High-Concurrency Locking in R-Trees

Marcel Kornacker*
Universität Hamburg
22527 Hamburg, Germany
kornacke@dbis1.informatik.uni-hamburg.de

Douglas Banks
University of California at Berkeley
Berkeley, CA 94720-1776, U.S.A
dbanks@cs.berkeley.edu

Abstract

In this paper we present a solution to the problem of concurrent operations in the R-tree, 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, insert and delete algorithms for R-link trees are designed to completely avoid holding locks during I/O operations and to allow concurrent modifications of the tree structure. In addition, we further describe how to achieve degree 3 consistency with an inexpensive predicate locking mechanism and demonstrate how to make R-link trees recoverable in a write-ahead logging environment. Experiments verify the performance advantage of R-link trees over simpler locking protocols.

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 as 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 multidimensional 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 the siblings on each level via rightward-pointing links and compensates for unfinished splits by moving across these links. This technique avoids holding locks during I/O operations and has recently been shown to offer the highest degree of concurrency among locking protocols 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 [BKS94], an extension of R-trees motivated by Lehman and Yao's work that shows similiar locking behavior and therefore offers the same high degree 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. Our deletion algorithm removes nodes as soon as they become empty without the need for a separate reorganization phase, a novel feature for link-style trees.

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 sketches the deletion algorithm. Section 4 shows how to make scan results serializable. Next, section 5 presents a way to make R-link trees recoverable in a write-ahead logging environment. Section 6 presents performance results and section 7 provides a discussion of related

* This work was done while the author was visiting the University of California, Berkeley. It was supported by the Defense Advanced Research Projects Agency under grant T63-92-C-0007 and the Army Research Office under grant 91-G-1083. The author's new address: University of California at Berkeley, Berkeley, CA 94720-1776, U.S.A.; e-mail: marcel@postgres.berkeley.edu.

Proceedings of the 21st VLDB Conference
Zürich, Switzerland 1995

work. Finally, section 8 gives a brief summary.

2 Background and Motivation

2.1 R-Trees

An R-tree is a hierarchical, height-balanced indexing structure similar to a B-tree. Like B-trees, 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 in B-trees, the keys in R-trees are multi-dimensional objects 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 has 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 the 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 parameters 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] minimizes 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, we attempt to descend to the geometrically optimal leaf by picking at each level the subtree with the optimal bounding rectangle. 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. When doing lock-coupling, locks are held during I/O operations, which should be particularly detrimental to high concurrency of insert and delete operations in R-trees. When descending the tree via lock-coupling, locks can be acquired in shared mode, allowing many search and update operations to descend the tree concurrently. But update operations can block on coupled read locks during tree ascent. For B-trees, tree ascent only takes place as a result of a node split or deletion. For R-trees, it also takes place in order to propagate a changed bounding rectangle. The latter can be expected to occur far more frequently and unpredictably than node splits or deletions.

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, it is inserted into the rightlink chain directly to the right of the old one. 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

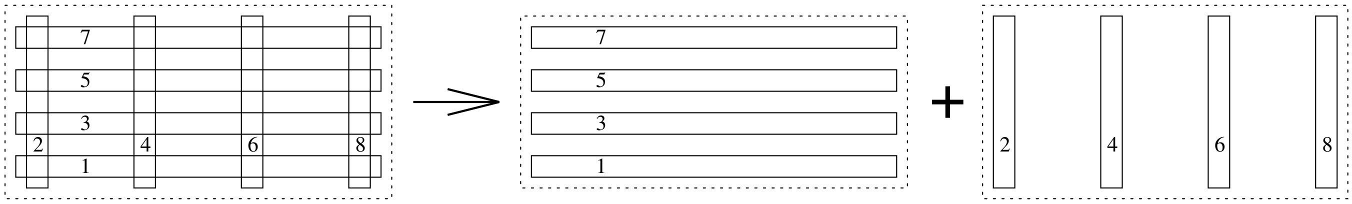

Figure 1: Overlap can be unavoidable after a split.

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, 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. It is impossible to apply the same strategy to R-trees. First of all, keys cannot conclusively tell us when a node has split. 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 also 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). But even if we are sure that a node has split, it is impossible to limit the extent to which we move right by doing key comparisons. Adjacent nodes in the rightlink chain might have a bounding rectangle that intersects our search range, even though they did not take part in the split we detected. As mentioned before, we must not visit these nodes via rightlink traversal because we will visit them later on while searching a different path in the tree.

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

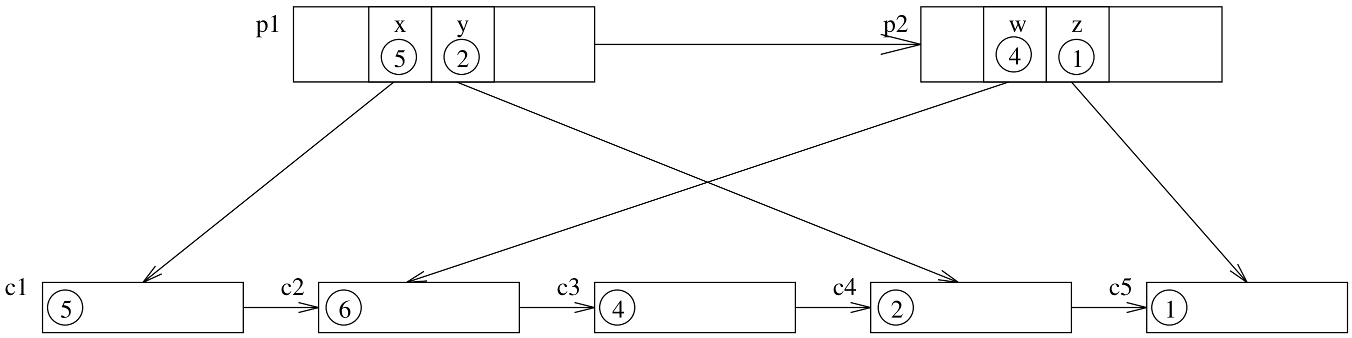

Figure 2: A subsection of an R-link tree (circled numbers are LSNs).

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. *(normal case – child-level structure fully reflected in parent)*
   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.

2. *(uninstalled split in child level compensated by rightlink)*
   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

```
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)
      end
      for all entries e of p
      intersecting r:
        push(stack,
          [node-pointer(e), LSN(e)])
      r-unlock(p)
    end
  end
```

Figure 3: The search algorithm

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 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. 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.[1] The latter step is missing in B-trees.

When descending the tree to a leaf, we choose the geometrically optimal subtree at each level. 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[2] 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. 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 would update the key, unaware of the previous changes to the child node. The key would not reflect the bounding rectangle of the child anymore and the tree structure would be incorrect.[3] This is a deviation from the locking behavior in B-link trees, where lock-coupling during ascent is not necessary. In practice, this should make little difference to the achievable degree of concurrency, because parent nodes can be expected to still reside in main memory and therefore no locks are held during I/O operations.

The implementation of the insert algorithm is shown in figure 4. 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.

There is one restriction to consider when installing a new child entry in a parent node, which might not be immediately obvious. 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 5. Suppose a search process is looking for item x contained in leaf node c (situation a).

[1] These two changes have to be applied atomically in order to guarantee the R-link tree properties of section 3.1. Atomic changes are further dealt with in section 5.

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

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

```
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)
    end
  end

Stack findLeaf(RTreeNode p, Rect r, LSN p-lsn):
  if p is leaf
    w-lock(p)
  else
    r-lock(p)
  end
  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
  end
  if p is leaf
    push(stack, p)
    return stack
  else
    e = entry on p leading to
      geometrically optimal subtree
      for r
    push(stack, p)
    r-unlock(p)
    return findLeaf(e, r, LSN(e))
  end

extendParent(RTreeNode p, Rect p-rect,
 LSN p-lsn, RTreeNode q, Rect q-rect,
 LSN q-lsn, Stack stack):
  if empty(stack)
    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(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(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)
      end
    end
  end

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(p)
    update key in entry with p-rect
    if bounding-rect(parent) changed
      updateParent(parent,
        bounding-rect(parent), stack))
    else
      w-unlock(parent)
    end
  end
```

Figure 4: The insertion algorithm

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

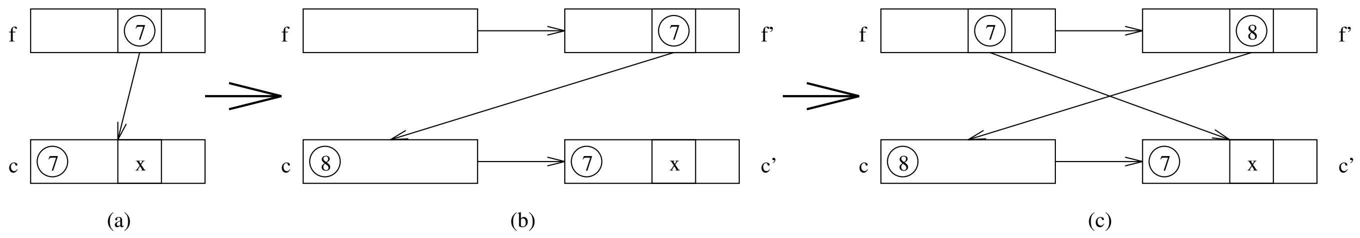

Figure 5: An incorrect structure modification.

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 would 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.

3.4 Deletion

A key deletion from a leaf node is a combination of the search and insert algorithms. First, we have to find the leaf holding the key we are interested in. After the key is removed from the leaf, we have to propagate the changed bounding rectangle up the tree, in the same manner as *updateParent* does for an insertion. The locking behavior is the same as for the search and insert algorithms and therefore key deletion is also deadlock-free.

If we want to maintain a high storage utilization, we have to remove nodes as soon as they become empty, so that they can be reused for subsequent splits in the tree. Since we do not employ lock-coupling when descending the tree, we have to deal with the problem of invalid pointers when removing nodes. In order to let descending operations detect that they followed an invalid pointer and reached a deleted node, we introduce a tree-global generation counter, which is a logical timestamp similar to an LSN. Each node in the tree stores its generation value in its header. On node removal, the global counter is incremented and the new value assigned to the deleted node, so that removed nodes have chronologically increasing generation values. A descending operation can now detect an invalid pointer by remembering the value of the global generation counter when reading the pointer and comparing this value to the one stored in the node header when finally visiting the node. A higher counter in the node indicates that it must have been removed after the pointer was read. In this case, a descending operation has to be restarted from the lowest valid ancestor node. The same compensating measure is necessary when the operation follows an invalid rightlink. This is detected if the generation value of the right sibling is higher than the global counter at the time the pointer to the *original* child node was read (at the time the parent, not the original child, was examined). The R-link tree structure as postulated in section 3.1 has to be slightly relaxed to accomodate the case where a node pointer is invalid.

The removal of the node and its corresponding entry in the parent are carried out recursively, also removing the parent if it becomes empty and so forth. Note that it is impossible for us to cheaply access the left sibling of the deleted node in order to reset its rightlink. But because a rightlink in the R-link tree is only needed to compensate for splits as long as these are not fully reflected in the parent level, we need not fix the rightlink chain when removing a node. An operation that crosses an invalid rightlink detects this and restarts itself; all subsequent operations will see a parent that fully reflects the child level and will not have to traverse the rightlink anymore. We can therefore safely ignore the invalid rightlink and as a consequence do not have to access the left sibling at all. Contrast this with B-link trees, where the rightlink chain at the leaf level is also used by range scans and a node deletion would therefore have to reset the rightlinks of neighboring nodes.

During the entire node deletion process, we only have to keep two nodes locked at a time: the node to be deleted and its parent node.

4 Consistency

A common requirement for concurrent 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.

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

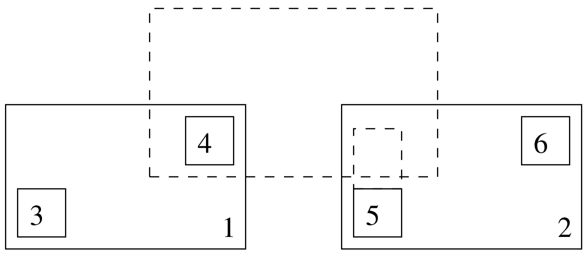

Figure 6: An example where 2-phase locking of leaves cannot guarantee RR.

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 still-active insertions and suspend itself if its query rectangle collides with any of the uncommitted new keys. A new insertion would in turn check the active scans and also suspend itself on a collision with a query rectangle. If a scan commits and leaves the system, 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. Key values and query rectangles can be organized into an in-memory spatial data structure to speed up checking for lock conflicts.

5 Recovery

The main ideas of the recovery method we present for R-link trees are drawn from [MoLe89] and [LoSa92]. Like these two papers, we also split an update operation into its contents-changing and its structure-modifying part. A contents change is an insert or delete of a key on a leaf; a structure modification can be a node split or deletion, an update of an index entry on an inner node or the insertion or deletion of an index entry on an inner node. By separating these two, the semantic effects of an index operation are attributed to the initiating transaction whereas the structure modification is handled independently of any transaction. This is necessary so that structure modifications can be made visible immediately after their completion and do not have to be locked until the initiating transaction commits. Like [MoLe89] and [LoSa92], we assume that write-ahead logging (WAL) is used for recovery purposes. In addition, we also assume that logical undo is supported. This is a necessity for the R-link tree, which is explained shortly.

When changing the contents of a leaf through an insert or delete of a single key, we log this operation within the context of the executing transaction, obeying the WAL protocol. This ensures that these leaf changes can always be undone if the transaction aborts and redone if it commits. Structural modifications, on the other hand, are treated as separate recoverable units ("atomic actions" in [LoSa92] and "nested top actions" in [MoLe89]). For example, a node split is an atomic action, which is caused by a transaction trying to insert a key on a full leaf node. The atomic action is logged and recovered separately from the surrounding transaction. Also, the atomic action is "committed" as soon as it finishes[4]and not rolled back even if the transaction that caused it fails.

The obvious advantage is that the effects of structure modifications can be made visible to other transactions although the initiating transation has not committed, without creating an abort dependency between the transactions. In other words, we do not have to keep the pages involved in a split locked until the end of transaction. A prerequisite of this strict separation is the ability to support logical undo as opposed to a purely page-oriented undo. As an example, consider the case where a transaction inserts a key on a page and the page is subsequently split, with the new key moved to the right sibling. If later on the transaction aborts, the inserted key cannot be found on its original leaf and the tree has to be traversed, undoing the insert with a delete operation. Logical undo is important in R-link trees for another reason, which the following example illustrates. When deleting a key, we also adjust the ancestors' bounding rectangles to reflect the missing key. Like all structure modifications, these updates are committed immediately. When rolling back the delete op-

[4]Note that the atomic action's log records need not be flushed when the atomic action finishes. It is sufficient that they be flushed with the log records of the next committing transaction, because no previously committed transaction could possibly have seen the effects of the atomic action.

eration, it is seldomly sufficient to do a page-oriented undo and just re-insert the key on the leaf page. We might also have to adjust the bounding rectangles and therefore perform a complete insertion operation.

Like [LoSa92], we break up entire structure modifications such as node split or delete propagation into several atomic actions that operate on at most two levels of the tree and are executed serially. These atomic actions are:

- splitting a leaf node
- changing an existing entry in an individual invocation of *updateParent*
- changing an existing entry and adding a new entry in an individual invocation of *extendParent* (this might include a split)
- incrementing the global generation counter by 1 and removing a node and its entry in the parent (the node is marked as "removed" by assigning it the new generation counter value)

The boundaries of atomic actions are chosen to coincide with the visibility boundaries induced by the locking protocol. Put differently, an atomic action "commits" before it releases the locks it holds. Only for node removal does an atomic action span more than a single level of the tree. The reason is that we might otherwise be left with an invalid pointer after having incremented the generation counter and marked the node as deleted. If the system crashed before removing the entry from the parent, a descending operation would later on not be able to detect that the pointer is invalid.

Since the structural modifications are carried out level by level, it is possible that they are interrupted by a system crash, possibly leaving behind index entries in inner nodes with "wide" bounding rectangles or nodes without a corresponding parent entry. This does not render the tree inconsistent, but it deteriorates the structure because subsequent search operations might be misled into unnecessarily following a pointer or forced to do rightlink traversals for lack of a parent entry. We can repair these structural deficiencies on the fly. Wide bounding rectangles are implicitly repaired by a subsequent insert or delete operation that adjusts bounding rectangles along the same path. When an ascending structure modification notices that a node on its path, which it reached by following a rightlink , does still not have a parent entry, it has detected an unfinished split. This has to be explicitly repaired by supplying the missing entry to the parent. When doing this, one has to be careful about obeying the required locking order and possibly release and reacquire some locks already held.

When combining this recovery mechanism with our proposed predicate locks, a rolling-back transaction is guaranteed never to be involved in a deadlock. This characteristic stems from a) the deadlock-free locking protocol inside the tree and b) the fact that predicate locks can be fully acquired before the tree operation is started and any node locks are acquired. The consequence of the latter is that there cannot be a deadlock between a predicate lock and a node lock. Because of this freedom from deadlock we can also use cheap latches [MoLe89] instead of locks when locking the tree nodes for physical consistency.

6 Performance Measurements

To asses the performance of R-link trees relative to non-link style R-tree concurrency mechanisms, we implemented R-link trees as a new access method for the Illustra database engine [Illu94] and compared it to the existing R-tree implementation. The performance numbers were obtained from concurrent client processes accessing a fully-functional database system, as opposed to simulation results or calls to a stand-alone access method implementation.

The existing R-tree implementation employs a variation of Bayer-Schkolnick-style lock-coupling for concurrent insertions. An insert operation obtains update locks on all nodes from the root to the leaf, holding on to them until the operation is finished. An update lock is incompatible with write locks and other update locks. This prevents two insert operations from getting a lock on the same node, thereby allowing a lock holder to escalate the update lock to a write lock without running the risk of a deadlock. This locking strategy allows multiple readers to be active in an index but serializes insertions, because the update lock on the root can only be help by one transaction at a time. Both R-link and R-trees employed the same page splitting strategy, a simplified version of Guttman's original quadratic split algorithm. We did not implement all of the recovery strategy outlined in section 5, because the no-overwrite storage manager [Ston87] of the Illustra database server does not undo the changes of aborted transactions. Also the deletion algorithm was not implemented as described in section 3.4, because index reorganization is performed off-line.

We investigated the throughput and response time of search and insert operations on the index. Analogous to [SrCa91], we defined three different workloads: the low-contention workload consists of a varying number of searchers and a single inserter, the moderate-contention workload of inserters only and the high-contention workload of an equal number of searchers and inserters. The effect of multiple users is obtained by concurrently running client processes which continuously send requests to a database server. The Illustra database engine has a process-per-user architecture, so that every process had its own server. Both insert and search operations were executed as SQL statements inserting or retrieving a single rectangle. The statements were executed within the context of a transaction, requiring the writing of a log record.

Prior to each run at a particular multiprogramming level,

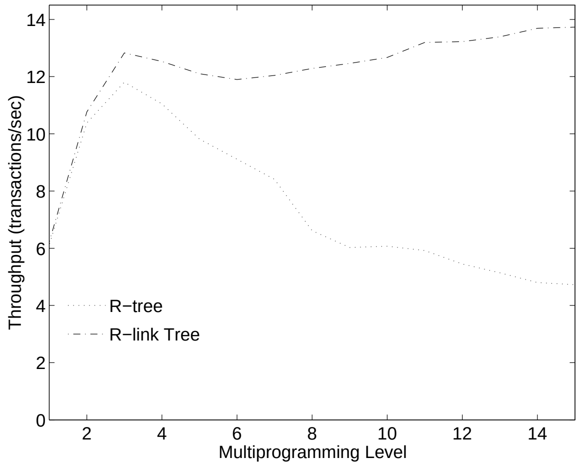

Figure 7: Total Throughput

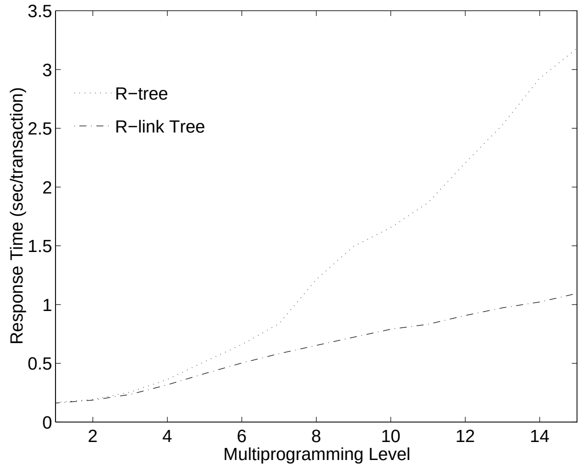

Figure 8: Response Time

the table was preloaded with 30,600 2-dimensional, non-overlapping rectangles, each measuring 10x10. Together, the rectangles imposed a grid pattern on the area 1700x1800. Rectangles inserted during the benchmark measure 8x8 and were placed within a randomly chosen rectangle from the pre-loaded data set. The intention was to avoid propagating bounding box changes up the tree, so that we did not have to investigate the effects of locking for consistency. The search transactions also returned a single random rectangle from the pre-loaded data set.

The benchmarks were run on a dual-processor SPARCstation 10 equipped with three DEC RZ28 disks attached to a single controller. Both the table and the index used for the benchmark were striped across all 3 disks in a simple round-robin fashion. In order to compensate for the relatively small data set we limited the number of page buffers to 64. The page size was 8K, which resulted in a page capacity of 70 tuples for the relation, 155 index entries for R-link trees and 181 index entries for R-trees.

The results we obtained for search transactions were essentially identical for both R-link and R-trees, under both the low-contention and the high-contention workload. This is not surprising, considering that neither R-link trees nor "Bayer-Schkolnick" R-trees lock out search operations when insertions are active in the tree. Also, the high-contention workload showed similar results for insert transactions as the medium-contention workload, only the numbers were scaled back due to the presence of resource-consuming searchers. For those reasons and space considerations, we decided to omit the results obtained from the low- and high-contention workloads altogether and focus on the all-inserters scenario.

The effects of the different locking strategies on throughput and response time can be seen in figure 7 and 8, respectively. Both R-link and R-trees reach their maximum throughput with three concurrent insertion processes. Whereas the R-link tree throughput is generally higher and stays about constant, the R-tree throughput deteriorates considerably after that point. A likely explanation is that the process holding the root lock gets descheduled, thereby adding to the wait time of all of the blocked processes. The system then thrashes trying to find another process to run. The same effects can be seen in figure 8, where the response time for R-trees increases at a much higher rate than for R-link trees.

7 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 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.

[Moha89, MoLe89] and [LoSa92] discuss recoverable access structures in a write-ahead logging environment; the former paper 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. 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 require logical undo, which in turn requires 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. The tree must be consistent whenever logical undo could be necessary; this is achieved by waiting for ongoing structure modifications to finish. To avoid the phantom problem when doing record-level locking, 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 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. The nodes of a Π-tree partition the key space, thus making it possible for descending operations to detect splits solely by key comparisons. Node consolidation requires lock-coupling during descent to avoid invalid pointers. As a consequence, lock-coupling cannot be employed during ascent without risking deadlocks. As mentioned in section 5, the Π-tree solution for recovery forms the basis of recovery in the R-link tree. The two approaches differ in how unfinished splits are detected and repaired. A descending operation in a Π-tree takes a rightlink traversal as an indication of a possibly unfinished split and then checks the parent for the entry suspected missing. In an R-link tree, an unfinished split is unambiguously detected during ascent.

8 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. To cope with invalid pointers resulting from node deletions, we keep track of when a node was deleted by assigning it a generation number. This allows us to do online node deletion without employing lock-coupling during tree descent. 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. For this to be possible, it is crucial that the recovery strategy support logical undo. A comparison with a lock-coupling concurrency mechanism for R-trees shows that R-link trees are capable of sustaining high throughput and low response times as the load increases.

An area for further research is the problem of enforcing degree 3 consistency of index scans. We outlined an inexpensive variant of predicate locking that we intend to investigate in more detail. The question remains whether popular techniques for B-trees such as key-range locking can be made to work for spatial data structures such as R-trees.

Acknowledgement

We would like to thank Paul Aoki, Paul Brown, Bob Devine, Shel Finkelstein, Joe Hellerstein, Mike Olson, Sunita Sarawagi and Mike Stonebraker for their comments on this paper. We would also like to acknowledge Megan Metters who helped to make this paper more readable. The people at Illustra were also very helpful. In particular, we have to thank Jeff Meredith, Wei Hong and Mike Ubell for their support. Dan Seguin deserves special thanks for help with hardware emergencies.

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.

[BKS94] D. Banks, M. Kornacker, M. Stonebraker, "High-Concurrency Locking in R-Trees," Sequoia 2000 Technical Report 94/56, June 1994.

[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.

[Illu94] Illustra Information Technologies, Inc., "Illustra User's Guide," Illustra Server Release 2.1, June 1994.

[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.

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

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

[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.

[Ston87] M. Stonebraker, "The Design of the Postgres Storage System," *Proc. 13th Int'l Conf. on Very Large Databases (VLDB),* Sep. 1987.