Concurrency and Recovery in Generalized Search Trees

Marcel Kornacker*
U. C. Berkeley
http://www.cs.berkeley.edu/~marcel
marcel@cs.berkeley.edu

C. Mohan
IBM Research Division
http://www.almaden.ibm.com/cs/people/mohan
mohan@almaden.ibm.com

Joseph M. Hellerstein
U. C. Berkeley
http://www.cs.berkeley.edu/~jmh
jmh@cs.berkeley.edu

Abstract

This paper presents general algorithms for concurrency control in tree-based access methods as well as a recovery protocol and a mechanism for ensuring repeatable read. The algorithms are developed in the context of the Generalized Search Tree (GiST) data structure, an index structure supporting an extensible set of queries and data types. Although developed in a GiST context, the algorithms are generally applicable to many tree-based access methods. The concurrency control protocol is based on an extension of the link technique originally developed for B-trees, and completely avoids holding node locks during I/Os. Repeatable read isolation is achieved with a novel combination of predicate locks and two-phase locking of data records. To our knowledge, this is the first time that isolation issues have been addressed outside the context of B-trees. A discussion of the fundamental structural differences between B-trees and more general tree structures like GiSTs explains why the algorithms developed here deviate from their B-tree counterparts. An implementation of GiSTs emulating B-trees in DB2/Common Server is underway.

1 Introduction

The increasing popularity of object-relational features within database systems—essentially, being able to store and access non-traditional datatypes such as images, audio and video, text, web pages, etc.—reflects the growing importance of having traditional database retrieval functionality for non-traditional applications. A key feature of these systems is fast, index-based access to the data as well as support for datatype-specific operations along with the standard features of multiuser access, transactional isolation and recoverability.

The research community has already developed a number of index structures for many kinds of non-traditional datatypes. Unfortunately, close to none of these are available in today's database systems. The reason is not that these new access methods have no performance benefits, but that access methods are hard to implement and integrate into DBMSs. In order for an access method to be useful in a DBMS, performance constraints often require it to handle concurrent access. Furthermore, the access method also should support the degrees of transactional isolation offered by the query language of the DBMS. Finally, the access method must fit in with the recovery mechanism that guarantees the integrity of the DBMS's data. Most research on novel access methods completely ignores these issues, and algorithms for their support are scarce. In addition, experience with B-tree implementations has shown that these features cause much of the complexity of actual implementations and account for a major fraction of the code. It appears, that the performance benefits of any particular access method other than B-trees do not outweigh the substantial implementation effort.

To solve the problem of making innovative access methods easier to integrate into DBMSs, [HNP95] developed a Generalized Search Tree (GiST), a "template" index structure supporting an extensible set of queries and datatypes. A GiST can be specialized to any particular tree-based access method by letting the implementor provide a small number of extension methods which customize the behavior of the tree with respect to the data type and query, thereby substantially reducing the amount of code required for a new access method. Examples of access methods, that can be reformulated as GiST specializations are: R-tree ([Gut84]) and its variants, TV-tree ([LJF94]), P-tree ([Jag90]) and Ch-tree ([KKAD89]). Although the GiST structure considerably reduces the effort involved in implementing a new access method from scratch, [HNP95] still does not address the areas of concurrent access, transactional isolation or recovery.

In this paper we describe general algorithms for concurrency control, recovery and repeatable read isolation that can be applied to a very broad class of access methods. These algorithms are presented in the context of the GiST, but can also be applied individually to any particular access method that complies with the GiST structure. This is not a very restrictive requirement, because it only excludes access methods that are either not proper trees (e.g., the hB-tree, as described in [LS90]) or that have other structural peculiarity (for instance, R^+-trees, described in [SRF87], replicate leaf entries). Although the GiST structure is similar to that of a B-tree, it generalizes the B-tree structure in a way which makes most of the extensively researched concurrency control techniques for B-trees inapplicable in the less restricted context of the GiST structure.

Our concurrency control protocol is based on an extension of the link technique initially developed for B-trees and completely avoids holding tree node locks during I/Os. Repeatable read isolation is achieved with an efficient hybrid locking mechanism, which combines traditional two-phase locking of data records with predicate locking. We show how the structure of an access method affects the available choices of concurrency control techniques and explain why existing B-tree techniques cannot be directly applied to more general tree structures. We also address practical issues

* Part of this work was done while Kornacker was working at the IBM Almaden Research Center. Kornacker and Hellerstein were supported by NASA grant FDNAGW-5198.

such as support for unique indices and savepoints. In conjunction with these algorithms, the GiST can be the basis for access method extensibility in a commercial DBMS, allowing the addition of new access methods by providing a few hundred lines of extension code without having to deal with recovery or concurrency.

The rest of this paper is organized as follows. Section 2 contains a brief description of the basic GiST structure and section 3 extends this structure for concurrent access. Section 4 outlines our design of the hybrid locking mechanism. After these preliminaries, the subsequent four sections explain the algorithms for index lookup, key insertion into non-unique and unique indices and key deletion. Logging and recovery are described in section 9; section 10 discusses a variety of implementation issues. Section 11 discusses some of the implications of the structure of an access method for concurrency control techniques and explains why most of the prior work on B-trees cannot be directly applied in the GiST context. Section 12 concludes this paper with a summary.

2 Basic GiST Structure

A GiST is a balanced tree which provides "template" algorithms for navigating the tree structure and modifying the tree structure through node splits and deletes. Like all other (secondary) index trees, the GiST stores *(key, RID)* pairs in the leaves; the RIDs (record identifiers) point to the corresponding records on the data pages. Internal nodes contain *(predicate, child page pointer)* pairs; the predicate evaluates to true for any of the keys contained in or reachable from the associated child page. This captures the essence of a tree-based index structure: a hierarchy of predicates, in which each predicate holds true for all keys stored under it in the hierarchy. A B-tree is a well known example with those properties: the entries in internal nodes represent ranges which bound values of keys in the leaves of the respective subtrees. Another example is the R-tree, which contains bounding rectangles as predicates in the internal nodes. The predicates in the internal nodes of a search tree will subsequently be referred to as bounding predicates (BPs).

Apart from these structural requirements, a GiST does not impose any restrictions on the key data stored within the tree or their organization within and across nodes. In particular, the key space need not be ordered, thereby allowing multidimensional data. Moreover, the nodes of a single level need not partition or even cover the entire key space, meaning that (a) overlapping BPs of entries at the same tree level are allowed and (b) the union of all BPs can have "holes" when compared to the entire key space. The leaves, however, partition the set of stored RIDs, so that exactly one GiST leaf entry points to a given data record.[1]

A GiST supports the standard index operations: SEARCH, which takes a predicate and returns all leaf entries satisfying that predicate; INSERT, which adds a *(key, RID)* pair to the tree; and DELETE, which removes such a pair from the tree. It implements these operations with the help of a set of extension methods supplied by the access method developer. The GiST can be specialized to one of a number of particular access methods by providing a set of extension methods specific to that access method; these extension methods encapsulate the exact behavior of the search operation as well as the organization of keys within the tree.

We now provide a sketch of the implementation of the operations and how they use the extension methods. For a more detailed description, together with examples of B-tree and R-tree extension methods, see the original paper ([HNP95]).

SEARCH In order to find all leaf entries satisfying the search predicate, we recursively descend *all* subtrees for which the parent entry's predicate is consistent with the search predicate (employing the user-supplied extension method *consistent()*).

INSERT Given a new *(key, RID)* pair, we must find a leaf to insert it on. Note that because GiSTs allow overlapping BPs, there may be more than a single leaf where the key could be inserted. A user-supplied extension method *penalty()* compares a key and predicate and computes a domain-specific penalty for inserting the key within the subtree whose bounds are given by the predicate. The penalty typically reflects how much the predicate has to be expanded to accommodate the new key. Using this extension method, we traverse a single path from root to leaf, following branches with the lowest insertion penalty.

Extension code also manages the organization of keys within the tree. If the leaf overflows and must be split, a extension method *pickSplit()* is invoked to determine how to distribute the keys between two leaves. If, as a result, the parent also overflows, the splitting is carried out recursively, from bottom to top.

If the leaf's ancestors' predicates do not include the new key, they must be expanded, so that the path from the root to the leaf reflects the new key. The update is done with a extension method *union()*, which takes two predicates, one of which is the new key, and returns their union. Like node splitting, expansion of predicates in parent entries is carried out bottom-up until we find an ancestor node whose predicate does not require expansion.

DELETE In order to find the leaf which holds the key we want to delete, we again traverse multiple subtrees as in SEARCH. Once the leaf is located and the key is found on it, we remove the *(key, RID)* pair and, if possible, shrink the ancestors' BPs.

To avoid having to check all entries in a node sequentially when traversing it, either for search, insert or delete operations, a GiST implementation should provide an interface to specialize the intranode layout of entries. This way, the nodes can store and compress predicates in any way that is efficient for the particular key domain and set of supported queries. For example, a GiST implementation of a B-tree would use an ordered sequence in order to be able to do binary search.

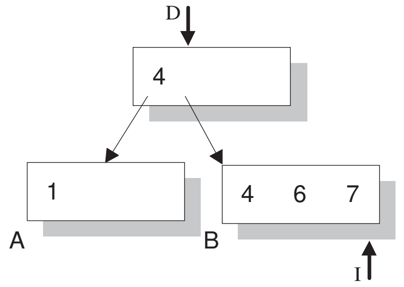
3 GiST Extension for Concurrency

When multiple operations are carried out on a GiST in parallel, their interactions may be interleaved in a way that leads to incorrect results. An example for a B-tree is shown in Figure 1, where a node split changes the location of some keys, which causes a concurrently executing search operation to miss them.

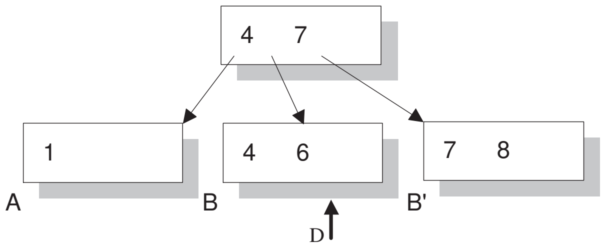
In order to avoid such situations, we apply the B-link tree strategy ([LY81]) of adding a link between a node and its split-off right sibling. All the nodes at each level are chained together via links to their right siblings; the addition of this *rightlink* allows operations traversing this node to compensate for missed splits by following the rightlink. Of course, rightlinks cannot be followed blindly every time a node is traversed, or parts of the tree would be scanned multiple times. For the link strategy to work, a traversing operation must be able to (1) detect a node split and (2) determine when to stop following rightlinks (a node can have split multiple times, in which case the traversing operation must follow as many rightlinks as there were node splits).

For B-trees, both of these questions can be answered by examining the keys in the node, since the key domain is ordered and the keys are partitioned across the leaves. In particular, a comparison of the search key and the highest key on the node will tell a traversing

[1] This structural requirement excludes R^+-trees ([SRF87]) from conforming to the GiST structure.

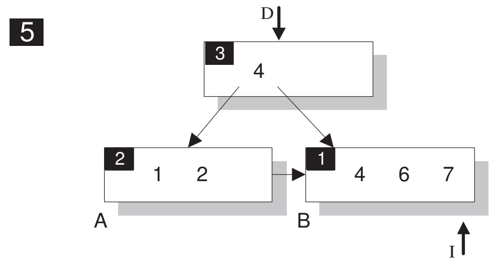


An insert operation of key 8 and a delete operation of key 7 are executing concurrently in the tree. The insert operation reaches leaf B, which is full and must be split. The delete operation examines the parent and discovers that it must also examine leaf B (arrows indicate scan positions).

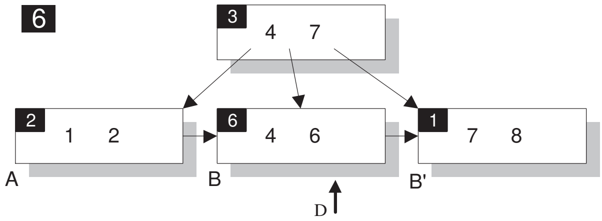


The delete operation reaches leaf B after it was split and incorrectly concludes that key 7 does not exist.

Figure 1: Incorrect Interleaving of Key Search and Node Split



The example B-tree GiST of Figure 1 has been extended with NSNs and rightlinks. Again, an insertion of key 8 and a deletion of key 7 are executing concurrently. The insertion is about to split leaf B. The delete operation records the value of the global counter (5) before it goes to B.



The insertion splits B, increments the global counter value from 5 to 6 and assigns the new value to B. The new sibling receives the old NSN and rightlink of B. When the delete operation reaches leaf B, it determines that B must have split, because the recorded global values (then 5) is less than the current NSN (6) of B. Likewise, when it follows the rightlink to B', it can determine that this is the end of the "split chain" it has to traverse, because the NSN of B' (1) is less than or equal to the recorded global counter values (5).

Figure 2: Example of Extended Tree Structure

operation if a node has split and if a right sibling might contain entries intersecting the search range. GiSTs do not impose these restrictions on the key domain, which means that the B-link strategy by itself is insufficient. [KB95] adapts the link strategy to R-trees by assigning sequence numbers to the nodes and incrementing these during node splits. The sequence number is also recorded in the parent entry, which allows traversing operations to reconstruct the "lineage" of a node after it was split. In contrast to B-link trees, this strategy does not rely on the semantics of the keys in a node but extends the tree structure in order to make node splits visible. The drawback of this design is that it requires a sequence number in each entry of an internal node, turning the *(predicate, pointer)* pair into a *(predicate, pointer, sequence number)* triple and thus reducing node fanout.

The concurrency protocol for GiSTs also extends every node with a node sequence number (NSN) and a rightlink and uses these to detect splits; it improves on the R-link tree design by eliminating the space overhead in internal index entries. The NSN is taken from a tree-global, monotonically increasing counter variable.[2] During a node split, this counter is incremented and its new value assigned to the original node; the new sibling node receives the original node's prior NSN and rightlink. Figure 2 illustrates the extended tree structure and shows how a traversing operation can take advantage of the NSNs and the counter variable to detect splits and determine how many times a node was split. In general, a traversing operation can now detect a split by memorizing the global counter value when reading the parent entry and comparing it with the NSN of the current node. If the latter is higher, the node must have been split and the operation follows rightlinks until it sees a node with an NSN less than or equal to the one originally memorized. A node with an NSN less than or equal to the memorized one cannot have been split after the parent has been visited and therefore its rightlink need not be traversed; since nodes are always split to the right, this node must demarcate the end of the rightlink sequence that the traversing operation has to follow.

4 Repeatable Read Consistency

The highest degree of transactional isolation is defined as Degree 3 consistency or repeatable read isolation ([Gra78]). It implies that if a search operation is run twice within the same transaction it must return the exact same result (even if that result set is empty). This section presents an overview of the mechanism that allows repeatable read isolation for search operations in GiSTs; it is a synthesis of two-phase data record locking and predicate locking.

The simplest solution would be to lock all involved tables for the duration of the entire transaction. Unfortunately, this leads to an unacceptably low degree of concurrency. DBMSs try to avoid this by accessing the tables through index structures and explicitly locking only as much as needed to guarantee repeatable read. One part of what is locked is the set of data records returned by the search; this will prevent modification or deletion of the data records in the result set. This is not sufficient, because it is also necessary to protect those regions of the search range for which no data records were returned; this will prevent subsequent insertions into the search range. These insertions are known as phantom insertions ([EGLT76]) and can result either from new data records or rolling-back deletions of data records.

[2] Management of the global counter variable is discussed in section 10.

4.1 A B-Tree Solution: Key-Range Locking

One solution to the phantom problem in an ordered key domain is a technique called key-range locking,[3] which works as follows. Each data item with key k_i is treated as a surrogate lock name for the key range $(k_{i-1}; k_i]$. For a scan with a given search range, we retrieve and two-phase lock all the data records[4] within the range and we also lock the first data record *past the right end of the range*; this is typically done with the help of a B-tree index. As a result, all the key ranges $(k_{i-1}; k_i]$ intersecting the search range are locked. Before a leaf entry insertion, we check the data record to the right of the insertion point for existing locks, thereby making sure that the "gap" we are inserting into is not locked by any scan.

Essentially, key-range locking requires the *ordering property of the key domain* and the *correspondence between logical key order and physical order of (key, RID) pairs* within and across nodes. This allows conflicting search and insert operations to agree on a sequence of physical data records as a surrogate for a logical search range. This strategy is very efficient for two reasons: (1) the surrogate lock names are easy to compute (they are either the returned data records or the data record right next to the last one in the sequence); (2) a logical lock range has been transfered into a sequence of purely physical locks, which can be set and checked very efficiently. [5]

4.2 Predicate Locking

Key-range locking is not applicable to GiSTs, because we cannot assume an ordered key domain and therefore cannot count on finding a physically contiguous sequence of leaf entries as a surrogate for the possibly multi-dimensional search range. An alternative technique, predicate locking ([EGLT76]), circumvents this problem. Instead of two-phase locking of data records, the search operations register their search predicates in a tree-global table, so that insert and delete operations can check for conflicting concurrent search operations. Symmetrically, insert and delete operations register their keys as predicates in a tree-global table, which is checked by search operations for conflicts with the search predicate. An operation can only set its own predicate lock and start traversing the tree after it verified that there are no conflicting predicates. This single predicate lock is sufficient for a search operation to protect its entire search range, including the existing data records, so that no locks have to be placed on data records. Compared to individual locks on data records, predicate locking has two disadvantages:

- Predicate locks are less efficient to set and check. Because the GiST has no information about the nature of the predicates, it cannot arrange them in an in-memory search data structure for efficient lookup such as a hash table when checking for conflicts. Instead, every check must go through the entire tree-global list of existing predicates, which can be very time-consuming.

- A search operation must set its predicate lock before the index is accessed and any data records are retrieved. Unlike key-range locking, the locked key range is not expanded gradually, which can be very detrimental to concurrency, if the search is done as part of an interactive cursor.[6]

[3]Other terms are key-value locking ([Moh90a]), or next-key locking ([GR93]).

[4]As in the data-only locking approach of ARIES/IM ([ML92]).

[5]A drawback is that the next key lock might cover a substantially wider range than is necessary, depending on the key value. See [Moh90b], [Lom93] and [Moh95] for some techniques to address this problem.

[6]The key range can be expanded gradually, but this would become relatively costly. For gradual predicate expansion, the insert and delete predicates would have to be re-checked every time the search predicate was expanded.

4.3 A Hybrid Mechanism

Instead of resorting to pure predicate locking, we use an efficient hybrid mechanism, which synthesizes two-phase locking of data records with predicate locking. We describe its general features here; the details are presented in the following three sections. The underlying idea is to use two-phase locking for existing data records and augment that technique with a restricted, more efficient version of predicate locking for phantom avoidance. In the hybrid mechanism, data records that are scanned, inserted or deleted are still protected by the two-phase locking protocol. In addition, search operations set predicate locks to prevent phantom insertions. Furthermore, these predicate locks are not registered in a tree-global list before the search operation starts traversing the tree; instead, they are *directly attached to the nodes*. Predicate attachments are performed so that the following invariant is true at all times:[7] if a search operation's predicate is consistent with a node's BP, the predicate must be attached to the node. An insert operation can therefore limit itself to checking only the predicates attached to its target leaf. A delete operation must be carried out as a *logical* delete, meaning that the respective leaf entry is not physically deleted but is only marked as deleted ([Moh90b]). The physical presence of the leaf entry and the lock on the corresponding data record ensure that search operations will block on the deleted entry until it is committed.

To ensure that search predicates are attached to the nodes they apply to, search operations attach their predicates to the nodes as they are visited. The predicates and their node attachments are only removed when the owner transaction terminates. Since the tree structure changes dynamically as nodes split and BPs are expanded during key insertions, the predicate attachments have to adapt to the structural changes. We distinguish two cases that require existing predicate attachments to be replicated at other nodes. The first case is a node split, which creates a new node whose BP might be consistent with some of the predicates attached to the original node. The invariant is maintained by attaching those predicates to the new node. The second case involves the expansion of a node's BP, causing it to become consistent with additional search predicates. Again, those predicates have to be attached to the node; they are found in the ancestor nodes accessed during the BP update phase of the insertion operation, and they are "percolated" to all the child nodes whose expanded BP is consistent with the predicate.

The advantage of this hybrid approach over pure predicate locking is that two-phase locking of data records is still used as much as possible, eliminating the need for search and delete operations to check for conflicting predicates. Only insert operations check for search predicates; furthermore, they only check the set of predicates attached to their target leaves, which are typically substantially smaller than the tree-global set of predicates.

A drawback that the hybrid mechanism retains from predicate locking is that the lock range is not expanded gradually. The reason is that predicates have to be attached to the visited nodes top-down, starting with the root. This can block an insertion into the search range, even if the leaf where the insertion takes place has not been visited by the search operation. Even in this respect, the hybrid mechanism is an improvement over pure predicate locking, because the insertion will only be blocked if it requires BP updates in ancestor nodes where the search predicate is already attached.

We proceed to present the details of the search, insert and delete algorithms.

5 Key Search

The search operation returns the set of leaf entries satisfying the search predicate; it does this by traversing nodes whose bounding

[7]Management of predicates and their attachments is discussed in Section 10.3.

```
search(search-pred)
  nsn = global NSN;
  push(stack, [root, nsn]);
  while (stack is not empty)
    [c, c-NSN] = pop(stack);
    latch(c, S-mode);
    if (c-NSN < NSN(c))
      push(stack, [rightlink(c), c-NSN]);
    end
    if (c is leaf)
      for each leaf entry [key, RID]:
        if (consistent(key, search-pred))
          lock(RID, S-mode);
          add entry to search result set;
        end
    else
      nsn = global NSN;
      for each node entry [pred, ptr]:
        if (consistent(pred, search-pred))
          push(stack, [ptr, nsn]);
        end
    end
    attach search predicate to c;
    unlatch(c);
  end
```

Figure 3: The Search Algorithm

predicates are consistent with the search predicate. An internal node can contain many consistent entries, so a stack is used during traversal to memorize the nodes with consistent BPs that have yet to be visited. The search operation starts by pushing the root pointer on the stack. It then repeatedly pops an entry off the stack, visits the corresponding node and pushes all entries with consistent predicates back on the stack. This results in a depth-first traversal of the tree and is repeated until the stack is empty.

While examining a node for consistent entries, we hold it latched[8] to prevent concurrent modifications. The latch is released as soon as we are done with the node and before going to the next node, avoiding latch-coupling across I/Os.

To recognize node splits, we timestamp every page pointer stored on the stack with the value of the global counter as of the time the page pointer was read. When the stack entry is used to visit a node, the recorded counter value is compared with the node's NSN. If the latter is higher, the node has been split, and we also push the rightlink pointer of the node together with the originally recorded counter value on the stack. This guarantees that the right siblings split off the original node will also be examined later on.

While traversing the tree, the search operation also attaches its predicate in a top-down fashion to every node that it visits. In addition, the RIDs of retrieved leaf entries are also locked as part of the hybrid locking protocol. The function `search()`, shown in Figure 3, implements the search operation as described in this section. It returns the set of leaf entries consistent with the search predicate. To simplify the presentation, the latches are not released when blocking on the lock on a data record lock, creating an opportunity for a deadlock between the latch-holder and the owner of the conflicting lock (either an inserting or deleting transaction). To deal with this situation, the latches have to be released at first and then reacquired when the search operation is unblocked. Since the latched leaf can be split in the meantime, we might have to traverse rightlinks, guided by the node's original NSN.[9]

6 Key Insertion

Key insertion is carried out in several phases:

1. the new data record (stored elsewhere in the database) is X-locked *before* the tree insertion is initiated;

2. the insertion operation begins by traversing the tree along a single path from the root to a leaf, following branches with the lowest insert penalty;

3. if the chosen leaf would overflow as a result of the key insertion, it has to be split beforehand, which in turn might cause recursive splitting of ancestor nodes; during recursive splitting, we attach to every new node all of the original node's predicates that are consistent with the new node's BP;

4. if the insertion of the new leaf entry will change the leaf's BP, the new BP is propagated to the parent entries by backing up the tree, until an ancestor node is encountered whose BP does not need to expand; during this phase, we also "percolate" search predicates from ancestor to child nodes, if the ancestors' predicates are consistent with the child's updated BP;

5. the new *(key, RID)* pair is inserted on the leaf;

6. we then check the list of predicates attached to the leaf and block on the conflicting ones until their owner transactions commit.

Note that in contrast to B-trees, an insertion operation may back up the tree for two reasons: splitting a node requires the installation of a new parent entry and expanding a leaf's BP requires the adjustment of parent entries. The latter step is missing in B-trees.

When traversing the tree to find an appropriate leaf, we do not employ lock coupling, and compensate for missed splits by following rightlinks, in the same manner as a search operation. That is, we memorize the global counter value when reading the parent entry and pointer and compare it to the NSN found in the node. If the memorized counter value is higher, we must examine some number of right siblings in addition to the current node. Eventually, after a sequence of down- and rightlink traversals, we arrive at the target leaf.

When backing up the tree to update parent entries, either because of a split or to expand their predicates, it can also become necessary to recognize splits of ancestor nodes that have taken place since the nodes were initially traversed on the way down to the leaf. In this instance, it is not even necessary to compare NSNs. If a parent node does not contain the child's pointer anymore, it must have been split and the search for the child's pointer is continued in the right sibling.

To make the new entry visible in the tree, we expand the leaf's BP to include the new key and propagate this up the tree. The ancestor nodes involved are latched by backing up the tree to the lowest ancestor node n (or the root) whose BP does not need to be expanded. Unlike recursive splitting, BP update propagation is then performed top-down starting at node n; this facilitates logging and is discussed in greater detail in Section 9. While updating a parent

[8] Latches differ from locks in two aspects: (1) latches, like mutexes, are addressed physically and can therefore be set and checked much more efficiently than locks, which are usually organized into a hash table; (2) existing latches are not checked for deadlock by the DBMS, which requires the latch holders to make sure their usage pattern is deadlock-free. Latches are commonly used to synchronize access to physical "objects" of the DBMS such as buffer pool frames. Also, latches do not interact with locks, so that it is possible to latch the buffer pool frame holding a particular node while some other transaction holds a lock on the node. See [MHL+92] for more details.

[9] When revisiting a leaf, we have to make sure that leaf entries are not included in the result set multiple times, even if the leaf has been modified or split in the meantime. We can keep track of which entries of a particular leaf have already been processed by including their *data* RIDs (not the entries' RIDs themselves) in a list. Note that this is only possible with leaf entries, not with entries of internal nodes; the discussion in Section 7.2 will make this clear.

```
insert([new key, RID])
  leaf = locateLeaf(&stack, new key);
  if (not enough space on leaf)
    splitNode(leaf, stack);
    release all latches at ancestor levels;
  end
  updateBP(leaf, union(BP(leaf), new key), stack);
  insert [new key, RID] on leaf;
  if (consistent(predicates on leaf, key of new item))
    block until predicate-owning transactions
      terminate;
  end
  unlatch(leaf);

locateLeaf(stack, new key)
  p = root;
  p-NSN = global NSN;
  loop
    if (p is leaf)
      latchmode = X-mode;
    else
      latchmode = S-mode;
    end
    latch(p, latchmode);
    if (p-NSN < NSN(p))
      p = node with smallest insert penalty in
        rightlink chain delimited by p-NSN;
    end
    if (p is not leaf)
      push(stack, [p, NSN(p)]);
      find entry [pred, child-ptr] on p with smallest
        insert penalty for new key;
      p-NSN = global NSN;
      unlatch(p);
      p = child-ptr;
    else
      return p;
    end
  end
```

```
splitNode(p, stack)
  latch(parent(p, stack), X-mode);
  if (NSN(parent) changed since first visited)
    parent = node in rightlink chain starting with
        parent, holding entry for p;
    latch correct parent;
  end
  create new node p';
  latch(p', X-mode);
  pickSplit(p, p');
  NSN(p') = NSN(p);
  global NSN = global NSN + 1;
  NSN(p) = global NSN;
  if (not enough space on parent)
    splitNode(parent, stack);
  end
  insert entry for p' on parent;
  update pred for p on parent;
  for each predicate attached to p:
    if (consistent(pred, BP(p'))
      attach pred to p';
    end

updateBP(p, union-BP, stack)
  if (union-BP != BP(p))
    latch(parent(p, stack), X-mode);
    if (NSN(parent) changed since first visited)
      parent = node in rightlink chain starting with
          parent, holding entry for p;
      latch correct parent;
    end
    updateBP(parent, union(BP(parent), union-BP),
        stack);
    for each predicate attached to parent:
      if (consistent(pred, union-BP) and
        not(consistent(pred, BP(p)))
        replicate pred on p;
      end
    end
    BP(p) = union-BP;
    update pred in parent entry of p with union-BP;
    unlatch(parent);
  end
```

Figure 4: The Insert Algorithm

entry and the corresponding child's BP, we also add to the child node the parent's predicates that are consistent with the child's new BP.

After inserting the new key on the leaf, we check the leaf's list of search predicates for conflicts. If there are any, the insert operation must be suspended until the conflicting search transactions are terminated. When resuming the insertion, the leaf originally located might have been split. This again is recognized by comparing the memorized NSN with the leaf's current NSN, followed by zero or more rightlink traversals.

The implementation of the insertion algorithm is shown in Figure 4. Note that the function `consistent()`, which is used to detect conflicting predicates, is the same user-supplied function that is also used by the search operation to navigate within the tree. The function `pickSplit()` was introduced in Section 2 as an extension method that determines the distribution of keys between two leaves. For brevity, we omit a detailed discussion of root splits.

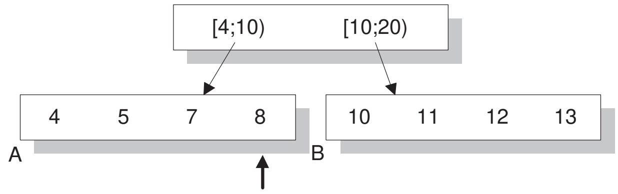
7 Key Deletion

In order to delete a key from a tree, it is first necessary to locate the key on a leaf, which is equivalent to a search operation with an equality predicate. The item on the leaf will not be physically deleted but only marked deleted, i.e., a logical deletion will be performed. The physical presence of this deleted key is necessary for compliance with the two-phase locking protocol; it ensures, that Degree 3 isolated search operations have an opportunity to be suspended when they encounter such a key.[10] For the same reason, the delete operation must not shrink parent entries after the key has been marked, because this would remove the path to the key and make the key inaccessible for concurrent search operations.

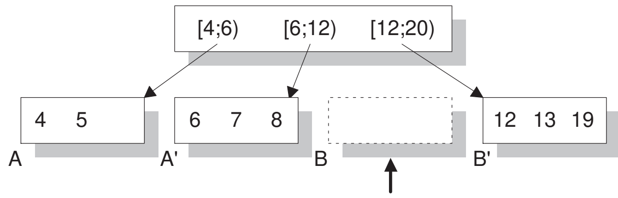
7.1 Garbage Collection of Deleted Leaf Entries

The key may only be physically removed and the parent entries shrunk after the deleting transaction has been committed. The physical update operations are then performed as garbage collection by other operations which happen to pass through the affected nodes. A node reorganization removes all those entries from a leaf which have been marked deleted and for which the initiating transactions

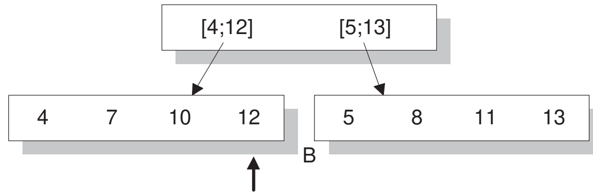
[10] In B-trees, the delete operation could physically remove the key, but would have to leave a lock on the next key. This is not applicable to GiSTs for the reasons already mentioned in section 4. Even in B-trees, logical deletion is preferable if increased concurrency is important (see [Moh90b] for further discussions of logical deletions). In fact, the new index manager of DB2/MVS V4 has adopted it for that reason.



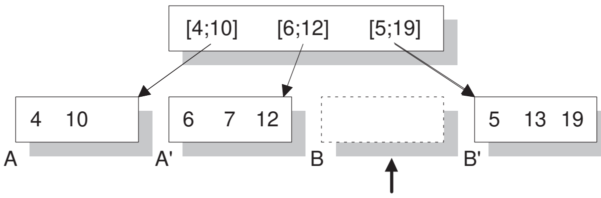
A regular B-tree, where entries in internal nodes are shown as intervals for clarity, although they would only be stored with their left or right boundary. A scan of the key range $[5; 20]$ starts in leaf A.



After the scan is done with leaf A, leaf A and B are split as a result of inserting keys 6 and 19; additionally, keys 10 and 11 are deleted and as a result, leaf B is also deleted. The scan notices this and repositions itself in the parent node, remembering that it already covered range $[5; 10)$ (by having scanned former leaf A). The next leaves to visit are A' and B'. Leaf A' does not contain any items contained in the remaining scan range ($[10; 12]$), and the scan moves on to leaf B'.



A fictitious non-partitioning B-tree, where the key space is not partitioned across the leaves and as a result internal entries have overlapping intervals. A scan of the key range $[5; 20]$ again starts in leaf A.



After the scan is done with leaf A, leaves A and B are split as a result of inserting keys 6 and 19. Keys 8 and 11 are deleted, resulting in the removal of leaf B. The scan notices this, and, in order to continue correctly, would have to go to leaf B' and ignore A'. Notice that it is impossible to determine this by repositioning in the parent node: (1) although the scan has already covered former leaf A with the interval $[4; 12]$, it would be incorrect to go to leaf B' and ignore entry 5; (2) for that same reason, if the scan would go to leaf A', it would incorrectly repeat scanning of entries 7 and 12.

Figure 5: Why Repositioning Requires Partitioning

have committed.[11] As a result of a node reorganization, the BP of that node may have shrunk, which can then be propagated to the parent nodes.

7.2 Node Deletion

A node reorganization may also leave a node completely empty, in which case it should be removed from the tree so that it can be reused for later node splits. Unfortunately, it becomes impossible to simply remove the node's parent entry and retire the node, because ongoing tree operations might still have pointers to the node on their stacks. There are two alternatives to deal with this problem: either a tree operation visiting a reallocated node detects this and recovers from it by repositioning itself within the tree (as in ARIES/IM [ML92]); or we avoid incorrect pointers altogether by delaying a node deletion until there can be no more active tree operations with pointers to it (introduced in [KL80] as the "drain technique").

The first alternative, recovery from incorrect pointers, is impossible for a variety of reasons. If the node has been split before it is deleted, a tree operation visiting it after the deletion may still need to traverse its rightlink. This is impossible, because the node might have been reused, in which case the original node content, including the rightlink and NSN, would be lost. Repositioning within the tree by revisiting the parent is also not possible, because the parent itself might have been split or simply changed (see Figure 5 for an example). When that happens, it is impossible to determine which nodes have been split off the deleted one.

It is clear for these reasons that a node cannot be deleted while active tree operations still hold direct or indirect pointers to it. A direct pointer is a node pointer recorded in some operation's stack. A node pointer indirectly references some of that node's right siblings, if the stack entry also contains an NSN which would lead the operation to cross rightlinks to them. To make node deletion operations aware of existing direct pointers, an operation must place a *signaling* lock on a node if it pushes a pointer to that node onto the stack. The signaling lock can be implemented as a regular S-mode lock on the node, so that other insert and delete operations are not prevented from physically accessing and modifying that node (a lock on a node, unlike a latch, does not restrict physical access to the buffer pool frame holding that node). A node deletion checks for signaling locks by trying to acquire an X-mode lock on the respective node. To extend the deletion protection to those nodes which have been split off and are therefore referenced indirectly, a node-splitting transaction copies the list of signaling locks placed on the original node to the new right sibling. A signaling lock is released as soon as the operation that set it visits that node. The only exception to that rule is the signaling lock set on the target leaf of an insert or delete operation. The lock has to be retained until the end of the inserting or deleting transaction for recovery purposes, otherwise recovery-relevant parts of the link chain would be interrupted (details are given in Section 9).

8 Key Insertion into Unique Indices

When inserting a duplicate into a unique index, database semantics require that the insertion operation return with an error message. Moreover, this error message must be reproducible if the inserting transaction runs with repeatable read isolation. To insert into a unique index, we therefore combine the steps of a search operation with those of an insert operation. The search operation preceding the insertion verifies that no duplicates will be introduced. If it finds

[11] [Moh90b] shows how this can be done cheaply in a WAL environment. Essentially, if the page's LSN is less than the LSN of the oldest active transaction, then all entries must belong to committed transactions and no additional locks have to be tested.

the new value in the tree, it returns an error condition. To make this error repeatable, it adheres to the two-phase locking protocol and requests an S-mode lock on the corresponding data record. Note that in this case no predicate locks need to be left behind by the search operation; the lock on the data record alone is sufficient to ensure repeatability of the error condition.

If the inserted value is not found in the tree, the phases of the regular insert operation are carried out: the leaf is located and after key insertion the parents' entries are updated, if necessary. To avoid a race between two insertion operations of the same value, the search phase of the insertion operation leaves behind predicates of the form "$= key$" on every visited node. If two insert operations happen to be interleaved in a way that they both miss the other operation's new key during their search phases, the predicates will ensure that each operation blocks on the other's predicate. This will result in a deadlock, which can be resolved in a standard manner by the lock manager. Once the insert operation is finished, the predicates left behind from the search phase can be released.

9 Logging and Recovery

The goal of recovery is twofold. First, it ensures that the tree only reflects insertions and deletions of committed transactions and none of those of uncommitted transactions. Second, the tree structure must be brought back into a consistent state after a system crash or failure of a single update operation. As an example of an inconsistent state, consider a node split that was interrupted by a system crash before a parent entry could be installed for the new child. Since the global counter value has been incremented, a subsequent traversal operation will not recognize the "missed split" and the tree structure will remain corrupted. The following GiST recovery protocol is targeted at a write-ahead logging environment with page-oriented redo and logical undo (for example, as in ARIES [MHL+92]).

9.1 Structure Modifications

In order to obtain high concurrency, a logging protocol must separate update operations into their content-changing (key insertion and logical deletion at the leaf level) and structure-modifying parts (node splits, parent entry updates, node deletions). This allows us to ascribe the content-changing operations to the initiating transactions, whereas structure modifications are carried out separately from any transactions as individually committed atomic units of work.[12] The advantage of this approach is that a structure modification can be "committed" as soon as it finishes and the latches on the involved nodes can be released immediately (within an atomic unit of work, we employ two-phase latching: once acquired, a latch is only released when the atomic unit of work finishes). As an example of what would happen without atomic units of work, consider how a parent entry update would be carried out if it were to execute as part of the same transaction that initiated the preceding key insertion. In this case, the updated parent entry would have to be locked until the end of the insert transaction, otherwise another update to the same parent entry by a different transaction would be incorrectly rolled back if the insert transaction were aborted. Locking the parent entry, however, would serialize all key insertions that happen to propagate through the same parent entry.

Apart from key insertion and logical deletion on leaf nodes, sections 6 and 7 used as the building blocks of insert and delete operations the following atomically executed structure modifications: (1) node split, which is carried out recursively and includes splitting of all necessary ancestor nodes and installation of the parent entries for the new nodes; (2) parent entry update on a single ancestor node; (3) deletion of a node, including the deletion of the parent entry; (4) garbage collection of a node. The types of log records as well as the corresponding redo and undo actions are shown in Table 1. Note that no additional user-supplied extension code is required to write the log records, so that logging can be handled independently by the core DBMS component.

[12] These atomic units of work have also been called "atomic actions" ([LS92]) or "nested top actions" ([MHL+92]) in the research literature. A technique for executing a sequence of page updates as individually committed atomic units of work is described in [MHL+92]. Essentially, the log records written for these page updates are separated from those of the surrounding transaction by appropriate setting of the backchain pointers in the log record headers.

9.2 Undo Recovery

All of the structure modification atomic actions are undone in a page-oriented fashion, i.e., they only involve visiting and modifying those pages recorded in the log record. On the other hand, key insertion and logical deletion at the leaf level have to be undone logically, i.e., by revisiting the leaf holding the key, which may require rightlink traversal. The reason why in the latter case rightlink traversal may be necessary is that between the time the index operation was performed and the time the transaction is aborted, the tree structure could have changed (for the same reason, logical undo is also employed in ARIES/IM [ML92] and ARIES/KVL [Moh90a]). If a split takes place in the meantime, the relevant entries may be moved rightward onto other leaves.

When performing undo recovery after a system crash, all node latches acquired prior to the crash are lost. Since there might still be unfinished structure modifications in the tree, which are unprotected as a result of the crash, the undo recovery phase must not execute any structure modifications as part of a logical undo of a leaf entry insertion or deletion. The GiST logging protocol fulfills this requirement, because (a) undoing a leaf entry deletion only involves unmarking the entry, not updating the parent entries; (b) undoing a leaf entry insertion only involves physical deletion of the entry; even if the node becomes empty, no node deletion is performed.

10 Implementation Issues

The previous sections describe the algorithms for concurrent operations in a GiST, but omit the implementation of some important details. This section discusses some of the issues that arise during the implementation of a high-concurrency GiST, namely: sequence number generation, savepoints and partial rollback as well as predicate management.

10.1 Node Sequence Numbers

A key ingredient of the concurrency protocol for GiSTs is the tree-global counter used to generate node sequence numbers. This counter is incremented during splits and has to be made recoverable in order for split detection to work after a crash. In a write-ahead logging environment, this can be achieved easily without having to write additional log records, using the existing infrastructure.

First, instead of maintaining a separate counter for each index in a database, it is possible to use a single database-wide counter. WAL-based recovery systems often have log sequence numbers (LSNs) associated with their log records, which are reflected in the page they were generated for and facilitate restart (for a description of logging and restart in WAL environments, see [MHL+92] and [GR93]). These LSNs are guaranteed to be monotonically increasing, which makes the LSN of the last log record written an ideal candidate for the global counter value. During a split atomic action, a log record has to be written for the splitting of the original node, which implicitly increments the global counter. Furthermore, this

Log Record	Fields	Redo Action	Undo Action
Parent-Entry-Update	new BP, child page ID, parent page ID	update BP in child and corresponding slot in parent	none (redo-only log record)
Split[1]	original page ID, new page ID, list of keys for new page	orig. page: delete keys in logrec, recompute and reset BP new page: insert keys in log record, recompute and reset BP	orig. page: insert keys in log record, recompute and reset BP new page: no action necessary
Garbage-Collection	page ID, RID list of garbage-collected entries	remove entries from leaf	none (redo-only log record)
Internal-Entry-Add[1]	page ID, new key	insert entry on page	remove entry from page
Internal-Entry-Update[1]	page ID, new BP, old BP	set BP in entry to new BP	set BP in entry to old BP
Internal-Entry-Delete[2]	page ID, entry	remove entry from page	insert entry on page
Add-Leaf-Entry	page ID, new entry	add entry to page	Logical Undo: locate leaf and remove the entry; do immediate garbage collection (shrink BP of page and update ancestor BPs) if not in restart recovery
Mark-Leaf-Entry	page ID, old entry	mark entry on page as "deleted"	Logical Undo: locate leaf and unmark entry
Get-Page[1]	page ID	mark page as "unavailable"	mark page as "available"
Free-Page[2]	page ID	mark page as "available"	mark page as "unavailable"

[1] written during recursive split
[2] written during node deletion

Table 1: Log Records and Their Redo and Undo Actions

counter is automatically recoverable without having to write any log records.

Using end-of-log LSNs to generate NSNs gives us an opportunity for a second optimization. When a descending operation examines a node, it has to memorize the end-of-log LSN along with all the qualifying subtree pointers. This could be a problem, because reading the end-of-log LSN requires synchronization within the log manager, which in turn could cause it to become a bottleneck. To alleviate the traffic on the end-of-log LSN, descending operations can memorize the node's LSN instead. This is possible, because the LSN and the NSN of a page come from the same source, which implies that the parent LSN will always be greater than any of the child NSNs, except for those whose recent splits are not reflected in the parent.[13]

10.2 Savepoints and Partial Rollback

Some commercial DBMSs support the concept of savepoints. These allow a user to establish alternate targets of rollback within a transaction apart from the transaction start. A rollback to a savepoint restores the database to its state as of the time the transaction established the savepoint; in addition, since the transaction is still active at that time, the positions of open cursors must also be restored. In order to record the position of a GiST search operation when establishing a savepoint, it is necessary to record the then-current stack. The search operation traverses the tree in a depth-first fashion, so that the amount of storage required for a copy of the stack (and the stack itself) is proportional to the page capacity times the height of the tree.

In Section 7.2 we noted that in order to prevent nodes from being deleted, a search operation sets signaling locks on the nodes it records on the stack. These are released as soon as the node is visited and the corresponding entry removed from the stack. With the establishment of savepoints, this behavior must be modified slightly. We have to make sure that the signaling locks that exist when the savepoint is established are not released later on.

10.3 Predicate Management

Predicate locking is one of the two centerpieces for repeatable read isolation and plays a crucial role in the algorithms for the search and insert operations. A predicate manager component can be used in conjunction with the regular lock manager to offer the required functionality efficiently.

The predicate management functions required by the search and insert operations are: (1) attaching search predicates to nodes; (2) removing search predicates from nodes; (3) checking all of the predicates attached to a node for conflicts with the new key of an insert operation; (4) replicating predicate attachments at child nodes during the BP update and percolation phase of an insertion; (5) replicating predicate attachments at sibling nodes when doing a node split. These functions are best realized by a predicate manager component, which can be implemented along the lines of a lock manager within a DBMS (see [GR93] for an example). The major data structures within a predicate manager would be:

- a list of predicates per transaction;
- a list of node attachments for each predicate;
- a list of the predicates attached to each node.

With the help of the standard lock manager, an operation can block "on a predicate" by blocking on the owner transaction of the predicate. This is easily achieved by constructing a lock name from the owner transaction ID and requesting an S-mode lock on that name, assuming that every transaction acquires an X-mode lock on its own ID when it starts up.

The alert reader will have noticed that insert operations as described in Chapter 6 are susceptible to starvation if they block on a predicate when checking their target leaf's list for conflicts. The reason is that while the insert operation is blocked, new scan operations are allowed to set additional predicates on the leaf. These

[13]The underlying assumption is that the splitting of the child and the update of the parent entry will each generate a log record, and that those log records will be written in that same order. Hence the parent node will have a higher LSN than the child's NSN, which is the LSN of its split log record.

could force the insert operation to block again as soon as it is unblocked, which can be carried on indefinitely. The solution is to require insertion operations to add their key as an insert predicate to the leaf, allowing subsequent scans to block on it. The predicate management component can then enforce fair locking behavior by ordering predicates (which are lock requests) attached to a particular leaf in a FIFO list and checking each new predicate against those ahead of it in the list for conflicts.

Analogous to the replication of predicate attachments as a consequence of node splits, it is also necessary to replicate the signaling locks set on a node. This requires an extension of the standard lock manager functionality.

11 Related Work

In this section, we will compare our approach to concurrency and recovery wih the prior work in this area, which has mostly been restricted to B-trees ([BS77, LY81, Sag86, SG88, Moh90a, ML92]). The comparison will show that structural differences between B-trees and the class of trees represented by GiSTs make most of the concurrency techniques developed so far for B-trees inapplicable in the GiST context and hence in the context of structures such as R-trees, TV-trees and so forth.

The link technique, on which GiST concurrency is based, was first introduced in ([LY81]) as the B-link tree, and has been the basis of many subsequent papers on B-tree concurrency (for example [Sag86]). Its superiority over subtree-locking concurrency protocols, as described in [BS77], has been confirmed by two performance studies ([SC91, JS93]).

A different approach than node linking was taken in ARIES/IM ([ML92]), which employs a conventional non-link tree structure and allows latch-coupling during tree descent, but is still able to propagate splits bottom-up without having to lock subtrees. Instead of following rightlinks, the traversing operations recognize an ongoing split and compensate for it by repositioning themselves in an ancestor node and retraversing the tree from there. Since repositioning a search operation within an ancestor node requires partitioning of the key space (see Figure 5 for an example), this technique is, like the original B-link tree technique, also not applicable in a GiST context.

Another implication of a main difference between B-trees and GiSTs—partitioning vs. non-partitioning of the key space—is the effectiveness of latch-coupling for avoiding incorrect pointers caused by node deletions. A search operation in a GiST might have to traverse multiple subtrees and would have to—if it wanted to use latch-coupling—start the traversal of a particular subtree from the respective parent node. In other words, the search operation would have to reposition itself within a parent node before it started traversing the next subtree, which is impossible as explained in Section 7.2. Hence, latch-coupling is impossible in any non-partitioning tree structure and the method for avoiding incorrect pointers introduced in Section 7.2 appears to be the only efficient solution.

Another general tree structure for which concurrency algorithms were developed is the (so-called) π-tree ([LS92]), which was designed for multidimensional point data, and partitions its key space at each level of the tree. It deviates from GiSTs in that it is not a proper tree but a DAG: two index entries on different parent nodes can point to the same child node. The π-tree also employs the link technique to allow traversing operations to recover from missed splits. Because each tree level partitions the key space, it can rely on BPs to detect splits. Although it is not mentioned in the π-tree paper, search operations may have to descend multiple subtrees. Node deletion is possible by latch-coupling during descent and repositioning in an ancestor node for each traversal of a subtree—the latter aspect is not mentioned in the paper. No algorithms for transactional isolation were developed for the π-tree, but a slight modification of the the method described in Sections 4 to 6 is also applicable to them.

The fundamental ideas for access method recovery—essentially, the requirement for separate atomic units of work to carry out structure modifications—have been recognized early on and have been published in a number of articles ([ML92, Moh90a, GR93, LS92]). The GiST logging and recovery protocol as presented in Section 9 directly builds on that prior work.

The basic GiST structure itself is described in [HNP95]. The paper also gives three examples of implementations of specific trees within the GiST structure and tries to analyze the GiST performance in a general way. The basis of the GiST concurrency protocol was developed in [KB95] in the context of R-trees, which have the same structural properties as GiSTs (non-partitioning, non-linear keys). The paper does not sufficiently address the problems of transactional isolation, recovery and node deletion. The data-only locking approach and logical deletion have been adopted from [ML92, Moh95] and [Moh90b].

12 Conclusion

This paper presents algorithms for concurrency, recovery and transactional isolation for a broad class of search trees. The algorithms are presented in the context of the GiST, an abstract hierarchical index structure, which is designed to support an extensible set of data types and queries, and which can be specialized to any particular access method that complies with its overall structure. In conjunction with the algorithms developed in this paper, the GiST structure can serve as the basis of truly extensible indexing in commercial-strength database systems. The core DBMS plus GiST can be extended with a new access method simply by supplying it with a set of pre-specified methods, which specialize the abstract GiST structure into the desired access method. Details such as concurrency and recovery—which usually account for a major fraction of the complexity of the code, are error-prone and hard to debug—can be ignored by the extension code; these are handled by the GiST structure extended with the mechanisms presented in this paper.

The key features of the GiST concurrency protocol is that it does not hold any latches during I/Os and is deadlock free, resulting in a degree of concurrency that should match that of the best B-tree concurrency protocols. The basic idea is derived from the link technique pioneered for B-trees, which allows compensating for unexpected splits by moving across rightlinks. To make this work for a broader class of tree structures, it is necessary to add sequence numbers to the nodes in order to make node splits visible without reference to the stored keys.

In order to ensure repeatable read transactional isolation, which guarantees the absence of "phantoms" across search operations, we propose a hybrid locking mechanism, which combines two-phase locking of data records with predicate locking. Predicate locking is responsible for avoiding phantoms, whereas more efficient data record locking is used for retrieved index entries. This division of responsibilities is necessary, because data record locking alone cannot reasonably solve the phantom problem in a key space without linear order; it is efficient because it relies as much as possible on data record locks, which can be checked and set more cheaply than predicate locks.

These algorithms are general enough to work for all tree-based access methods with a "traditional" B-tree-like structure, because structural elements, not semantic knowledge about the data, are exploited. We are currently implementing GiSTs emulating B-trees in DB2/Common Server.

Acknowledgement

We would like to thank Paul Aoki and Bernhard Seeger for their comments on this paper.

References

[BS77] R. Bayer and M. Schkolnick. Concurrency of Operations on B-Trees. *Acta Informatica*, 9:1–21, 1977.

[EGLT76] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger. The notion of consistency and predicate locks in database systems. *Communications of the ACM*, 19(11):624–633, November 1976.

[GR93] J. Gray and A. Reuter. *Transaction Processing – Concepts and Techniques*. Morgan Kaufmann Publishers, 1993.

[Gra78] J. Gray. Notes on Database Operating Systems. In *Operating Systems – An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*. Springer-Verlag, 1978.

[Gut84] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD Conf.*, pages 47–57, June 1984.

[HNP95] J. Hellerstein, J. Naughton, and A. Pfeffer. Generalized Search Trees for Database Systems. In *Proc. 21st Int'l Conference on Very Large Databases (VLDB)*, pages 562–573, September 1995.

[Jag90] H. V. Jagadish. Spatial Search with Polyhedra. In *Proc. 6th IEEE Int'l Conf. on Data Engin.*, 1990.

[JS93] T. Johnson and D. Shasha. The Performance of Current B-Tree Algorithms. *ACM TODS*, 18(1), March 1993.

[KB95] M. Kornacker and D. Banks. High-Concurrency Locking in R-Trees. In *Proc. 21st Int'l Conference on Very Large Databases (VLDB)*, pages 134–145, September 1995.

[KKAD89] W. Kim, K.-C. Kim, and A. A. Dale. *Object-Oriented Concepts, Databases and Applications*, chapter Indexing Techniques for Object-Oriented Databases. ACM Press and Addison-Wesley Publishing Co., 1989.

[KL80] H. T. Kung and P. L. Lehman. Concurrent Manipulation of Binary Search Trees. *ACM TODS*, 5(3), 1980.

[LJF94] K. Lin, H. Jagadish, and C. Faloutsos. The TV-Tree: An Index Structure for High-Dimensional Data. *VLDB Journal*, 3, October 1994.

[Lom93] D. Lomet. Key Range Locking Strategies for Improved Concurrency. In *Proc. 19th Int'l Conf. on Very Large Databases (VLDB)*, August 1993.

[LS90] D. Lomet and B. Salzberg. The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance. *ACM TODS*, 15(4):625–685, December 1990.

[LS92] D. Lomet and B. Salzberg. Access Method Concurrency with Recovery. In *Proc. ACM SIGMOD Conf.*, pages 351–360, 1992.

[LY81] P.L. Lehmann and S.B. Yao. Efficient Locking for Concurrent Operations on B-Trees. *ACM TODS*, 6(4):650–670, December 1981.

[MHL+92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM TODS*, 17(1), March 1992.

[ML92] C. Mohan and F. Levine. ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging. In *Proc. ACM SIGMOD Conf.*, June 1992.

[Moh90a] C. Mohan. ARIES/KVL: A Key-Value Locking Method for Concurrency Control of Multiaction Transactions Operating on B-Tree Indexes. In *Proc. 16th Int'l Conference on Very Large Databases (VLDB)*, August 1990.

[Moh90b] C. Mohan. Commit_LSN: A Novel and Simple Method for Reducing Locking and Latching in Transaction Processing Systems. In *Proc. 16th Int'l Conference on Very Large Databases (VLDB)*, August 1990.

[Moh95] C. Mohan. *Performance of Concurrency Control Mechanisms in Centralized Database Systems*, chapter Concurrency Control and Recovery Methods for B+-Tree Indexes: ARIES/KVL and ARIES/IM. Prentice Hall, Englewood Cliffs, New Jersey, 1995.

[Sag86] Y. Sagiv. Concurrent Operations on B*-Trees with Overtaking. Journal of Computer and System Sciences, 33(2):275–296, 1986.

[SC91] V. Srinivasan and M. Carey. Performance of B-Tree Concurrency Control Algorithms. In *Proc. ACM SIGMOD Conf.*, pages 416–425, 1991.

[SG88] D. Shasha and N. Goodman. Concurrent Search Structure Algorithms. *ACM TODS*, 13(1), March 1988.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-Tree: A Dynamic Index for Multidimensional Objects. In *Proc. 13th Int'l Conference on Very Large Databases (VLDB)*, pages 507–518, September 1987.