ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes

C. MOHAN

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

Abstract This paper presents a method, called ARIES/KVL (*Algorithm for Recovery and Isolation Exploiting Semantics using Key-Value Locking*), for concurrency control in B-tree indexes. A transaction may perform any number of nonindex and index operations, including range scans. ARIES/KVL guarantees serializability and it supports very high concurrency during tree traversals, structure modifications, and other operations. Unlike in System R, when one transaction is waiting for a lock on a key value in a page, reads and modifications of that page by other transactions are allowed. Further, transactions that are rolling back will never get into deadlocks. ARIES/KVL, by also using for key value locking the IX and SIX lock modes that were intended originally for table level locking, is able to better exploit the semantics of the operations to improve concurrency, compared to the System R index protocols. These techniques are also applicable to the concurrency control of the classical links-based storage and access structures which are beginning to appear in modern systems also.

1. Introduction

Methods for controlling concurrent access to B-trees have been studied for a long time (see [BaSc77, Mino84, Sagi86, Shas85, ShGo88] and references in them). None of those papers considered thoroughly the problem of efficiently guaranteeing serializability [EGLT76] of transactions containing multiple operations on B-trees, in the face of transaction and system failures, and concurrent accesses by different transactions with fine-granularity locking. [FuKa89] presents an incomplete (in the *not found* case and locking for range scans) and expensive (using nested transactions) solution to the problem. Unfortunately, the details of the algorithms used in existing systems like System R [GMBLL81], SQL/DS [ChGY81], NonStop SQL[1] [Tand87], and DB2[1] [HaJa84] have not been published. 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 the System R protocols, which are used in the IBM product SQL/DS, has been found to be inadequate by some customers [IBM85].

Proceedings of the 16th VLDB Conference
Brisbane, Australia, August 1990

The primary goal of our work was to modify the System R index concurrency control method to drastically improve its concurrency, performance, and functionality characteristics. Serializable executions had to be supported with efficient storage management and high concurrency. We present a method, called ARIES/KVL (*Algorithm for Recovery and Isolation Exploiting Semantics using Key-Value Locking*), which supports very high concurrency during tree traversals, structure modifications (i.e., page splits and page deletes), and other operations (i.e., key inserts, deletes, fetches and range scans). When one transaction is waiting for a lock on a key value in a page, reads and modifications of that page by other transactions are allowed. Transactions that are rolling back will never get into deadlocks, unlike in System R [GMBLL81] and R* [MoLO86]. Although the logging and recovery aspects of ARIES/KVL are not covered in this paper, the concurrent executions permitted by the locking protocols are such that correct logging and recovery are made possible. ARIES/KVL may be used with write-ahead logging (WAL) [MHLPS89, MoLe89, MoPi90, RoMo89] or the shadow-page recovery method [GMBLL81, MHLPS89]. We explain the subtleties involved in index concurrency control, especially with a richer set of index primitives like range scans and with conditions like '<', '<=', '>' and '>=' being associated with the input key value and the key value to be fetched. Most papers on index concurrency control ignore these very important operations.

1.1. Overview

For the benefit of the reader who may at first like to have an overview of ARIES/KVL's locking, the table in Figure 1, summarizes the locks acquired during different operations. In the section "5.5. Discussion", we try to provide an intuitive explanation of ARIES/KVL's locking behavior. Example locking scenarios are sprinkled throughout the paper. This paper is part of the ARIES series of papers that we have authored. These papers describe an *integrated* set of concurrency control and recovery protocols which provide high concurrency and efficient recovery by exploiting the semantics of the user operations.

The rest of the paper is organized as follows. In section 2, we introduce some of the basics relating to locking and latching, and the tree architecture. In order to gradually introduce the reader to the complexities and subtleties involved in index concurrency control, initially, a very simplified view of the index concurrency control problem is presented in section 3 and then a simple algorithm is described. In the rest of the paper, this simple algorithm is enhanced to provide more function and higher levels of concurrency. Section 4 introduces the algorithm for tree traversal, while section 5 presents

		NEXT KEY VALUE	CURRENT KEY VALUE
FETCH & FETCH NEXT			**S** for **Commit** Duration
INSERT	Unique Index	**IX** for **Instant** Duration	**IX** for **Commit** Duration if Next Key Value *Not* Previously Locked in S, X, or SIX Mode **X** for **Commit** Duration, if Next Key Value Previously Locked in S, X, or SIX Mode
	Nonunique Index	**IX** for **Instant** Duration, if *Apparently* Insert Key Value *Doesn't* Already Exist **No Lock**, if Insert Key Value Already Exists	**IX** for **Commit** Duration, if (1) Next Key Not Locked During This Call, *OR* (2) Next Key Locked Now But Next Key *Not* Previously Locked in S, X, or SIX Mode **X** for **Commit** Duration, if Next Key Locked Now and It had Already Been Locked in S, X, or SIX Mode
DELETE	Unique Index	**X** for **Commit** Duration	**X** for **Instant** Duration
	Nonunique Index	**X** for **Commit** Duration, if *Apparently* Delete Key Value Will No Longer Exist **No Lock**, if Value Will Definitely Continue to Exist	**X** for **Instant** Duration, if Delete Key Value Will *Not* Definitely Exist After the Delete **X** for **Commit** Duration, if Delete Key Value *May* or Will Still Exist After the Delete

Figure 1: Summary of Locking in ARIES/KVL

all the locking algorithms for leaf-level operations like key fetch, range scan, key insert and key delete, and for structure modification operations. The locking algorithms used in System R and the experiences with those algorithms in IBM products are described for comparison purposes in section 6. Section 7 concludes the paper by discussing the application of our ideas to other access structures (e.g., links) and other index concurrency control protocols.

2. The Basics

In this section, we introduce some of the basic concepts of locking and latching that are of interest here. We also introduce the index tree architecture that we are assuming in our discussions.

2.1. Locks and Latches

We use locks and latches for synchronizing concurrent activities. ***Latches*** are like semaphores. Usually, latches are used to guarantee physical consistency of data, while ***locks*** are used to assure logical consistency of data. Typically, latches are owned by *processes* whereas locks are owned by *transactions*. The distinction between processes and transactions makes a difference in a system like R* [LHMWY84] in which, even without nested transactions being supported, multiple processes may be working on behalf of a single transaction. We do not exclude the possibility of support for nested transactions in a system which implements ARIES/KVL. In fact, in [RoMo89], ARIES has been extended to support a very general model of nested transactions.

Latches are usually held for a much shorter period of time than are locks. Also, the deadlock detector is not informed about latch waits. Latches are requested in such a manner so as to avoid deadlocks involving latches alone, or involving latches and locks. Acquiring a latch is much cheaper than acquiring a lock (in the no-conflict case, 10s of instructions versus 100s of instructions), because the latch control information is always in virtual memory in a fixed place, and direct addressability to the latch information is possible given the latch name. On the other hand, storage for locks is dynamically managed and hence more instructions need to be executed to acquire and release locks.

The compatibility relationships amongst the different modes (S, X, IS, IX, SIX) of locking that were invented in the context of System R [Gray78] are shown in Figure 2. A check mark ('$\surd$') indicates that the corresponding modes are ***compatible*** which means that two different transactions may hold a lock simultaneously in those modes. ARIES/KVL, by using for key value locking also the IX and SIX lock modes, which were intended originally for table level locking, is able to better exploit the semantics of the index operations to improve concurrency, compared to the System R index protocols. These modes are used in addition to the S and X modes which were the only ones that were originally used in System R for key value locking.

Lock requests may be made with the *conditional* or the *unconditional* option. A ***conditional*** request means that the requestor is not willing to wait if the lock is not grantable immediately at the time the request is processed. An ***unconditional*** request means that the requestor is willing to wait until the lock becomes grantable. Locks may be held for different *durations*. An unconditional request for an ***instant duration*** lock means that the lock is not to be actually granted, but the lock man-

1 DB2, IBM and OS/2 are trademarks of the International Business Machines Corp. NonStop SQL and Tandem are trademarks of Tandem Computers, Inc.

	S	X	IS	IX	SIX
S	√		√		
X					
IS	√		√	√	√
IX			√	√	
SIX			√		

Figure 2: Lock Mode Compatibility Matrix

ager has to delay returning the lock call with the *success* status until the lock becomes grantable. ***Manual duration*** locks are released some time after they are acquired and, typically, long before transaction termination. ***Commit duration*** locks are released only at the time of termination of the transaction, i.e., after commit or abort is completed. The above discussions concerning conditional calls, S and X modes, and durations, except for commit duration, apply to latches also. When a lock request for a resource returns successfully, the lock manager will indicate whether the current transaction was already *holding* (and not yet released) a lock on that resource *before* the current request was issued. In this case, the mode of the previously acquired lock will be returned.

Transactions may request different ***levels of isolation*** (or ***consistency***) with respect to each other. In the context of System R, levels 0, 1, 2, and 3 were discussed [Gray78]. The IBM products SQL/DS, the OS/2 Extended Edition Database Manager[1] and DB2, and Tandem's NonStop SQL support the isolation levels ***cursor stability*** (*consistency level 2* of System R) and ***repeatable read*** (*consistency level 3* of System R). These consistency levels are referred to as ***CS*** and ***RR***, respectively. Both return only committed data to the transactions, unless the accessed data is uncommitted data belonging to the accessing transaction. Due to lack of space, CS transactions are not treated in this paper (see [Moha89]).

With RR, locks are held on all the accessed data until the end of the transaction. Actually, locks are somehow held even on nonexistent data which could have satisfied the query. Later in this paper, we discuss how this is done when the accesses are made via indexes. With RR, if a certain query were to be posed at a certain point in a transaction, and a little later the same query were to be posed within the *same transaction*, then the response to the query would be the same, even if it were a negative response like *not found*, unless the same transaction had changed the data base to cause a difference to be introduced in the responses. If all the transactions are run with RR, then their concurrent executions would be ***serializable*** in the sense of [EGLT76]. That is, the concurrent execution would be equivalent to some *serial* execution of those transactions.

2.2. Conventions and Storage Structures

The data storage model that we assume is that of System R, in which the data (i.e., the records of the table) are stored in a set of ***data pages***, which are separate from the indexes. All the indexes on the table contain only the key values and record identifiers (RIDs) of records containing those key values. The RID of a record identifies the record's location in the set of data pages. All the leaf pages of an index contain ***key-value,RID*** pairs, where the RID is treated as if it were an extra key field. Without loss of generality, we assume that the keys are maintained in ascending collating order on all the key fields, including the RID. The leaf pages alone are forward and backward chained using the ***PrevPage*** and ***NextPage*** fields so that ascending and descending range scans could be supported (see Figure 3).

Every nonleaf page contains a certain number of child page pointers (page numbers) 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 (last) child's page pointer. The high key stored in the nonleaf page for a given child page is always **greater than** the highest key actually stored in the corresponding child page (note that RID is included in the high key).

In most systems, when a nonunique index contains duplicate instances of a key value, the key value is stored **only once in each leaf page** where it appears. The single value is followed by as many RIDs as would fit on that page (see Figure 3, which shows a leaf page - the key value *H* has duplicates (RIDs 4 and 8)). We call this a ***cluster of duplicates***.

The use of the ***key map***, which points to the keys in ascending key sequence, allows one to do binary search even when dealing with varying length keys or nonunique indexes (see Figure 3). In the nonleaf pages also, the key map is used and the map points to *high key,page pointer* pairs in high key sequence. The pointer to the last child alone is stored separately in the page header. The nonleaves which are the parents of leaves have a flag so that, when a leaf is about to be accessed, the *latch* on the leaf can be obtained in the appropriate mode, depending on the operation to be performed on the leaf.

We refer to the page split and page deletion operations as ***structure modification operations (SMOs)***. A page is removed from the tree at the time the only key in the page is deleted. Page splits and deletions are propagated up the tree from the leaves toward the root (i.e., *bottom-up*). To prevent deadlocks involving latches, the page latches acquired at a lower level of the tree (e.g, leaf) will be released before requesting latch(es) at the higher level (e.g., nonleaf). SMOs are serialized by acquiring a ***tree latch*** in the X mode. Typically, no I/Os will be done while the tree latch is held and hence this serialization should not cause any significant reduction in concurrency (see [MoLe89] for more discussions and ways to relax this restriction). During the actual page split or deletion process, the tree structure may be inconsistent. That is, a child may be split (deleted, respectively) but the parent does not yet reflect the effect of that split (deletion). This localized path inconsistency is detected by noticing that a bit called the ***SM_Bit (Structure Modification Bit)*** has a value of '1'. The latter is set by the structure modifying transaction in the affected leaf and nonleaf pages. In order to recover from a problem caused by such an inconsistency and to ensure that incorrect data base recovery does not occur as a result of attempting to perform a key insert or delete on a leaf which is a

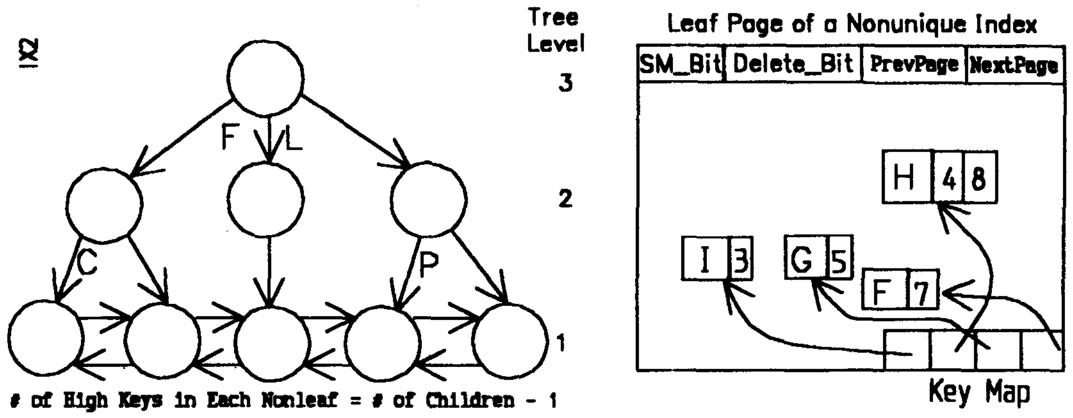

IX
Tree Level
3
2
1
F
L
C
P
# of High Keys in Each Nonleaf = # of Children - 1
Leaf Page of a Nonunique Index
SM_Bit
Delete_Bit
PrevPage
NextPage
H 4 8
I 3
G 5
F 7
Key Map

Figure 3: Tree Architecture

participant in an incomplete SMO (see [MoLe89] for details), the transaction merely requests the tree latch in the S mode, thereby being forced to wait for any incomplete SMO to complete and for a ***point of structural consistency (POSC)*** to be established. Thus, our design principles of ***indication of incomplete SMO via SM_Bit*** and ***SMO serialization via tree latch*** allow us to perform operations only in a valid state of the tree.

We assume that once an index is created its root page does not change. That is, when a root split is needed, the root page's contents are copied to a new page, which becomes the root's child and which is then split. During a split of a leaf, we split to the "right". The new ***high key*** for the split page will be the smallest key that is moved to the new leaf page. We assume that every index page has a ***version number (VN)*** in its header and that every time the page is latched and modified, the VN is updated so that it is a monotonically increasing value. In systems which log index changes (e.g., DB2), VN could be the log sequence number (LSN) of the log record describing the most recent change to the page. In systems which do not log those changes (e.g., System R), it could be a counter which is incremented by one for every change.

3. A Simple Locking Algorithm

In this section, we start with a simple set of index operations and discuss the simple locking algorithm needed to support them. Then, we show why the simple locking algorithm needs to be changed when additional operations and concurrency requirements are to be supported. We delay discussing the algorithm followed during the traversal of the tree until the section "4. Tree Traversal in ARIES/KVL".

3.1. Simple Operations

Let us first consider only the three index operations:

1. ***Fetch***: Given a key value, fetch the RID(s) associated with it, if the key value is present.
2. ***Insert***: Insert the given key (*key-value,RID*).
3. ***Delete***: Delete the given key (*key-value,RID*).

We assume that before the insert and delete calls are made to the index manager, an X lock would have been obtained on the underlying data (either a record lock on the record with the input RID or a data page lock on the data page containing the record, depending on the granularity of data locking). In the algorithm that supports only these operations, the obvious choice for the lock name would be to derive it using the key value. Since key values can vary in size and since lock managers typically deal with only fixed length lock names, the key value would have to be hashed to construct a fixed length lock name. The index ID will also be concatenated with the hash value to make the lock name unique across all the indexes in the data base.

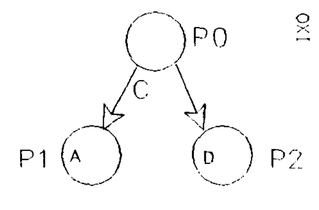
The locking algorithm to be followed is very simple. Once the index manager is invoked with one of the above operations, before the index manager starts traversing the tree, it would first acquire the appropriate mode lock (S for Fetch, and X for Insert and Delete) on the key value. The S lock continues to be held even if the looked-up key is *not* found. This is the way to ensure that no other transaction is able to insert the same key value before the transaction that did the look-up terminates. Hence, the latter will be guaranteed the RR property.

Note that the modes of the locks acquired by readers and modifiers must be incompatible in order to ensure that (1) an uncommitted inserted key of one transaction is not fetched by another transaction and (2) a fetching transaction is blocked when there is an uncommitted delete of the requested key by another transaction. The choice of the S and X modes to ensure this serializability property has the following bad consequence. Even though 2 different transactions are deleting (or inserting) 2 different records belonging to a particular table, if both the records have the same key value for a particular (nonunique) index, then one of them will be forced to wait for the other to terminate. There is no reason to prevent these two operations from happening concurrently. One way to permit this level of concurrency is to let the key delete (or insert) operation acquire the *IX* lock instead of the *X* lock. Since IX is incompatible with S, we still guarantee RR for readers. With these changes, inserts and deletes of the *same key value* can go on concurrently by different transactions. These changes are correct, as long as we consider *only nonunique indexes*.

If we now consider *unique* indexes, we could have a problem with a scenario like the following: (1) T1 deletes the key value 'A' (with RID 20) after acquiring the *IX* lock on 'A'; (2) T2 inserts the key value 'A' (with RID 10) after acquiring the *IX* lock on 'A' (this is possible since IX and IX are compatible and T2 does not find the value 'A' already in the index - T2 does not realize that there is an *uncommitted* delete of 'A'); (3) T2 commits; (4) T1 aborts; (5) T1 puts back the value 'A' (with RID 20). Clearly, there is a problem now since the uniqueness constraint of the index is being violated. Not letting T1 put back the value would violate the transaction atomicity property. The way out of this problem is to change the locking algorithm *for unique indexes alone* so that the insert and delete operations get the *X* lock, instead of the *IX* lock.

3.2. More Powerful Fetch Operation

Now let us consider enhancing the functionality of the index manager so that in the Fetch call, in addition to asking for the RID(s) associated with the given key value, one could also ask for the retrieval of that key value that is present in the index and that is higher than the given key value (e.g., for the leaf page of Figure 3, the Fetch call '> F' should return G,5). The problem here is that before the index manager starts traversing the tree it does not know what lock to obtain. This means that the locking for the Fetch call must be postponed until a key value matching the search criterion is found or it is determined that no such key exists. The same problem will arise, if we enhanced the Fetch call to let the caller specify *only a prefix of a key value* and ask for the retrieval of the first key value that matches the given prefix. Once the key value is determined and a lock is requested, the lock may not be granted right away since the key value could be in the uncommitted state. This condition must be handled carefully. Otherwise, holding the latch on the leaf page and waiting for a lock could lead to a reduction in concurrency and, more importantly, to a deadlock that goes undetected. This requires that the lock be first requested *conditionally* (i.e., while holding the latch on the leaf page). If the lock is not granted, then the latch must be released and the lock must be requested *unconditionally*. Once the lock is finally granted, it may not be correct to return the locked key value as the result, due to situations like the one illustrated in Figure 4. We need to verify that the previously determined information is still valid. We call this design principle ***revalidation after unconditional locking***.



Time	*Tran*	*Action*
10	T1	*X lock C and insert C on P2*
20	T2	Looks up key > A; accesses P1 first and then P2; requests S lock on C and waits
30	T2	Gets lock, notices P2 hasn't changed since going into wait - reads C

Serializability Problem if at Time 25 T1 Inserts B and then Commits, but T2 Does Not Read B

Solution: If Waited on Lock for First Key on Page, Even if Page Has Not Changed, Restart Search From Previous Page if Search Originally Began on Previous Page. Redetermine Key Satisfying Search Condition

Figure 4: Repositioning After Lock Wait

Even after we ensure that what is locked is the right value to be returning to the caller (possibly after being forced to lock a different value from the one that was originally locked due to the changes that occurred while waiting for the first lock to be granted), there may still be problems. This comes from the fact that what the Fetch call locked will not prevent another transaction from later inserting a key value that (better) satisfies the Fetch call of the first transaction. If such an insert were to be permitted and the inserting transaction were to commit later, and then the Fetching transaction were to reissue its call, a different result would be returned, thereby violating the RR guarantee. For example, if the call were to fetch a key value '> F', then 'G' would be locked and returned. But this lock will not prevent another transaction from inserting the value 'FF', which would be returned, instead of 'G', if the Fetch call were to be repeated by the first transaction.

Somehow, the fetching transaction needs to communicate, to inserting transactions, the fact that no new key should be inserted in the *gap* (i.e., between 'F' and 'G'). It would be extremely inefficient to make Fetch acquire an S lock on all possible key values in the gap (i.e., the nonexistent keys). The simplest way this is accomplished is for the Fetch operation to leave behind an S lock on the key value that satisfied the search condition *or the next higher value that is present*, if no existing key satisfied the search condition. Thus, a lock on a key value is really a ***range lock*** on the range of keys spanning the values from the *preceding* key value that is currently present in the index to the locked key value. For this range-locking protocol to work, the inserting transaction must check the lock on the *next key value, before* it does the insert of a given key value. The mode of Insert's lock request must be such that it is incompatible with the S lock acquired by Fetch.

For the time being, let us assume that the mode of Insert's next key lock request is X, as is the case in System R. If the lock on the next key is not grantable right away, then the insert must be delayed until that lock becomes grantable. Instead of using predicate locking, we are using **next key locking** to get a similar effect.

We call this design principle ***range locking via next key locking***. This is a conservative approach since the nongrantability of the X lock on the next key is interpreted to imply that the holder of that lock does not want the insert to happen until the holder terminates, when in fact the holder might have the lock on the next key for a totally different reason (e.g., it is an uncommitted insert, or a Fetch was done specifically for that next key and that Fetch request would not be satisfied by the value about to be inserted). The lock on the next key need not be held after the insert operation has been performed. Only as of the time of an insert, it must be ensured that the lock on the next key is not held by another transaction. Hence, the next key lock is requested for *instant duration* during an insert operation.

Just as Insert needs to do next key locking, Delete also needs to do such locking. Otherwise, when a key is in the *uncommitted deleted state*, Fetch will not be aware of such a key since Fetch locks only those keys that are actually present in the index. As a result, Fetch would wind up retrieving the next key erroneously, when in fact the deleted key would have been the right one to retrieve had it been present. This would cause an RR violation, if the delete were to be rolled back and the Fetch were to be repeated. To handle this correctly, Delete needs to leave behind a lock on the next key which would cause Fetch to wait until the deleting transaction terminates. For this to happen, the mode of the next key lock must be incompatible with the S lock that would be acquired by Fetch. The next key lock must be *held until commit* by the deleting transaction. For the time being, let us assume that the mode of this request is X, as is the case in System R. Contrast the *commit duration* next key lock required in this case with the *instant duration* next key lock that is sufficient for the Insert case - the difference comes from the fact that an uncommitted inserted key is *visible* to transactions that visit the page later, whereas an uncommitted deleted key is not visible, since the key is physically removed from the page.

It should be obvious that the next key locking cannot be done before the tree is accessed since it is not known as to what currently exists in the index as the next key, until the leaf page is accessed.

In the rest of this paper, we enhance this simple algorithm in order to improve its concurrency and performance characteristics. But before presenting the details of the improved algorithm, called ARIES/KVL, we discuss next the problems that we were interested in solving.

3.3. Problems

We would like to use key values as the objects of locking. This is to be contrasted with the **data-only locking** approach taken in ARIES/IM [MoLe89], where the locking is always done on the underlying data record, which is stored elsewhere and whose key is the one in the index entry to be locked. ARIES/KVL's **index-specific key value locking** would be necessary where the records are stored in the index itself and an index entry contains the corresponding record, instead of a record identifier, as in Tandem's NonStop SQL. It could also potentially lead to higher concurrency compared to the data-only locking feature of ARIES/IM, but with an increase in the locking overhead. Given that, the following problems arise:

1. How to support serializability and locking during range scans, without requiring support for predicate locking?

   We would like to avoid having to support predicate locking, even with only simple predicates, since checking for compatibility among predicates is much more expensive than checking for compatibility among locks. With the latter, the data structure is usually a hash table that can be looked up very efficiently. No such simple data structure exists for organizing a collection of predicates and comparing them.

2. If locking a range of keys is going to be supported as in System R (i.e., lock the smallest key that exists in the index that is higher than the end point of the range - see the section "6. System R"), how to allow one transaction to insert a key value that is just smaller than an already existing uncommitted key inserted by another transaction?

3. In a nonunique index, how to let more than one transaction perform concurrently inserts of different index entries with the same key value?

4. In a nonunique index, how to let different transactions concurrently perform deletes of some duplicate instances of keys in two neighboring clusters of duplicates with different key values?

4. Tree Traversal in ARIES/KVL

The basic search routine, ***Search***, takes as input the key (or prefix of key) value, the target level (at which to stop), the action routine (Fetch, Insert, or Delete) to continue processing after Search has reached its target page, and other parameters. Search traverses the index tree from one page to that page's child by holding an S latch on the parent while requesting a latch (in S mode, if nontarget child, else in S or X mode, depending on the action to be performed) on the child. This protocol is called ***lock-coupling*** in [BaSc77]. Our design principle of ***latch-coupling*** allows us to validate the path from the parent to the child.

The pseudo-code for this procedure is given in Figure 5. To simplify the presentation, we have shown only the case where the target level is a leaf. We have not specified the case where the root is a leaf.

If the child is participating in a structure modification, which is done bottom-up, by another transaction, then the traverser waits for the propagation of the SM to be completed.[2] This is done by requesting the tree latch in the S mode, after releasing the page latches. Once the tree latch is granted, Search restarts its tree traversal. We may optimize the retraversal by restarting at the parent, if the parent had not been modified since it was seen last - a modification of an ancestor can be detected by remembering, as Search traversed down the tree, the VNs of all the accessed pages from the root; even if the

```
  /* for simplicity, root = leaf case not specified here */
  S Latch Root and Note Root's VN
  Child := Root
  Parent := NIL
Descend:
  IF Child is a Leaf Page AND Operation is (Insert OR Delete) THEN X Latch Child
  ELSE S Latch Child
  Note Child's VN
  IF Child is a Nonleaf Page OR (Child.SM_Bit = '1') THEN
    IF Child.SM_Bit = '0' THEN                  /* Not part of an ongoing SM */
      IF Parent <> NIL THEN Unlatch Parent
      Parent := Child
      Child := Page_Search(Child)               /* Search Child to Decide Which Page to Access Next */
      Go to Descend
    ELSE                                        /* Unfinished Structure Modification */
      Unlatch Parent & Child
      S Latch Tree for Instant Duration         /* Wait for Unfinished SM to Finish */
      Unwind Recursion as far as Necessary Based on Noted Page VNs and Go Down Again
  ELSE                                   /* Child is Leaf; Appropriate Latch (S or X) Held on Child */
    Unlatch Parent
    1st_Leaf := Child
    CASE Operation OF
      Fetch:  ...                               /* Invoke Fetch Action Routine  */
      Insert: ...                               /* Invoke Insert Action Routine */
      Delete: ...                               /* Invoke Delete Action Routine */
    END
```

Figure 5: Search Procedure for Tree Traversal

parent has changed, we will still be able to restart from that page, as long as the key of interest is still covered by that parent. Otherwise, we can restart from the grandparent and so on recursively, if necessary. Once Search reaches the leaf level, it leaves a latch on the leaf (call it the ***1st leaf***) and passes control to the appropriate action routine (Fetch, Insert, or Delete).

5. Basic Operations in ARIES/KVL

There are four basic index operations that ARIES/KVL 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.
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, Search is called to look for only the key value. For a nonunique index, the whole new key is provided as the Search key.
4. ***Delete***: Delete the given key (*key-value,RID*).

5.1. Fetch

The pseudo-code for the Fetch action routine is given in Figure 6. When Fetch is called with an S latch held on the 1st leaf, it searches the leaf to find a ***satisfying key*** (i.e., the *smallest* key value that matches the starting condition, or, if there is no such key value, then the next higher key value). The 1st leaf could be in one of two states. In the following, we discuss the two cases and explain how Fetch deals with each one of them.

1. *A satisfying key is found in the 1st leaf.*

   This will be the most common case.
2. *1st leaf does not have a satisfying key.*

   Fetch S latches the successor page (call it the ***2nd leaf***). If the 2nd leaf is empty (SM_Bit must be equal to '1'), then Fetch unlatches both pages and requests the tree latch in the S mode. Once the tree latch is granted, it restarts the search. If a satisfying key is found in the 2nd leaf (SM_Bit may be equal to '1'), then Fetch continues to hold the latch on the 1st_Leaf. This is done to make sure that a key satisfying the request does not suddenly appear "behind the back of the searcher" in the 1st leaf, without the knowledge of the searcher, due to the abort of a key delete operation by another transaction or due to an insert by a transaction in forward processing. This much care is required to guarantee RR.

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 value'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

2 We can do much better than this. For simplicity of presentation here, we are not showing how Search can traverse down the tree even when the child is involved in a page split, thereby improving concurrency. Fetch can also deal with a leaf which is a participant in an on-going split. The interested reader is referred to [Moha89, MoLe89] for the details of how these are accomplished.

```
/* Satisfying Key Value - Requested Key, if it Exists; Otherwise, Next Higher Key Value */
IF Satisfying Key Value Not Found in 1st_Leaf THEN
  2nd_Leaf := 1st_Leaf.NextPage
  IF 2nd_Leaf <> NIL THEN                    /* 2nd_Leaf exists */
    S Latch 2nd_Leaf
    IF 2nd_Leaf has a Satisfying Key THEN    /* 2nd_Leaf is Definitely Not Empty */
      Child := 2nd_Leaf
      Found Key := Satisfying Key in 2nd_Leaf
    ELSE  /* 2nd_Leaf Must be Empty - wait for tree latch */
      Unlatch 1st_Leaf and 2nd_Leaf
      S Latch Tree for Instant Duration        /* Wait for Unfinished SM to Finish */
      Unwind Recursion as far as Necessary Based on Noted Page VNs and Go Down Again
  ELSE Found Key := End_Of_File              /* 2nd_Leaf Doesn't Exist */
ELSE                                         /* 1st_Leaf has a Satisfying Key */
  Found Key := Satisfying Key in 1st_Leaf

/* 1st_Leaf & 2nd_Leaf, if Accessed, Will be Held Latched in S Mode */

S Lock Found Key Value for Commit Duration (Maybe End_Of_File)

Unlatch 1st_Leaf and 2nd_Leaf, if Accessed
IF Locked Key Satisfied Search Condition THEN Return Key
ELSE Return Not Found
```

Figure 6: Pseudo-Code for Fetch Action Routine

caller. In any case, while holding the page latch(es), a *conditional* S lock is requested on the found key value.[3] If the conditional request is not granted, then, in order to avoid a deadlock involving latches, the page latches must be released and then the lock must be requested *unconditionally*. Once the unconditional request is granted, then the page must be reexamined to make sure that the previously retrieved information is still the correct one. If the state is not the same, then the new satisfying key must be determined and locked.

With the leaf page as depicted in Figure 3, if the call were to locate the key *GG*, then the lock would be acquired on the key value *H* of *H,4*, which is in the next cluster of duplicates; on the other hand, if the requested key had been G, then the lock would be acquired on the key value G of *G,5*. Even if the requested key value is not found, the next key value is locked to make sure that the requested key does not suddenly appear (due to an insert by another transaction) before the current transaction *terminates* and prevent RR from being possible. As we will see later, the next key value locking done during inserts makes it possible to guarantee RR. 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 the last instance of a key value leaves a trace of its action by X locking the next key value for commit duration.

If the *conditional* lock request is not granted, then the found key and the current page's VN are remembered. Then, the current page and any other page that is still held latched are unlatched and an *unconditional*, manual duration, S lock is requested on the found key value. After the lock is granted, the leaf page which contained the key is relatched. This is done to enforce our earlier described design principle of ***revalidation after unconditional locking***. The VN was remembered before unlatching to make the cost of revalidation cheap. If the page was the 1st leaf page (this check is to handle the problem illustrated in Figure 4) and the page's VN is still the same, no further work is required. This means that the page had not changed since it was seen last. The page is unlatched and control is returned to the caller with the appropriate status, and, possibly, the key.

If the page's VN is different and the page is (1) empty, (2) is no longer part of that index, (3) is no longer a leaf page, or (4) is nonempty but it was the 2nd leaf and the first key in the page is greater than or equal to the *searched-for key* (may not necessarily be the *locked key*), then Fetch restarts Search. If the first key in the page is *less than* the searched-for key, but the page's VN has changed and the highest key in the page is *equal to or greater than* the searched-for key, then Fetch searches that page again; otherwise, it restarts Search. If the key value found now matches the previously locked key value, then the page is unlatched and the appropriate status, and, possibly, the key are returned to the caller. If a different key value is found, then the old key value is unlocked and the new key value is locked using the above algorithm in a recursive manner as necessary.

For a Fetch call that is being issued at the start of a range scan (i.e., the subsequent calls will be Fetch Next calls), if a key is being returned to the caller, then Fetch

3 From now on, to shorten the paper, all the lock calls are shown or described as if they would be granted right away (i.e., when requested initially with the page latches held). To avoid deadlocks and to increase concurrency, as discussed before, 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 corrective action is taken, if a change of interest had occurred when the latches were not held. Before unlatching the pages, the VNs would be noted to make the detection of no changes a cheap operation.

```
IF 1st_Leaf Needs to be Split THEN                 /* No Space for Insert of Key */
  Invoke the Page_Split Procedure and Return

IF Insert Key Value Already in 1st_Leaf THEN       /* No Need to Lock Next Key */
  IF Unique Index THEN
    S Lock Insert Key Value for Commit Duration
    Return Unique Key Violation Status
  ELSE                                             /* Nonunique Index and Insert Key Value Already on 1st_Leaf */
    IX Lock Insert Key Value for Commit Duration
    Insert Key, Log, Unlatch 1st_Leaf and Return

/* Insert Key Value NOT Already in 1st_Leaf */

IF No Higher Key Value in 1st_Leaf AND 2nd_Leaf Exists THEN
  S Latch 2nd_Leaf                                 /* While Holding X Latch on 1st_Leaf */
  IF 2nd_Leaf is Empty THEN                        /* Page Delete in Progress - Wait for it to be Over */
    Unlatch 1st_Leaf and 2nd_Leaf
    S Latch Tree for Instant Duration
    Unwind Recursion as far as Necessary Based on Noted Page VNs and Go Down Again
  ELSE                                             /* 2nd_Leaf is NOT Empty */
    IF Insert Key Value Found in 2nd_Leaf THEN     /* This Can't be a Unique Index */
      IX Lock Insert Key Value for Commit Duration
      Unlatch 2nd_Leaf, Insert Key in 1st_Leaf, Log, Unlatch 1st_Leaf and Return
    ELSE Next Key Value := First Key Value in 2nd_Leaf
ELSE
  IF No Higher Key Value in 1st_Leaf THEN Next Key Value := End_Of_File
  ELSE Next Key Value := Higher Key Value in 1st_Leaf

IX Lock Next Key Value for Instant Duration

Unlatch 2nd_Leaf, if Accessed
IF Next Key Already Locked in X, S or SIX Mode by Current Transaction THEN Lmode := 'X'
ELSE Lmode:= 'IX'
Lock in Lmode Insert Key Value for Commit Duration
Insert Key in 1st_Leaf, Log, Unlatch 1st_Leaf and Return
```

Figure 7: Pseudo-Code for Insert Action Routine

remembers the returned key's key map slot number, and, for a nonunique index, the ordinal position of the returned RID in the RID list of the duplicate cluster. This position information will be used during a subsequent Fetch Next call to avoid a binary search to locate the current key, if the page's VN had not changed in between. To detect a change to the page in between, the page's VN is remembered in the cursor's control block.

5.2. 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 VN is different from the VN 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 before in a Fetch call.

The difference from a Fetch call is that, if the next pair's key value is the same as the current position's key value, then there is no need to lock that value again. If the key value is different and the stopping condition is violated (and hence a *not found* condition needs to be returned to the caller), then that higher key value has to be locked, unless the stopping condition was "=" and the stopping key was the most recently returned key value. In the latter case, the lock already held on the stopping key value will be sufficient to prevent future insertions, by other transactions, of other *key-value,RID* pairs with the stopping key value.

With the leaf page of our example of Figure 3, if the stopping condition is = *F*, then, after *F,7* is returned, if there is a Fetch Next call, then there would be no need to lock G before a *not found* is returned. On the other hand, if the stopping condition is < G or = *FF* then G would be locked.

5.3. Insert

The pseudo-code for the Insert action routine is given in Figure 7. If there isn't enough space to insert the key, then the page splitting algorithm is executed. The pseudo-code for the page split procedure is presented in Figure 8. Note that the tree latch is acquired only after all the affected pages have been brought into the buffer pool. This is done to minimize the serialization delays due to the X tree latch. Note also that the effects of the split are completely propagated up the tree *before* the insert which caused the split is performed. Points like this are further discussed and rationalized in [Moha89, MoLe89].

```
2nd_Leaf := 1st_Leaf.NextPage
IF 2nd_Leaf <> NIL THEN
  Fix 2nd_Leaf in Buffer Pool

X Latch Tree for Manual Duration

Allocate New_Page and X Latch it
New_Page.SM_Bit := '1'
New_Page.PrevPage := 1st_Leaf
New_Page.NextPage := 2nd_Leaf
1st_Leaf.SM_Bit := '1'
1st_Leaf.NextPage := New_Page
Move Some Keys from 1st_Leaf to New_Page
  and Log Changes
Unlatch New_Page
IF 2nd_Leaf <> NIL THEN
  X Latch 2nd_Leaf
  2nd_Leaf.PrevPage := New_Page
  Unlatch 2nd_Leaf
Unlatch 1st_Leaf
Propagate Split Up the Tree, Reset SM_Bits to '0'
Unlatch Tree and Then Restart Insert
```

Figure 8: Pseudo-Code for Page_Split Procedure

If there is enough space, 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 the key value to be inserted is already in the leaf and this is a *nonunique* index, then a commit duration IX lock needs to be acquired on that key value and there is no need to lock the next key value. For the example of Figure 3, an insert of *H,6* would require locking in IX mode the key value *H*. Getting only an IX lock, as opposed to an X lock, allows the insertion of the same key value or a smaller key value by multiple transactions concurrently since IX is compatible with IX. Such concurrent activities are not possible in System R since the inserted key values are always locked in the X mode in that system. Note that since IX is incompatible with S, readers will still not be able to read uncommitted data, unless the reader is the only inserter of that uncommitted value. In the latter case, the held lock mode will become SIX. Since SIX is incompatible with IX, inserts by other transactions will be delayed, as is required.

If the key value to be inserted does not already exist in the 1st leaf and Insert is positioned past the last key in the 1st leaf, then Insert S latches the 2nd leaf and positions on the next *key-value,RID* pair, if there is one, in a manner similar to what happens in Fetch. Note that only the 1st leaf needs to be latched in the X mode. The 2nd leaf needs to be latched only in the S mode since Insert is only trying to prevent a reader from scanning from the highest key in the 1st leaf to the next key in the 2nd leaf.

If Insert were positioned at an equal key value in a *unique* index, then it requests an S lock on the found key value 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 a commit duration one to make sure that the error condition is repeatable.

If the to-be-inserted key value is not already present, then Insert requests an *instant duration* IX lock on the next key value. In System R, the next key lock is always obtained. Furthermore, it is always requested in the X mode. ARIES/KVL's IX locking permits the insert to occur when the next key value is an uncommitted insert of another transaction, whereas System R causes a wait under those conditions. In ARIES/KVL, with the example of Figure 3, an insert of *HH,9* would require acquiring an instant duration IX lock on *I* and later a commit duration IX lock on *HH*.

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 [EGLT76] and to guarantee RR. Note that in a *nonunique* index, when we are adding one more instance to an already existing key value, the IX lock obtained on the key value being added is itself sufficient to make sure that no other concurrent reader's previously read state is being disturbed. Not having to lock the next key value should lead to higher concurrency in this case compared to the other cases. The System R method does not have this optimization. The advantage of not locking the next key value, if we can avoid it, is that the inserter does not have to wait even if the next key value is currently locked by another transaction in an incompatible mode (an S lock due to a read, or an X or SIX lock due to an uncommitted insert or delete - see below). It will possibly save even an I/O if the next key value is not in the same page.

In the case of a *unique* index, with the next key value 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.

When the instant duration lock on the next key value is granted, using the return code from the lock manager, Insert checks whether the *current* transaction already held that lock in the X, SIX, or S mode. If the lock was held in one of those modes, then the key value being inserted must be locked in the *X mode for commit duration*. This we call ***lock state replication via next key locking***. We are essentially transferring a range lock from the next key to the current key. Otherwise, a *commit duration IX lock* must be obtained on the key value being inserted. Note that nothing special needs to be done if the next key value is already held in the *IX* mode by the current transaction (i.e., the next key value is an uncommitted insert of the same transaction).

The reasons for this difference in the mode of locking (X versus IX) of the key value being inserted are subtle.[4] Let us consider an index with the key values *A*, *B*, *E*, *K*,

```
IF >1 Instance of Delete Key Value in 1st_Leaf THEN   /* No Need to Lock Next Key Value */
  X Lock Delete Key Value for Commit Duration
  Delete Key, Log, Unlatch 1st_Leaf and Return

/* Only 1 Instance of Delete Key Value in 1st_Leaf */

IF No Higher Key Value in 1st_Leaf AND 2nd_Leaf Exists THEN
  S Latch 2nd_Leaf                          /* While Holding X Latch on 1st_Leaf */
  IF 2nd_Leaf is Empty THEN                          /* Page Delete in Progress - Wait for it to be Over */
    Unlatch 1st_Leaf and 2nd_Leaf
    S Latch Tree for Instant Duration
    Unwind Recursion as far as Necessary Based on Noted Page VNs and Go Down Again
  ELSE                                       /* 2nd_Leaf is not Empty */
    IF Delete Key Value Found in 2nd_Leaf THEN
      X Lock Delete Key Value for Commit Duration
      IF 1st_Leaf Will Become Empty After Key Delete THEN
        Invoke the Page_Delete Procedure and Return
      ELSE Unlatch 2nd_Leaf, Delete Key from 1st_Leaf, Log, Unlatch 1st_Leaf and Return
    ELSE Next Key Value := First Key Value in 2nd_Leaf
ELSE
  IF No Higher Key Value in 1st_Leaf THEN Next Key Value := End_Of_File
  ELSE Next Key Value := Higher Key Value in 1st_Leaf

X Lock Next Key Value for Commit Duration

Unlatch 2nd_Leaf, if Accessed
IF Delete Key Value is Smallest Key Value in 1st_Leaf AND Nonunique Index THEN
  /* Other Instances of Delete Key May Exist in the Predecessor of 1st_Leaf */
  X Lock Delete Key Value for Commit Duration
  IF 1st_Leaf will Become Empty After Key Delete THEN
    Invoke the Page_Delete Procedure and Return
  Delete Key from 1st_Leaf, Log, Unlatch 1st_Leaf and Return
ELSE                                  /* Delete Key Value Not Smallest in 1st_Leaf */
  X Lock Delete Key Value for Instant Duration
  Delete Key from 1st_Leaf, Log, Unlatch 1st_Leaf and Return
```

Figure 9: Pseudo-Code for Delete Action Routine

and *M*. In the first scenario, let us assume that T1 had done a range scan from *B* through *K*. If now T1 were to insert G and it were to lock G only in the IX mode, then that would permit T2 to insert *F* (T2's request of the IX lock on G would be compatible with the IX lock held by T1) and commit. If now T1 were to repeat its scan, then it would retrieve *F*, which would be a violation of the RR guarantee. When T1 requested IX on *K*, during the insert of G, and found that it already had an S lock on *K*, then it should have obtained an X lock on G. The latter would have prevented T2 from inserting *F* until T1 committed.

In the second scenario, T1 might have an SIX lock on *K* because it had first inserted *K* (getting a commit duration IX on it) and later did a scan of *B* through *K* (getting a commit duration S lock on *K*, which causes the resultant hold mode to be changed from IX to SIX). In the third scenario, T1 might have an X lock on *K* because it had deleted *F* (deletion causes instant duration X lock to be acquired on the deleted key and a commit duration X lock on the next key value, assuming a unique index - see the section "5.4. Delete"). Now, if G were to be inserted by T1 and locked only in the IX mode, then that would permit T2 to insert *F* and commit. If later T1 were to rollback then it would put back its *F* and introduce duplicate keys in a unique index! This is the reason T1 should have noticed that it already held *K* in the X mode and hence should have locked G in the X mode also, thereby preventing the insertion of any key immediately behind G by any other transaction.

After obtaining the X or IX lock request on the key value being inserted, Insert inserts the key in the 1st leaf, 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 value.

5.4. Delete

The pseudo-code for the Delete action routine is given in Figure 9. After searching the leaf page, Delete should be positioned at the key to be deleted. Only if (1) this is a unique index or (2) this is a nonunique index and this key deletion is definitely or, *possibly*, causing the only instance of the key value to be deleted, then the next key value is determined. A *commit duration X lock* is then requested on the next key value. 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. Note that if this weren't the only instance of the to-be-deleted key value currently

4 At this point, the reader may wish to refer to the section "5.5. Discussion" for an intuitive explanation of ARIES/KVL's locking behavior.

in the index, then the commit duration X lock that will be obtained on the to-be-deleted key value itself would be sufficient to let other transactions know about the uncommitted delete and there is no need to lock the next key value.

In System R, the next key value is always locked in the X mode. The advantage of not locking the next key value, if we can safely avoid it, is that it allows new keys to be added after the deleted key value (i.e., key values larger than the deleted key value) and the next key value to be deleted by other transactions, even before the current transaction commits its delete. Also, the deleter does not have to wait, even if the next key value is currently locked by another transaction. Additionally, it allows other transactions to start scanning from the next key value. Furthermore, if the next key value is in a different page, then not having to lock that value will potentially save an I/O.

Next, if this is a unique index or this is a nonunique index and definitely the only instance of the key value in the index is being deleted, then Delete has to X lock for *instant duration* the to-be-deleted key value - this is to make sure that the key value is not currently locked by an active transaction which has performed an index-only scan; if (1) the next key value did not have to be locked or (2) the next key value was locked and it is not definite that the only instance of the key value is being deleted, then the to-be-deleted key value has to be X locked for *commit duration*. The advantage of an instant duration lock, compared to a commit duration one, is that the former does not consume any storage and it does not cause a hash synonym chain in the lock table to become longer. 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. The pseudo-code for the latter is given in Figure 10. 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.).

```
0th_Leaf := 1st_Leaf.PrevPage
IF 0th_Leaf <> NIL THEN Fix 0th_Leaf in Buffer Pool
X Latch Tree for Manual Duration
IF 2nd_Leaf <> NIL THEN Unlatch 2nd_Leaf
Delete Key from 1st_Leaf and Log
1st_Leaf.SM_Bit := '1'
Deallocate 1st_Leaf
IF 2nd_Leaf <> NIL THEN
  X Latch 2nd_Leaf
  2nd_Leaf.PrevPage := 1st_Leaf.PrevPage
  Unlatch 2nd_Leaf
Unlatch 1st_Leaf
IF 0th_Leaf <> NIL THEN
  X Latch 0th_Leaf
  0th_Leaf.NextPage := 2nd_Leaf
  Unlatch 0th_Leaf
Propagate the Delete Up the Tree, Reset SM_Bits to '0'
Unlatch Tree
```

Figure 10: Pseudo-Code for Page_Delete Procedure

5.5. *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.

In the case of Delete, unlike in the case of Insert, the lock mode for the deleted key value and the next key value has to be X instead of IX. The reason is a subtle one. If the next key value lock mode had been IX during a delete, then that would permit another transaction to do an insert of a key value less than the *next* key value, before the commit of the deletion by the first transaction. The newly inserted value may be less than, equal to, or greater than the *deleted* key value. If the newly inserted value happened to be greater than the deleted key value, then RR cannot be guaranteed. For example, let T1 delete *G,5* and lock *H* only in the IX mode for commit duration. This would permit T2 to insert the value *GG*, which it would lock in the IX mode for commit duration. Before actually inserting that key value, T2 would also request an IX lock on *H* which would be granted since it is compatible with the IX lock held by T1. Now, if T2, were to look for G it would not find it and it would then request an S lock on GG which would be granted. Then, T1 might rollback and put back G,5. Now, if T2 were to repeat its search, then it will find G, thereby violating RR!

To see why the mode of the lock on the deleted key has to be X instead of IX, in our example, assume that *H,4* is an uncommitted insert of T1. This means that T1 would be holding an IX lock on *H* for commit duration. Now, let T2 try to delete *H,8*. Since T2 is not deleting the only instance of *H*, it would request a commit duration lock on *H*. Let it be in the IX mode, instead of X. T2's IX mode lock request would be granted since it is compatible with T1's IX mode lock and T2 would delete *H,8* successfully. Then, T1 could rollback removing its *H,4*, now the only instance of *H*. T3 might then try to fetch *H* and not finding it, T3 will lock *I* in the S mode. Then, T2 could rollback and put back *H,8*. Then, if T3 were to repeat its search it would find *H*, thereby violating RR!

The asymmetry between insert and delete partly comes from the fact that an uncommitted insert is "visible" since the inserted key exists in the index, whereas an uncommitted delete is not visible since the deleted key disappears from the index. So, in the latter case, we need to leave behind a "strong" lock on a still-existing key for others to "trip on" (i.e., conflict on a lock request). 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. In the case of an insert, the inserted key itself serves as the tripping point, whereas for delete the tripping point may have to be another key value or if it is the same key value, then it must be guaranteed to be a "stable" one. The reader should now be able to map the above examples to these analogies to visualize what is going on.

Note that, if we are not careful, a transaction which has deleted a key might itself create a wall behind its tripping point, thereby allowing another transaction to create a wall behind the first one's wall, which then enables a violation of RR. To take an example, let T1 try to fetch the range of key values from *F* to those less than G. T1 will fetch *F* and acquire S locks on *F* and G. Then, T1 inserts *FK*, acquiring an instant duration IX lock on G (the next key value) and a commit duration *IX* lock on *FK*. Now, T2 can insert *FC*, acquiring an instant IX lock on *FK* (the next key value) and a commit duration IX lock on *FC*. Then, T2 commits. Now, if T1 were to repeat its range scan, it will retrieve *FC*, thereby violating RR. It is to prevent situations like this that in ARIES/KVL, during an insert, we get an X lock on the inserted key value, if the next key value needs to be locked and that next key value was already locked by the inserting transaction in the S, X, or SIX mode. Thus, the inserter, while "erecting" a wall right behind its own tripping point "replicates" its tripping point on the newly inserted value. This is what we earlier called ***lock state replication via next key locking***.

6. System R

As far as we know, System R was the first system to do key-value locking and support RR. Unfortunately, the System R concurrency control method for index locking was never documented in the literature. ARIES/KVL has some similarities to the System R method, but we have also adapted many of the ideas reported in [MHLPS89, MoLe89] along with other innovations to improve performance and concurrency.

A feature of the System R locking method is that many times (especially during inserts, deletes, and at the end of range scans, and sometimes during fetches) the key value (termed the ***next key value***) following the one(s) of interest is locked in S mode during read operations and in X mode, otherwise. A bigger range of key values (from the one preceding the one(s) of interest to the next key value) gets locked due to this feature. This has been termed the **adjacent key conflict** problem and customers have suffered reduced concurrency due to this also [IBM85]. One way of reducing the occurrence of this problem is to avoid acquiring such locks whenever it is safe to avoid them. Some of the differences between the System R method and ARIES/KVL accomplish this reduction in lock conflicts and the ranges of key values locked.

System R uses page **locks** for physical consistency, while doing key value locking for logical consistency. Unfortunately, all these page locks are not released until the end of the RSS (the data manager) call. This means that these *index and data pages' locks are held even during I/Os and lock waits*. Depending on the operation to be performed, read or write, the page lock will be acquired in the S or the X mode. Typically there will be many I/Os during a single RSS call. The waits for physical locks caused by prolonged holding of the page locks causes deadlocks and unnecessary delays to other transactions. Unfortunately, this approach of holding all the page locks until the end of the RSS call, which amounts to treating each RSS call as a mini-transaction, is also suggested by others that discuss multilevel transaction management (see, e.g., [Weik87]). From practical experience with the SQL/DS product, it has been found that a significant percentage of deadlocks are caused by the page locks when record/key locking is being done. The reduction in concurrency due to the next key locking has also been a cause for concern, especially because the VM/SP Shared File System uses the SQL/DS index manager to store meta-information about ordinary user files, etc.

Since pages are locked even during rollbacks, a transaction that is rolling back may get into a deadlock. System R and R* serialize the execution within RSS by the rolling back transactions to avoid a deadlock involving only such transactions [MoLO86]. Since ARIES/KVL acquires only latches during rollbacks and latches never get involved in deadlocks, transactions that are rolling back will never get into deadlocks.

7. Conclusions

We presented a method called ARIES/KVL for concurrency control in B-tree indexes. Some of the design principles that we adopted in the design of ARIES/KVL to improve concurrency and performance are: (1) use of latches instead of locks for physical consistency, (2) releasing latches during lock waits, (3) revalidation after unconditional locking, (4) use of VN to detect page state changes, (5) range locking via next key locking, (6) lock state replication via next key locking, (7) SMO serialization via tree latch, (8) indication of incomplete SMO via SM_Bit and (9) latch-coupling. The table in Figure 1 summarizes the locking performed by the different leaf-level operations. At most 2 page latches are held simultaneously. ARIES/KVL can used in conjunction with two-phase locking for the table data. As far as we know, compared to the published papers, this is the only paper which presents a comprehensive, and a high concurrency, efficient solution to the problem of providing concurrency control of multiaction transactions operating on B-tree indexes. Due to lack of space, we have not discussed backward scans, protocols for cursor stability and recovery in this paper. The latter and ways to improve concurrency during structure modifications are presented in depth in [MoLe89]. Variations of the presented protocols for cursor stability are discussed in [Moha89]. ARIES/KVL brings us closer to the power of predicate locking using only traditional locking and without using any additional lock modes other than the ones introduced in System R. We have studied alternatives to key-value locking to improve concurrency in indexes in [MHWC90, Moha90, MoLe89].

Many of the design principles of ARIES/KVL are also applicable to the concurrency control of the classical links-based storage and access structures which are beginning to appear in more modern systems also [ShCa89]. If the children records of a parent record are linked together and scans along such links are permitted, then, in order to guarantee RR scans, inserters and deleters of children records would have to do next key locking. Then, our ideas would apply. Our techniques may also be combined with the data-only locking approach of ARIES/IM [MoLe89] to improve concurrency further in ARIES/IM. The basic idea is to make record inserters obtain IX locks rather than X locks on the records or data pages, depending on the locking granularity in use. In the case of page locking, this permits multiple transactions to insert on the same page. For inserts alone, for the price of page locking, we can get the concurrency of record locking! With this change to ARIES/IM, in the index, during a key insert, the lock on the data of the next index entry will be requested in the IX mode rather than in the X mode. If the current transaction is found to have already a lock on the next index entry's data in any mode other than IX, then the lock on the inserted index entry's data is converted to the X mode, if it is not already held in that mode. Thus, some of the features of ARIES/KVL are being implemented in the context of ARIES/IM.

Acknowledgements I would like to thank Laura Haas, Yehoshua Sagiv, Beau Shekita and the anonymous referees for their comments on earlier versions of this paper.

8. References

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

ChGY81 Chamberlin, D., Gilbert, A., Yost, R. *A History of System R and SQL/Data System*, **Proc. 7th International Conference on Very Large Data Bases**, Cannes, September 1981.

EGLT76 Eswaran, K.P., Gray, J., Lorie, R., Traiger, I. *The Notion of Consistency and Predicate Locks in a Database System*, **Communications of the ACM**, Vol. 19, No. 11, November 1976.

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., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I. *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 - An Advanced Course**, Lecture Notes in Computer Science, Volume 60, Springer-Verlag, 1978.

HaJa84 Haderle, D., Jackson, R. *IBM Database 2 Overview*, **IBM Systems Journal**, Vol. 23, No. 2, 1984.

IBM85 IBM *SQL/Data System Diagnosis Guide for VM/ System Product Release 3.5*, **SY24-5230**, November 1985.

LHMWY84 Lindsay, B., Haas, L., Mohan, C., Wilms, P., Yost, R. *Computation and Communication in R*: A Distributed Database Manager*, **ACM Transactions on Computer Systems**, Vol. 2, No. 1, February 1984.

MHLPS89 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*, To Appear in **ACM Transactions on Database Systems**. Also Available as **IBM Research Report RJ6649**, IBM Almaden Research Center, January 1989.

MHWC90 Mohan, C., Haderle, D., Wang, Y., Cheng, J. *Single Table Access Using Multiple Indexes: Optimization, Execution and Concurrency Control Techniques*, **Proc. International Conference on Extending Data Base Technology**, Venice, March 1990, Springer-Verlag. An expanded version is available as **IBM Research Report RJ7341**, IBM Almaden Research Center, March 1990.

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.

Moha89 Mohan, C. *ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes*, **IBM Research Report RJ7008**, IBM Almaden Research Center, September 1989.

Moha90 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. Also available as **IBM Research Report RJ7344**, IBM Almaden Research Center, February 1990.

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.

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. Also IBM Research Report RJ5037, IBM Almaden Research Center, February 1986.

MoPi90 Mohan, C., Pirahesh, H. *ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method*, **IBM Research Report RJ7342**, IBM Almaden Research Center, February 1990.

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 available as **IBM Research Report RJ6650**, IBM Almaden Research Center, January 1989.

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

Shas85 Shasha, D. *What Good are Concurrent Search Structure Algorithms for Databases Anyway?*, **IEEE Database Engineering**, Vol. 8, No. 2, June 1985.

ShCa89 Shekita, E., Carey, M. *Performance Enhancement Through Replication in an Object-Oriented DBMS*, **Proc. ACM-SIGMOD International Conference on Management of Data**, Portland, June 1989.

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

Tand87 The Tandem Database Group *NonStop SQL: A Distributed, High-Performance, High-Availability Implementation of SQL*, **Proc. 2nd International Workshop on High Performance Transaction Systems**, Asilomar, September 1987.

Weik87 Weikum, G. *Principles and Realisation Strategies of Multi-Level Transaction Management*, **Technical Report DVSI-1987-T1**, Technical University of Darmstadt, 1987.