ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging

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

FRANK LEVINE
IBM, 11400 Burnet Road, Austin, TX 78758, USA

Abstract This paper provides a comprehensive treatment of index management in transaction systems. We present a method, called ***ARIES/IM*** (***Algorithm for Recovery and Isolation Exploiting Semantics for Index Management***), for concurrency control and recovery of B^+-trees. ARIES/IM guarantees serializability and uses write-ahead logging for recovery. It supports very high concurrency and good performance by (1) treating as the lock of a key the same lock as the one on the corresponding record data in a data page (e.g., at the record level), (2) not acquiring, in the interest of permitting very high concurrency, commit duration locks on index pages even during index structure modification operations (SMOs) like page splits and page deletions, and (3) allowing retrievals, inserts, and deletes to go on concurrently with SMOs. During restart recovery, any necessary *redos* of index changes are always performed in a page-oriented fashion (i.e., without traversing the index tree) and, during normal processing and restart recovery, whenever possible *undos* are performed in a page-oriented fashion. ARIES/IM permits different granularities of locking to be supported in a flexible manner. A subset of ARIES/IM has been implemented in the OS/2 Extended Edition Database Manager.[1] Since the locking ideas of ARIES/IM have general applicability, some of them have also been implemented in SQL/DS and the VM Shared File System, even though those systems use the shadow-page technique for recovery.

1. Introduction

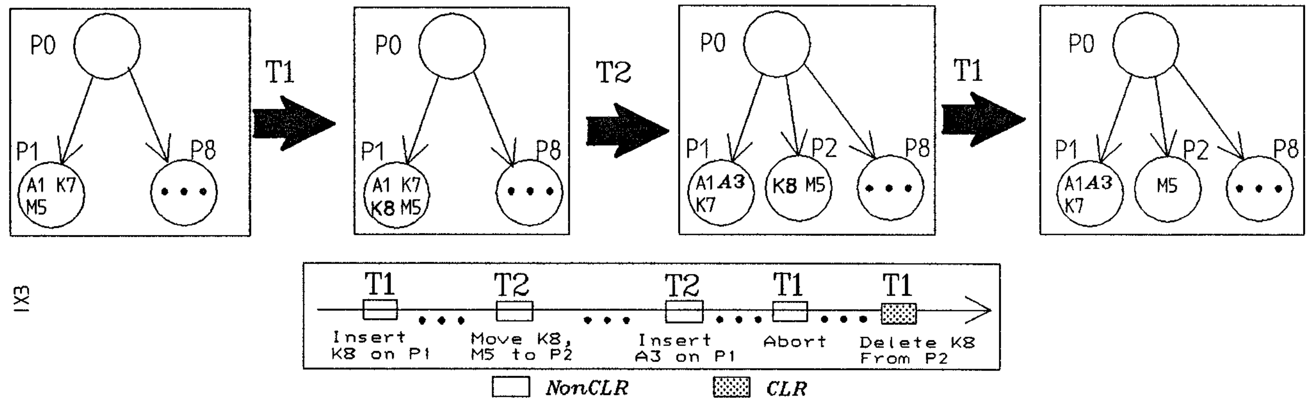
Protocols for controlling concurrent access to B-trees and their variants have been studied for a long time (see [BaSc77, LeYa81, Mino84, Sagi86, ShGo88] and references in them). None of those papers considered the problem of guaranteeing atomicity and serializability of transactions containing multiple operations (like Fetch, Insert, Delete, etc.) on B^+-trees, in the face of transaction, system and media failures, and concurrent accesses by different transactions. [FuKa89] presents an incorrect (e.g., insufficient locking in the *not found* case and while locking for range scans) and expensive (using nested transactions) solution to the problem (see [MoLe89] for details). The index managers of data base management systems (DBMSs) like DB2[1], the OS/2 Extended Edition Database Manager[1], System R, NonStop SQL[1] and SQL/DS support serializability (***repeatable read (RR)*** or ***degree 3 consistency*** [Gray78]). For recovery, DB2, NonStop SQL and the OS/2 Extended Edition Database Manager use write-ahead logging (WAL) [Gray78, MHLPS92], while System R and SQL/DS use the shadow-page technique [GMBLL81]. Unfortunately, the details of the algorithms used in most of the above systems have never been published. In this paper, we present a concurrency control and recovery method, called ***ARIES/IM*** (***Algorithm for Recovery and Isolation Exploiting Semantics for Index Management***), for B^+-tree index data. We developed ARIES/IM as part of designing the OS/2 Extended Edition Database Manager product.

For the first time, most of the details of the System R approach to index locking were described by us in [Moha90a], as part of our ARIES/KVL work which improved that approach's concurrency and locking overhead characteristics. In spite of the fine-granularity locking provided via record locking for data and *key value* locking for the index information, the level of concurrency supported by System R, which originated the IBM product SQL/DS, has been found to be inadequate by customers. The concurrency enhancements provided in ARIES/KVL are still inadequate since even in ARIES/KVL locks are acquired on key values, rather than on individual keys. The latter makes a significant difference in the case of nonunique indexes. Furthermore, the number of locks acquired for even single record operations like record insert or delete is very high in System R. While designing ARIES/IM, our primary goals were to modify the System R algorithm to use WAL and to drastically improve its concurrency, performance, and functionality characteristics. Serializable executions had to be supported with efficient recovery and storage management, and high concurrency. ARIES/IM satisfies these requirements.

ARIES/IM is based on the ARIES recovery and concurrency control 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[1] product suite, and in the University of Wisconsin's Gamma data base machine and EXODUS extensible DBMS. In ARIES/IM, minimal number of locks are acquired while providing a high level of concurrency. Restart and normal performance are improved by implementing undos and redos efficiently and by avoiding deadlocks during undos. Our measure of concurrency is the qualitative one defined in [KuPa79] which basically states that the more the number of permitted different interleavings of the actions of a set of transactions the higher the level of concurrency. Our measures of efficiency are the number of locks acquired, the number of pages accessed during redo, undo, and normal operations, the number of passes of the log made during media recovery, and the number of required synchronous data base page and log I/Os.

The rest of the paper is organized as follows. In the rest of this section, we first introduce the tree architecture and list some of the problems involved in index concurrency control

1992 ACM SIGMOD - 6/92/CA, USA
© 1992 ACM 0-89791-522-4/92/0005/0371...$1.50



This scenario shows why the page (P2) affected during the rollback of an action (key insert) may be different from the one (P1) affected during forward processing. Such a ***logical undo*** may require retraversing the tree from the root to locate the key (K8). This is caused by an intervening page split (by T2) which moves the originally inserted key to a different page. Writing compensation log records (CLRs) during rollback actions allows the change to the different page to be logged.

Figure 1: Logical Undo Scenario

and recovery. We then introduce the ARIES recovery and concurrency control method since ARIES/IM's recovery is based on ARIES. Section 2 then presents the concurrency control features of ARIES/IM, while section 3 discusses the recovery aspects. In section 4, we explain how deadlocks involving latches are avoided and why rolling back transactions never get involved in deadlocks. We conclude, in section 5, with discussions about implementations of ARIES/IM.

1.1. Tree Architecture and Problems

A ***key*** in a leaf page is a ***key-value,record-ID*** pair, where ***record-ID (RID)*** is the identifier of the record containing that key value. The records themselves are stored elsewhere in ***data pages*** (i.e., outside of the index tree). The leaf pages alone are forward and backward chained. Every nonleaf page contains a certain number of child page pointers and *one less* number of high keys - each high key is associated with one child page pointer and there is no high key associated with the rightmost child. A high key stored in the nonleaf page for a given child page is always **greater than** the highest key that is actually stored in that child page.

There are four basic index operations that ARIES/IM supports:

1. ***Fetch***: Given a key value or a partial key value (its prefix), check if it is in the index and fetch the full key. A ***starting condition*** (=, >, *or* >=) will also be given with the input value.
2. ***Fetch Next***: Having opened a range scan with a Fetch call, fetch the next key satisfying the *key-range specification* (e.g., a **stopping key** and a comparison operator (<, =, *or* <=)).
3. ***Insert***: Insert the given key (*key-value,RID*). For a *unique* index, the *search* logic is called to look for only the key value since duplicates must be avoided. For a nonunique index, the whole new key is provided as input for search.
4. ***Delete***: Delete the given key.

There are many problems involved in supporting recoverable, concurrent modifications to an index tree. Some of the questions to be answered are: (1) how to log the changes to the index so that, during recovery after a system failure, any missing updates can be reapplied efficiently? (2) how to ensure that, if an ***SMO (structure modification operation*** - i.e., a page split or a page deletion operation) were to be in progress at the time of a system failure and some of the effects of that SMO had already been reflected in the disk version of the data base, then the system is able to restore the structural consistency of the tree at the time of system restart (see Figure 11 for a failure scenario which causes structural inconsistencies)? (3) how to perform the changes to index pages so as to minimize the interference caused to concurrent accessors of the tree? (4) how to ensure that, even if a transaction were to rollback after successfully *completing* an SMO, it does not undo the SMO, since doing so might result in the loss of some updates performed by other transactions in the intervening period to the pages affected by the SMO? (5) how to detect that a key that had been inserted by a transaction T1 in page P1 had been moved, due to a subsequent SMO by T2, to P2 so that if T1 were to rollback, then P2 is accessed and the key is deleted (see Figure 1 for an example of such a ***logical undo***)? (6) how to detect that a key that had been deleted by T1 from P1 no longer belongs on P1 but only on P2 due to subsequent SMOs by other transactions, so that if T1 were to rollback, then P2 is accessed and the key is inserted in it? (7) how to avoid a deadlock involving a transaction that is rolling back so that no special logic is needed to handle a deadlock involving only rolling back transactions? (8) how to support different granularities of locking and what to designate as the objects of locking? (9) how to lock the "not found" condition efficiently to guarantee RR (i.e., handling the *phantom problem*)? (10) how to guarantee that in a *unique* index if a key value were to be deleted by one transaction, then no other transaction is permitted to insert the *same* key value before the former transaction commits? (11) how to let tree traversals go on even as SMOs are in progress and still

[1] DB2, IBM and OS/2 are trademarks of International Business Machines Corp. NonStop SQL and Tandem are trademarks of Tandem Computers, Inc. Transarc is a registered trademark of Transarc Corp. Encina is a trademark of Transarc Corp.

ensure that the traversing transactions are able to recover if they run into the effects of the SMOs that are still in progress (see Figure 3 for an illustration of a problem scenario)?

1.2. ARIES

In this subsection, we briefly describe the ARIES recovery method. The reader is referred to [MHLPS92] for the details about ARIES, to [MoPi91] for the presentation of some optimizations to the original ARIES method, to [MoNa91] for the descriptions of enhancements to ARIES to handle the shared disks environment, and to [RoMo89] for the description of ARIES/NT, which is the extension of ARIES to the nested transactions model. We assume that the reader is familiar with the concept of a *latch*, the different degrees of consistency (*repeatable read*, *cursor stability*), the different durations (*instant*, *commit*) and modes (*S, X, IS, IX, SIX*) of locking and latching, and the differences between locks and latches, as described in [Gray78, MHLPS92, Moha90a, Moha90b].

In ARIES, every data base page has a ***page_LSN*** field which contains the ***log sequence number (LSN)*** of the log record that describes the most recent update to the page. Since LSNs monotonically increase over time, by comparing at recovery time a page_LSN with the LSN of a log record for that page, we can unambiguously determine whether that version of the page contains that log record's update. That is, if the page_LSN is *less than* the log record's LSN, then the effect of the latter is *not* present in the page. ARIES uses latches on pages to assure physical consistency of the accessed information, while it uses locks on data to assure logical consistency. ARIES supports fine-granularity (e.g., record) locking with semantically-rich lock modes (e.g., increment/decrement-type locks), partial rollbacks, nested transactions, write-ahead logging, selective and deferred restart, fuzzy image copies (archive dumps), media recovery, and the *steal* and *no-force* buffer management policies.

In ARIES, restart recovery after a system failure consists of three passes of the log: *analysis*, *redo* and *undo*. First the log is scanned, starting from the log record of the last complete checkpoint, up to the end of the log. This ***analysis pass*** determines the starting point for the log scan of the next pass. It also provides the list of in-flight and in-doubt transactions. In the ***redo pass***, ARIES ***repeats history*** by redoing those updates logged on stable storage but whose effects on the data base pages did not get reflected on disk before the crash. This is done for the updates of *all* transactions, *including the updates of in-flight transactions*. The redo pass also reacquires the locks needed to protect the uncommitted updates of the *in-doubt* transactions.

The next pass is the ***undo pass*** during which all in-flight transactions' updates are rolled back, in reverse chronological order, in a single sweep of the log. In addition to logging updates performed during forward processing of transactions, ARIES also logs, typically using compensation log records (***CLRs***), updates performed during partial or total rollbacks of transactions. CLRs have the property that they are redo-only log records. By appropriate *chaining* of the CLRs to log records written during forward processing, a bounded amount of logging is ensured during rollbacks, even in the face of repeated failures during restart recovery or of nested rollbacks. When the undo of a log record (***nonCLR***) causes a CLR to be written, the CLR is made to point, via the ***UndoNxtLSN*** field of the CLR, to the *predecessor* (i.e., setting it equal to the ***PrevLSN*** value) of the log record being undone.

There are times when we would like some changes of a transaction to be committed irrespective of whether later on the transaction as a whole commits or not. We do need the atomicity property for these changes themselves. A few of the many situations where this is very useful are: for performing page splits and page deletes in indexes as we show later in this paper, and for relocating records in a hash-based storage method [Moha92]. ARIES supports this via the concept of ***nested top actions***. The desired effect is accomplished by writing a ***dummy CLR*** at the end of the nested top action (see Figure 9). The dummy CLR has as its UndoNxtLSN the LSN of the most recent log record written by the current transaction just before it started the nested top action. Thus, the dummy CLR lets ARIES bypass the log records of the nested top action if the transaction were to be rolled back *after* the completion of the nested top action. ARIES's repeating history feature ensures that the nested top action's changes would be redone, if necessary, after a system failure even though they may be changes performed by an in-flight transaction. If a system failure were to occur *before* the dummy CLR is written to stable storage, then the incomplete nested top action will be undone since the nested top action's log records are written as undo-redo (as opposed to redo-only) log records. This provides the desired atomicity property for the nested top action itself.

2. Concurrency Control in ARIES/IM

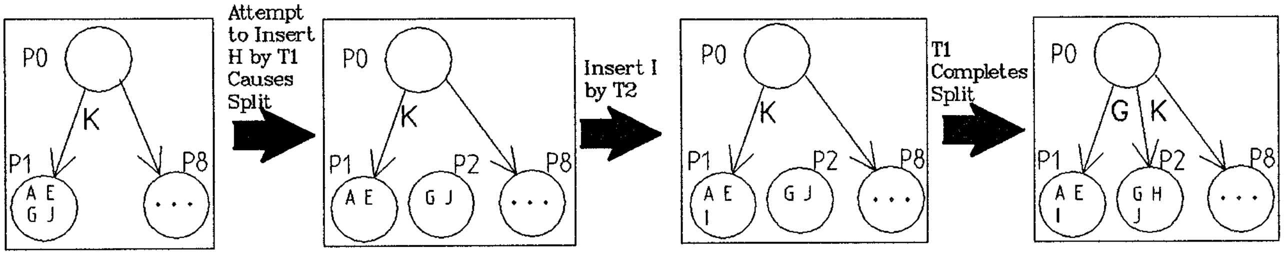
In this section, we present those features of ARIES/IM which are intended for concurrency control purposes. We first give an overview of those features in a general way and then present the specifics for the different basic index operations. Those features of ARIES/IM which are intended to allow recovery to be performed correctly are presented in the section "3. Recovery in ARIES/IM". The recovery requirements will be shown to place further restrictions on the allowed concurrency during certain operations. More details are presented in [MoLe89].

2.1. Overview of Locking and Latching

Locking The table in Figure 2, summarizes the locks acquired during different operations. ARIES/IM supports very high concurrency and good performance by (1) treating as the lock of a key the same lock as the one on the corresponding record data in a data page (at the locking granu-

	NEXT KEY	CURRENT KEY
FETCH & FETCH NEXT		S for commit duration
INSERT	X for instant duration	X for commit duration if index-specific locking is used
DELETE	X for commit duration	X for instant duration if index-specific locking is used

Figure 2: Summary of Locking in ARIES/IM



T3 Looks Up for I, Goes to P2 and Declares I Does Not Exist

ERROR!

Figure 3: Undesirable Interaction Between a Structure Modifying Transaction and an Insert Transaction

larity (page, record, ...) associated with the table/file), (2) **not** acquiring, in the interest of permitting very high concurrency, commit duration locks on *index pages* even during SMOs, and (3) allowing key retrievals, inserts, and deletes to go on concurrently with SMOs. To lock a key, ARIES/IM locks the record whose *record ID* is present in the key (or the *data page* ID which is part of the record ID, if the locking granularity is a page). A lock on a key is really a lock on the corresponding piece of data which contains the key. We call this ***data-only locking***. This is to be contrasted with the *key value* locking approaches of System R and ARIES/KVL [Moha90a], and the index (mini)page locking approaches of DB2 and System R (with page locking). We call those other approaches to locking as ***index-specific locking***. Actually, ARIES/IM can be easily modified to perform index-specific locking also for slightly more concurrency compared to data-only locking, but with extra locking costs (see [MoLe89]). With data-only locking, the *current key* is not explicitly locked by the *index* manager during key deletes and inserts, since the *record* manager would have *already* locked the corresponding data with a commit duration X lock during the *data page* operation. The *explicit* locking of the deleted or inserted key by the index manager is needed only if index-specific locking is being done. Since, with data-only locking, during fetch and fetch next calls, the index manager locks the current key, the record manager does *not* have to lock the corresponding record during the *subsequent* record retrieval from the data page. Obviously, with index-specific locking, the record manager would have to do that locking also.

During the insert or delete of a key, a lock is requested on the *next* key currently in the index in order to support RR (thus solving the *phantom problem*) and also to guarantee, in the case of a unique index, that multiple keys with the *same key value* do not show up due to transaction rollbacks (see the sections "2.4. Insert" and "2.5. Delete"). During a fetch or fetch next operation also, in order to guarantee RR, the next key is locked if the requested key value is not present (see the sections "2.2. Fetch" and "2.3. Fetch Next").

Latching Only page *latches* are used to provide *physical* consistency of information, when the tree is being traversed for any kind of operation. These minimize the number of *locks* acquired and improve performance in terms of both pathlength and concurrency. Not more than 2 index pages are held latched simultaneously at anytime. In order to improve concurrency and to avoid deadlocks involving latches, even those latches are not held while waiting for a lock which is not immediately grantable. No *data page* latch is held or acquired during an index access. ***Latch coupling*** is used while traversing the tree - i.e., the latch on a parent page is held while requesting a latch on a child page. The steps executed during a tree traversal are as follows:

1. S latch the root making it the current page.
2. Examine the current page, identify the child page to be latched, and make the child page the current page. If the current page is not a leaf, S latch it, unlatch the parent page, and do step (2) again. If the current page is a leaf, X (respectively, S) latch it if the operation is a key insert/delete (fetch/fetch next).

Later, some checks will be added to the above logic to take care of conditions caused by concurrent activities (SMO, etc.) of other transactions (see Figure 4).

Structure Modification Operations An SMO (page split or delete) is performed by the same *transaction* that encountered a need for it, unlike in some other methods [Sagi86, ShGo88]. When a page is split by a transaction, other transactions are not prevented from reading that page or even modifying that page before the transaction which performed the split *commits*. To improve concurrency, the effects of SMOs are propagated in the tree in a bottom-up manner (i.e., from the leaves to the nonleaves), without the notion of a *safe node (page)* (see [BaSc77]). To avoid deadlocks involving latches, the latches on the lower-level pages are released before the higher-level page(s) is latched and modified. This leaves open the possibility of a traverser seeing an inconsistent tree (see Figure 3). The rationale for this design decision is discussed in [MoLe89]. Splits are done to the "right". That is, the higher valued keys are moved to the *new* page. A page that becomes empty is deleted from the tree. It is ensured that under no circumstances an empty page remains a part of the index with no SMO remaining to be completed (i.e., with the SM_Bit (see below) equal to '0' and the page being reachable from the root of the tree). In order to avoid undesirable interactions between an SMO of one transaction and the actions of the other transactions (see, e.g., Figure 3), ARIES/IM does the following:

SMOs within a single index tree are serialized using an **X tree latch** *that is specific to this index.*

The tree latch is acquired in the X mode by a transaction just before it performs the SMO at the leaf level and is held until ALL the effects of that SMO are propagated up the tree. As the SMO is performed at the leaf level and is propagated up the tree, a bit, call it the ***SM_Bit***, is set to '1' in each page affected by the SMO to warn other transactions. The SM_Bit is used to avoid problems like the one where an insert is performed on the wrong page because of an incomplete SMO. The SM_Bit can be reset to '0' once the SMO which caused it to be set to '1' has been completed.

In order to support high levels of concurrency, the tree latch is **not** acquired during tree traversals. But ARIES/IM uses the tree latch also to synchronize, under certain conditions, the different transactions performing key inserts, key deletes, and SMOs involving a particular tree. The tree latch is acquired by a transaction traversing a tree when a page which has participated in a yet-to-be-completed SMO is encountered (i.e., SM_Bit='1' for the page) and (1) there is ambiguity, when the page is a nonleaf, about whether or not it is correct to traverse further down that subtree[2], or (2) the page is a leaf and it needs to be modified.[3] Under these conditions, the tree latch is requested in the S mode to wait for the SMO to be completed (see Figure 4, Figure 6 and Figure 7). Because ARIES/IM propagates SMOs bottom-up, the high key in a parent page for a particular child page can be trusted only if latches are held on both the parent and the child, and the SM_Bit is equal to '0' on the child. This means that the parent-child path is valid.

Next, we describe the actions taken after we reach a leaf page for the different operations (Fetch, Insert, ...).

2.2. Fetch

The logic for Fetch is shown in Figure 5. If Fetch reaches the *last* (i.e., the *rightmost*) leaf page and no matching or higher key value is found, then it is treated as the *EOF (End Of File)* situation and a special lock name unique to this index is used as the found key's lock name. If the requested key value was not found but a higher valued key was found or it is the EOF case, then the *not found* status will be returned to the caller. If the first leaf examined during the tree traversal does not have a key value equal to or greater than the one searched for, then the next leaf would be latched and accessed while continuing to hold the latch on the first leaf in order to find the next higher key (see [MoLe89] for details). In any case, while holding the page latch(es), an S lock is requested on the found key.\ Getting the lock while holding the latch(es) guarantees that the inferred information (e.g., the fact that the requested key exists or does not exist) is correct (i.e., the inferred state is the committed state, unless of course the inferred state is the uncommitted state of the same transaction).

```
/* for simplicity, root = leaf case not specified here */
S latch root and note root's page_LSN
Child := Root
Parent := NIL
Descend:
IF child is a leaf AND Op is (insert OR delete) THEN
  X latch child
ELSE S latch child
Note child's page_LSN
IF child is a nonleaf page THEN
  IF nonempty child & ((input key<=highest key in child)
      OR ((input key>highest key in child) & SM_Bit='0'))
      THEN                    /* Not an ambiguous case */
    IF parent <> NIL THEN unlatch parent
    Parent := Child
    Child := Page_Search(Child)
        /* Search child to decide next page to access */
    Go to Descend
  ELSE            /* Unfinished SMO causing ambiguity */
    Unlatch parent & child
    S latch tree for instant duration
                /* Wait for unfinished SMO to finish */
    Unwind recursion as far as necessary based on
      noted page_LSNs and go down again
ELSE     /* Child is leaf; S or X latch held on child */
  CASE Op OF
    Fetch:  ...       /* invoke fetch action routine  */
    Insert: ...       /* invoke insert action routine */
    Delete: ...       /* invoke delete action routine */
  END
```

Figure 4: Search Logic During Tree Traversal

```
Find requested or next higher key (maybe on NextPage)
Unlatch parent
Request conditional S lock on found key
IF lock granted THEN
  Unlatch child & return found key
ELSE
  Note LSN and key position and unlatch child
  Request unconditional lock on key
  Once lock granted backup & search if needed
```

Figure 5: Pseudo-Code for Fetch Action Routine

To shorten the paper, in the following, all the lock calls are described as if they would be granted right away (i.e., when requested *conditionally* while holding tree and/or page latch(es)). To avoid deadlocks and to increase concurrency, if the lock is not granted when requested conditionally, then the following steps must be taken: (1) all the latches must be released, (2) the lock must be requested *unconditionally*, and (3) once the lock is granted, a verification must be performed to ensure that a corrective action (e.g., requesting another lock) is taken if a change of interest had occurred when the latches were not held (e.g., the locked key may no longer exist or a smaller key of interest has appeared in the index). Before unlatching the pages, their page_LSNs would be noted to make the detection of no changes a cheap operation. The tree latch also should not be requested *unconditionally* while holding other latches. As for locks, if the tree latch is obtained without holding page latches, then a validation of the previously inferred information must be performed (this may require reexamining the ancestors of the child page).

Even if the requested key is not found, the next higher key which is currently present in the index is locked to make sure that the requested key does not appear (due to an insert by another transaction) before the current transaction *terminates* and prevent RR from being possible. As we will see later, this locking, in conjunction with the next key locking done during key inserts, makes it possible to guarantee RR and hence serializability. This locking in Fetch also makes sure that the requested key has not been deleted by another transaction which has not yet committed. As we will see later, the deleter of a key leaves a trace of its action by X locking the next key for commit duration.

2.3. Fetch Next

If the current cursor position already satisfies the stopping key specification (unique index and a stopping condition of =), then Fetch Next returns right away to the caller with a *not found* status. Otherwise, the leaf page which is expected to contain the key on which the cursor is currently positioned is latched and a check is made to see if the page's current

LSN is different from the LSN remembered at the time of the last positioning. The current key (current cursor position) may not be in the index anymore due to a key deletion earlier by the same transaction. If a change is noticed, then repositioning to the next *key-value,RID* pair is done as in a Fetch call. Except in the case mentioned above, once the next key is located it has to be locked. If the next key satisfies the key range specification, then it will be returned; otherwise, a *not found* condition needs to be returned to the caller.

2.4. Insert

If there is enough space on the leaf page, then, after the page is searched, Insert is positioned at a key with the same key value, positioned at a key with a higher value, or positioned past the last key in the page. If Insert were positioned at an equal key value in a *unique* index, then it requests an S lock on the found key to make sure that the key value is in the committed state, unless of course it is an uncommitted insert of the same transaction. After this lock is granted, if Insert discovers that the previously found key value is still in the index, then it returns the *unique key violation* status to the caller. The lock is obtained for commit duration to make sure that the error condition is repeatable.

In the other cases (see Figure 6), Insert requests an *instant duration* X lock on the next key. This may involve, as in Fetch, having to access the next leaf page to identify the next key. In such a case, the latches on both leaves will be held while the lock is requested. One of the purposes of the *instant duration* lock that is requested on the *next key value* is to determine if, as of the time the X latch was acquired on the leaf (hence the *instant* duration rather than *commit* duration lock), there was any other concurrently running transaction which had looked for and not found the key value being inserted. This is to handle the *phantom problem* and to guarantee RR. In the case of a *unique* index, with next key locking, Insert is also trying to determine if there exists an uncommitted delete by another transaction of the same key value as the one to be inserted.

After doing the next key locking, Insert inserts the key in the correct leaf page, unlatches the page(s), and returns to the user with the *success* status. The latching protocol is used to guarantee that the instant lock was requested on the correct next key. If there isn't enough space to insert the key, then the page splitting algorithm is executed (see Figure 8). The tree latch is acquired in the X mode only after all the affected pages have been brought into the buffer pool. This is done to minimize the serialization delays caused by the X tree latch. The effects of the split are completely propagated up the tree *before* the insert which caused the split is performed. The reason for this delaying of the insert is discussed in the section "3. Recovery in ARIES/IM" (see also [Moha90a, MoLe89]).

```
IF SM_Bit | Delete_Bit = '1' THEN
  Instant S latch tree, set Bits to '0'
Unlatch parent
Find key > insert key & X lock it for instant duration
 /* Next key may be on next page */
 /* Latch next page while holding latch on current page*/
 /* Lock next key while holding latch on next page also*/
 /* Unlatch next page after acquiring next key lock */
Insert key, log and update page_LSN
Release child latch
```

Figure 6: Pseudo-Code for Key Insert Action Routine (No Uniqueness Violation and No Page Split Case)

2.5. Delete

The logic for Delete is shown in Figure 7. After searching the leaf page, Delete should be positioned at the key to be deleted. A *commit duration X lock* is then requested on the next key. This lock is necessary to warn other transactions, which may be looking to insert or retrieve the key value being deleted, about the uncommitted delete. If the key to be deleted is the smallest or the largest one in the page, a *conditional* S latch on the tree is requested. The reason for holding the tree latch when the key to be deleted is a ***boundary key*** is related to recovery (see the section "3. Recovery in ARIES/IM" for further explanations).

After this locking is done successfully, usually Delete deletes the specified key, unlatches the page(s) and returns to the caller. But, if the key to be deleted is the only key in the page, which would make the page become empty after the key delete is completed, Delete invokes the page deletion procedure (see Figure 8). This procedure, like the page split procedure, requests the X latch on the tree *after* ensuring that all the affected pages are already in the buffer pool to minimize the time during which the X latch is held. On obtaining the latch, it deletes the key and then performs the page delete related processing (modifying the neighboring pages' pointers, propagating the page deletion, etc.).

2.6. Discussion

In this section, we try to explain why there are some significant differences in the locking protocols that are followed during the different leaf-level operations. The asymmetry in the next key locking duration (*instant* versus *commit*) for insert and delete comes from the fact that an uncommitted insert is *visible* since a key once inserted begins to exist in the index, whereas a key once deleted is not visible anymore since it disappears from the index. So, in the latter

```
IF SM_Bit = '1' THEN
  Instant S latch tree and set SM_Bit to '0'
Set Delete_Bit to '1'
Unlatch parent
Find key > delete key & X lock it for commit duration
IF delete key is smallest/largest on page THEN
  S latch tree and set Delete_Bit to '0'
Delete key, log and update page_LSN
Release child latch and tree latch, if held
```

Figure 7: Pseudo-Code for Key Delete Action Routine (No Page Delete Case)

[2] In the algorithms that adopt the B^{link}-tree architecture [LeYa81, Sagi86], this ambiguity is prevented from arising by storing explicitly in a nonleaf page the high key associated with its rightmost child also. We discuss in [MoLe89] the negative implications of using B^{link}-trees.

[3] Note that in the case of a leaf and a key delete/insert operation, even if there is no ambiguity about whether that is the right leaf to be at, ARIES/IM waits for any incomplete SMO to be completed. The reason for this has to do with recovery (see the section "3. Recovery in ARIES/IM").

```
Fix needed neighbouring pages in buffer pool
X latch tree and unlatch parent
IF key delete THEN do it as before (Figure 7)
Remember LSN of last log record of transaction
Perform SMO at leaf, set SM_Bit = '1', modify
  neighbouring pages' pointers, log, and unlatch pages
Propagate SMO to higher levels setting SM_Bit to '1'
Write Dummy CLR pointing to remembered LSN
Reset SM_Bit to '0' in affected pages (optional)
IF key insert THEN do it as before (Figure 6)
Release tree latch
```

Figure 8: Pseudo-Code for Structure Modification During Forward Processing

case, as long as the key is in the uncommitted deleted state, we need to leave behind a *strong* lock on a still-existing key for other transactions to *trip on* (i.e., conflict on a lock request) and realize that there is an uncommitted delete. The lock has to be strong enough to prevent others from building a *wall* behind the *tripping point* such that the wall hides the tripping point from the point of deletion. Permitting such a wall to be built would allow some transaction to conclude that a key does not exist when in fact it is still an uncommitted delete and the deletion could get rolled back anytime. In the case of an insert, the inserted key itself serves as the tripping point, whereas for delete the tripping point has to be another key which must be guaranteed to be a *stable* one (i.e., nondeletable by other transactions). The reader should now be able to visualize what is going on. More discussions along these lines with examples may be found in [Moha90a].

3. Recovery in ARIES/IM

Recovery in general works as in ARIES, as we briefly described in the section "1.2. ARIES". In this section, we address those aspects of recovery that are specific to index management. We discuss how some of the concurrency control aspects discussed in the previous section are impacted by recovery considerations. Some of these are very subtle and require careful analysis to understand the problems and the solutions.

Logging In ARIES/IM, all index changes, including those performed as part of undo of updates, are logged such that each log record contains the identity of the affected page and the inserted or the deleted key. The changes performed during undo are *typically* logged using CLRs. During restart, any required redos are performed in a page-oriented manner. In System R, index changes are not logged. Hence, during recovery, any required redos and undos are always performed logically, based on log records for the data pages. To work correctly, that method depends crucially on the shadow-page recovery technique.

Structure Modification Operations Even if a transaction which performed an SMO were to roll back, if all the effects of the SMO had been propagated successfully up the tree before the rollback is initiated, then that earlier SMO is not undone in a page-oriented fashion in order to avoid wiping out the subsequent changes, by other transactions, to the pages involved in that earlier SMO. This is accomplished by taking the following steps:

- The SMO is performed as a nested top action.

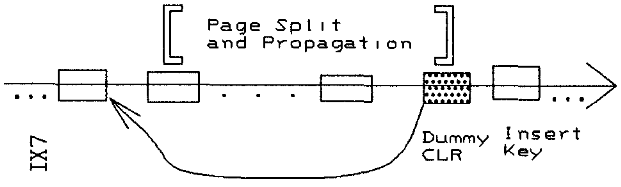


Figure 9: Page Split During Forward Processing

- If an insert requires a page split, then all the actions relating to that split (the leaf-level actions, the propagation up the tree and the writing of the dummy CLR) are completed **before** the insert which necessitated the split is performed (see Figure 8 and Figure 9).
- If the deletion of a key necessitates a page deletion (because the page became empty), then the key deletion is first performed and logged and then all the actions relating to that page deletion are completed. The dummy CLR will point to the key deletion log record (see Figure 8 and Figure 10).

Thus, the dummy CLR lets the transaction, if it were to rollback after completing the SMO, bypass the log records relating to the SMO. At the same time, by performing the key insert/delete causing the SMO outside of the nested top action, it is ensured that, on a rollback, the insert/delete operation causing the SMO will definitely be undone. Partially completed SMOs are undone in a page-oriented fashion to restore the structural consistency of the tree. This undo is acceptable since no other transactions would have been allowed (by the use of the SM_Bit and the tree latch - see below) to modify those pages after this SMO started. At the time of restart recovery, no special processing is performed to determine which indexes are structurally inconsistent. There is no special handling of such indexes. If an incomplete SMO is being undone during normal processing due to a process failure, then the tree latch would be released after all the log records relating to the incomplete SMO are undone.

During a key insert or delete operation (see Figure 6 and Figure 7), if the leaf to be modified is found to have the SM_Bit set to '1', then, even if it is **not** ambiguous whether that is the page to be affected (e.g., in the scenario of Figure 3, the value B, instead of I, is to be inserted by T2), the modification is still delayed until it is ensured that any in-progress SMO has completed (i.e., the SM_Bit can be reset to '0'). This is important since if the insert or delete were to be permitted prematurely then the inserting or deleting transaction could commit and after that the incomplete SMO might have to be undone due to a process or a system failure. The undoing of the incomplete SMO in a page-oriented fashion will cause the state of the leaf page to be restored to the state which existed prior to the beginning of that SMO,

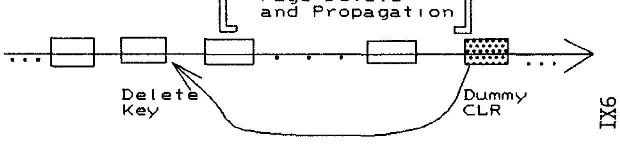
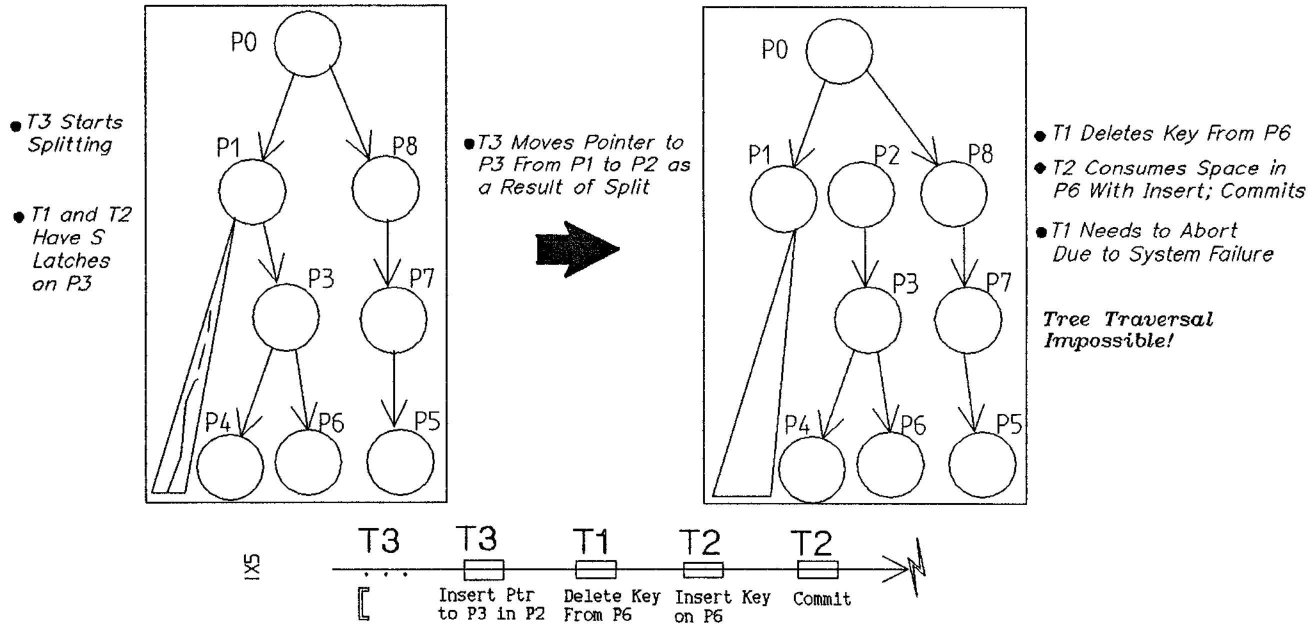


Figure 10: Page Deletion During Forward Processing


T3 Starts Splitting
T1 and T2 Have S Latches on P3
P0
P1
P8
P3
P7
P4
P6
P5
T3 Moves Pointer to P3 From P1 to P2 as a Result of Split
P0
P1
P2
P8
P3
P7
P4
P6
P5
T1 Deletes Key From P6
T2 Consumes Space in P6 With Insert; Commits
T1 Needs to Abort Due to System Failure
Tree Traversal Impossible!
T3
T3
T1
T2
T2
Insert Ptr to P3 in P2
Delete Key From P6
Insert Key on P6
Commit

In this scenario, the space freed by T1's key deletion on P6 is consumed by T2's key insertion which is soon committed. When the key deletion and consumption are happening, the affected leaf page P6 is inaccessible from the root due to an incomplete page split caused by T3 higher up in the tree. Before anything else happens, the system crashes and at restart time the undoing of T1's key deletion necessitates a page split of P6 (since T2 consumed the space) which requires traversing the tree from the root to reach P6. This is impossible since T3's incomplete page split which made P6 inaccessible is not yet undone. Using a Delete_Bit will avoid this sort of a problem since T2 would have realized that before it consumes space it should ensure that there is no ongoing structure modification which is making the leaf inaccessible from the root.

Figure 11: Interaction Between an Incomplete SMO and a Space Consuming Operation

thereby wiping out the effect of the committed key insert or delete operation of the other transaction.

Undo Processing During normal and restart undo processing, undos of key inserts and deletes are performed in a page-oriented fashion, whenever possible. That is, when a key insert/delete needs to be undone, ARIES/IM *first* accesses the page mentioned in the to-be-undone log record and checks to see if that is the right page to perform the undo on, given the current state of that page. Sometimes, undos may have to be performed logically (i.e., by going back through the root of the index, as during forward processing). This will be necessary, for example (see Figure 1), if originally a key K8 was inserted by transaction T1 on page P1, later another transaction T2 split P1 and moved K8 to P2, and then T1 rolled back. Other cases where logical undos are needed are discussed below.

During undo processing, SMOs (both splits and page deletions) may have to be performed. Such SMO related actions will be logged using **regular** (i.e., nonCLR) log records, as in forward processing. This is so that, if such an SMO were to be interrupted by a failure before the completion of the SMO, then, during the subsequent restart recovery, the actions could be undone and tree consistency restored. This is an exception to the general practice, in ARIES, of writing only CLRs during undo processing. ARIES/IM's exceptional logging is necessary since CLRs are redo-only log records and hence their changes would not be undone.

Restart Undo Considerations In order to guarantee that no *traversal* of a particular tree will be attempted at restart recovery time before any *incomplete* SMO for that tree is undone (and also in order to avoid undesirable interactions between an SMO of one transaction and the actions of the other transactions during normal processing, as discussed in the previous section), ARIES/IM serializes SMOs using the tree latch mentioned before.

The X tree latch is released once the dummy CLR, which signals the completion of the SMO, is written to the buffer version of the log. ARIES/IM also uses the tree latch to synchronize, under certain conditions, the different transactions performing key inserts, key deletes, and SMOs involving a particular tree. The idea is to allow the logging relating to key inserts and key deletes for a particular tree by different transactions to go on concurrently with logging relating to an SMO for the same tree by another transaction as long as the latter could not adversely affect the former operations' (key inserts' and deletes') correctness or those operations' subsequent undo if a system failure were to occur. To explain this further, we need to discuss in detail under what conditions logical undos will be necessary.

If an operation performed originally at time t1 needs to be undone at time t2, then, during such an undo, *tree traversal is performed (i.e., logical undo)*, ***only if*** *page-oriented undo cannot be performed due to*

1. lack of enough free space on the original page to undo a key delete, thereby necessitating a page split SMO (i.e., the space freed by the original key delete was consumed by other transactions for their inserts in the time between t1 and t2);

2. the key *definitely* does not belong on the original page anymore: in the undo of a key insert case, the key is not on the page anymore (caused by an intervening page split SMO); in the undo of a key delete case, the original page is no longer a leaf page (caused by an intervening page delete SMO);
3. it is *ambiguous* whether the key belongs on the original page or not: undo of a key delete case - the original page is still a leaf page but the key to be put back is **not** *bound* on the page (***bound*** means that both a higher key and a lower key than the one to be inserted are present in the page); or
4. the undo causes the original page to become empty, thereby necessitating a page delete SMO: undo of a key insert case - since at the time of the original insert there must have been at least one other key on the page (guaranteed by page split logic), it means that there must have been a delete of a boundary key in the time between t1 and t2.

A certain precautionary step must be taken while performing (at time between t1 and t2) any operation on a given page that follows an earlier performed operation (at time t1) on the same page and that could *potentially* cause one of the above four conditions to arise *subsequently* during the undo (at time t2) of the *earlier* performed (at time t1) action, thereby *potentially* forcing the previously logged action's undo to necessitate a tree traversal at the time of a restart undo. Figure 11 illustrates such a problem scenario: during the insert by T2 on P6 which follows the delete by T1 on P6, T2 needs to take the precautionary step so that if T1's delete on P6 were to be rolled back later thereby T1 being forced to perform a logical undo, then T1 will not encounter a structurally inconsistent tree due to which it is unable to reach the leaf level. The precautionary step is to first ensure that a ***point of structural consistency (POSC)*** is reached by requesting the tree latch in the S mode, *before* performing the action (i.e., T2 establishes a POSC before performing its key insert on P6 which consumes the space released by T1). This guarantees that if a system failure were to occur, then by the time the undo pass reaches, if at all necessary, the POSC, the tree would be structurally consistent. The undo pass will access the portion of the log preceding the POSC only if some transaction which started before the POSC had to be rolled back.

Even if the leaf in which a key delete/insert is being attempted is not a participant in an incomplete SMO (i.e., the SM_Bit on the leaf page is equal to '0'), such an operation *may have* to be delayed, if an SMO is going on elsewhere in the tree, until that SMO completes (see Figure 11 for an illustration of the problem). The delaying is necessary only if a system failure which happens after the key insert/delete operation finished but before the inserting or deleting transaction had committed could cause the retraversal of the tree from the root to undo the key insert/delete. Under those circumstances, it must be guaranteed that the tree would be structurally consistent and fit for traversal. We call as the ***region of structural inconsistency (ROSI)*** that portion of the log from the point at which the first SMO related log record is written (indicated by the symbol ⟦) to the point at which the dummy CLR for that SMO is written (indicated by the symbol ⟧). In that region, if another transaction's operation on that index is allowed to be logged then we must be sure that that operation can be undone in a page-oriented fashion. If we are not sure that a logical undo would not be necessary in case the system were to fail right after that action is performed and logged, then we must delay that operation and wait for the ROSI to end and a POSC to be established.

To take care of the situation described under the first of the four reasons given above for tree traversal during restart undo, ARIES/IM makes use of a bit, called the ***Delete_Bit***, on every page. This bit is set to '1' by the transaction doing a key delete on a leaf page (see Figure 7). When a key insert is being attempted on a leaf whose Delete_Bit is set to '1', ARIES/IM first ensures that no SMO is in progress before it allows the insert to proceed (see Figure 6). In the example of Figure 11, note that T2 would notice that the Delete_Bit is equal to '1' and hence it would establish a POSC before resetting the Delete_Bit to '0' and doing its insert. Thus, T2 protects T1 from encountering a structurally inconsistent tree if a system failure were to happen soon after T2's commit of its insert action and T1 had to be rolled back. An alternative to using something like the Delete_Bit to handle the above situation would have been to require that every delete be performed (and logged) only when no SMO is in progress anywhere in the tree. We did not use that option since it would cause too much unnecessary synchronization and reduce concurrency. The cost of this unnecessary synchronization will be even more pronounced when the tree *latch* is converted to a *lock*, in order to allow concurrent SMOs (see the section "5. Conclusions"). Acquiring and releasing a latch costs tens of instructions compared to the hundreds of instructions it costs to acquire and release a lock, even when there is no conflict. The negative concurrency and pathlength implications will become significantly worse in the shared disks environment [MoNa91] where the tree latch will become a **global lock** costing thousands of instructions!

The situations described under the second reason for tree traversal should not cause any problems. This is because the fact that between t1 and t2 an SMO (page split or delete) must have happened for the logical undo to become necessary ensures that, even if the action performed at t1 was in a ROSI, a POSC would have been subsequently established before the logical-undo-causing SMO was allowed to happen. The establishment of the POSC is ensured due to the serialization of all SMOs using the X tree latch.

Because of the situation described under the third reason for tree traversal, ARIES/IM has to make sure, before allowing the (logging of a) delete of a boundary key (i.e., smallest or largest key on a page), that there is no ongoing SMO higher up in the tree which could make the leaf page inaccessible from the root if a failure were to occur. It does this conservatively by establishing a POSC before performing the delete of a boundary key (see Figure 7). Further, it avoids the logging of such a delete during a ROSI by not releasing the S tree latch until the delete operation is completed.

The situation described under the fourth reason for tree traversal should not cause any problems since, as mentioned before, between t1 and t2 the deletion of a boundary key must have taken place. That boundary key deletion, due to the above logic, would have ensured that a POSC was established between t1 and t2.

It is important to note that, in general, the holder of the tree latch will not do any I/Os while holding the tree latch and hence the time interval for which the latch is held should be very small.

4. Deadlocks

The protocols that are followed in acquiring latches guarantee that there will not be any deadlock involving latches. Even though most of the time a latch on an index page is held when a latch on another index page is requested *unconditionally*, no deadlocks are possible because there is a hierarchical ordering amongst the latches (hold parent's latch and request child's latch, or hold leaf's latch and request the **next** leaf's latch). Even when a leaf-level operation (page split or delete) has to be propagated up the tree, the nonleaf page's latch is requested only after the leaf-level latches have been released. No lock is requested unconditionally when one or more latches are held. So, there will not be any lock waits when a latch is held.

It turns out even the tree latch will not be involved in a deadlock. This is so because the holder of a tree latch waits, if at all, only for acquiring latches for propagating the consequences of the leaf-level actions up the tree. No **locks** are requested *unconditionally* while holding the tree latch. The holders of those latches that delay the holder of the tree latch themselves will not wait for any locks or the tree latch, while holding those latches. A rolling back transaction will not be involved in a deadlock since (1) no locks would be requested and (2) only latches on accessed pages will be requested. The exception is that the tree latch may need to be reacquired, in addition to the latches of accessed pages, if a logical undo needs to be performed. Since these latches never get involved in deadlocks, a rolling back transaction will never get into deadlocks.

5. Conclusions

Compared to the System R protocols, ARIES/IM gains a significant amount of concurrency and performance by doing the following: (1) locking individual keys rather than key values, and (2) acquiring latches on pages rather than locks, and holding those latches for much shorter durations and avoiding deadlocks involving them. ARIES/IM reduces the number of locks for single-record operations by performing data-only locking rather than index-specific locking. The OS/2 Extended Edition Database Manager implements a subset of ARIES/IM. Some of the features of ARIES/IM have also been incorporated in SQL/DS V2R2 and V2R3, and in VM's Shared File System which originally used the System R protocols. ARIES/IM supports page-oriented media recovery for indexes - i.e., dumps of indexes can be taken and when there is a problem in reading a page (because, e.g., a crash had occurred when that page was being written), the page can be loaded from the last dump and then, by rolling forward using the log, the page can be brought up-to-date. Details concerning media recovery, deferred restart, etc. are presented in [MHLPS92].

Serialization of SMOs via X latches on the tree was specified earlier only to make the presentation simple. Concurrent SMOs can be easily permitted by changing the tree *latch* into a *lock*. This change to a lock is needed since deadlocks may occur if multiple SMOs are permitted concurrently and since latch deadlocks cannot be allowed to occur as they will not be detected. With this change, while *leaf-level* SMOs are being performed, transactions will acquire the tree lock in the *IX* mode. If a *nonleaf-level* SMO is required, then they will upgrade the IX lock to an X lock (it is due to this upgrading that deadlocks may then be possible since two transactions may attempt upgrading concurrently). In order to avoid rolling back transactions from ever getting involved in deadlocks, such transactions will be made to obtain the tree lock in the *X* mode even as they perform *leaf-level* SMOs.

Acknowledgements Our thanks go to Luis-Felipe Cabrera, Don Haderle, Rajiv Jauhari, Sharad Mehrotra, Inderpal Narang, Rajeev Rastogi and Avi Silberschatz for their comments on earlier versions of this paper.

6. References

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

FuKa89 Fu, A., Kameda, T. *Concurrency Control for Nested Transactions Accessing B-Trees*, **Proc. 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems**, Philadelphia, March 1989.

GMBLL81 Gray, J., et al. *The Recovery Manager of the System R Database Manager*, **ACM Computing Surveys**, Vol. 13, No. 2, June 1981.

Gray78 Gray, J. *Notes on Data Base Operating Systems*, In **Operating Systems**, R. Bayer et al. (Eds.), LNCS Volume 60, Springer-Verlag, 1978.

KuPa79 Kung, H.T., Papadimitriou, C. *An Optimality Theory of Concurrency Control for Databases*, **ACM-SIGMOD International Conference on Management of Data**, Boston, May 1979.

LeYa81 Lehman, P., Yao, S.B. *Efficient Locking for Concurrent Operations on B-Trees*, **ACM Transactions on Database Systems**, Vol. 6, No. 4, December 1981.

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. Also available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989.

Mino84 Minoura, T. *Multi-Level Concurrency Control of a Database System*, **Proc. 4th IEEE Symposium on Reliability in Distributed Software and Database Systems**, Silver Spring, October 1984.

Moha90a 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. A different version of this paper is available as **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.

Moha90b Mohan, C. *Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems*, **Proc. 16th International Conference on Very Large Data Bases**, Brisbane, August 1990.

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

MoLe89 Mohan, C., Levine, F. *ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging*, **IBM Research Report RJ6846**, IBM Almaden Research Center, August 1989.

MoNa91 Mohan, C., Narang, I. *Recovery and Coherency-Control Protocols for Fast Intersystem Page Transfer and Fine-Granularity Locking in a Shared Disks Transaction Environment*, **Proc. 17th International Conference on Very Large Data Bases**, Barcelona, September 1991.

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.

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.

Sagi86 Sagiv, Y. *Concurrent Operations on B*-Trees with Overtaking*, **Journal of Computer and System Sciences**, Vol. 33, No. 2, p275-296, 1986.

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