High-Concurrency Locking in R-Trees*

Douglas Banks Marcel Kornacker Michael Stonebraker

Computer Science Div., Dept. of EECS
University of California
Berkeley, California 94720

Abstract

In this paper we present a solution to the problem of concurrent operations in R-trees, a dynamic access structure capable of storing multidimensional and spatial data. We describe the R-link tree, a variant of the R-tree that adds sibling pointers to nodes, a technique first deployed in B-link trees, to compensate for concurrent structure modifications. The main obstacle to the use of sibling pointers is the lack of linear ordering among the keys in an R-tree; we overcome this by assigning sequence numbers to nodes that let us reconstruct the "lineage" of a node at any point in time. The search, insertion and deletion algorithms for R-link trees are designed to lock at most two nodes at a time and the locking can be shown to be deadlock-free. In addition, we describe how R-link trees can be made recoverable so that they are instantly available after a crash and we further describe how to achieve degree 3 consistency with an inexpensive predicate locking mechanism.

1 Introduction

One of the future requirements for databases is the ability to support multidimensional and spatial data. This support is crucial for non-traditional database applications such a s CAD, Geographical Information Systems (GIS) or temporal databases, to name a few. A fundamental aspect of support for spatial data is efficient handling of range queries along multiple dimensions; one example is the retrieval of points that intersect a given query rectangle. The most widespread access method, the B-tree [BaMc72], does not handle multi-dimensional data very well.

[Gutt84] proposed a spatial access method designed to handle multidimensional point and spatial data. Unlike other spatial access methods [Bent75, Niev84, Robi81, LoSa90], R-trees are not restricted to storing multidimensional points, but can directly store multidimensional spatial objects, which are represented by their minimal bounding box. R-trees have not benefited greatly from the many refinements and optimizations of concurrency mechanisms that have been designed for B-trees. A particular modification of B-trees, the B-link tree [LeYa81], connects

*This research was sponsored by the National Science Foundation under grant IRI-9107455, the Defense Advanced Research Projects Agency under grant T63-92-C-0007, and the Army Research Office under grant 91-G-0183. Additional support was provided by the University of California and Digital Equipment Corporation under Research Grant 1243. Other industrial and government partners include the State of California Department of Water Resources, United States Geological Survey, Construction Engineering Research Laboratory (CERL) of the U.S. Army Corps of Engineers, the National Aeronautics and Space Administration (NASA), Epoch Systems, Inc., Hewlett-Packard Corp., Hughes Aircraft Company, MCI, Metrum Corporation, PictureTel Corporation, Research Systems Inc., Science Applications International Corporation, Siemens Inc., and TRW Space and Electronics.

the siblings on each level via rightward-pointing links and compensates for unfinished splits by moving across these links. It has recently been shown that this technique offers the highest performance among concurrency mechanisms for B-trees [SrCa91, JoSh93]. Unfortunately, the B-link tree technique expects the underlying key space to have a linear order and therefore cannot be directly applied to R-trees.

In this paper we present R-link trees, an extension of R-trees motivated by Lehman and Yao's work that offers the same high level of concurrency as B-link trees. We circumvent the requirement for linearly ordered keys by introducing a system of sequence numbers that are assigned to each page and are used to determine when and how to traverse sibling links.

The remainder of this paper is organized as follows. Section 2 provides background on R-trees and B-link trees. Section 3 goes into detail on the difficulties in applying the structural modification of B-link trees to R-trees, presents the formal definition of an R-link tree and describes the search and insert algorithms. It also outlines the deletion algorithm. Section 4 presents a way to make R-link trees recoverable so that they are immediately available at restart. Next, section 5 shows how to make scan results serializable. Section 6 provides a discussion of related work. Finally, section 7 gives a brief summary.

2 Background and Motivation

2.1 R-Trees

R-Trees are a hierarchical, height-balanced indexing structure similar to B-Trees. Like a B-Tree, R-Trees have leaf nodes and internal nodes with entries in leaf node pointing to disk records and entries in internal nodes pointing to other internal nodes or leaf nodes. A node corresponds to a disk page and has between m and M entries ($1 < m \leq M$). The only exception is the root, which may hold between 1 and M entries. Unlike B-trees, R-trees are indexed on multi-dimensional keys that have no linear order defined on them.

An entry in a leaf node of an R-Tree contains a disk tuple identifier and the key, which is either a multidimensional point or a rectangular outline of the spatial object it represents. An entry in an internal node summarizes the node it points to by storing as the key the minimum bounding rectangle that tightly encloses all the keys in the child node.

The information contained in an R-Tree is thus hierarchically organized and every level in the tree provides more detail than its ancestor level. A pointer to an indexed object is stored in the tree only once, but keys at all levels are allowed to overlap, possibly making it necessary even for point queries to descend multiple subtrees. Since multidimensional keys cannot be linearly ordered there is no single "correct" place for a particular key; consequently, it can conceivably be stored on any leaf.

The search process in an R-Tree is very different from that in a B-Tree due to the lack of ordering and the possible overlap among keys. For example, to find all rectangles intersecting a given range the search process will have to descend all subtrees that intersect or fully contain the range specification. Furthermore, since an entry in an internal node summarizes the child node with a bounding rectangle, there is no guarantee that the child contains any keys of interest, even if its bounding rectangle intersects our search range.

The strategy for placing entries on leaf nodes should therefore create an efficient index structure that optimizes retrieval performance. The literature has identified a variety of parameters for the layout of keys on nodes that affect retrieval performance [BKSS90, SRF87]. These pa-

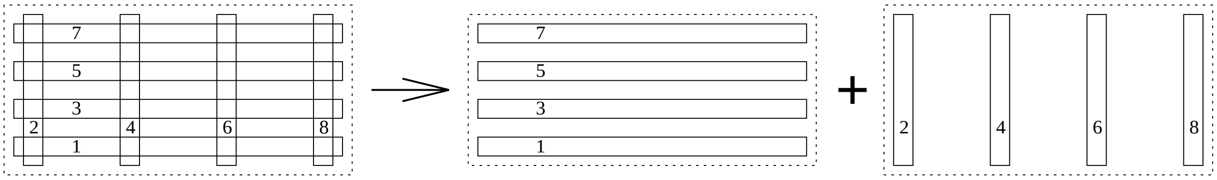

Figure 1: Overlap can be unavoidable after a split.

rameters are: minimal node area, minimal overlap between nodes, minimal node margins or maximized node utilization. It is impossible to optimize all of these parameters simultaneously. For instance, the original R-tree proposal [Gutt84] minimized overlap between nodes; the R*-Tree variation [BKSS90] minimizes overlap for internal nodes and minimizes the covered area for leaf nodes.

When a new key has to be inserted in an R-tree, it is necessary to descend to the leaf that optimizes the parameter chosen by the particular R-tree variant. In contrast to B-trees, R-trees have to recursively update the ancestor keys if a leaf's bounding rectangle changes. Splitting a node also deviates noticeably from the B-tree pattern. Whereas the B-tree simply “cuts” the sequence of keys stored in the overflowing node in half, the R-tree will partition the key sequence according to its layout strategy. Figure 1 illustrates the scenario of a split where the layout strategy is minimal overlap. Note in this example that it is impossible to completely avoid any overlap.

2.2 Concurrency in B-Trees

When multiple search and insertion processes are carried out on a B-Tree in parallel, their interactions may be interleaved in a way that leads to incorrect results. Simple solutions to this problem have the insertion process lock the entire tree or the subtree that needs to be modified due to anticipated splits. Variations thereof lock the upper levels of the subtree so that only readers can still access it [BaSc77]. In essence all of these methods employ top-down lock-coupling: when descending the tree a lock on a parent node can only be released after the lock on the child node is granted. Consequently, locks are held during an I/O operations, which reduces concurrency.

A radically different approach was proposed in [LeYa81]. Instead of avoiding possible conflicts by lock-coupling, the tree structure is modified so that the search process has the opportunity to compensate for a missed split. The crucial addition is the rightlink, a pointer going from every node to its right sibling on the same level (excluding the rightmost nodes). When a node is split and a new right sibling is created, the old node's rightlink is copied to the new node and is then changed to point to the new node itself. The effect is that all nodes at the same level are chained together through the rightlinks. Furthermore, the sequence of the nodes in the rightlink chain reflects the sequence of their corresponding entries in the ancestor level; in short, the rightlink chain orders the nodes by their keys. This is true for every level of the B-Tree and is a result of the splitting strategy in B-Trees, where the upper half of the key sequence is moved to the new right sibling.

Searching in a B-link tree can therefore be done without lock-coupling. When descending to a node that was split after examining the parent, the search process discovers that the highest key on that node is lower than the key it is looking for and correctly concludes that a split must have taken place. It compensates for this split, or multiple splits, by moving right until it comes to a node where the highest key exceeds the search key. Likewise, an insertion process does not have to employ lock-coupling when descending the tree to the correct leaf. If the leaf has to be split, it is also possible to avoid lock-coupling when installing a new entry in the parent, as is shown in [LaSh86] and [Sagi86]. As soon as the page has been split and the new right sibling inserted into the rightlink chain, the insertion process can drop the lock on the leaf that was overflowing and then acquire a lock on the parent,[1] possibly moving right to compensate for concurrent splits and possibly splitting the parent itself, leading to recursive splits up the tree. This locking strategy is deadlock-free and offers very high concurrency because search and insertion processes only need to hold one node locked at a time.

3 R-Link Trees

We would like to achieve high concurrency for operations on R-trees, and given the similarities in structure and functionality between B-trees and R-trees, it would seem natural to try to apply the ideas and algorithms of [LeYa81] to create an "R-link tree." This is not a trivial matter, however, because R-trees differ from B-trees on a number of important points and the B-link tree strategy itself is insufficient.

The source of this problem is the lack of ordering on R-tree keys. The core of the link-tree strategy is to account for splits that have not updated the parent by moving to the right. To implement that strategy we must answer two questions: how do we detect that the child has split and how do we limit the extent to which we move right. For R-trees, the latter question is not only relevant for efficiency, it is relevant because we descend multiple subtrees and may therefore end up visiting the same node twice if we move too far to the right.

For B-link trees, the answer to those questions lies in the linear ordering that is defined on the key space and the fact that the nodes on a single level are ordered through the rightlink chain by their keys. This allows us to detect a split and to determine when to stop moving right based on key comparisons. For spatial data, there is no such ordering, and therefore it is impossible to apply the same strategy and to do key comparisons. Consider the following two situations:

- It is possible that the key of an entry in the parent intersects the search range, even if the keys in the child do not. In this case, it would be wrong to conclude that the child has split and move right. Using a notion analogous to the high key in a B-tree, we could recompute the bounding rectangle of the child node and compare that to the key seen in the parent in order to detect a split. Doing so might cause us to miss a split because taking entries out of a node does not necessarily change its bounding rectangle (see figure 1).
- Even if we are sure that a node has split, it is infeasible to limit the extent to which we move right by doing key comparisons. The number of matching keys in the original node is between 0 and M, but it is unknown in advance. In order to not miss any keys, we would generally have to move right until we reach the end of the rightlink chain. In a multi-gigabyte index, this strategy could force us to scan millions of leaf nodes for every search request. Again, the basic problem is that the keys do not provide enough information to correctly delimit the range of nodes to visit.

[1] Note that this makes it possible that an insertion is overtaken by another insertion on its way up the tree. This can cause problems if the B-tree accepts duplicate keys. If two insertion processes split the same node and the updates in the parent are done in reverse order, the order of entries in the parent will not reflect the order of nodes in the rightlink chain on the child level. This destroys the tree structure.

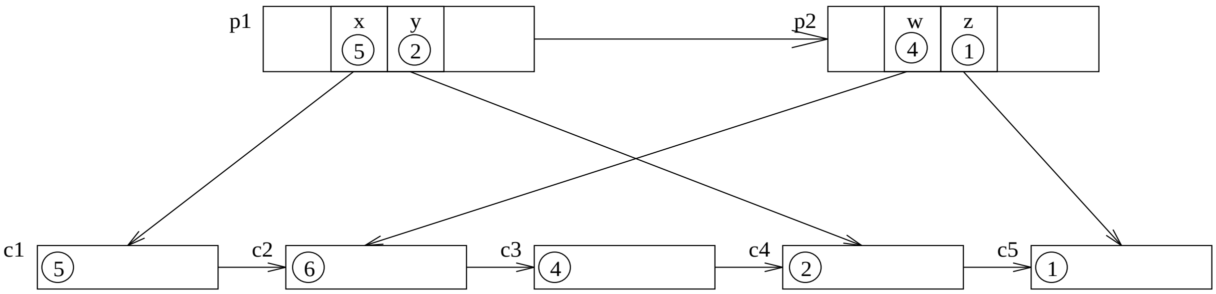

Figure 2: A subsection of an R-link tree (circled numbers are LSNs).

We need to provide each operation on an R-tree with a way of determining whether it has accurate information about the current state of any node it might examine, and how it should proceed if it finds that its information is obsolete.

3.1 Structure of an R-Link Tree

Clearly, if we are to provide high concurrency operations on R-trees through a rightlink-style approach, we need to add some additional information to the standard R-tree that can be used to correctly traverse a constantly-changing tree structure. We propose fulfilling this requirement by assigning logical sequence numbers (LSNs) to each node. These numbers are similar to timestamps in that they monotonically increase over time but are not synchronous with any real-time clock. The node entries and the search and insert algorithms are designed so that these LSNs can be used to make correct decisions about how to move through the tree.

An R-link tree is basically a standard R-tree, as described in section 2.1, with two key differences. First, like a B-link tree, all of the nodes on any given level are chained together in a singly-linked list via rightlinks. It is very important to note that, unlike the B-link tree, the chain of nodes on a given level does not represent an ordering of the keys from smallest to greatest, and, in general, it will not reflect the ordering of their corresponding entries in the nodes on the parent level. This is illustrated in figure 2. In the rightlink chain of the parent level, p_1 precedes p_2. However, c_4, which is a child of p_1, does not precede c_2. This situation can arise if p_1 splits and moves the entry for c_2 over to the new right sibling, p_2.

Second, the main structural addition is an LSN in each node that is unique within the tree. These LSNs give us a mechanism for determining when an operation's understanding of a given node is obsolete. Each entry in a node consists of a key rectangle, a pointer to the child node and the LSN that it expects the child node to have. If a node has to be split, the new right sibling is assigned the old node's LSN and the old node receives a new LSN. A process traversing the tree can detect the split even if it has not been installed in the parent by comparing the expected LSN, as taken from the entry in the parent node, with the actual LSN. If the latter is higher than the former, there was a split and the process moves right. When the process finally meets a node with the expected LSN, it knows that this is the rightmost node split off the old node.

R-link trees can be formally defined as a balanced tree in which index nodes consist of a set of entries and a rightlink r. On each level of the tree the rightlinks form the nodes on that level into a singly-linked list. Entries on internal nodes consist of a key rectangle k, a pointer p, and an expected LSN l so that either:

1. p points to a child node N, where l is the LSN of N, and the rightlink of N points to NULL or to some node R which is also pointed to by some entry in the level above. In figure 2, entry x points to node c_1; both x's LSN and c_1's LSN are matching and c_1's rightlink points to c_2, which is also pointed to by entry w in p_2. This situation represents the normal case, where the node structure on the child level is fully reflected in the entries on the parent level.

2. p points to a child node N, where the LSN of N is greater than l, and there exists a node N' whose LSN is l, which can be reached by following rightlinks from N through nodes with LSNs higher than l which are not pointed to by any entry in the level above. N' also has no entry in the level above, but its right sibling, if N' is not the end of the chain, does. An example from figure 2 is the entry w in p_2. The LSN in w is smaller than that of c_2 and equal to the LSN of c_3, which in turn can be reached from c_2 by following one rightlink. Node $c3$ does not yet have an entry in the level above, but its right sibling, node c_4, is pointed to by entry y in p_1. This situation was caused by a split of node c_2, which has not yet been installed in the parent node.

Note that in either case, the right sibling R of the node whose LSN matches the entry's expected LSN has an entry in some node on the parent level. This entry can generally be anywhere in the parent level. Node c_4 in figure 2 is an example where this entry is in a node to the left of the parent node of c_2.

3.2 The Search Algorithm

A search process has to find all the entries on leaf nodes that fall in the query range, and since keys can overlap, it will generally have to descend multiple subtrees within the index. The underlying data structure to support this is a stack, which is used to remember which nodes still have to be visited. The process starts by initially pushing the root on the stack. A node that has not yet been examined is popped off the stack and all entries in the node that qualify for the search condition are in turn pushed onto the stack and the whole process is repeated. If a leaf node is popped off the stack, we can return the qualifying entries that we find on it. The search is terminated when the stack is empty.

In order to remember a yet-to-be visited node on the stack, we push the pointer and the LSN we found in the corresponding entry. If we examine a node p and find that the LSN is higher than the one on the stack, we know that this node has been split in the meantime. To compensate for the split we must examine all of the nodes that have been split off from this node since we first pushed its entry. Therefore we push nodes to the right of p, up to and including the node with the LSN equal to the expected LSN for p.

The search process, as shown in figure 3 is implemented with an iterator-like interface. The first call to *search* will return the first record and subsequent calls to *continueSearch* will return all other matching items until the stack is empty.

3.3 The Insertion Algorithm

An insertion proceeds in two stages: first we must locate the leaf to insert the key on, remembering the path we take as we descend the tree; next, then the new key is inserted and the leaf possibly split. If the leaf's bounding rectangle has changed, we must propagate the change to its ancestor node. This is accomplished by backing up the tree until we arrive at a parent node that does not need to be changed. If the leaf was split we must also install a new entry in the

```
search(Rect r)
{
    push(stack, [root, root-lsn])
    return reduceStack(r)
}

continueSearch(Rect r)
{
    return reduceStack(r)
}

reduceStack(Rect r)
{
    while not empty(stack) {
        [p, p-lsn] = pop(stack)
        if (p is pointer to indexed tuple)
            return p
        else {
            r-lock(p)
            if p-lsn < LSN(p) {
                traverse the rightlink chain
                    starting at rightlink(p)
                    to the node with
                    LSN = p-lsn;
                for each node n along the
                rightlink chain:
                    r-lock(n)
                    push(stack, [n, LSN(n)])
                    r-unlock(n)
            }
            for all entries e of p
            intersecting r:
                push(stack,
                    [node-pointer(e), LSN(e)])
            r-unlock(p)
        }
    }
    return done
}
```

Figure 3: The Search Algorithm

parent node. If it is full, we recursively split nodes up the tree until we arrive at a node with enough free space or alternatively split the root. The latter case requires special attention and is further commented on below. Note that in contrast to a B-tree insertion, we must back up the tree for two reasons: splitting a node requires the installation of a new entry and changing the bounding rectangle requires the adjustment of the keys in the ancestor nodes.[2] The latter step is missing in B-trees.

When descending the tree to a leaf, we choose the geometrically optimal subtree. However, if we detect that a node has been split, we must take into consideration all the nodes to the right of the original node that were split off it. As in the search algorithm, this chain is delimited to the right by the node carrying the original LSN. When we are updating parent keys during ascent, we also must move right if the parent node has split. Notice in this case that no LSN is necessary to recognize the split or delimit the rightlink chain. An entry in a node can be uniquely identified by the node pointer[3] it contains; for that reason, we move right until we find the node with that particular entry.

When backing up the tree one level, we employ lock-coupling; that is, we hold the child node write-locked until we obtain a write-lock on the parent. As explained in subsection 2.2, this is generally not necessary in B-trees. It is necessary in R-trees because the key of the old entry is modified. If we do not couple the locks, another inserter causing a split can overtake us and install the changes before us. When it is finally our turn, we will update the key, unaware of the previous changes to the child node. The key will not reflect the bounding rectangle of the child anymore and the tree structure will be incorrect. It is important to recognize that it is not necessary to do lock-coupling when moving right. If another inserter overtakes us while we are moving right and splits the nodes we are examining, it is impossible for us to miss the entry for

[2]These two changes have to be applied atomically in order to guarantee the R-link tree properties of section 3.1. Atomicity is achieved by holding the lock on the parent node until both updates are done.

[3]Node pointers do not change after a split because the original node is kept in place.

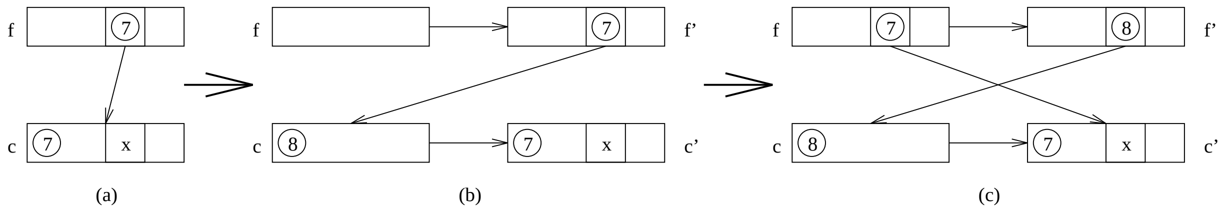

Figure 4: An incorrect structure modification.

the child node since a split can move entries only right.

After finding the parent node and updating the entry's key, we may still have to install a new entry for a new right sibling. If the parent node was split, we would like to insert the new entry on the geometrically optimal node in the chain. Unfortunately, this is not possible and the new entry has to be installed on the same page that contained the old entry (or its new right sibling if the insertion causes a split). The reason for this can be seen in figure 4. Suppose a search process is looking for item x contained in leaf node c (situation a). Node f and c are split independently and item x is moved to the new leaf c' (situation b). If the splitting of f is already reflected in the parent level, the search process will navigate directly to f'. In situation c, the entry for c' has been installed in f, because this results in a geometrically better node layout than a placement on f', and the entry in f' for c has also been updated (key and LSN). In this case, the search process will be unable to find leaf c' because it never considers going to f. On the other hand, if the entry for c' had been installed in f', the search will have been successful.

In principle, this requirement could deteriorate the tree structure by forcing its keys to have more overlap than necessary. We expect this potential drawback to have little effect in practice because only in rare cases will the geometrically optimal node differ from the node containing the entry for the old child.

The implementation of the insert algorithm is shown in figure 5. The individual procedures do the following: *findLeaf* descends to the geometrically optimal leaf, recording the path along the way and finally write-locks the leaf; *extendParent* is called after a leaf split to recursively install an entry for the new leaf in the parent and to propagate the changed bounding rectangle of the old leaf; *updateParent* is called only after a leaf's bounding rectangle has changed in order to recursively propagate the new bounding rectangle of that leaf.

To keep the algorithm as short as possible, we do not consider the case where multiple insertions are carried out at the same time and a splitting of the root by one inserter goes unnoticed by the others. This is problematic when the remaining inserters have to change the bounding rectangle of what they believe is the root or if the "root" has to be split a second time. A solution can be found in [LaSh86] and [Sagi86]; both suggest using an anchor page to make root splits visible to other insertion processes and allow for compensating actions.

3.4 Deletion

Deleting a key from an R-tree can be implemented by allowing each leaf to have fewer than its lower bound of m entries. The deletion algorithm then simply removes the key from the leaf, or multiple leaves if the deletion is given a key range, and adjusts the bounding rectangles of the ancestor nodes. This is effectively combining a search with a subset of the insertion algorithm, the actual implementation of which is not shown here. The adjustment phase of a deletion has

```
insert(Rect r)
{
    stack = findLeaf(root, r, root-lsn)
    leaf = pop(stack)
    insert r on leaf
    if leaf was split {
        extendParent(leaf,
            bounding-rect(leaf),
            LSN(leaf), right sibling,
            bounding-rect(right sibling),
            LSN(right sibling), stack)
    } else {
        if bounding-rect of leaf changed {
            updateParent(leaf,
                bounding-rect(leaf), stack)
        } else {
            w-unlock(leaf)
        }
    }
}

Stack findLeaf(RTreeNode p, Rect r, LSN p-lsn)
{
    if p is leaf {
        w-lock(p)
    } else {
        r-lock(p)
    }
    if p-lsn < LSN(p) {
        p = geometrically optimal node to take
            r in rightlink chain starting at p
            and ending at node with
            LSN = p-lsn
    }
    if p is leaf {
        push(stack, p)
        return stack
    } else {
        e = entry on p leading to
            geometrically optimal subtree
            for r
        entry = findBestEntry(p, r)
        push(stack, p)
        r-unlock(p)
        return findLeaf(e, r, LSN(e))
    }
}

extendParent(RTreeNode p, Rect p-rect,
    LSN p-lsn, RTreeNode q, Rect q-rect,
    LSN q-lsn, Stack stack)
{
    if empty(s) {
        create new root (w-locked) with 2
        entries:
        - for child, key: p-rect
        - for sibling, key: q-rect
        w-unlock(q)
        w-unlock(p)
        w-unlock(new root)
        return
    } else {
        parent = pop(s)
        w-lock(parent)
        find the entry for node p in parent
            or one of its right siblings;
            let parent = that node and
            entry = that entry
        w-unlock(q)
        w-unlock(p)
        update  entry with p-rect and p-lsn
        insert q on parent
        if parent split {
            extendParent(parent,
                bounding-rect(parent),
                LSN(parent),
                right sibling,
                boundingRect(right sibling),
                LSN(right sibling), stack)
        } else {
            if bounding-rect(parent) changed {
                updateParent(parent,
                    bounding-rect(parent),
                    stack)
            } else {
                 w-unlock(parent)
            }
        }
    }
}

updateParent(RTreeNode p, Rect p-rect,
    Stack stack)
{
    if empty(stack) {
        w-unlock(p)
        return
    } else {
        parent = pop(stack)
        w-lock(parent)
        find the entry for node p in parent or
            one of its right siblings; let
            parent = that node and
            entry = that entry
        w-unlock(child)
        update key in entry with p-rect
        if bounding-rect(parent) changed {
            updateParent(parent,
                bounding-rect(parent), stack))
        }
}
```

Figure 5: The Insertion Algorithm.

the same locking behaviour as that of an insertion. Since a deletion never merges any nodes, it interacts with other search, insertion, or deletion processes like an insertion.

A deletion algorithm that never attempts to merge nodes will not degrade space utilization in the tree to unacceptable levels if the rate of insertions and deletions is about the same. If there are occasional bursts of delete requests that together remove a large fraction of the tree's entries and cause the space utilization to drop considerably, the entire tree can be write-locked and reorganized off-line.

4 Recovery

R-link trees can be made instantly recoverable from system failures with the same strategy that was originally proposed for B-link trees. The updated nodes are immediately written to disk and the results of node splits are written back in a particular order: first, the newly created right child; next, the original left child; finally, the updated parent. This ordering guarantees that the disk copy maintains the structure of the tree as was defined in section 3.1, and regardless of when a crash intercepts the writes, no information in the tree is lost or doubly-visible at any time. Writing the new child clearly does not affect the disk copy of the tree, since no node entries on the disk copy point to it. Writing the old child also makes the now disk-resident right child visible at the same time. Although the disk copy of the parent node remains unaware of the split, the parent entry for the old child is correct according to the second part of the definition in 3.1. Note that no entries are hidden or doubly-visible at this point—the old child no longer contains the entries that were moved to the new child and the new child is visible through the rightlink pointer from the old child and the LSN in the parent entry. Finally, writing the parent also maintains the correctness of the tree on disk; in one indivisible step we insert an entry for the new child and change the key and expected LSN for the old child so that the new child will no longer be accessed via the old child's rightlink. If the parent splits the write-ordering is carried over to the next higher tree level.

A crash after the second write makes an unfinished split permanently visible in the tree. Although it does not violate the structural requirements of an R-link tree according to our definition, it has two undesirable side effects. First, when descending to a leaf through the region of an unfinished split, we compensate for it as we do for an ongoing split. This forces us to regularly traverse the rightlink and access an extra node, adding to the total number of I/Os. Second, unlike in a B-link tree, an insertion process now has to be aware of *unfinished* splits during the ascending phase. Under normal processing, splits are invisible to insert operations propagating changes up the tree because we do lock-coupling. An insert operation ascending the tree can safely assume that it can find a parent entry for any node it passed through on the way down. With an unfinished split, however, the insertion process will still not find an entry for a node it passes through on its way up. This becomes a problem when the bounding rectangle of that node changes or the node splits. In either case, a non-existent parent entry has to be updated. An example of a situation that can result from an unfinished split is shown in figure 2. An inserter expanding the bounding rectangle of node c_3 will not find a corresponding parent entry to update.

One way to take care of unfinished splits is to have a recovery phase at restart that traverses the entire tree and repairs it, making it unnecessary for inserters to take them into account but considerably delaying the availability of the index after a crash. We can avoid a restart phase if we extend the insertion algorithm to detect unfinished splits and repair them by supplying all of the missing entries to the parent and updating the original entry with a new key and LSN. The steps involved are as follows:

1. We become aware of the unfinished split during ascent when we cannot find an entry at the parent level for the child node that was split or updated. The entry in the parent level has to carry the valid LSN of the child or the pre-split LSN, otherwise it is obsolete and also indicates an unfinished split. Going back to figure 2, the LSN in the parent entry for node c_2 (4) differs from c_2's actual LSN (6) and shows that there is an unfinished split. Taking into account that the entry might be missing, we have to put a limit on how far we move right when looking for the entry. We must also remember the LSN of the parent node on the stack during descent in order to notice when we have to stop crossing rightlinks.

2. Once we know about an unfinished split, we have to acquire locks on the entire chain of nodes that are part of the unfinished split(s)[4] before we can do the repair. In order to identify the relevant part of the rightlink chain, we have to add two more items to each stack entry: the node through which we entered the particular tree level and the LSN we expected it to have. Consider figure 2 again for an example. If we had to pass through $c2$ and $c3$ on our way to a leaf, we would have remembered the pair $(c_2, 4)$ to indicate the starting point and extent of a potential unfinished split.

   Before acquiring the locks on the chain, we have to drop any locks we still hold, otherwise we violate the locking ordering for nodes and can cause a deadlock. After we locked the chain, we have to verify that the tree structure has not been repaired already by a competing insertion process. This can be done by checking the parent for a valid entry (with the matching LSN) for the starting node of the chain. In fact, a valid entry for any of the nodes on the chain tells us that the structure is already repaired because, once the locks are set the parent is updated atomically to reflect the entire chain of nodes. After repairing the unfinished split, we can proceed with the insertion as usual, propagating changes in the parent's bounding rectangle and possibly a split further up the tree.

The above approach of repairing the tree structure during normal processing does not add any extra overhead to an insertion process not encountering unfinished splits. Looking for an unfinished split can be done at no extra cost when propagating updates or splits up the tree. The only drawback in comparison to a separate repair phase at restart is that we cannot make any guarantees when unfinished splits will be repaired. Since the frequency of crashes is typically very low, we do not expect temporary unfinished splits in the tree to become a performance problem.

5 Consistency

A common requirement for concurrenct access in database systems is *degree 3 consistency*, or *repeatable read* (RR) [Gray78]. A simple solution employed for B-trees is to keep all leaf pages that were read by an index scan locked until the end of the transaction. This strategy depends on the linear order of the keys and leaves and the fact that index scans always visit a contiguous sequence of leaves.

[4] There can be at most three unparented nodes left after a crash: after a node is split and written back but before the parent is written back and unlocked, each of the resulting child nodes can be split exactly once. No further splits are possible, because the most recent insertion processes block on the parent node while still holding locks on the children.

In R-trees, keys can be inserted on arbitrary leaves and an insertion into the key range of a previous scan can succeed even though the scan locked all of the leaves it read. If the insertion commits, the new key will be visible to a re-scan, giving rise to a phantom. An example for a two-dimensional key space is shown in figure 6. Boxes 1 and 2 are the bounding rectangles of internal nodes, boxes 3 to 6 are the bounding rectangles of leaves and the dashed box is the query rectangle. If the scan is looking for overlapping keys, only leaf 4 qualifies and consequently it is the only leaf that is visited and locked. The insertion of a new key into leaf 5 extends its bounding box into the query rectangle, so that a re-scan will be able to see the new key, violating degree 3 consistency.

One way to avoid the phantom problem is for scans to keep every node they traversed locked until the end of the transaction, including internal nodes. This way, even a successful insertion into a leaf cannot propagate the new key so far up the tree that a scan with a conflicting key range can see it. The major disadvantages are that by setting locks on internal nodes it reduces concurrency more than necessary and also introduces deadlocks. A searcher descending the tree can now collide with an inserter propagating changes up the tree.

A more effective solution to the phantom problem is to use a simplified form of predicate locks [EGLT76], where exclusive predicates consist of a single key value and shared predicates consist of a query rectangle and scan operation such as inclusion or overlap. A new scan request would check the list of still-active insertions and suspend itself if its query rectangle collides with any of the unommitted new keys. A new insertion would in turn check the list of active scans and also suspend itself on a collision with a query rectangle. If a scan commits and leaves the system it is removed from the active list and the waiting inserters are rechecked to see if some can be activated; the case of an inserter committing is handled symmetrically. The advantages of this over the former page-locking scheme are that no deadlocks are possible and concurrency is not unnecessarily restricted, since an insertion can still propagate changes up to the root as long as it does not fall in the specified ranges of active scans. The disadvantages attributed to general-purpose predicate locks for tables, exponential runtime and overly pessimistic behaviour, do not apply here. To evaluate a predicate we simply check a key value against a query rectangle and a lock request is only rejected if there is a guaranteed collision with another active lock.

6 Related Work

So far there has not been much work published on the concurrency control problem in R-trees. None of the algorithms known to us attempt to adapt the B-link tree strategy to R-trees in order to achieve higher concurrency.

Ng and Kameda [NgKa93] present three algorithms varying in complexity and possible concurrency. The simplest of the three algorithms locks the entire tree so that an insertion would

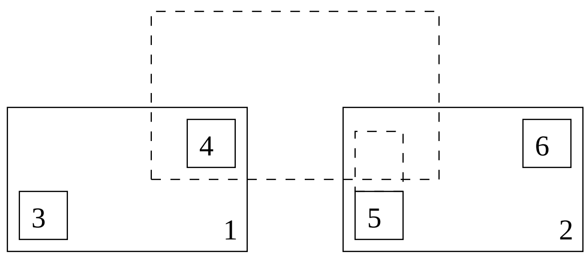

Figure 6: An example where 2-phase locking of leaves cannot guarantee RR.

exclude all searchers. The second algorithm postpones page splits by adding buffer space to each node to accommodate overflows. When overflows or underflows take place, a separate maintenance process exclusively locks the entire tree and reorganizes it, splitting and merging several nodes in the same run. Because an insertion never performs a split itself, there is no need for concurrent search processes to do lock-coupling. The highest-concurrency algorithm is modeled after one presented in [BaSc77] for B-trees. Readers do top-down lock-coupling when descending the tree in order to avoid having to deal with splitting pages. Insertions lock the entire subtree that needs modification on their way to the leaf.

Biliris [Bili89] presents an approach to B-tree locking that can be applied to R-trees. The structure of a regular B-tree node is modified to contain left and rightlink pointers. Unlike in [LeYa81], a search process does top-down lock-coupling in order to descend to a leaf and does not make use of the link pointers to account for splits. The link pointers are used to link side branches—the left or right halves of overflowing pages—into the tree. This is done by the insertion process without acquiring exclusive locks on the corresponding full nodes. After the insertion process creates the side branches bottom-up they are incorporated top-down into the tree, exclusively locking the entire subtree. For B-trees, the height of the subtree to lock is determined by the highest unsafe node on the path to the leaf. The author mentions that, for R-trees, the algorithm also must address the case where bounding rectangles have to be changed. This can easily be incorporated into the algorithm.

[Moha90, MoLe92] and [LoSa92] discuss making access structures recoverable in a write-ahead logging environment; the former papers also present a solution for guaranteeing degree 3 consistency with row-level locking.

ARIES/KVL and ARIES/IM both use a conventional, non-link tree structure, yet they are able to propagate splits bottom-up without locking subtrees and to let top-down traversing processes recover from them. Instead of following rightlinks, pages that are involved in a split are marked so that a search that runs into an ongoing split is able to notice it and retraverse the tree starting from the lowest unmodified parent node. A particular consideration for a write-ahead logging environment is that modifications of the tree structure would normally be logged within the context of the transaction that initiated the modification. Under theses circumstances, if the transaction were to do a page-oriented rollback it would also automatically undo the structure modifications. Therefore, access to the newly created or modified pages has to be prevented until the transaction commits, otherwise the updates of other transactions on those pages would also disappear. In order to circumvent this severe concurrency limitation, tree structure modifications are separated from all transactions via nested top level transactions. Unlike in a B-link tree, a partially executed structure modification may leave parts of the tree temporarily invisible. Taking into account that some operations might have to be rolled back logically, requiring tree traversal, it is necessary to serialize complete splits, including propagation, so that no two splits can take place at the same time. Moreover, there are situations in which insert or delete requests also have to be serialized with structure modifications. To avoid the phantom problem when doing record-level locking, ARIES/IM and ARIES/KVL employ next key locking, where a scan also sets a shared lock on the next-highest key past its scan range. Again, this is not applicable in R-trees because the key space is not linearly ordered and the notion of a next-highest key does not exist.

The Π-tree presented in [LoSa90] is a generalization of a B-link tree where nodes can have multiple parents, which turns the tree structure into a DAG. Their solution for recoverability capitalizes a the property of link-type trees, namely, that unfinished splits leave all parts of the tree accessible. As in [MoLe92], modifications of the tree structure are done separately from inserting or deleting transactions in order to avoid holding commit-duration locks on nodes. However, becaus the index remains fully accessible even if a split is interrupted by a crash and cannot be fully propagated upward, it is not necessary for structure modifications in the Π-tree to be serialized or for access operations on a Π-tree to synchronize with ongoing structure modifications. This recovery method is also applicable to B-link trees.

7 Summary

In this paper, we have presented R-link trees, an extension on R-trees designed to support high concurrency. R-link trees look and work very similar to B-link trees, with each operation holding only a few locks at one time and handling unexpected splits by moving across link pointers to sibling nodes on the same level. The key differences in the design of B-link trees and R-link trees are a result of the fact that spatial keys cannot be ordered linearly. Where B-link trees rely on the actual keys involved in the search to resolve unexpected splits, R-link trees have to use a system of sequence numbers assigned to each node. The degree of concurrency obtainable with R-link trees should be as good as the best B-tree algorithm, the B-link tree. Descending an R-link tree to a leaf requires no lock-coupling; consequently, only a single node needs to be locked at any time. An insert or delete process ascending the tree only needs to hold a maximum of two nodes locked, allowing many updates of the index to take place concurrently. An R-link tree can be made instantly recoverable from system crashes by forcing out changed nodes in a particular order. To enforce degree 3 consistency of index scans, an inexpensive variant of predicate locking is more effective than commit-duration locks of nodes.

Acknowledgement

We would like to thank Paul Aoki, Bob Devine, Sunita Sarawagi, Paul Brown, Mike Olson and Joe Hellerstein for their comments on this paper. Paul Aoki deserves special thanks for his almost inexhaustible patience. We would also like to acknowledge Megan Metters who helped to make this paper more readable.

References

[BaMc72] R. Bayer and E. McCreight, "Organization and Maintenance of Large Ordered Indexes," *Acta Informatica*, Vol. 1, No. 3, pp. 173–189, 1972.

[BaSc77] R. Bayer and M. Schkolnick, "Concurrency of Operations on B-Trees," *Acta Inf.*, Vol. 9(1977), pp. 1–21.

[Bent75] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *CACM*, September 1975, Vol. 18, No. 9, pp. 509–517.

[Bili89] A. Biliris, "Operation-Specific Locking in Balanced Structures," *Information Sciences*, June 1989, Vol.48, (No.1):27–51.

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. 1990 ACM SIGMOD Conf.*, pp. 322–331.

[EGLT76] K. Eswaren, J. Gray, R. Lorie and I. Traiger, "On the Notions of Consistency and Predicate Locks in a Database System," *Comm. ACM*, November 1976, Vol. 19, No. 11, pp. 624–633.

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

[Gutt84] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. 1984 ACM SIGMOD Conf.*, pp. 47–57.

[JoSh93] T. Johnson and D. Shasha, "The Performance of Current B-Tree Algorithms," *ACM TODS*, Vol. 18, No. 1, pp. 51–101, March 1993.

[LaSh86] V. Lanin and D. Shasha, "A Symmetric Concurrent B-Tree Algorithm," *1986 Fall Joint Computer Conference (Dallas, Tex., Nov. 1986)*, pp. 380–389.

[LeYa81] P. Lehman and S. Yao, "Efficient Locking for Concurrent Operations on B-Trees," *ACM TODS*, Vol 6, No. 4, December 1981.

[LoSa90] D. Lomet and B. Salzberg, "The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance," *ACM TODS*, Vol 15, No. 4, pp. 625–685, December 1990.

[LoSa92] D. Lomet and B. Salzberg, "Access Method Concurrency with Recovery," *Proc. 1992 ACM SIGMOD Conf.*, pp. 351–360.

[Moha90] C. Mohan, "ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions," *Proc. 16th Int'l Conf. on Very Large Databases (VLDB)*, Brisbane, August 1990.

[MoLe92] C. Mohan and F. Levine, "ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging," *Proc. 1992 ACM SIGMOD Conf.*, San Diego, June 1992.

[NgKa93] V. Ng and T. Kameda, "Concurrent Accesses to R-Trees," *Proceedings of Symposium on Large Spatial Databases*, pp. 142–61, Springer-Verlag, Berlin 1993.

[Niev84] J. Nievergelt, H. Hinterberger and K. C. Sevcik, "The Grid File: An Adaptable, Symmetric Multikey File Structure," *ACM TODS*, Vol. 9, No. 1, March 1984.

[Robi81] J. T. Robinson, "The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes," *Proc. 1981 ACM SIGMOD Conf.*, pp. 10–18.

[Sagi86] Y. Sagiv, "Concurrent Operations on B*-Trees with Overtaking," *Journal of Computer and System Sciences*, Vol. 33, No. 2, pp. 275–296, 1986.

[SrCa91] V. Srinivasan and M. Carey, "Performance of B-Tree Concurrency Control Algorithms," *Proc. 1991 ACM SIGMOD conf.*, pp. 416–425.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R+-tree: A Dynamic Index for Multidimensional Objects," *Proc. 13th Int'l Conf. on Very Large Databases (VLDB)*, Sep. 1987, pp. 507–518.