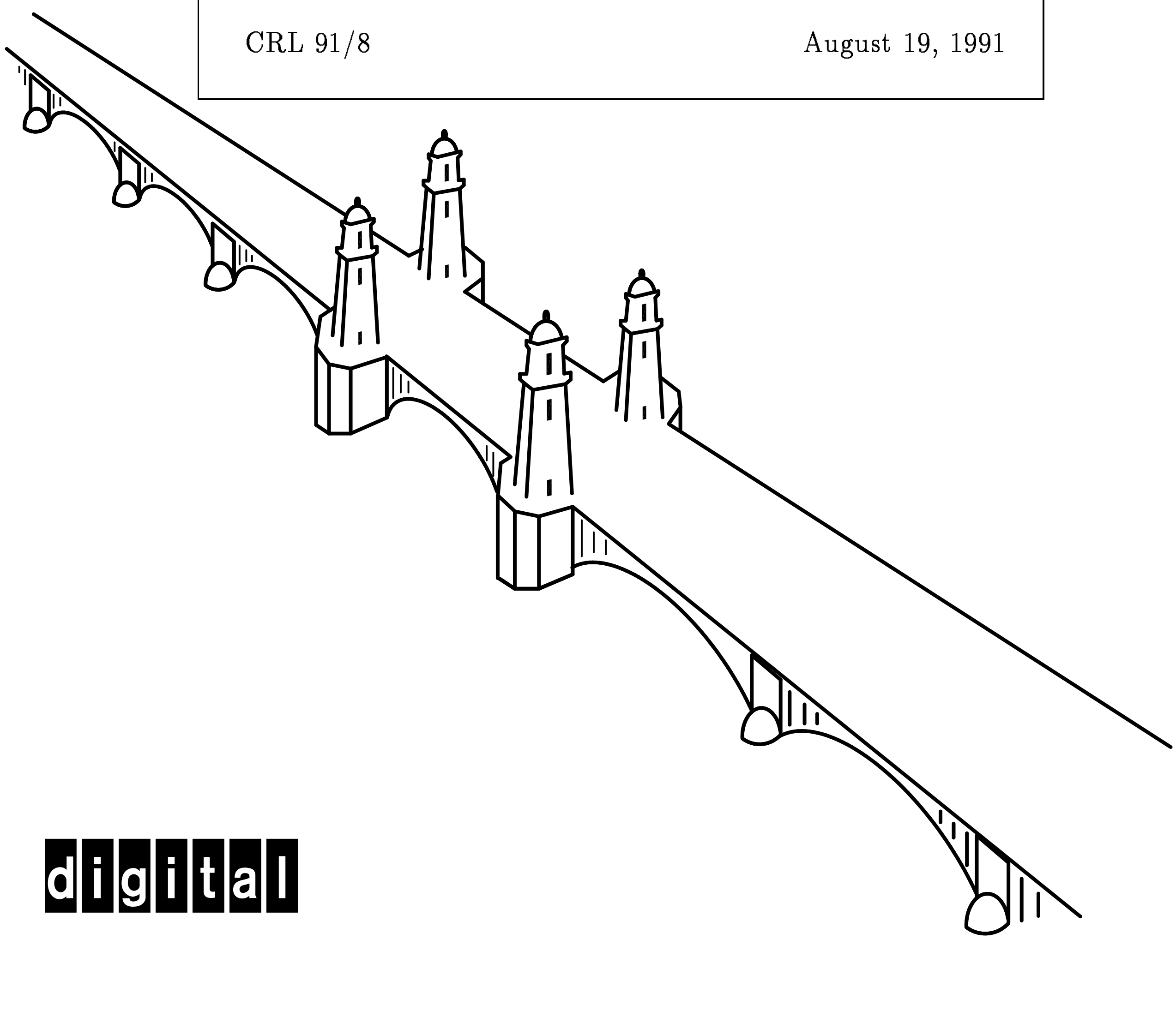
Concurrency and Recovery for Index Trees

David Lomet   Betty Salzberg

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/8   August 19, 1991



CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

Concurrency and Recovery for Index Trees

David Lomet Betty Salzberg [1]

Digital Equipment Corporation
Cambridge Research Lab

CRL 91/8 August 19, 1991

Abstract

Providing high concurrency in B^+-trees has been studied extensively. But few efforts have been documented for combining concurrency methods with a recovery scheme that preserves well-formed trees across system crashes. We describe an approach for this that works for a class of index trees that is a generalization of the B^{link}-tree. A major feature of our method is that it works with a range of different recovery methods. It achieves this by decomposing structure changes in an index tree 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.

Keywords: concurrency, recovery, indexing, access methods, B-trees

[1]College of Computer Science, Northeastern University, Boston, MA. This work was partially supported by NSF grant IRI-88-15707 and IRI-91-02821

1 Introduction

1.1 Background

The higher a node is in an index tree, the more likely it is to be accessed. In structures such as the B^+-tree, where all data is stored in the leaves, upper level index nodes are only written when the shape of the tree changes. It is important then that these structure changes hold only short-term locks on index nodes to facilitate concurrent access via the index.

The subject of concurrency in B^+-trees has a long history [1, 6, 13, 14, 15, 16]. Most papers, with the exception of [13], have not treated the problem of system crashes during structure changes. In this paper, we show how to manage **concurrency and recovery** for a wide class of index tree structures, single attribute, multiattribute, and versioned.

We have three goals.

1. Provide practical concurrency techniques for a wide class of index trees.

2. Provide an abstract description of our approach to make it clearly understandable, and to separate it from particular implementations.

3. Describe how the techniques mesh with surrounding recovery methods. This provides a rationale for choosing among various schemes.

1.2 Our Approach

We define a search structure, called a Π**-tree** , that is a generalization of the B^{link}-tree[6]. Our concurrency and recovery method is defined to work with all search structures in this class. This generality means that our technique has very broad applicability.

The three innovations that make it possible for us to hold only short term locks on non-leaf nodes are the following:

1. Π-tree structure changes consist of a **sequence** of atomic actions[7]. These actions are serializable and 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. Hence, complete structural changes are not serializable. It is the strength of the Π-tree that these intermediate states are well-formed.

2. We define separate actions for performing updates at each level of the tree. A structure change can occur as a result, but the structure change is confined to a single level. Thus, update actions and possible structure changes on non-leaf nodes can be separate from the transaction whose update triggers a structure change. Only re-structuring at the leaves of a tree may need to be within an updating transaction in such a way that locks associated with the re-structuring are held until the end of the transaction..

3. When a system crash occurs during the sequence of atomic actions that constitutes a complete Π-tree structure change, crash recovery takes no special measures. A crash may cause an intermediate state to persist for some time. The structure 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.

1.3 Organization of Paper

Section 2 defines the Π-tree. In section 3, atomic actions are described in general. Section 4 describes how Π-tree structure changes are decomposed into atomic actions, and how to cope with such decomposed changes. Section 5 presents the specific logic of atomic actions. We show how a wide array of search structures can be adapted so as to be forms of Π-trees in section 6. Section 7 is a short discussion of results.

2 The Π-tree

2.1 Structural Description

Informally, a Π-tree is a balanced tree, 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. All these terms are defined more precisely below.

2.1.1 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**.

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. A **level** of the Π-tree is a maximal connected subgraph of nodes and side pointer edges. Each level of the Π-tree is responsible for the entire key space. The first node at each level is responsible for the whole space, i.e. it is the containing node for the whole key space.

2.1.2 Multiple Levels

The Π-tree is split from the bottom, like the B-tree. **Data nodes**(leaves) are at level 0. Data nodes contain only data records 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. Hence, 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 data node. In Π-trees, as in $\mathrm{B^{link}}$-trees, parent nodes are **index nodes** which contain only index terms and/or sibling terms. Parents nodes are at a level one higher than their children. Unlike $\mathrm{B^{link}}$-trees, in the more general Π-trees, the same child can be referred to by two parents. This happens when the boundary of a parent split cuts across a child boundary. Then the union of the spaces for which children nodes are responsible may be larger than (strictly include) the space the index node directly contains.

2.1.3 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 de-allocated 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[6]. 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 [16], 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 the subspace of its containing node for which its referenced node is responsible.
3. each index term correctly describes the subspace of the index node for which its referenced child node is responsible;
4. the space which an index node directly contains, i.e., which it has not delegated to a sibling, is a subspace of the union of spaces that its child nodes are responsible for at the next lower level;
5. the lowest level nodes are data nodes.
6. a root exists that is responsible for the entire search space.

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

2.2 Π-tree Operations

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 structure changes into atomic actions. This is described in section 5, after we have discussed how to realize atomic actions and the difficulty of dealing with multiple atomic action structure changes in general.

2.2.1 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**. 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. This node will usually, but not always, contain the search point, at least for the Π-trees that we build. 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 data node level of the tree is reached. The record for the search point will be present in the data node whose directly contained space includes the search point, if it exists at all.

2.2.2 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) intersects 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 section 4.1).

A node split has the following steps:

1. Allocate space for the new node.

2. Partition the subspace directly contained by the original node into two parts. The original node continues to directly contain one part. The other part is delegated to the new sibling node. There is wide discretion in this step. The goal is usually to achieve as close to an even split of the original node's contents as possible, but there may be other considerations.

3. If the node being split is a data node, place in the sibling node all of the original node's data that are contained in the delegated space. Include any sibling terms to subspaces for which the new node is now

responsible. Remove from the original node all the data that it no longer directly contains. This partitions the data. (What is dealt with here is point data.)

4. If the node being split is an index node, we include in each node the index terms that refer to child nodes whose approximately contained spaces intersect the space directly contained by the node. 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.

5. Put a sibling term in the original node that refers to the new node.

6. Schedule the posting of an index term describing the split to the next higher level of the tree. The index term describes the new node and the space for which it 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 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 data 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.

The entries of index nodes denote subspaces, not merely points. 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.

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 needn't 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 structure changes. Using this mechanism does not usually increase the cost of the structure 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.

2.2.3 Node Consolidation

Node consolidation is scheduled when a node's storage utilization drops below some threshold. This threshold can be fairly arbitrary. It is desirable, however, to consolidate nodes before their utilization becomes too low so as to maintain adequate overall utilization.

When a node becomes under-utilized, it may be possible to consolidated it with either its containing node or one of its contained nodes. 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 de-allocated. For this to work well,

- 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. This is important as a node cannot be deleted until all references to it have been purged. It is also important so as to be sure that others cannot reference the node being deleted via a different path. The latch on the parent node (see below) protects the contained node from such referencing. 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 refering to multi-parent nodes. All other nodes are what we call **single parent nodes** and are subject to consolidation. We schedule consolidations only for nodes known to be single parent nodes.

In the hB-tree, the number of multi-parent nodes is small. No more than one such node is created per index node split. A subsequent split of a multi-parent node may cause the new (contained) node to be a multi-parent node. While only one of containing or contained node has directly contained space that intersects multiple parents' directly contained spaces, because we do not promptly post index terms, both nodes may end up being referenced by more than one parent. In B^{link}-trees, no multi-parent nodes exist. 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 causes no extra difficulties. For arbitrary Π-trees, there is no bound on the number of multi-parent nodes.

3 Atomic Actions for Updating

We need to assure that atomic actions are correctly serialized and have the all or nothing property required of them. How this is done is described in this section.

3.1 Latching for Atomic Actions

3.1.1 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**. Latches are short-term low-cost locks for which the holder's usage pattern guarantees the absence of deadlock. Latches normally come in two modes, **share(S)** mode which permits others with S latches to access the latched data simultaneously, and **exclusive(X)** mode which prevents all other access so that update can be performed.

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. With latches, the user assumes responsibility for ensuring that deadlock is avoided. Thus, latch acquisition does not involve the lock manager.

For deadlock avoidance, each atomic action claims(declares) the set of resources that it will use. Deadlock is avoided by PREVENTING cycles in a "potential delay" graph [8] which includes conflicts between both locks and claims. If resources are ordered and locked in that order, the potential delay graph can be kept cycle-free without materializing it. The claimed resources consist, implicitly, of those with higher numbers in the ordering than the last(highest) resource that was locked. As new, higher numbered resources are locked, the claim on lower numbered resources is implicitly relinquished, except for those explicitly locked.

Promoting a previously acquired lock violates the ordering of resources and compromises deadlock avoidance. Lock promotion is the most common cause of deadlock [5]. 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 [4] support lock promotion by retaining an exclusive claim on a resource that is currently shared [8]. They allow sharing by readers, but conflict with X or other U latches. An atomic action is not allowed to promote from a 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 locked by the requestor of the lock promotion. The rule that we observe is that the promotion request is not made while the requestor holds latches on higher numbered resources.

3.1.2 Latch Acquisition

Since a Π-tree is usually accessed in search order, we can 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. The first action is terminated and a second atomic action is initiated to complete the structure change.

3.1.3 Early (non-2PL) Release of Latches

In the absence of knowledge about concurrent atomic actions, using two phase locking is not only sufficient, it is also necessary, in order to guarantee serializability via locking [2]. However, when dealing with index trees, the types of possible atomic actions are known. Because of this, there are circumstances in which early release of latches does not compromise serializability.

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 serializability even when a higher level node has been changed.

3.2 Interaction with Database Transactions

3.2.1 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.

Should holders of database locks be required to wait for latches on data 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 updates record R1 in node N1 and releases its latch on N1 while holding its database lock on R1. Transaction T2 latches N1 in X mode and tries to update R1. It must wait. Transaction T1 now tries to update a second record in N1 and is forced to wait **for the N1 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 data nodes whenever we wait for database locks. However, latches on index nodes may be retained. Except for data node consolidation, no atomic action or database transaction **both**:(i) holds database locks; and (ii) uses other than S latches above the data 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.

3.2.2 Move Locks

Data 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, the records cannot be moved until this update transaction commits or aborts. In this case, a **move** lock is required that conflicts with any such update to prevent records from being moved until updating transactions complete. Since reads do not require undo, concurrent reads can be tolerated. Hence, move locks can be share mode 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 specifics of the lock manager. 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.

When the move lock is realized as a set of record locks, the need to wait for one of these locks means that the node latch 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 re-latched 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 structure change.

2. The structure change becomes unnecessary. Abort the structure change action.

3. The structure 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.

3.3 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.

3.3.1 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 a "commit" record to the log 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.

3.3.2 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" [12]. 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.

4 Multi-Action Structure Changes

The database activity that triggers a structure change is largely isolated from the change itself. It is this isolation that enables the high concurrency of our approach, independent of the particular recovery method that is used by the database. (This is in contrast to the ARIES/IM method [13], where the success of the method depends in an essential way on the fact that the ARIES recovery method is used.) Isolation results from dividing a structure change into a number of atomic actions.

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

4.1 Completing Structure 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 structure 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 structure changes need completion, and hence need an independent way of re-scheduling them.

1. A system crash may interrupt a structure 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 structure 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 structure 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.

Structure 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. 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.

4.2 Exploiting Saved State

Exploiting saved information is an important aspect of efficient index tree structure changes. The bad news of independence is that information about the Π-tree acquired by early atomic actions of the structure 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 is particularly of interest 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. We record state identifiers [9] that indicate the states of each node as well in order to perform the necessary verification. 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. State identifier checking is always useful, and is essential in the consolidation case (below) in verifying the validity of the saved information.

Whether node consolidation is possible has a major impact on how we handle this information. More than an additional "node consolidation" operation is involved. The invariants upon which other atomic actions depend are altered. Thus, the extent to which we can trust this saved information changes.

4.2.1 No Consolidate Case

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

This 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 "re-traversals" to find a parent always start with the remembered parent. State identifier equality can be used to avoid a second node search. 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. The vast

majority of nodes will have only a single parent, however, and this index term posting will complete the structure change.

4.2.2 Consolidate Case

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

De-allocated nodes 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 re-traversals of an index tree in later atomic actions very efficient, it needs to be verified before being trusted.

The effect this 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 de-allocated. How to deal with this contingency depends upon how node de-allocation is treated. There are two strategies for handling node de-allocation.

   (a) **De-allocation is NOT a Node Update:** A node's state identifier is unchanged by de-allocation. It is impossible to determine by state identifier examination if a node has been de-allocated. However, we ensure that the root does not move and is never de-allocated. Then, any node reachable from the root via a tree traversal is guaranteed to be allocated. Thus, tree re-traversals 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 re-traversal is limited to re-latching path nodes and comparing new state ids with remembered state ids, which will usually be equal.

(b) **De-allocation is a Node Update:** Node de-allocation changes not only space management information, but also the node's state identifier to indicate that de-allocation has taken place. This requires the posting of a log record and possibly an additional disk access. However, the remembered parent node in the path will always be allocated if its state identifier has not changed and re-traversals 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 re-traversal begins at this node. Since node de-allocation is rare, full re-traversals of the tree are usually avoided.

4.3 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 re-acquire data 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 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.

5 Structure Changes

There are many ways to realize Π-tree structure changes in detail. In this section, we describe one way of doing this. Tree updates are decomposed into a sequence of atomic actions, one for each node of the Π-tree that is being updated. 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.

5.1 Service Subroutines

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

5.1.1 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 re-traversals. To do this, each traversal updates the saved path associated with the structure change. With CP, re-traversals start either at the root (when de-allocation 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.) 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.

5.1.2 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 a sibling exists that is responsible for space containing, but not equal to the space denoted in the index term being posted, latch coupled searching of siblings continues until either it is determined that the node denoted by the new index term has been deleted (the prior case) or else that the node is found. In this case, true is returned, indicating that index posting remains appropriate. 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.

5.1.3 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 data node, then a move lock is requested. If a wait is required, the U latch on the current node is released. It is re-acquired 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. 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.

5.1.4 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 treate 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.

5.1.5 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 it has sufficient space to absorb the contained node contents. If not, consolidation is cancelled, 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 data nodes, a move lock is requested. If a wait is required for the move lock, the X latches on the data 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 de-allocated. The changes to containing and contained nodes are logged and ascribed to the node consolidate atomic action. Then X latches are dropped.

5.2 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 structure 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 [3] for the specifics of how this works with hB-trees.

- 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 data 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 data node, **Universal** executes 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. 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.

Verify Operation Need: Verify that the operation intended is still appropriate.

- Data 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 data 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 data 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 re-inserting the dropped index term.

Sparse Node Check: If NODE is now under-utilized 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 under-utilized, 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) de-allocate 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 data node update, however, the database locks are held by the surrounding database transaction, and remain held.

Note that the description above causes data node splits to be part of database transactions, such that the split is not committed until the transaction commits. This restricts concurrency somewhat at the data node level. However, it is this feature that permits the method to be used with many recovery schemes. It is possible to decompose the atomic actions so that data node splits can commit independently of the surrounding database transaction. However, it then becomes essential that the recovery scheme used be able to cope, e.g., as ARIES does, with records that have moved.

Other structure changes take place within independent atomic actions. These actions only execute for a short duration, and while occasionally having to wait, even for a database lock, they never hold latches that prevent normal database activity. The latches act to serialize other such independent atomic actions. Normal database activity commutes (is compatible) with these latches.

6 Applicability to Various Search Structures

We have used $\mathrm{B}^{\mathrm{link}}$-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 discuss a number of multi-attribute search trees, showing how they can be described as Π-trees, and hence exploit our concurrency control and recovery method.

6.1 Hyperplane Split Indexes

A multiattribute structure where all splits (both at the index and data nodes) are made along hyperplanes, can be regarded as a Π-tree. One direction is considered to be the "contained" region and the other to be the "containing". For example, we may assume the region with the higher values of the split attribute is the contained region. We can use the coordinates of the corners to indicate the boundaries of split-off regions in an arbitrary k-dimensional hyperplane-splitting structure as illustrated in Figure 1. This is a B^{link}-tree in the one-dimensional case.

6.2 The TSB-tree

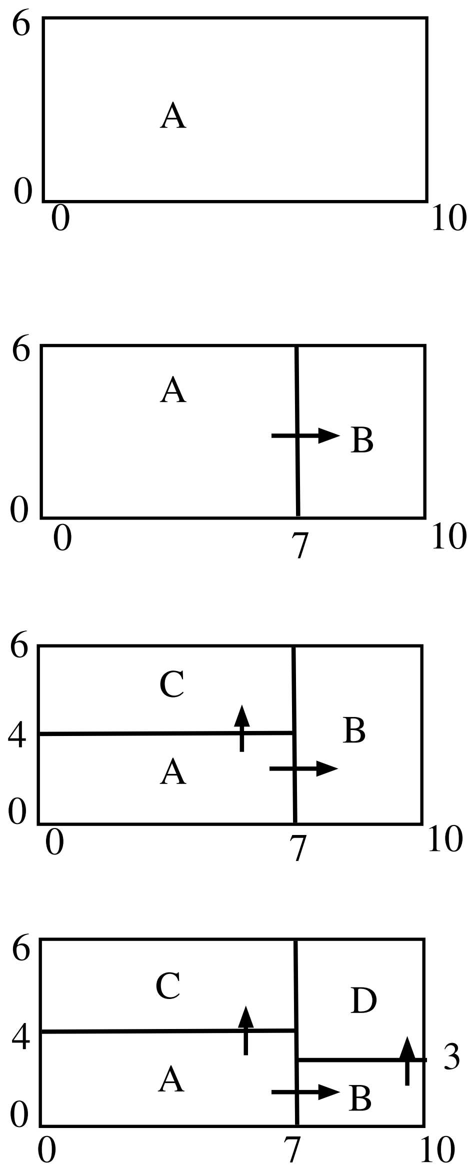
A TSB-tree [10] provides indexed access to multiple versions of key sequenced records. As a result, it indexes these records both by key and by time. We take advantage of the property that historical nodes (nodes created by a split in the time dimension) never split again. This implies that the historical nodes have constant boundaries and that key space is refined over time.

The index nodes have exact boundaries for historical nodes and possibly outdated boundaries for current nodes. At any level of the tree, some of the index entries for splits which have already occurred may be missing.

Splits in the TSB-tree can be made in two dimensions, either by time or by key. In Figure 2, 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 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



1. First split A along hyperplane x = 7

2. In A, make an indication that point with x > 7 are in B using the low and high x and y coordinates.
   B: (7,10, 0, 6)

3. In A, add an indication of the borders of C.
   B: (7,10, 0, 6)
   C: (0, 7, 4, 6)

4. In B, indicate the borders of D.
   D: (7, 10, 3, 6)

Figure 1: General multiattribute index showing form of side pointers. Splits are always by hyperplane. Sibling term space descriptions here are border coordinates. Each node contains sibling terms for every node split from it.

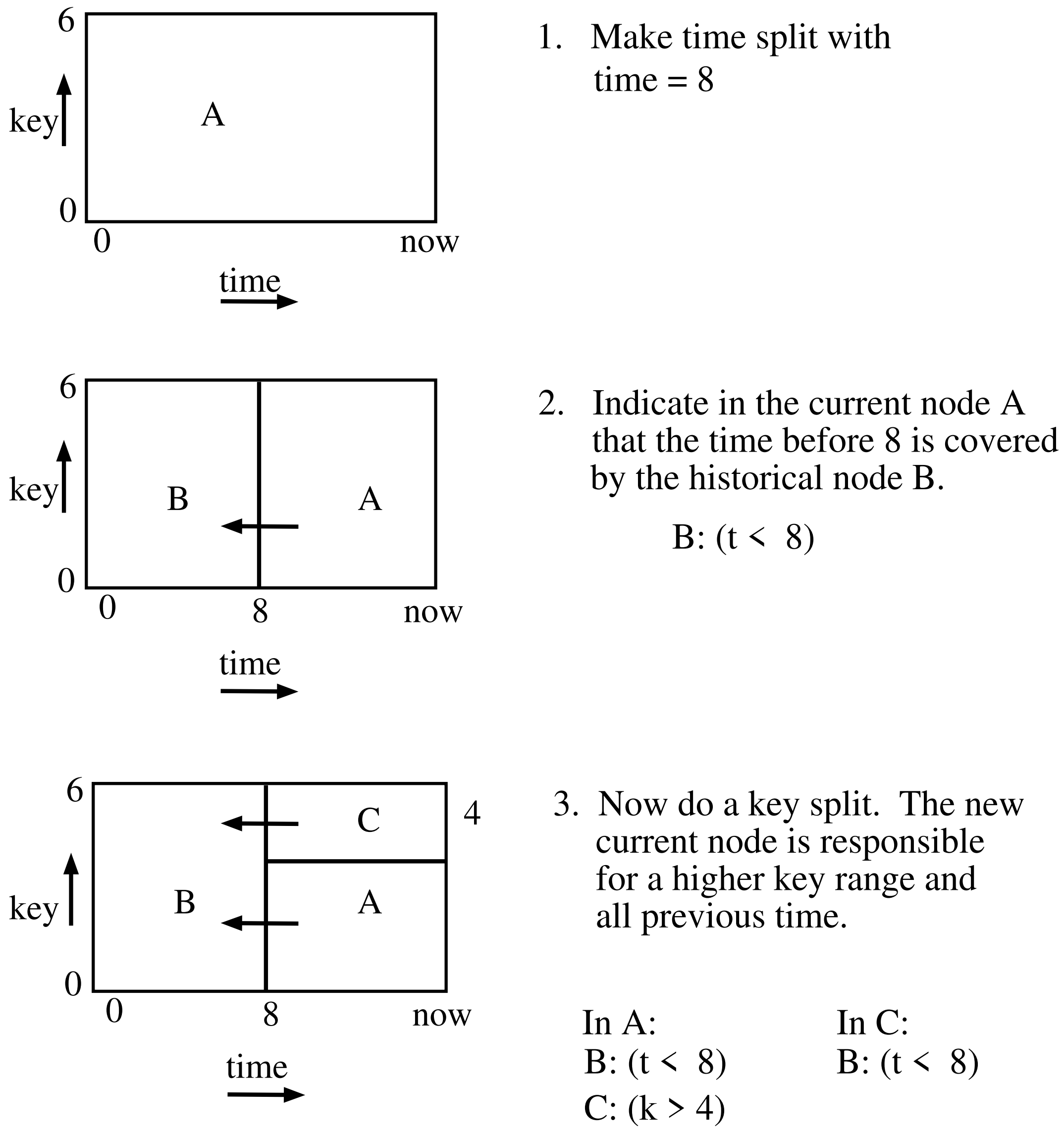


Figure 2: In the Time-Split B-tree, new current nodes contain copies of old history node pointers and old key pointers. New historic nodes contain copies of old history pointers. Current nodes are responsible for all previous time through their historical pointers and all higher key ranges through their key(side) pointers.

current key space, but for the entire history of this key space.

6.3 The hB-Tree

In the hB-tree [11], the idea of containing and contained nodes is explicit and is described with kd-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 [11], 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 Figure 3. A complete description and explanation of hB-tree concurrency, node splitting, and node consolidation is given in [3].

Any time a node containing entries representing spaces is split, it is possible for the split to also split the space described 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. The hB-tree splitting algorithm solves this problem by "clipping" the index terms whose spaces are split, producing nodes with multiple parents. Note that with hB-trees, at most one index term needs to be clipped per index node split, which minimizes the occurence of the problem.

7 Discussion

Our approach to index tree structure 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.

Our techniques permit multiple concurrent structure changes. In addition, all update activity and structure change activity above the data level executes in short independent atomic actions which do not impede normal database activity. Only data node splitting might execute in the context of a database transaction. This feature makes the approach usable with the diverse recovery mechanisms, while only impacting concurrency in a modest

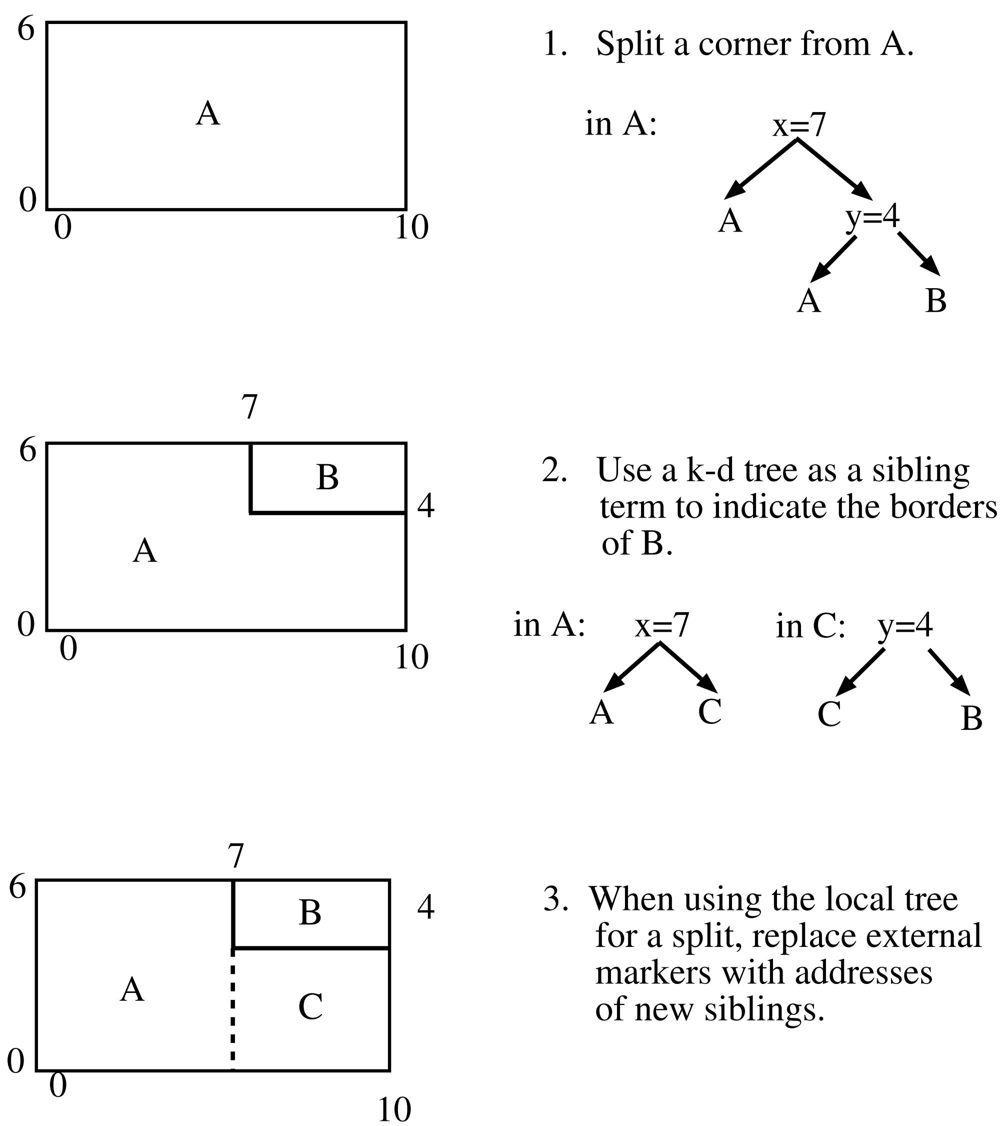


Figure 3: An hB-tree index showing the use of k-d trees for sibling terms. External markers (showing what spaces have been removed in creating "holes") have been replaced with sibling pointers.

way. Should the recovery method support "logical" undo in which updated records can move while still being subject to undo recovery, structure changes even at the data level can occur outside of the database transaction.

References

[1] Bayer, R., Schkolnick, M., Concurrency of operations on B-trees. *Acta Informatica* Vol 9 (1977) pp 1-21.

[2] Eswaren, K., Gray, J., Lorie, R. and Traiger, I. On the notions of consistency and predicate locks in a database system. Communications of ACM Vol 19, No 11 (Nov 1976) pp 624-633.

[3] Evangelidis, G., Lomet, D. and Salzberg, B., Modifications of the hB-tree for node consolidation and concurrency. [in preparation]

[4] Gray, J.N., Lorie, R. A., Putzulo, G. R., and Traiger, I. L., Granularity of locks and degrees of consistency in a shared data base. *IFIP Working Conf on Modeling of Data Base Management Systems*, (1976) pp 1-29.

[5] Gray, J. and Reuter, A.,*Transaction Processing: Techniques and Concepts*, book in preparation.

[6] Lehman, P., Yao, S.B., Efficient locking for concurrent operations on B-trees. *ACM Trans on Database Systems*, Vol 6, No 4 (Dec 1981) pp 650-670.

[7] Lomet, D. B. Process structuring, synchronization, and recovery using atomic actions. *Proc. ACM Conf. on Language Design for Reliable Software*, SIGPLAN Notices 12,3 (Mar 1977) pp 128-137.

[8] Lomet, D.B. Subsystems of processes with deadlock avoidance. *IEEE Trans. on Software Engineering*, vol SE-6, no. 3 (May 1980) pp. 297-304.

[9] Lomet, D.B. Recovery for shared disk systems using multiple redo logs. *Digital Equipment Corp. Technical Report* CRL90/4 (Oct. 1990) Cambridge Research Lab, Cambridge, MA.

[10] Lomet, D., Salzberg, B., Access methods for multiversion data, *Proc. ACM SIGMOD Conf. 1989*, Portland, OR pp. 315-324.

[11] Lomet, D. Salzberg, B., The hB-tree: a multiattribute indexing method with good guaranteed performance.[to appear in]*ACM Trans on Database Systems*, vol 15, no. 4 (Dec 1990).

[12] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, P., and Schwarz, P. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *IBM Research Report* RJ 6649, (Jan 1989) IBM Almaden Research Center, San Jose, CA

[13] Mohan, C. and Levine, F., ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. *IBM Research Report* RJ 6846, (August 1989) IBM Almaden Research Center, San Jose, CA

[14] Sagiv, Y., Concurrent operations on B* trees with overtaking. *Journal of Computer and System Sciences*, Vol 33, No 2, (1986) pp. 275-296

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

[16] Shasha, D., Goodman, N., Concurrent search structure algorithms. *ACM Trans. on Database Systems*, vol 13, No. 1 (March 1988) pp 53-90.