thereby ensuring that no records in the current page would be relocated. IR, DR and UR hold on to the S latch on ST until they end their execution since they keep computing home pages for different records during relocations and while looking for candidates for relocations. Since S is compatible with S, ARIES/LHS permits multiple record relocating operations by different transactions to go on in parallel (later, it will be seen that, for recovery reasons, the parallel relocations can happen as long as the sets of pages involved in the different relocating operations are disjoint).

```
PROCEDURE Find_Current_Page_With_Reloc(Key,Op,Pg)
S_latch(ST)  /* ensure no FE or FC in progress */
pg0 := home_addr(Key) /* note home addr of rec */
pg := pg0        /* start with home addr of rec */
probe# := 1    /* first separator to be probed */
found := false  /* not found current pg of rec */
SIX_latch(STE(pg)) /* ensure no ongoing reloc;
                      relocators have X or SIX */
WHILE ¬found DO      /* find current pg of rec */
| IF sign(Key, probe#) < STE(pg).separator THEN
| | found := true   /* found current pg of rec */
| | Upgrade SIX latch on STE(pg) to X latch
| |  /* ensure nobody is looking at separator  */
| |  /* can't cause a deadlock since a converter
| |     jumps wait queue, there can be only one
| |     converter on a page, no page latch held
| |     while converting, IS holders release
| |     latches "soon", and IS holders give up
| |     IS before requesting S latch          */
| | locate and X_latch(pg)
| ELSE    /* continue to search for current pg */
| | pg := pg+1
| | probe# := probe#+1
| | SIX_latch(STE(pg)) /* latch next STE before
| |   unlatching previous STE so that no
| |   relocator overtakes me and leads me to the
| |   wrong page                              */
| | unlatch(STE(pg-1))
END      /* of WHILE finding current pg of rec */
/* now the following latches must be held:
    X on current pg, X on STE(pg) and S on ST */
RETURN(success)
END Find_Current_Page_With_Reloc
```

Figure 5: Procedure for Locating Current Page When Record Relocations are Expected

The modes in which the **STE latches** are acquired during the different operations are described next.

The ***IS mode*** is used by DNR, INR, R and UNR during the scanning of the ST from the *home* page's STE until the *current* page's STE is reached. That is, when the separator value of a page is about to be read, its corresponding STE is latched in the IS mode. Once the next page's STE is latched, the previous page's STE's latch is released (see Figure 3). This is similar to the *lock-coupling* protocol used in [BaSc77] and the *latch-coupling* protocol used in [Moha90, MoLe92] in the context of B^+-trees. If the latch-coupling protocol were not followed in ARIES/LHS, then, after the current STE's latch is released but before the next STE's latch is acquired, a relocation operation may be performed by another transaction that invalidates the inference that was made which said that the current page for the record of interest is further down in the ST. Note that at the time the next STE latch is acquired by a first transaction, a relocation operation by another transaction may still be in progress (as indicated by the R_Bit of the STE being '1') and hence the separator value that is present now may change after a while. Under these conditions, it is acceptable for the first transaction to proceed forward as long as (1) it is performing an R operation or (2) it is performing a modification (I, D or U) operation and the STE being examined is **not** the one for the *current* page of the record of interest. The careful choice of relocation propagation order (top to bottom), and the latching (or upgrading) and unlatching (or downgrading) order during the execution of the operations causing relocations allow this without erroneous results being produced. In the case of the *current* page, the modification operations will have to wait for the relocation activity to complete before they can proceed since they make changes to the data page. Prematurely letting those changes happen might result in records being inserted on the wrong page or in committed changes being undone, if, later, the incomplete relocation activity needs to be undone due to a process or system failure [Moha92]. Note that if the STE were not the one for the current page, then letting the operations proceed forward may permit a nonrelocating operation to overtake a relocating operation. This is very desirable since the relocating operation (especially if it is in the first pass identifying candidate records for relocation) may have to wait for the I/Os of a set of data pages and hence we wouldn't want it to block other operations from making progress.

The ***SIX mode*** is used by DR, IR and UR during the scanning of the ST from the *home* page's STE until the *current* page's STE is reached (see Figure 5). Since SIX is compatible only with IS, getting the STE latch in the SIX mode ensures that no relocation by another transaction that has already affected that STE is still in progress (i.e., there is no need to check the R_Bit). This is guaranteed because the SIX latch is finally released for an STE that is involved in a relocation only after the relocation activity is completed across all the affected pages. Once the next page's STE is latched in the SIX mode, the previous page's STE's latch is released (latch-coupling). Once the *current* page is determined, the SIX latch on the current page's STE is upgraded to an X latch since the R_Bit must be set to '1' when no other transaction is examining the STE. Basically, we need to drive out any updating transaction for which this STE's file page may be the current page since we may modify and/or extract record information from that file page. This attempt to upgrade cannot result in a deadlock since no other transaction could also be attempting to do the same (since SIX is incompatible with SIX - this is the reason for making the relocators get SIX, instead of IS as for the nonrelocators, while coming down from the *home* page). Further, any holders of the IS latch on the STE cannot get involved in a deadlock with the upgrader since the latter is not holding any *other* latches which the former would request in a conflicting mode (see below for discussion about a deadlock involving the *same* STE latch). Later, the X is downgraded to SIX. This SIX latch continues to be held on the STEs of all the data pages (current page and one or more following pages) which are not currently being modified but on which some relocation-related changes might have already been made (see Figure 7) or from which information about some candidate records for relocation might have already been extracted (see Figure 6). In addition to the R_Bit on the STE being set to '1' to indicate the existence of an ongoing relocation activity involving this page, the STE latch is also held in the SIX mode. The latter is done so that any other transaction which notices that the R_Bit value is '1' and hence it needs to wait for the relocation activity to terminate can do so by re-

```
PROCEDURE Do_UR_With_Rec_Contract_OR_DR (Key,Op,Pg,Rec)
/* X latches on current page (pg) and STE(pg), and S latch on ST must be held before this call    */
Search page for record with Key
IF record not found THEN
  unlatch(pg)
  unlatch(STE(pg))
  unlatch(ST)
  RETURN(¬found)
delete/update record & log it        /* relocation must be performed after operation is performed */
Note LSN of tran's latest log record /* LSN saved for dummy CLR of relocation's nested top action */
firstpage := pg                      /* remember lowest numbered page to be changed by relocation */
     /* identify records not in their home pages by accessing all pages from (current page + 1)  */
WHILE STE(pg).separator <> MaxSig+1 DO /* to the nearest following page with separator = MaxSig+1 */
| unlatch(pg)
| STE(pg).R_Bit := '1' /* indicate relocation in progress to prevent updates until identification */
|                      /* of candidate records for relocation and actual relocations are over     */
| Downgrade X latch to SIX on STE(pg)      /* this allows reads of pg but no updates. readers and */
|                      /* nonrelocating updaters can overtake current tran now; it is OK since we */
|              /* are not changing separators now but are only identifying records for relocation */
| pg := pg+1
| locate(pg)           /* do I/O without holding pg's STE & page latches to allow more concurrency*/
| X_latch STE(pg)
| S_latch(pg)
| identify records on pg for which pg is not home page       /* no modifications are made to page */
END                                                /* of loop identifying candidates for relocation */
lastpage := pg
unlatch(pg)
STE(pg).R_Bit := '1'
Downgrade X latch to SIX on STE(pg)            /* this allows reads of pg but no updates          */
/* identification of candidate records for relocation is over; time to perform relocations        */
Upgrade SIX on STE(firstpage) to X        /* can't cause a deadlock since a converter jumps wait */
       /* queue, there can be only one converter on a page, no page latch held while converting, */
      /* IS holders release latches "soon", and IS holders give up IS before requesting S        */
FOR pg := firstpage TO lastpage DO                                 /* now perform the relocations */
| X_latch(pg)
| insert as many remaining identified records as possible for which this page is a home page or is
|   a page following their home pages; delete records already moved to earlier pages; log changes
| update STE(pg).separator and log it
| unlatch(pg)
| IF pg <> lastpage THEN     /* get X latch on next STE before downgrading current STE's latch to */
| | upgrade SIX latch on STE(pg+1) to X latch   /* prevent overtaking by readers or nonrelocating */
| |                     /* updaters. this is necessary since we have already updated the separator */
| downgrade X latch on STE(pg) to SIX latch          /* this allows reads of pg but no updates  */
END
write dummy CLR and set its UndoNxtLSN to LSN noted above
      /* this ensures that even if transaction rolls back later, relocations will not be undone  */
FOR pg := firstpage to lastpage DO
| STE(pg).R_Bit := '0'                          /* indicate relocation is over                    */
| unlatch(STE(pg))                              /* give up SIX latch on STE(pg)                   */
END
unlatch(ST)                                     /* permit file expansion & contraction to occur */
RETURN(success)
END Do_UR_With_Rec_Contract_OR_DR
```

Figure 6: Procedure for Deleting/Updating Record on Current Page with Possibly Upward Record Relocations

questing the STE latch in the S mode and getting blocked until the SIX mode latch being held by the relocating action of another transaction is released at the end of that action. This is similar to the way the *tree* latch is used in ARIES/IM to synchronize the actions of a structure modification transaction with other transactions which are waiting for the structure modification to be over [MoLe92].

The ***S mode*** is used by DNR, INR and UNR when they need to wait until an in-progress relocation operation (as indicated by the R_Bit for the STE under examination being

```
PROCEDURE Do_UR_With_Rec_Expand_OR_IR (Key,Op,Pg,Rec)
/* X latches on current page (pg) and STE(pg), and S latch on ST must be held before this call    */
Note LSN of tran's latest log record /* LSN saved for dummy CLR of relocation's nested top action */
delete records to be relocated and possibly make place for record to be inserted/updated; log changes
   /* don't insert new record or update old record still, even if this page will be current page */
   /* for record ultimately; those actions are to be performed only after all relocations are    */
   /* done; do move old version of record to that page where the updated version will exist      */
update STE(pg).separator and log it
STE(pg).R_Bit := '1'                                        /* indicate relocation is in progress */
firstpage := pg                    /* remember lowest numbered page being changed by relocation */
WHILE records remain for relocation DO
| unlatch(pg)                                                  /* page is now available to others */
| X_latch(STE(pg+1))    /* X_latch next STE before downgrading latch on current page's ST entry    */
|      /* this is so that no reader or nonrelocating modifier will overtake current tran. this     */
|      /* is necessary since we have already updated separator but relocation is not yet over      */
| Downgrade X latch to SIX on STE(pg)/* this allows reads of pg but no updates (due to R_Bit='1')*/
| pg := pg+1
| locate and X_latch(pg)
| insert as many overflowed records as possible and delete records to be overflowed
| log changes to page
| STE(pg).R_Bit := '1'                                      /* indicate relocation is in progress */
| update STE(pg).separator and log it
END                                               /* of while loop relocating overflowed records */
lastpage := pg                   /* remember highest numbered page to be affected by relocation */
write dummy CLR and set its UndoNxtLSN to LSN noted above
       /* this ensures that even if transaction rolls back later, relocations will not be undone */
unlatch(pg)
Downgrade X latch to SIX on STE(pg)                  /* this allows reads of pg but no updates   */
IF op = 'I' THEN
| insert new record on the correct page after X latching that page
| log action and unlatch page
ELSE                                                       /* overflow causing update operation */
| update record after X latching its current page (may not be original page)
| log action and unlatch page
FOR pg := firstpage to lastpage DO
| STE(pg).R_Bit := '0'                                 /* indicate relocation not in progress  */
| unlatch(STE(pg))                                             /* give up SIX latch on ST entry */
END
unlatch(ST)                                        /* let file expansion or contraction happen */
RETURN(success)
END Do_UR_With_Rec_Expand_OR_IR
```

Figure 7: Procedure for Inserting/Updating Record on Current Page with Possibly Downward Record Relocations

'1') terminates. In order to avoid a deadlock with the holder of the SIX latch on the STE who may be trying to upgrade it to an X latch, the S latch request is made after giving up the already-held IS latch on the STE (see Figure 3). Since no latch is held on the STE for a period of time before the S latch is obtained, once the S latch is granted, a rescanning of the ST must be initiated from the *home* page if the current page is not the same as the home page. This is necessary because what was initially thought to be the *current* page may no longer be the current page due to some other transactions' activities during the time when no latch was held on the STE by the current transaction. If it is felt that this may lead to starvation, we can restart the operation as a relocating one (DR, IR or UR).

The ***X mode*** is used by DR, IR and UR when they are modifying the separator value or the R_Bit ('0' to '1' transition only) of an STE, or when they are deleting or inserting records on data pages, as part of relocation activities. Since IR (and UR when the updated record expands requiring relocations) does its relocations in a single top-down pass (see Figure 7), the X latching of the STE (starting from the STE of the *current* page of the record being inserted by the user) is followed by the X latching of the corresponding data page. Once the latter is manipulated appropriately, it is unlatched and the next STE is latched. After that the X latch on the previous STE is downgraded to SIX. The idea behind the *X-latch-coupling* on the STE is to prevent any other transaction from overtaking the relocating transaction since such an overtaking may result in a record that is in the process of being relocated *down* not being located by the overtaker. The downgrading from X to SIX is done so that *readers* may access those pages on which relocation has been completed. In the case of DR (and UR when the updated record contracts thereby

allowing relocations), since a two-pass relocation strategy needs to be used, during the first pass, which does not perform any changes to data pages or the separators, it is not incorrect to allow overtaking. This is the reason the X latch on an STE involved in a relocation is downgraded to SIX *before* the next STE is latched in the X mode (see Figure 6). During the second pass, we do not allow any overtaking.

Latch-coupling is also used between the STE latch and the corresponding data page latch, for similar reasons as before. That is, the *current* page's STE's latch (and, in the case of relocators, for the following pages also) is released only *after* the corresponding file page's latch is obtained. This guarantees consistency between the STE's separator information and the actual data content of the corresponding file page.

3. Conclusions

In this paper, first we briefly summarized Larson's *Linear Hashing with Separators* hash-based storage scheme. Then, we proceeded to deal with the problems introduced by supporting the hashing method in a transaction context with the associated efficient recovery and high-concurrency requirements. We exploited the flexibility and the efficiency characteristics of the ARIES recovery method to design the ARIES/LHS recovery and concurrency control method for supporting Larson's storage scheme. Due to space constraints, the recovery aspects of ARIES/LHS were not discussed in detail here (see [Moha92]). The flexibility and efficiency attributes of ARIES (e.g., with respect to buffer management and media recovery) carry over to ARIES/LHS also. The latching protocols of ARIES/LHS support very high concurrency. Furthermore, they do not cause deadlocks. The concurrency control aspects of ARIES/LHS can be used with other recovery methods also (e.g., with shadow paging). Since ARIES has been extended to support nested transactions [RoMo89], ARIES/LHS also can support nested transactions, thereby permitting parallelism within transactions. Most of the ideas of ARIES/LHS are applicable to other hashing methods that have been proposed in the literature. As far as we know, ARIES/LHS is the first published method to comprehensively deal with the problem of providing recovery from transaction and system failures for a general model of transactions with fine-granularity locking and logical undo for a hashing-based storage method using write-ahead logging. Discussions concerning range queries and prefetching of data may be found in [Moha92].

4. References

BaSc77 Bayer, R., Schkolnick, M. *Concurrency of Operations on B-Trees*, **Acta Informatica**, Vol. 9, No. 1, p1-21, 1977.

Elli83 Ellis, C. *Extendible Hashing for Concurrent Operations and Distributed Data*, **Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems**, Atlanta, March 1983.

Elli87 Ellis, C. *Concurrency in Linear Hashing*, **ACM Transactions on Database Systems**, Vol. 12, No. 2, p195-217, June 1987.

EnDu88 Enbody, R.J., Du, H.C. *Dynamic Hashing Schemes*, **ACM Computing Surveys**, Vol. 20, No. 2, p85-113, June 1988.

Gray78 Gray, J. *Notes on Data Base Operating Systems*, In **Operating Systems - An Advanced Course**, R. Bayer, R. Graham and G. Seegmuller (Eds.), Lecture Notes in Computer Science, Vol 60, Springer-Verlag, 1978.

HsTY90 Hsu, M., Tung, S.-S., Yang, W.-P. *Concurrent Operations in Linear Hashing*, **Information Sciences**, Vol. 51, No. 2, July 1990.

HsYa86 Hsu, M., Yang, W.-P. *Concurrent Operations in Extendible Hashing*, **Proc. 12th International Conference on Very Large Data Bases**, Kyoto, August 1986.

KeRu88 Kelley, K., Rusinkiewicz, M. *Multikey, Extensible Hashing for Relational Databases*, **IEEE Software**, July 1988.

Kuma89 Kumar, V. *Concurrency Control on Extendible Hashing*, **Information Processing Letters**, Vol. 31, April 1989.

Lars88 Larson, P.-A. *Linear Hashing with Separators - A Dynamic Hashing Scheme Achieving One-Access Retrieval*, **ACM Transactions on Database Systems**, Vol. 13, No. 3, September 1988.

MHLPS92 Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P. *ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging*, **ACM Transactions on Database Systems**, Vol. 17, No. 1, March 1992.

Moha90 Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990. An expanded version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.

Moha92 Mohan, C. *ARIES/LHS: A Concurrency Control and Recovery Method Using Write-Ahead Logging for Linear Hashing with Separators*, **IBM Research Report RJ8682**, IBM Almaden Research Center, March 1992.

MoLe92 Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **Proc. ACM SIGMOD International Conference on Management of Data**, San Diego, June 1992. A longer version of this paper is available as **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.

MoLO86 Mohan, C., Lindsay, B., Obermarck, R. *Transaction Management in the R* Distributed Data Base Management System*, **ACM Transactions on Database Systems**, Vol. 11, No. 4, December 1986.

MoPi91 Mohan, C., Pirahesh, H. *ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method*, **Proc. 7th International Conference on Data Engineering**, Kobe, April 1991.

OuAb89 Ouksel, M., Abdul-Ghaffar, J. *Concurrency in Multidimensional Linear Hashing*, **Proc. 3rd International Conference on Foundations of Data Organization and Algorithms**, Paris, June 1989.

RoMo89 Rothermel, K., Mohan, C. *ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions*, **Proc. 15th International Conference on Very Large Data Bases**, Amsterdam, August 1989. A longer version of this paper is available as **IBM Research Report RJ6650**, IBM Almaden Research Center, January 1989.

ShGo88 Shasha, D., Goodman, N. *Concurrent Search Structure Algorithms*, **ACM Transactions on Database Systems**, Vol. 13, No. 1, March 1988.