The VLDB Journal (1997) 6: 224–240

The VLDB Journal

Concurrency and recovery for index trees

David Lomet[1], **Betty Salzberg**[2]

[1] Microsoft Corporation, One Microsoft Way, Bldg 9, Redmond, WA 98052-6399, USA; lomet@microsoft.com
[2] College of Computer Science, Northeastern University, Boston, MA 02115, USA; salzberg@ccs.neu.edu

Edited by A. Reuter. Received August 1995 / accepted July 1996

Abstract. Although many suggestions have been made for concurrency in B^+-trees, few of these have considered recovery as well. We describe an approach which provides high concurrency while preserving well-formed trees across system crashes. Our approach works for a class of index trees that is a generalization of the B^{link}-tree. This class includes some multi-attribute indexes and temporal indexes. Structural changes in an index tree are decomposed into a sequence of atomic actions, each one leaving the tree well-formed and each working on a separate level of the tree. All atomic actions on levels of the tree above the leaf level are independent of database transactions, and so are of short duration. Incomplete structural changes are detected in normal operations and trigger completion.

Key words: Concurrency – Recovery – Indexing – Access methods – B-trees

1 Introduction

In this paper, we describe a concurrency algorithm for a class of trees which includes a type of B^+-tree containing sibling links (called a *B^{link}-tree*) and some spatial and temporal indexes. Only data-node splitting sometimes takes place within database transactions. All other parts of index tree restructuring are independent of such transactions. The basic principles of this algorithm were exposed in (Lomet and Salzberg 1992). This paper gives a step-by-step description of the algorithm details.

In order to use this algorithm, a search structure must have several structural and behavioral properties.

- It must *partition* the search space at each level of the tree. That is, the set of spaces associated with nodes at a given level covers the search space and no two nodes' spaces overlap.
- When a node is split, a pointer in the old node must be inserted, which indicates the address of the new node.

Other properties are similar to those of the standard B^+-tree:

- The data is all in the leaves.
- Insertion must first search down the tree, inserting the new record in a leaf if there is room and, if not, splitting the leaf and creating a new leaf and posting the information to the parent.
- Splitting and posting continues up the tree if needed.
- When nodes get sparse, two "adjacent" nodes may sometimes be consolidated when they share the same parent.

We call such a tree a Π-tree, and we give a formal definition of the Π-tree in Sect. 3.

A B^{link}-tree (Lehman and Yao 1981), which is a B^+-tree with sibling links, is a Π-tree. A time-split B-tree (or *TSB-tree*) (Lomet and Salzberg 1989) can be made into a Π-tree by adding sibling links. The TSB-tree is a temporal index, where the search space is a two-dimensional space based on time and database key. The hB^{Π}-tree (Evangelidis et al. 1995, 1997) is a spatial search structure on any number of dimensions which is a Π-tree. The R-tree (Guttman 1984) is not a Π-tree (and could not easily be made into one by adding sibling links), because the spaces associated with nodes which are on the same level of the tree overlap.

The algorithm described here can be used on any Π-tree. It provides a high degree of concurrency because it breaks down structural changes into a series of short-term atomic actions – splitting a node, posting split information to a parent or consolidating a node with one of its siblings. Consolidation requires three nodes to be locked – the parent and the two siblings being consolidated. Splitting index nodes requires only the node being split to be locked. Posting requires the node receiving the new split information (the parent) to be locked and also must have a short-term lock on the child to verify that posting is still needed. The details of this algorithm, with explicit directions for locking, are in Sect. 6.

The subject of concurrency in B^+-trees has a long history (Bayer 1977; Lehman and Yao 1981; Mohan and Levine 1992; Sagiv 1986; Salzberg 1985; Shasha and Goodman 1988). Most work, with the exception of Mohan and Levine (1992), and Gray and Reuter (1993) has not treated the problem of system crashes during structural changes. In this paper, we show how to manage both **concurrency and recovery** for a wide class of index tree structures.

1.1 Our approach

There are four innovations which make it possible for us to provide high concurrency for a large class of index trees and to mesh with several recovery methods. These are:

1. We define a search structure, called a Π-**tree** , that is a generalization of the B^{link}-tree (Lehman and Yao 1981). Our concurrency and recovery method is defined to work with all search structures in this class.
2. Π-tree structural changes consist of a **sequence** of atomic actions (Lomet 1977). These actions are guaranteed to have the all-or-nothing property by the recovery method. Searchers can see the intermediate states of the Π-tree that exist between these atomic actions. No locks are held between atomic actions. However, incomplete structural changes do not affect search correctness.
3. We define separate actions for performing updates at each level of the tree. Three actions are defined: node consolidation, node splitting and posting node-split information. Node consolidation holds three locks. Node splitting and posting hold at most two locks. Atomic actions on non-leaf nodes can be separate from the transaction whose update triggers a structural change. No node consolidation, even at the leaf level, need be part of an updating transaction. Only node splitting at the leaves of a tree may need to be within an updating transaction in such a way that locks associated with the atomic action are held until the end of the transaction. This occurs only in systems which do not support non-page-oriented (logical) UNDO. This feature is especially important in systems where a time limit is given for database transactions.
4. When a system crash occurs during the sequence of atomic actions that constitutes a complete Π-tree structural change, *crash recovery takes no special measures*. A crash may cause an intermediate state to persist for some time. The structural change is completed when the intermediate state is detected during **normal** subsequent processing by scheduling a completing atomic action. The state is tested again in the completing atomic action to assure the idempotence of completion.
   The log could be used to detect and complete structural changes or to complete other actions as in (Zou and Salzberg 1996). We decided explicitly against this option for the following reasons:
   (a) making recovery simple and fast is important: our lazy approach requires no special logging and no special recovery processing;
   (b) a system failure does not often occur leaving a structural change incomplete; and
   (c) even when this happens, some incomplete structural changes will not affect performance, since the relevant nodes may not be visited again.

1.2 Organization of paper

Section 2 gives an overview of the method. Section 3 formally defines the Π-tree. Our concurrency algorithm can be used by any search structure which satisfies this definition. In Sect. 4, issues of deadlock avoidance and of page-oriented UNDO are discussed. This affects the decisions on the type of locking done by our atomic actions. Section 5 describes how atomic actions are scheduled and how information from previous atomic actions about the search path can sometimes be used. Most of the material in Sects. 3, 4 and 5 appeared in (Lomet and Salzberg 1992) and is repeated here to make this paper self-contained. Section 6 presents the algorithm. We give examples of adaptations of two search structures so as to be forms of Π-trees in Sect. 7. Section 8 is a short discussion of results.

2 Overview

We are concerned with structural changes in index trees that are a generalization of B^{link}-trees, which are a variation of B^{+}-trees. In particular, our index tree is multi-level, with all leaves on the same level. Leaves contain data records if the tree is used as a primary index. They contain entries referencing data records if the tree is used as a secondary index. These entries have the form <secondary key, primary key> or <secondary key, record address>. All insertions of new records or of entries which are references to new records therefore occur only at the leaf, and structural changes are propagated upwards.

The two basic structural changes occur when a node is split and when an underutilized node is consolidated with a sibling. Although information about these changes does not need to be **posted** (placed in the parents of the changed node(s)) for correct search, for efficiency reasons, change information should eventually be posted. Before outlining the two basic structural changes, we first explain the terms **lock** and **latch**.

2.1 Locks and latches

We control concurrency by exploiting two forms of “locks”, which we refer to here as “database locks” (but subsequently simply refer to as “locks”) and “latches”.

Database locks are handled by a lock manager, which maintains a graph of who delays whom. The lock manager detects cycles in this delay graph, which indicate that deadlock has occurred, and aborts one of the parties involved.

Latches are short-term low-cost locks for which the holder’s usage pattern guarantees the absence of deadlock. Thus, latches do not involve the lock manager, can be associated directly with the protected data, and can be manipulated with in-line code.

2.2 Splits

Nodes are split when there is no space for an insertion. In our index tree structure, this can occur when a new record is added to the database. Here, the leaf must split if there is no available space. Some of the leaf node content remains where it is and some is copied to a new leaf. The information about the split must be posted to the next level of the tree.

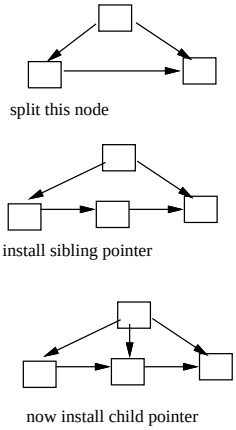

Fig. 1. The sequence of state changes to a B^{link}-tree that are triggered by an insertion into a leaf node

Since nodes at the next level may also be too full to receive the new posting information, the process of node-splitting can occur recursively at higher levels of the tree.

Suppose a database transaction wishes to insert a new record in the database. A primary index would require that a new data record be placed directly in its leaf node. Each secondary index would require a new entry in one of its leaf nodes when the new data record is inserted. In either case, suppose a leaf node split is required. For some recovery methods, the split must be part of the database transaction. But for no recovery methods does our method require that the posting of index information to the next (and occasionally higher) level of the tree be part of the database transaction. This is because we use a generalization of the B^{link}-tree. If the index information is not posted, search follows side pointers which are installed at the split. So the search will be correct, even if the information about the new node is not immediately posted to its parent.

In this way, we are able to decompose a split operation on a tree into a sequence of separate atomic actions, one at each affected level of the tree. First, a split is made at the leaf level of the tree. A posting operation at the next level is then scheduled for a later time. The database transaction can proceed without further action and without ever latching any nodes above the leaf level. When the posting action occurs, if a further split and a higher posting is required, only the split at the level of the posting is made. The higher level operation is scheduled as a separate atomic action. A split above the leaf level latches only the node to be split and drops the latch as soon as the split is complete. A posting action latches the node where the information is to be posted and also briefly latches the split child to verify the information and then drops its latches when the information has been posted. Fig. 1 illustrates the progression of atomic actions that can be triggered by an initial leaf insertion.

Thus structural changes caused by insertion are decomposed into a sequence of actions, one at each level of the tree. Each such action is logged and is atomic. If a system failure occurs before a complete structural change is made, some of the separate actions in the sequence may have completed and others may not have, but after recovery at restart, no action will be part-way done. That is, an index tree will be operation-consistent, where the operations involved here are our single tree-level atomic actions.

No attempt is made to stably record the progress of an entire structural change. Thus, this information is lost across system crashes. Incomplete structural changes caused by insertions are completed when some transaction or atomic action is forced to follow a side pointer. This is an indication of a missing index-posting. The missing posting is then rescheduled. If no transaction ever follows the side pointer where the posting is missing, the posting is never made. However, in this case, the missing posting does not cause any performance penalty.

2.3 Node consolidation

Consolidation of underutilized nodes is an optimization. Many commercial systems do not support this optimization. In our method, if a system wishes to support this optimization, a node consolidation is scheduled when a transaction visits an underutilized node. Node consolidation is never part of a database transaction.

Suppose a transaction encounters an underutilized node. It then schedules a node consolidation. The node consolidation atomic action works at two levels: the level of the underutilized node and that of its parent.

A number of constraints are enforced. For example, the dropped node must have only one parent. (This is only an issue in multi-attribute trees; B^{link}-tree nodes always have only one parent). The node to be dropped must be matched with an “adjacent” sibling (below, in the general case, we will define “adjacency” in terms of “containment”) which has enough space to absorb the contents of the dropped node. (Either the dropped node or its sibling may be the original underutilized node.) If these conditions are satisfied, we latch the parent and the two siblings and proceed with the consolidation.

To perform node consolidation, we erase the index term for the node to be dropped and move the contents of the dropped node to its sibling. The index term for the sibling must be changed to reflect its new larger contents. If this consolidation causes the parent of the dropped node to be underutilized, we schedule a separate node consolidation atomic action for the parent.

2.4 Scheduling atomic actions

We have used the term **scheduling** to describe the initiation of the separate atomic actions that comprise a complete structural change. This term was chosen to emphasize the **independence** of the atomic actions from each other. That is, *each can succeed or fail independently of the success or failure of the other atomic actions*. (Of course, an atomic action that is scheduled to post an index term describing a split will only perform the posting if the node allocated in the split action is present when the index-posting operation executes.) This independence is important for the recoverability of our structural changes. We exploit it through making recoverable only the separate atomic actions. Entire structural changes need not be recoverable as a unit. By having a smaller unit of recovery, concurrency is enhanced.

Independence of atomic actions is essential for the correct operation of our structural changes. However, it is useful to schedule atomic actions proximate to related atomic actions for two reasons.

1. Proximate scheduling enhances our ability to exploit saved state information across atomic actions. This saved state information has to be validated (checked to assure that it still describes correctly the current state). But, in so far as the state has not changed, it permits us to avoid all or part of index tree searches.
2. Proximate scheduling frequently avoids the process or thread switch that might be required were scheduling deferred and done via an asynchronous control path. Since process- (and thread-) switching costs can be more than a thousand instructions, proximate scheduling is important.

Thus, while we consistently describe, e.g., a node split, as resulting in the scheduling of a subsequent index term posting atomic action, we expect that the execution of this action will normally occur promptly and in the same control path.

We need to be careful here in that scheduling is subject to one important constraint. The scheduled action will normally be required to be performed after the currently executing atomic action drops its latches and locks. This prevents deadlocks between searchers which descend the tree and structural changes which ascend the tree. Deadlock prevention is described in detail in Sect. 4.

If node splitting is part of a database transaction (i.e. if the node is a leaf node), the index posting can occur before the database transaction commits if non-page-oriented UNDO is supported. Here, the only consideration is that the split completes and drops its latches before the index-posting action begins. In contrast, in page-oriented UNDO systems (explained in Sect. 4.3), when a leaf split occurs as part of a database transaction, the index-posting action must be deferred until after the database transaction commits.

When the need for an index-posting is discovered during a search that requires a sibling link traversal, scheduling of the posting can always be done promptly, i.e., immediately following the release of the latches acquired on the parent during the search by the current control path. Here, a separate control path is used for the index-posting. If the same control path were to be used, the latches on the level of the sibling traversal would have to be released as well before the index-posting could begin.

Node consolidation is handled analogously to index-posting. Consolidation can be promptly executed when dealing with index nodes, i.e., as soon as the search latches are released. For leaf nodes in page-oriented UNDO systems, consolidation will not be executed until after any transaction with locks on the nodes completes. In particular, the control path that detects the need for consolidation must not hold any locks on the nodes subject to consolidation at the time consolidation is executed. This may require that leaf consolidation actions be scheduled after the completion of the current database transaction.

3 The Π-tree

In this section, we define the properties a search structure must have to be able to use our concurrency and recovery method. First, we formally describe the structural properties, and then we describe the behavioral properties. A structure which satisfies this definition is called a Π-tree.

3.1 Structural description

Informally, a Π-tree is a balanced tree, (all leaves are at the same *level*) and we measure the level of a node by the number of child edges on any path between the node and a leaf node. More precisely, however, a Π-tree is a rooted DAG, because, like the B^{link}-tree, nodes have edges to sibling nodes as well as child nodes. Edges between sibling nodes go in only one direction. Some commercial implementations of the B^{+}-tree have doubly linked lists at the leaf level. Our Π-tree does not have doubly linked lists. All these terms are defined more formally below.

3.1.1 Within one level

Each node is **responsible for** a specific part of the key space, and it retains that responsibility for as long as it is allocated. A node can meet its space responsibility in two ways. It can **directly contain** entries (data or index terms) for the space. Alternatively, it can **delegate** responsibility for part of the space to a **sibling node**.

A node delegates space to a new sibling node during a node split. A **sibling term** describes a key space for which a sibling node is responsible and includes a **side pointer** to the sibling. A node containing a sibling term is called the **containing node** and the sibling node to which it refers is called the **contained node**. Note that this relationship can change over time. If A is a containing node and B is its contained node and A splits again, allocating a new node C, the side pointer from A to B may be copied into the new node C. Then C contains B and A contains C.

For example, an index or data node A in a B^{link}-tree is said to be responsible for a key range from some lower value to the maximum key value, say [500, MAX]. When it splits, all keys with value equal to or greater than 700, say, go to the new sibling, B. B is then responsible for [700, MAX]. The original node A is still responsible for [500, MAX], but now contains a sibling term indicating that the keys greater than or equal to 700 are at another address, B. This is indicated in Fig. 2a and b.

If the node A responsible for [500, MAX] splits again, moving all the keys greater or equal to 600 to another new sibling, C, C will obtain the address of B as its side pointer, and A will have a sibling term indicating that keys greater than or equal to 600 are in C. This is illustrated in Fig. 2c.

Any node except the root can contain sibling terms to contained nodes. Further, a Π-tree node is not constrained to have only a single sibling, but may have several. (This does not happen in the B^{link}-tree.) Formally, a **level** of the Π-tree is a maximal connected subgraph of nodes and side pointer edges. The first node at each level is responsible for

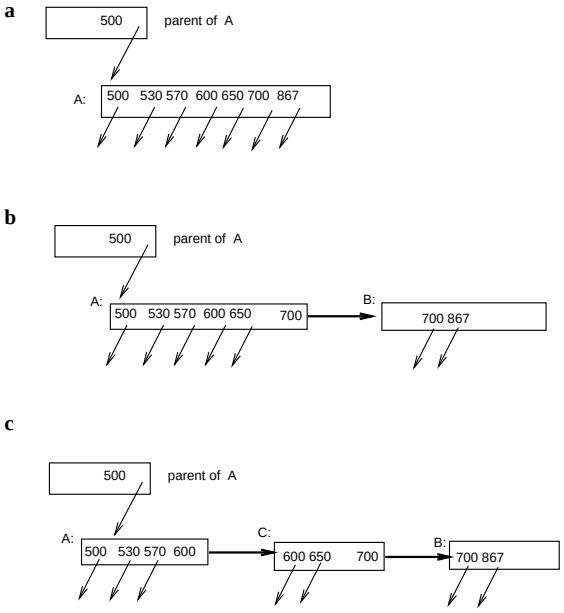

Fig. 2a–c. Contained and containing nodes; space responsibility. **a** At this point, A is responsible for [500, MAX] and directly contains all entries in its level in [500,MAX]. Then A splits and delegates some of its space to a contained sibling B. **b** Now A is a container node and B is a contained node. This is because A now contains a sibling term (700) which descibes the space [700,MAX] for which B is responsible. A is still responsible for [500,MAX], but now its directly contained space is [500,700). **c** Now A is the container node for C and C is the container node for B. A contains a sibling term for C and C has a sibling term for B. A is responsible for [500,MAX], but its directly contained space is [500,600). C is responsible for [600,MAX], but its directly contained space is only [600,700)

the whole space, i.e., it is the containing node for the whole key space. New siblings are always on the same level as the nodes from which they split.

3.1.2 Multiple levels

The Π-tree is split from the bottom, like the B-tree. **Leaf nodes** are at level 0. Leaf nodes contain only data records (in the case of primary trees) or entries consisting of secondary keys and primary keys or references to data pages (in the case of secondary trees) and/or sibling terms. As the Π-tree grows in height via splitting of a root, new levels are formed.

A split is normally described by an index term. Each **index term**, when posted, includes a **child pointer** to a **child node** and a description of a key space for which the child node is responsible. A node containing the index term for a child node is called a **parent** node. A parent node indicates the containment ordering of its children based on the spaces for which the children indexed are responsible.

A parent node directly contains the space for which it is responsible and which it has not delegated, exactly as with a leaf node. In Π-trees, as in B^{link}-trees, parent nodes are **index nodes** which contain only index terms and/or sibling terms. Leaf nodes are not index nodes.

Parent nodes are at a level one higher than their children. Unlike B^{link}-trees, in the more general Π-trees such as TSB-trees and hB^{Π}-trees, the same child can be referred to by two (or more) parents. This happens when the boundary of a parent split cuts across a child boundary.

3.1.3 Well-formed Π-trees

Going down from the root, each level describes a partition of the space into subspaces directly contained by nodes of that level. This gives the Π-tree its name.

Side pointers and child pointers must refer to nodes which are responsible for spaces that contain the indicated subspaces. A pointer can never refer to a deallocated node. Further, an index node must contain index terms that refer to child nodes that are responsible for spaces, the union of which contains the subspace directly contained by the index node. However, each node at a level need not have a parent node at the next higher level. This is an abstraction and generalization of the idea introduced in the B^{link}-tree (Lehman and Yao 1981). That is, having a new node connected in the B^{link}-tree only via a side pointer is acceptable. We never know whether a node directly contains the space of interest or whether it is merely responsible for the space until we examine the sibling terms.

Like (Shasha and Goodman 1988), we define the requirements of a well-formed general search structure. Thus, a Π-tree is **well-formed** if

1. each node is responsible for a subspace of the search space;
2. each sibling term correctly describes a subspace of the (responsible) space of its containing node for which its referenced node is responsible.
3. each index term correctly describes a subspace of the (responsible) space of the index node for which its referenced child node is responsible;
4. the union of the spaces described by the index terms and the sibling terms equals the space an index node is responsible for.
5. the lowest level nodes are leaf nodes.
6. a root exists that is responsible for the entire search space.

The well-formedness description above defines a correct search structure. All structural changing atomic actions must preserve this well-formedness. We will need additional constraints on structure changing actions to facilitate node consolidation (deletion).

3.2 Π-tree behavioral description

Here we describe the operations on Π-trees in a very general way. The steps do not describe how to deal with either concurrent operations or with failures. In particular, we do not show how to decompose structural changes into atomic actions. This section shows how Π-tree searches, splits and node consolidations must behave if the recovery and concurrency algorithm of this paper is to be applicable.

3.2.1 Searching

Searches start at the root of the Π-tree. The root is an index node that directly contains the entire search space. In an

index node whose directly contained space includes a search point, an index term must exist that references a child node that is responsible for the space that contains the search point. There may be several such child nodes. Proceeding to any such child node is correct in that the search will eventually succeed. However, it is desirable to follow the child pointer to the node that directly contains the search point. This avoids subsequent sibling traversals at the next lower level.

Index terms describe the space for which a child is responsible, not its directly contained space. Because the posting of index terms can be delayed, we can only calculate the space **approximately contained** by a child with respect to a given parent. This is *the difference between that part of the space of the parent node the child is responsible for and the subspaces that it has delegated to other child nodes referenced by index terms that are present in the index node*. Before we visit a node, we have only approximate or partial information about its contents.

In Fig. 2b, the approximately contained space of node A with respect to its parent includes all values greater than or equal to 500. Its directly contained space includes only the values greater than or equal to 500 and smaller than 700. This is because the index term for B has not yet been posted. Thus, as far as a visitor to the parent can tell, A has (approximately) all values greater than or equal to 500.

When all index terms for child nodes that have been delegated space from a child C have been posted to an index node I, the approximately contained space for C relative to I equals the intersection of its directly contained space and the directly contained space of I. With this precise information, a side pointer from C would not have to be followed after the search proceeds from I to C.

Thus, we minimize our search cost by proceeding to the child that approximately contains the search point. Because we attempt to make structural changes complete, this node will usually, but not always, contain the search point. If the directly contained space of a node does not include the search point, a side pointer is followed to the sibling node that has been delegated the subspace containing the search point. Eventually, a sibling is found whose directly contained space includes the search point.

The search continues until the leaf node level of the tree is reached. In primary trees, the record for the search point will be present in the leaf node whose directly contained space includes the search point, if it exists at all. In secondary trees, the key (which is the search point) will be in the leaf if the corresponding record is in the database.

3.2.2 Node-splitting

We wish to build our Π-tree so as to permit our search procedure to minimize side pointer traversals. Thus, we want the children of an index node to be exactly the nodes at the next lower level with directly contained spaces that intersect the directly contained space of the index node. However, when we split index nodes, our information is incomplete. The best that we can do is to partition index terms based on the spaces that their child nodes approximately contain. Index terms are thus placed in the resulting index node(s) whose directly contained space(s) intersect(s) the approximately contained space of the index term. This is acceptable in that searches will still be effective. Over time, the missing index terms will be correctly posted (see Sect. 4.1).

Let O stand for the original node to be split. Let S stand for the new sibling node. A node split has the following steps:

1. Allocate space for S.
2. Partition the subspace directly contained by O into two parts. O continues to directly contain one part. The other part is delegated to S.
3. If O is a leaf node and the data is point data, place in S all of O's data that are contained in the delegated space. (In a secondary tree, "data" consists of secondary keys and primary keys or pointers to database pages.) Include any sibling terms to subspaces for which S is now responsible. Remove from O all the data that it no longer directly contains.
4. If O is a leaf node and the data has some extent (for example, in the TSB-tree, data has a time interval as well as a database key), place in S all of O's data which *intersects* the delegated space. (In a secondary tree, "data" consists of secondary keys and primary keys or pointers to database pages.) Include any sibling terms to subspaces for which S is now responsible. Remove from O all the data that its directly contained space does not intersect. This implies that data items which intersect both spaces will have copies in both nodes.
5. If O is an index node, we retain in O the index terms i that refer to child nodes $C(i)$ whose approximately contained spaces intersect the now smaller space directly contained by O. Similarly, if the approximately contained space of $C(i)$ intersects S's space, i is placed in S. Because an index node split can divide the approximately contained space of a child node, the index term for that node can end up in both of the resulting index nodes. (This does not happen in the B^{link}-tree, but can happen in the hB^{Π}-tree or the TSB-tree.)
6. Put a sibling term in O that refers to S.
7. Schedule the posting of an index term describing the split to the next higher level of the tree. The index term contains a reference to S and describes the space for which S is responsible. Posting occurs in a separate atomic action from the action that performs the split.

Example: In a B^{link}-tree, an index or sibling term is represented by a key value and node pointer. It denotes that the child node referenced is responsible for the entire space greater than or equal to the key. To perform a node split, first allocate a new node. Find the key value that evenly divides the records of the node. Copy all records ("records" may be index entries in index nodes or data records in leaf nodes) from the original node to the new node whose keys are ordered after the middle record's key. The new node has been delegated the high-order key subspace. Copy the link (sibling term) from the old node to the new node. Then remove the copied records from the old node. Replace the link in the old node with a new sibling term (address of the new node and the split key value). Finally, post the address of the new node and the split key value to the parent. This is the index term.

3.2.3 Clipping

The entries of index nodes denote subspaces, not merely points. (In B^{link}-trees, subspaces are key intervals.) When an index node is split, it is simplest, if possible, to delegate to the new sibling a space which is the union of the approximately contained spaces of a subset of child nodes. This is what happens in the B^{link}-tree. Then, there will not be an index term that needs to appear in both nodes resulting from the split. It can be difficult to split a multi-attribute index node in this way, because either the space partitioning is too complex, resulting in very large index and sibling terms, or because the division between original and new sibling nodes is too unbalanced, reducing storage utilization. In the TSB-tree, splitting by any given time value will usually result in cutting across the timespan of several of its children, for example.

This approach to splitting nodes whose entries describe spatial information by storing the entry in both nodes is called "clipping". When a child node is referenced from two index nodes (or more) because its index term was clipped, then posting index terms describing the splitting of this child may involve the updating of several of these parent index nodes. We must be prepared to deal with this complication.

Because of the redundant paths to data that are provided by Π-trees, we need not post index terms to all parents of a splitting node atomically. Instead, we post an index term only to the parent that is on the current search path to the splitting node. This is the lowest cost way of updating this parent, since it has already been read, and merely needs to be updated and written to complete the index-posting.

Other parents can be updated when they are on a search path that results in a sibling traversal to the new node. This exploits a mechanism that is already present to cope with system failures in the midst of Π-tree structural changes. Using this mechanism does not usually increase the cost of the structural change. Instead of reading a second parent and writing it, we perform the write of the second parent later and incur an extra read to do the sibling traversal. Subsequently, when we refer to "the parent", we intend this to denote the parent that is on the current search path.

3.2.4 Node consolidation

Node consolidation is scheduled when a node's storage utilization drops below some threshold. When a node N becomes underutilized, it may be possible to consolidated it with either its containing node (the sibling of N which contains a sibling term referring to N) or one of its contained nodes (nodes referred to by sibling terms inside N). We always move the node contents from contained node to containing node, regardless of which is the underutilized node. Then the index term for the contained node is deleted and the contained node is deallocated. For this to be simple,

- both containing and contained node must be referenced by index terms in the same parent node, and
- the contained node must only be referenced by this parent.

These conditions mean that only the single parent of the contained node need be updated during a consolidation. This node will also be a parent of the containing node. These conditions are used to simplify the consolidation. Refusing to consolidate other nodes means that we will consolidate fewer nodes. But the search structure will remain well-formed.

There is a difficulty with the above constraints. Whether a node is referenced by more than one parent is not derivable from the index term information we have described thus far. However, multi-parent nodes are only formed when (1) an index node (the parent) splits, clipping one or more of its index terms, or (2) when a child with more than one parent is split, possibly requiring posting in more than one place. We mark these clipped index terms as referring to multi-parent nodes. All other nodes are what we call **single parent nodes** and are subject to consolidation.

4 Atomic actions for updating

We need to assure that atomic actions are correctly serialized and have the all-or-nothing property required of them. Interactions between atomic actions must not cause undetected deadlocks or incorrect searches. How this is done is described in this section.

4.1 Latching for atomic actions

4.1.1 Resource ordering and deadlock avoidance

The only "locks" required for atomic actions that change an index tree at the index levels, i.e., above the leaf level, are *latches*. For deadlock avoidance, resources are ordered. Latches will be held on Π-tree nodes and will be acquired in the same order by each atomic action, thus preventing deadlock. Parents are latched before children and containers before contained nodes.

Promoting a previously acquired latch violates the ordering of resources and compromises deadlock avoidance. Promotion is the most common cause of deadlock (Gray and Reuter 1993). For example, when two transactions set S-latches on the same object to be updated, and then subsequently desire to promote their latches to X, a deadlock results.

Update(U) latches (Gray et al. 1976) support latch promotion by retaining an exclusive claim on a resource that is currently shared (Lomet 1980). They allow sharing by readers, but conflict with X or other U-latches. An atomic action is not allowed to promote from an S- to an X-latch, because this increases its claim. But it may promote from a U-latch to an X-latch.

However, a U-latch may only be safely promoted to X under restricted circumstances. We must prevent another action with an S-latch on the resource from having to wait for higher numbered resources that might be already be latched by the requester of the latch promotion. The rule that we observe is that the promotion request on node N is not made while the requester holds latches on nodes lower than N in the tree or further along than N in the partial order of siblings made by sibling pointers.

4.1.2 Latch acquisition

The resource ordering we have chosen for our algorithm, which will enable us to prevent deadlock, is presented in this subsection. Latches are acquired in search order, parent nodes prior to their children and containing nodes prior to the contained nodes referenced via their side pointers. Whenever a node might be written, a U-latch is used.

Space management information can be ordered last. Node-splitting and consolidation access it, but other updates and accesses do not. Changes in space management information follow a prior tree traversal and update attempt. Hence, latching and accessing this information last is convenient and shortens its latch hold time.

When the order above might be violated, as it would in an upward propagation of node-splitting, the activity is decomposed into separate atomic actions, each one of which follows the resource ordering we have chosen. The first action is terminated, all its latches and locks are dropped, and a second atomic action is initiated to complete the structural change.

4.1.3 Release of latches by atomic actions

When dealing with index trees, the types of possible atomic actions are known. Because of this, there are circumstances in which release of latches before termination of the atomic action does not compromise correctness.

We do not claim that database transactions are serializable with respect to atomic actions. If a database transaction makes two searches of a tree, one before and one after a node split, for example, it may see two versions of the tree. Both searches will be correct.

Suppose, for example, an atomic action holds a latch on the node whose subspace contains the entry of interest. The higher level nodes are not revisited in the atomic action. Hence, latches on the higher level nodes can be released. An atomic action commutes with other atomic actions that are accessing or manipulating nodes outside the subtree rooted by the latched node.

Other cases where early release is acceptable include (i) releasing a latch when the resource guarded has not been changed and the state observed will not be relied upon for subsequent execution, and (ii) demoting a latch from X- to U-mode when a lower level latch is sufficient to provide correctness even when a node has been changed.

4.2 Interaction with database transactions

4.2.1 Avoiding latch-lock deadlocks

There are two situations where an index tree atomic action may interact with database transactions and also require locks. Sometimes, but not always, these actions are within a database transaction.

1. Normal accessing of a database record (fetch, insert, delete, or update of a data record) requires a lock on the record.
2. Moving data records, whether to split or consolidate nodes, may require database locks on the records to be moved. This is explained in the section on page-oriented UNDO.

Note that, in secondary index trees, the “records” in question may be the secondary keys and primary keys or pointers in the leaves, rather than the actual data records. In this case, database transactions may hold locks on the leaf records (keys and pointers) during insertion or deletion, or when range queries are made (Mohan and Levine 1992; Mohan 1990). For the same reasons, these leaf records in secondary indexes may require database locks when moved due to splits and consolidations.

Should holders of database locks be required to wait for latches on leaf nodes, this wait is not known to the lock manager and can result in an undetected deadlock even though no deadlock involving only latches is possible. For example, transaction $T1$ inserts record R in node N and releases its latch on N while holding its database lock on R. Transaction $T2$ latches N in X mode and tries to delete R. It must wait. Transaction $T1$ now tries to insert a second record in N and is forced to wait **for the N-latch**.

To avoid latch-lock deadlocks, we observe the

– **No-Wait rule:** actions do not wait for database locks while holding a latch that can conflict with a holder of a database lock.

A universal strategy for dealing with an action that waits for a database lock while holding a latch is to abort it, releasing all its latches and undoing its effects. When the requested locks are acquired, the atomic action is re-executed in its entirety, making use of saved information where appropriate. However, for the specific operations of our method, this is not necessary. Only certain latches need be released, i.e., those that can conflict with the holder of a database lock. We then wait for the needed locks to be granted, and resume the atomic action.

For our index tree operations, we must release latches on leaf nodes whenever we wait for database locks. However, latches on index nodes (i.e., nodes above the leaf level) may be retained. Except for leaf node consolidation, no atomic action or database transaction **both**:(i) holds database locks; and (ii) uses other than S-latches above the leaf node level. S-latches on index nodes never conflict with database transactions, only with index change atomic actions. Except for consolidate, these actions never hold database locks. And consolidate never requests a U-latch on the index node to be updated while it holds database locks. Hence, its holding of this U-latch cannot conflict with another consolidate (or any other action) that holds database locks.

4.3 Logical UNDOs

4.3.1 Multi-level system view

Logical UNDO allows updates on records to be UNDONE on a different page from the one the record was on when the update was made. However, this creates a delicate situation when very high concurrency B^+-tree implementations,

including ours, release B$^+$-tree node locks early. This early release of node locks permits multiple transactions to be concurrently updating different parts of the tree, including within a subtree whose root is being split. These updates may be uncommitted at the time of a system crash. If we allow *logical* UNDO and if these updates have been moved because of a completed structure change lower in the tree, they will need UNDO that requires a tree search involving a path that includes the node whose split is incomplete.

The **multi-level transaction** (Weikum 1986) way of looking at this is that there are two **levels of abstraction**, an ordered record abstraction, and the B$^+$-tree implementation that supports it. Record update is considered to be a higher level abstraction than B$^+$-tree structure changing. (In this subsection of this paper, we use the word **level** to mean a *level of abstraction*, not a level of the B$^+$-tree.)

The logical operations (UNDOs in this case) are expressed in terms of the ordered record abstraction. For these operations to execute correctly, the path that they need from root to leaf must be well-formed. But the B$^+$-tree implementation layer is being changed as well, via a page split in this case. We must guarantee that the changes leave the path well-formed when we need to perform the record update UNDOs. This means that incomplete structure changes on such paths need to be recovered before the record update is UNDONE.

4.3.2 Careful recovery needed

We assume here that our recovery method performs REDO recovery first, repeating history, during a forward pass over the log. Then it performs UNDO. UNDO recovery is usually accomplished by starting at the tail of the log and scanning backwards, undoing operations of uncommitted transactions and of uncommitted B$^+$-tree restructurings, until all required UNDO operations have been executed. Without modification, or additional care, this recovery paradigm will occasionally fail.

Recall that a B$^+$-tree split will touch multiple pages, and hence there will be multiple log records, spread out among other operations on the log. Further, some of the changes produced by the split may have been flushed to the disk. Despite this, if all log records describing a B$^+$-tree restructuring on a path are naturally guaranteed to occur later on the log than the log records for the logical record update operations that require the path to be well-formed, then no extra care is required. But this is not guaranteed.

One way of dealing with this problem is to ensure that log records for B$^+$-tree operations always appear later in the log than the log records for the logical UNDOs that need a well-formed path to their update. That way, the tree will be recovered before the logical UNDO that needs it. One can use concurrency control to restrict the sequence in which operations are performed. For example, the ARIES/IM (Mohan and Levine 1992) B$^+$-tree method provides a single structure modification lock (SMO), which is locked in exclusive (X) mode when a structure modification is in progress. Only a single modification can be active at a time.

4.3.3 Multi-level UNDO

Maximum concurrency permits B$^+$-tree structure modifications to occur simultaneously, whether on the same path or not. The bottom-up method we describe here permits this.

Recovery to cope with multi-level subtransactions is designed specifically for situations like these. Instead of performing UNDO recovery from the log tail back in a single sequential pass of the log, multi-level methods (Weikum et al. 1990; Lomet 1992) perform UNDO recovery level by level, begining at the lowest level of abstraction. In the case of B$^+$-trees, this ensures that the B$^+$-tree is fully recovered prior to undoing logical record operations that need well-formed paths.

The MLR method (Lomet 1992) extends to an arbitrary number of levels. Using MLR, B$^+$-tree structure modifications can be incorporated into any scheme involving multiple levels of abstraction, regardless of the number of other levels. It uses a single log, exploits physiological operations for REDO, and repeats history à la ARIES.

4.4 Page-oriented UNDO

4.4.1 Non-commutative updates

Leaf node splitting and consolidation require database locks for some (but not all) recovery protocols. For example, if UNDOs of updates on database records must take place on the same page (leaf node) as the original update, (**page-oriented UNDO**) the records cannot be moved until their updating transaction commits or aborts. No updates can be permitted on records moved by uncommitted structural changes, since undoing the move would cause those records to move. Finally, no update can be permitted that makes the undoing of the move impossible. Such updates are those that consume space in the node that is needed in order to consolidate nodes split by a transaction. Only operations (together with their inverses) that commute with the structural change can be permitted.

When a structural change is part of an independent atomic action, the latches needed for the structural change are two-phased but only persist for the duration of this action. All node consolidation is like this. Some leaf-node-splitting can also be done in an independent atomic action. If a transaction, T, whose update triggers the need for a node split, has not yet updated any record to be moved by the split, the split can be performed in an action independent of and before T. Then, updates that do not commute with the structural change are only blocked during this independent action. Further, of course, the structural change will not be undone if T aborts.

Other leaf node splits in page-oriented UNDO systems must be done within an updating database transaction. In this case, the database locks are held to the end of transaction and the structural change must be undone if the transaction aborts.

4.4.2 Move locks

In this section, we describe the requirements of move locks. They may be implemented in several ways, depending on

the lock granularities and modes available in the underlying system.

For page-oriented UNDO, a **move** lock is required that conflicts with non-commutative updates. The move lock causes the structure change operation to wait until all transactions that are updating records to be moved have completed. Further, it blocks updating transactions from changing records moved until the moving transaction completes. Finally, the move lock keeps updates from consuming space that would prevent the undoing of the move. Since reads do not require UNDO, concurrent reads can be tolerated. Hence, move locks may be compatible with share-mode locks. (This assumes that all access to records is through the Π-tree, following links when sibling terms indicate that it is necessary. Otherwise, say with a table scan using page locks, readers can miss a moved record or see it twice.)

When leaf-node-splitting occurs in a system with page-oriented UNDO, the move lock must be held to the end of the transaction T that does the splitting. The posting of the index term for splits cannot occur until and unless T commits, so that UNDO of the split is possible if T aborts. For the same reason, any other transaction which traverses the sibling pointer created by T's split may not post the index term until T commits. Therefore, a move lock must be distinguished from a share lock. A transaction encountering a move lock on a sibling traversal does not schedule an index-posting. This implies that, in a database with page-oriented UNDO, transactions must set database locks at leaf level when traversing links, in order to detect possible move locks.

A move lock can be realized with a set of individual record locks, a page-level lock, a key-range lock, or even a lock on the whole relation. This depends on the implementation specifics. If the move lock is implemented using a lock whose granule is a node size or larger, once granted, no update activity can alter the locking required. This one lock is sufficient.

Should the move lock be realized as a set of record locks, the need to wait for one of these locks means that the latch on the splitting node must be released. This permits changes to the node that can result in the need for additional records to be locked. Since the space involved (one node) is limited, the frequency of this problem should be low. The node is relatched and examined for changes (records inserted or deleted). The following outcomes are possible.

1. No change is required to the locks needed to implement the move lock. Proceed with the structural change.
2. The structural change becomes unnecessary. Abort the structural change action.
3. The structural change remains necessary, but different locks are needed to implement the move lock. Request the new locks. If a wait is required, release the node latch and repeat this sequence until all needed locks are held.

4.5 Providing all-or-nothing atomicity

We want our approach to index tree concurrency and recovery to work with a large number of recovery methods. Thus, we indicate what our approach requires from a recovery method, without specifying exactly how these requirements are satisfied.

4.5.1 Logging

We assume that write-ahead logging (the WAL protocol) is used to ensure that actions are atomic, i.e., all or nothing. The WAL protocol assures that actions are logged so as to permit their UNDO, prior to making changes in the stable database.

Our atomic actions are not user-visible and do not involve user-commitment promises. Atomic actions need only be "relatively" durable. That is, they must be durable prior to the commitment of transactions that use their results. Thus, it is not necessary to *force to disk* a "commit" log record when an atomic action completes. This "commit" record can be written when the next transaction commits, forcing the log. This transaction is the first one that might depend on the results of the atomic action. When the log record of the commit of the transaction and the last log record of the atomic action (its "commit" log record) are on disk, both the transaction and the atomic action are durable. This optimization assumes that any transaction which might depend on these results uses the same log.

4.5.2 Identifying an atomic action

Atomic actions must complete or partial executions must be rolled back. Hence, the recovery manager needs to know about atomic actions, as it is the database system component responsible for the atomicity property, i.e., the all-or-nothing execution of the action.

Three possible ways of identifying an atomic action to the recovery manager are as (i) a separate database transaction, (ii) a special system transaction, or (iii) as a "nested top-level action" (Mohan et al. 1992). Our approach works with any of these techniques, or any other that guarantees atomicity. One strength of the method is that it realizes high concurrency while providing independence from the details of the surrounding database system.

5 Multi-action structural changes

The database activity that triggers a structural change is largely isolated from the change itself. It is this isolation that enables the high concurrency of our approach.

Only if the multiple atomic actions involved in a structural change are truly independent can they be scheduled independently. Only then can an intervening system crash interrupt the structural change, delaying its completion for a potentially long period while leaving the Π-tree well-formed.

5.1 Completing structural changes

There is a window between the time a node splits in one atomic action and the index term describing it is posted in

another. Between these atomic actions, a Π-tree is said to be in an intermediate state. These states are, of course, well-formed and can be successfully searched. However, searching a tree in an intermediate state may result in more nodes on the search path or in the existence of underutilized nodes which should be deleted. Hence, we try to complete all structural changes. And, it is not always the case that we have already scheduled atomic actions to do this.

There are at least two reasons why we "lose track" of which structural changes need completion, and hence need an independent way of rescheduling them.

1. A system crash may interrupt a structural change after some of its atomic actions have been executed, but not all. The key to this is to detect the intermediate states during normal processing, and then schedule atomic actions that remove them. Hence, database crash recovery does not need to know about interrupted structural changes.
2. We only schedule the posting of an index term to a single parent. We rely on subsequent detection of intermediate states to complete multi-parent structural changes. This avoids the considerable complexity of trying to post index terms to all parents, either atomically or via the scheduling of multiple atomic actions.

Structural changes are detected as being incomplete by a tree traversal that includes following a side pointer. At this time, we schedule an atomic action to post the index term. In the case of leaf-level tree traversal in a system where move locks may be in place, a transaction following a side pointer must test for such a lock by, for example, attempting to place an instant duration lock incompatible with move locks. If a move lock is detected, no index-posting is scheduled.

Several tree traversals may follow the same side pointer, and hence try to post the index term multiple times. A subsequent node consolidation may have removed the need to post the index term. These are acceptable, because the state of the tree is **testable**. Before posting the index term, we test that the posting has not already been done and still needs to be done.

The need to perform node consolidation is indicated by encountering an underutilized node. At this point, a node consolidation is scheduled. As with node-splitting, the Π-tree state is tested to make sure that the consolidation is only performed once, and only when appropriate.

5.2 Exploiting saved state

Exploiting saved information is an important aspect of efficient index tree structural changes. The bad news of independence is that information about the Π-tree acquired by early atomic actions of the structural change may have changed, and so cannot be trusted by later atomic actions. The Π-tree may have been altered in the interim. Thus, saved information may need to be verified before it is used, and in general, later atomic actions must verify that their execution remains appropriate.

The information that we save consists of search key, nodes traversed on the path from root to data node containing the search key, and the location of the relevant index terms within those nodes. This information can permit us to locate nodes to be restructured without a second search of the nodes on the path, and to find a location within an index node where a new index term is to be inserted or an old one deleted.

To verify saved information, we use state identifiers (Lomet 1990) within nodes to indicate the states of each node. We record these identifiers as part of our saved path. The basic idea is that if a node and its state id (stored in the node) equal a remembered node and state id, then there have not been any updates to the remembered node since the previous traversal. Hence, the remembered descendent can be used, avoiding a second search of the node. (Log sequence numbers are used as state identifiers in many commercial systems.)

Whether node consolidation is possible has a major impact on how we handle saved information. The extent to which we can trust this saved information changes when node consolidation is allowed. We outline the effects of supporting or not supporting node consolidation here.

5.2.1 No-consolidate case

Consolidation Not Supported [CNS] Invariant: *A node, once responsible for a key subspace, is always responsible for the subspace.*

CNS has three effects on our tree operations.

1. During a tree traversal, an index node is searched for an index or sibling term for the pointer to the next node to be searched. We need not hold latches so as to assure the pointer's continued validity. The latch on an index node can be released after a search and prior to latching a child or sibling node. Only one latch at a time is held during a traversal.
2. When posting an index term in a parent node, it is not necessary to verify the existence of the nodes resulting from the split. These nodes are immortal and remain responsible for the key space assigned to them during the split.
3. During a node split, the parent index node to be updated is either the one remembered from the original traversal (the usual case) or a node that can be reached by following sibling pointers. Thus "retraversals" to find a parent always start with the remembered parent. If the state identifier in the parent is the same as the remembered state identifier, the index term is posted to the remembered parent. Should state identifiers be unequal, the parent may have delegated responsibility for part of its subspace to a sibling. But there is a side pointer from the parent to its sibling which can be followed to find the entry of interest. We choose to update only the first parent node encountered that contains an index term for the split node that needs now to include an index term for the new node. Subsequent sibling traversals will complete the updating required for multiple parent nodes.

5.2.2 Consolidate case

Consolidation Possible [CP] Invariant: *A node, once responsible for a key subspace, remains responsible for the subspace only until it is deallocated.*

Deallocated pages are not responsible for any key subspace. When re-allocated, they may be used in any way, including being assigned responsibility for different key subspaces, or being used in other indexes. This affects the "validity" of remembered state. While saved path information can make retraversals of an index tree in later atomic actions very efficient, it needs to be verified before being trusted.

The effect CP has on the tree operations is as follows:

1. During a tree traversal, latch-coupling is used to ensure that a node referenced via a pointer is not freed before the pointer de-referencing is completed. The latch on the referenced node is acquired prior to the release of the latch on the referencing node. Thus, two latches need to be held simultaneously during a traversal.
2. When posting an index term in a parent node, we must verify that the node produced by the split continues to exist. Thus, in the atomic operation that posts the index term, we also verify that the node that it describes exists by continuing our traversal down to this node. When deleting an index term, we consolidate the node into its containing node in the same atomic action as the index deletion.
3. During a node split, the remembered parent node to be updated may have been deallocated. How to deal with this contingency depends upon how node deallocation is treated. There are two strategies for handling node deallocation.
   a) **Deallocation is NOT a node update:** A page's state identifier is unchanged by deallocation. It is impossible to determine by state identifier examination if a page has been deallocated. However, we ensure that the root does not move and is never deallocated. Then, any page reachable from the root via a tree traversal is guaranteed to be allocated. Thus, tree retraversals start at the root. A node on the path is accessed and latched using latch-coupling, just as in the original traversal. Typically, a path retraversal is limited to relatching path nodes and comparing new state ids with remembered state ids, which will usually be equal.
   b) **Deallocation is a node update:** Node deallocation changes not only space management information, but also the page's state identifier to indicate that deallocation has taken place. This requires the posting of a log record and possibly an additional disk access to write the modified page should the page not be reused before it needs to be flushed. However, the remembered parent node in the path will always be allocated if its state identifier has not changed and retraversals can begin from there. If it has changed, however, one must go up the path, setting and releasing latches until a node with an unchanged state id is found or the root is encountered. A path retraversal begins at this node. Since node deallocation is rare, full retraversals of the tree are usually avoided.

5.3 Scheduling atomic actions

Atomic actions that are spawned as a result of a database transaction need to be scheduled to run. Their performance is improved if they can exploit saved state. Thus, in the scheduling of atomic actions, provision is made to associate saved state with these actions.

1. **Required database locks:** locks that were identified as needed ahead of time are indicated. When the action is executed, it will request these locks prior to requesting any latches. This will frequently avoid the need to release and then reacquire leaf node latches.
2. **Saved path:** It is always potentially useful to save the path traversed by earlier atomic actions. Usually, this path information will remain valid, and hence traversals during subsequent actions can be dramatically faster. The saved information in the B^{link}-tree case consists of <node, state id, record location> for each node of the path and a search key. An equal comparison of the saved state id for a node and its present state id replaces the search within the node and permits us to proceed to the next node on the saved path without any check of the node contents. Saved location is useful to avoid searching for the place in an index node where a new index term should be posted because of a split of one of its children.

6 Structural changes

In this section, we present the step-by-step details of our concurrency and recovery algorithm. Tree updates are decomposed into a sequence of atomic actions, one for each level of the Π-tree that is being updated. The details of latching and locking for each atomic action are presented here.

A node split is triggered by an update of the original node. Node consolidation, which makes changes at two levels of the Π-tree and moves information from one node to another is considered to be an update at the level of the parent of the consolidated nodes (where an index term is deleted). Each atomic action is an instance of a single universal action, regardless of the specifics of the update. This program treats both the CP and CNS cases.

6.1 Service subroutines

We identify a number of subroutines that will be invoked as part of the universal action at appropriate places.

6.1.1 Find_Node

Our **Find_Node** returns the address of a node at LEVEL whose approximately contained space includes a KEY. The parent node to this node is left S-latched. Latch-coupling is used with CP, but a parent node latch can be released before acquiring a child or contained sibling node latch with CNS.

This routine handles both new traversals and retraversals. To do this, each traversal updates the saved path associated with the structural change. With CP, retraversals start either at the root (when deallocation is not an update) or else at the lowest unchanged node of the path (when deallocation

is an update). With CNS, the saved parent of a node can be simply latched and used.

When a side pointer is traversed during **Find_Node**, an index-posting action is scheduled for the parent level of the tree. (The root is not allowed to have side pointers.) An exception is made if a move lock is detected in a system using page-oriented UNDO. In this case, no index-posting is scheduled, as the transaction splitting the leaf could still abort and have to UNDO the split. (In systems supporting non-page-oriented UNDO, once the page is split and the latches are released, the split is not undone and no move locks are necessary.)

Similarly (with CP), when an underutilized node is encountered, except at the root level, an index delete action, which also consolidates nodes, is scheduled for the parent level of the underutilized node.

6.1.2 Verify_Split

Verify_Split (needed only with CP) confirms that the node referenced by a new index term still exists. The index NODE to which the term is to be posted has been found and update latched beforehand. If the index term has already been posted, false is returned, indicating that the posting is inappropriate.

Otherwise, the child node which is the original splitting node is S-latched. It is accessed to determine whether a side pointer refers to a sibling node that is responsible for the space that contains the space denoted in the new index term. If not, then the node whose index term is being posted has already been deleted and false is returned. If so, true is returned, indicating that index-posting remains appropriate.

If a sibling exists that is responsible for space containing, but not equal to the space denoted in the index term being posted, this sibling becomes the one whose index term is posted. (This happens if the original containing node is split again before the posting of the index term of the first split.) The S-latches are dropped here so that the U latch on the parent node can be safely promoted to an X-latch. The new node whose index term is being posted cannot be consolidated while a latch is held on a parent.

6.1.3 Split_Node

Split_Node divides the contents of a current node between the current node and a new node. It is invoked whenever the current node has insufficient space to absorb an update. The current node has been U-latched beforehand. If the current node is a leaf node, and non-page-oriented UNDO is not supported, a move lock is requested. If a wait is required, the U-latch on the current node is released. It is reacquired after the move lock has been granted.

The U-latch on the current node is promoted to X. The space management information is X-latched and a new node is allocated. The key space and contents directly contained by the current node are divided, such that the new node becomes responsible for a subspace of the key space. A sibling term is placed in the current node that references the new node and its key subspace. The change to the current node and the creation of the new node are logged. These changes are ascribed to the surrounding atomic action or database transaction.

If the split node is not the root, an index term is generated containing the new node's address as a child pointer, and an index-posting operation is scheduled for the parent of the current node. In the case of a split leaf node in a page-oriented UNDO system, the index-posting is not scheduled until after the transaction commits.

If the split node is the root, a second node is allocated. The current node's contents are removed from the root and put into this new node. A pair of index terms is generated that describe the two new nodes, and they are posted to the root. These changes are logged.

6.1.4 Verify_Consolidate

Verify_Consolidate checks whether a sparse node can be consolidated with another node. The parent of the sparse node is already U-latched. If the consolidation has already taken place, **Verify_Consolidate** returns, indicating that consolidation is inappropriate.

We prefer to treat the sparse node as the contained node, and move its contents to its containing node as there is less data to move. This is possible, space permitting, when the sparse node is a single parent node and its containing node is a child of its parent. In this case, containing and contained nodes are uniquely identified and **Verify_Consolidate** returns, indicating which nodes are to be consolidated.

When the above condition does not exist, we make the sparse node the containing node in the consolidation and try to find an appropriate contained node. There may not be a unique contained node, and one may not even exist. Either return, indicating that consolidation is inappropriate, or select one contained node, and attempt consolidation with it. No latches are left on any nodes checked.

6.1.5 Consolidate_Nodes

Consolidate_Nodes absorbs a contained node into its containing node. It is invoked as part of the atomic action that deletes the contained node index term. The single parent of the contained node has been U-latched previously. First the containing node is X-latched, then the contained node. The containing node is checked to determine if it has a side pointer to the contained node and has sufficient space to absorb the contained node contents. If not, consolidation is canceled, the X-latches are dropped, and the parent U-latch is promoted to X so as to enable the reinsertion of the previously deleted index term. Otherwise, consolidation continues.

If the nodes to be consolidated are leaf nodes, a move lock is requested. If a wait is required for the move lock, the X-latches on the leaf nodes are released, but the U-latch on the parent is retained. When the move lock is obtained, **Consolidate_Nodes** is re-executed from the start.

The contents of the contained node are then moved to the containing node. The appropriate space management information is X-latched, and the contained node is deallocated.

The changes to containing and contained nodes are logged and ascribed to the node consolidate atomic action. Then X-latches are dropped.

6.2 The universal action

One should regard our universal action (called **Universal**) as encompassing the operations necessary to perform an update at exactly one level of the Π-tree. The form of the update will vary. During its execution, however, it may be necessary to make a structural change to the Π-tree.

Universal takes the following arguments.

- LEVEL of the tree to be updated;
- KEY value for the search; The KEY value can be more complex than a simple byte string value. Such complexity is ignored here. For example, see Evangelidis et al. (1995) and Evangelidis et al. (1997) for the specifics of how this works with hB-trees.
- PATH that was saved for index-posting and for consolidation.
- LOCKS that need to be acquired in order for the operation to complete;
- OPERATION which is one of (i) posting an index term, (ii) dropping an index term and consolidating two nodes, or (iii) accessing or updating a leaf node. (The description below is written in terms of updates to simplify the discussion. The access case is simpler and uses S-latches instead of U-latches.) Again, we ignore the complexities of dealing with specific data structures, both those representing the node and those representing the update.

When dealing with a leaf node (case iii), **Universal** executes as part of a database transaction. (Consolidating leaf nodes and dropping the resulting index term is not considered a leaf node operation and does not execute as part of a database transaction.) However, posting or deleting index terms for index nodes are all in short duration independent atomic actions.

Universal performs the following steps.

Request Initial Locks. If database locks are known to be needed, get them now, prior to holding any latches. This avoids having to release latches subsequently in order to get them.

Search. Execute **Find_Node** to find a node(NODE) at LEVEL whose approximately contained space includes KEY. For CP, the parent of NODE is left S-latched, ensuring that re-searching the tree is avoided and that node consolidations involving children of this node will not occur during this action. (When the update is to the root, do not invoke **Find_Node** and do not latch any nodes.)
Using a KEY, from a search argument, instead of attempting to locate a subspace means that only one parent of a split child node will be found. Most of the time, i.e., for single parent nodes, updating this node will complete the index-term-posting for a split. When multiple parent nodes exist for a split node, several executions of **Universal** may be needed before all parents are updated. And these are scheduled as a result of searches with other keys that required side pointer traversals.

Get Target Node. U latch NODE. Traverse sibling pointers, U-latching each node, and for CP, latch coupling, until the node is found whose directly contained space includes KEY. Set NODE to be this node. NODE is left U-latched. U-latches are used because we do not know which node on this level will be updated until we read it. For CP, the parent of NODE can be unlatched now if NODE is above the leaf level. If NODE is at leaf level, NODE may be unlatched later to wait for database locks, and its parent latch will be needed to make sure NODE is not deallocated.

Verify Operation Need. Verify that the operation intended is still appropriate.
- Leaf node UPDATE: the action is always appropriate.
- Index POSTING: invoke **Verify_Split** to verify that posting the index term remains appropriate.
- Index DROPPING: invoke **Verify_Consolidate** to verify that deleting an index term and consolidating nodes remains appropriate.

If the action is now inappropriate, terminate the atomic action.

Space Test. Test NODE for sufficient space to accommodate the update. If sufficient, then X-latch NODE and proceed to **Request Remaining Locks**. Otherwise, split NODE by invoking **Split_Node**. (This will not occur for index-dropping.) Then check which resulting node has a directly contained space that includes KEY, and make that NODE. This can require descending one more level in the Π-tree should NODE have been the root where the split causes the tree to increase in height. Release the X-latch on the other node, but retain the X-latch on NODE. Repeat this **Space Test** step.

Request Remaining Locks. If NODE is a leaf node and database locks have not been acquired because it was not known which were needed a priori, they are requested here. If a wait is necessary, the U-latch on NODE is released. After the database lock(s) are acquired, return to **Get Target Node**.

Update Node. Update NODE by performing the requested operation. Post a log record describing the update to the log. If NODE is a leaf node, this log record is associated with the database transaction. Otherwise, it is associated with an independent atomic action. If the update is not an index-dropping, proceed to **Sparse Node Check**. Otherwise, demote the X-latch on NODE to U and proceed to **Consolidate**.

Consolidate. Invoke **Consolidate_Nodes** to consolidate the lower level nodes. If it fails, cancel the index-dropping atomic action, which undoes the prior NODE update. Note that the U-latch retained on the index node permits us to perform the UNDO by promoting the U-latch to X and reinserting the dropped index term.

Sparse Node Check. If NODE is now underutilized and NODE is not the root, schedule an index dropping atomic action to delete an index term in the parent of NODE by consolidating NODE with a sibling. If NODE is the root and it is underutilized, but has more than one child, we let this condition persist.
If NODE is the root, and it has only a single child, we can schedule a special atomic action that consolidates this child with the root, thus reducing the height of the tree by

one. This action is similar to other node consolidates in that it must: (i) test that it is still appropriate, (ii) acquire appropriate latches and the necessary move lock, and (iii) move the contents of the child node into the root, (iv) deallocate the child node, and (v) log the effects of the action. It differs from ordinary consolidations only in that the parent serves as the containing node, and that no index-term-dropping is involved.

Complete. Release all latches still held by **Universal**. If **Universal** was an independent atomic action, release its database locks and commit the action by writing an appropriate commit record. If this is a leaf node update, however, the database locks are held by the surrounding database transaction, and remain held. Note that move locks are never acquired in non-page-oriented UNDO systems, so the releasing of the page latches for a split at leaf level commits the split (makes it a nested top-level action) as in (Mohan and Levine 1992).

Leaf-level splits in page-oriented UNDO systems are part of database transactions. Other structural changes take place within independent atomic actions. These actions only execute for a short duration.

7 Applicability to various search structures

We have used B$^{\text{link}}$-trees as a running example of how our concurrency method works. This is the simplest case, since only a single attribute is being indexed. In this section, we describe briefly how the TSB-tree and the hB$^{\Pi}$-tree can be described as Π-trees, and hence exploit our concurrency control and recovery method.

7.1 The TSB-tree

A TSB-tree (Lomet and Salzberg 1989) provides indexed access to multiple versions of key-sequenced records. As a result, it indexes these records both by key and by time. We take advantage of the property that historical nodes (nodes created by a split in the time dimension) never split again. This implies that the historical nodes have constant boundaries and that key space is refined over time.

Splits in the TSB-tree can be made in two dimensions, either by time or by key. In Fig. 3, the region covered by a current node after a number of splits is in the lower right-hand corner of the key space it started with. A time split produces a new (historical) node, with the original node directly containing the more recent time. A key split produces a new (current) node, with the original node directly containing the lower part of the key space.

With time splits, a history sibling pointer in the current node refers to the history node. The new history node contains a copy of prior history sibling pointers. These pointers can be used to find all versions of a given record.

With key splits, a key sibling pointer in the current node refers to the new current node containing the higher part of the key space. The new node will contain not only records with the appropriate keys, but also a copy of the history sibling pointer. This pointer preserves the ability of the current

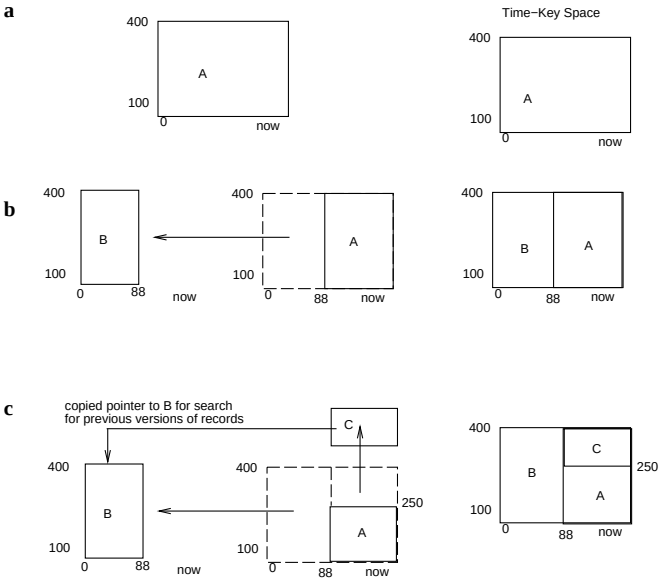

Fig. 3. The time-split B-tree. **a** First, A covers time from 0 to now and keys from 100 to 400 (400 is the maximum key value.) **b** A splits by time = 88 and delegates the previous time interval to B. A is still responsible for all of time, but only directly contains time after or equal to 88. **c** Now A splits by key. A now contains two sibling pointers. A is the container for C and A is the container for B. A is still responsible for all time and for keys between 100 and 400, but directly contains only records which intersect the time after 88 and the keys between 100 and 250

node directly containing a key space to access history nodes that contain the previous versions of records in that space. This split duplicates the history sibling pointer. It makes the new current node responsible for not merely its current key space, but for the entire history of this key space.

In the TSB-tree, many nodes may be multi-parent nodes, but these are all historical nodes. No historical nodes ever split and nodes are never consolidated. Thus, in the TSB-tree, the existence of multi-parent nodes and the fact that more than one history sibling pointer may point to the same historical node causes no extra difficulties.

7.2 The hB-tree

In the hB-tree (Lomet and Salzberg 1990), the idea of containing and contained nodes is explicit and is described with k-d-tree fragments. The "External" markers can be replaced with the addresses of the nodes which were extracted, and a linking network established with the desired properties. In addition, when the split is by a hyperplane, instead of eliminating the root of the local tree in the splitting node, as in Lomet and Salzberg (1990), one child of the root (say, the right child) points to the new sibling containing the contents of the right subtree. This makes the treatment of hyperplane splits consistent with that of other splits. This is illustrated in Fig. 4.

An hB-tree with these modifications is called an hB$^{\Pi}$-tree. A complete description and explanation of hB$^{\Pi}$-tree concurrency, node splitting, and node consolidation is given in Evangelidis et al. (1997).

Any time a node containing entries representing spaces is split, it is possible for the split to also split the space de-

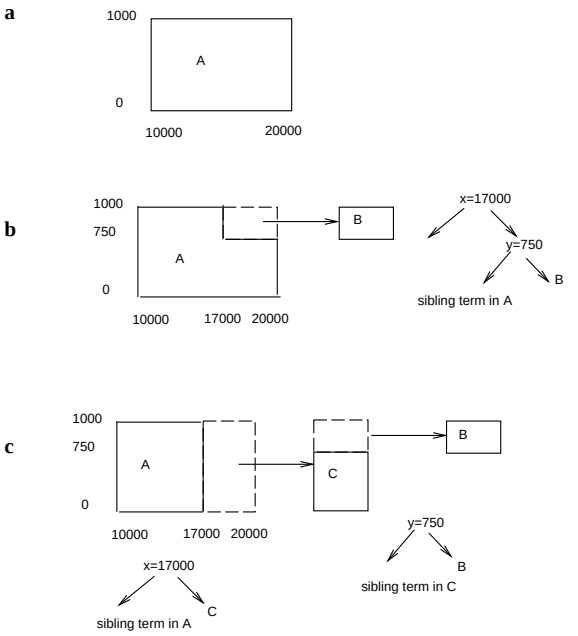

Fig. 4. An hB-tree index showing the use of k-d trees for sibling terms. External markers (showing what spaces have been removed in creating "holes") have been replaced with sibling pointers. **a** To begin, A contains the entire key space with x ranging from 10000 to 20000 and y ranging from 0 to 1000. **b** Then a corner split is made in A. A k-d tree is the sibling term which indicates the space for which B is responsible. **c** A splits again, using the k-d tree to make the split. The split is at the root of the k-d tree. A is now only directly containing points with x-coordinate less than 17000. A is still responsible for the whole space

scribed by one of the entries. This is an intrinsic problem for multi-attribute methods, where it is almost always the case that no simple partitioning of entries into simply described spaces exists that does not split an entry. One variant of the hB$^{\Pi}$-tree splitting algorithm disallows such splits. But the most general variant ("split anywhere") of the hB$^{\Pi}$-tree splitting algorithm solves this problem by "clipping" the index terms whose spaces are split, producing nodes with multiple parents. However, with hB$^{\Pi}$-trees, at most one index term needs to be clipped per index node split, which minimizes the occurrence of the problem.

8 Conclusion

Our approach to index tree structural changes provides high concurrency, while being usable with many recovery schemes and with many varieties of index trees. We have described it in an abstract way, which emphasizes its generality and hopefully makes the approach understandable.

Recently, there has been a surge of interest in multi-attribute search structures. Such structures are now being used for data mining and data warehousing. Many different attributes may be needed for determining patterns over time and for decision support. Thus, algorithms which work with spatial and temporal indexes as well as the usual B^{+}-tree may be of greater importance.

Our techniques permit multiple concurrent structural changes. In addition, all update activity and structural change activity above the data level executes in short independent atomic actions which do not impede normal database activity. Only leaf-node-splitting might execute in the context of a database transaction. Should the recovery method support "logical" UNDO, in which updated records can move while still being subject to UNDO recovery, structural changes even at the leaf level can occur outside of the database transaction. If an insertion triggers a major structural modification, it is useful to be able to postpone the greater part of the modification until after the transaction completes. In most cases, the postponed actions will be performed in a timely manner, soon after the triggering transaction, so that other transactions will not have to follow sibling pointers.

Acknowledgements. This work was partially supported by NSF grants IRI-88-15707 and IRI-91-02821 and IRI-93-03403.

References

Bayer R, Schkolnick M (1977) Concurrency of operations on B-trees. *Acta Inf* 9:1–21

Evangelidis G, Lomet D, Salzberg B (1995) The hB$^{\Pi}$-tree: a modified hB-tree supporting concurrency, recovery and node consolidation. In: Dayal U, Gray P, Nishio S (eds) *Proc. Very Large Databases Conf.*, Zurich. Morgan-Kaufman, San Francisco, pp 551–561

Evangelidis G, Lomet D, Salzberg B (1997) The hB$^{\Pi}$-tree: a multiattribute index supporting concurrency, recovery and node consolidation. *VLDB J* 6(1): 1–31

Gray JN, Lorie RA, Putzolu GR, Traiger IL (1976) Granularity of locks and degrees of consistency in a shared data base. In: *IFIP Working Conf on Modeling of Data Base Management Systems*, pp 1–29. Reprinted in: Stonebraker M (ed) *Readings in Database Systems*. Morgan Kaufman, San Francisco, 2nd edn 1994, pp 181–208

Gray J, Reuter A (1993) *Transaction Processing: Techniques and Concepts*. Morgan Kaufman, San Mateo, Calif.

Guttman A (1984) R-trees: A dynamic index structure for spatial searching. In: *Proc. ACM SIGMOD Conf.*. pp 47–54. Reprinted in: Stonebraker M (ed) *Readings in Database Systems*. Morgan Kaufman, San Francisco, 2nd edn 1994, pp 125–135

Lehman P, Yao SB (1981) Efficient locking for concurrent operations on B-trees. *ACM Trans Database Sys* 6:650–670

Lomet DB (1977) Process structuring, synchronization, and recovery using atomic actions. In: *Proc ACM Conf on Language Design for Reliable Software*, SIGPLAN Notices 12,3 pp 128–137

Lomet DB (1980) Subsystems of processes with deadlock avoidance. *IEEE Trans Software Eng* SE-6:297–304

Lomet D, Salzberg B (1989) Access methods for multiversion data. In: Clifford J, Lindsay B, Maier D (eds) *Proc ACM SIGMOD Conf*, Portland. ACM, New York pp 315–324

Lomet DB (1990) Recovery for shared disk systems using multiple redo logs. *Digital Equipment Corp Technical Report* CRL90/4 Cambridge Research Lab, Cambridge, Mass.

Lomet D, Salzberg B (1990) The hB-tree: a multiattribute indexing method with good guaranteed performance. *ACM Trans Database Sys* 15:625–658

Lomet D, Salzberg B (1992) Access method concurrency with recovery. In: Stonebraker M (ed) *Proc. ACM SIGMOD Conf.*, San Diego. ACM, New York, pp 351–360

Lomet DB (1992) MLR: A recovery method for multi-level systems. In: Stonebraker M (ed) *Proc. ACM SIGMOD Conf.*, San Diego. ACM, New York, pp 185–194

Mohan C (1990) ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-tree indexes. In: McLeod D, Sacks-Davis R, Schek H (eds) *Proc. Very Large Databases Conf.* Brisbane. Morgan Kaufman, Palo Alto, pp 392–405

Mohan C, Haderle D, Lindsay B, Pirahesh P, Schwarz P (1992) ARIES: a transaction recovery method supporting fine-granularity locking and

partial rollbacks using write-ahead logging. *ACM Trans Database Sys* 19(1): 94–162

Mohan C, Levine F (1992) ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. In: Stonebraker M (ed) *Proc. ACM SIGMOD Conf.* San Diego. ACM, New York, pp 371–380

Sagiv Y (1986) Concurrent operations on B* trees with overtaking. *J Comput Sys Sci* 33:275–296

Salzberg B (1985) Restructuring the Lehman-Yao tree. *Northeastern University Technical Report* TR BS-85-21, Boston, Mass.

Shasha D, Goodman N (1988) Concurrent search structure algorithms. *ACM Trans Database Sys* 13:53–90

Weikum G (1986) A theoretical foundation of multi-level concurrency control. *Proc ACM PODS Conference*, Cambridge, Mass. ACM, New York, pp 31–42

Weikum G, Hasse C, Broessler P, Muth P (1990) Multi-level recovery. *Proc ACM PODS Conf*, Nashville. ACM, New York, pp 109–123

Zou C, Salzberg B (1996) On-line Reorganization of Sparsely-Populated B^+-trees. In: Jagadish HV, Inderpal Singh Mumick (eds). ACM, New York, pp 115–124