Simple, Robust and Highly Concurrent B-trees with Node Deletion

David Lomet
Microsoft Research
One Microsoft Way, Redmond, WA
lomet@microsoft.com

Abstract

Why might B-tree concurrency control still be interesting? For two reasons: (i) currently exploited "real world" approaches are complicated; (ii) simpler proposals are not used because they are not sufficiently robust. In the "real world", systems need to deal robustly with node deletion, and this is an important reason why the currently exploited techniques are complicated. In our effort to simplify the world of robust and highly concurrent B-tree methods, we focus on exactly where b-tree concurrency control needs information about node deletes, and describe mechanisms that provide that information. We exploit the B^{link}-tree property of being "well-formed" even when index term posting for a node split has not been completed to greatly simplify our algorithms. **Our goal is to describe a very simple but nonetheless robust method.**

1. Introduction

1.1. Early Techniques

The importance of only holding short duration locks on B-tree nodes was recognized very early [1,9,16,17]. However, none of the early methods can be applied directly to real database systems because they cannot recover the B-tree correctly should the system crash during a structure modification (node split or node delete).

1.2. Techniques Supporting Recovery

A paper by Mohan and Levine [15] in SIGMOD'92, based on an earlier tech report, is the earliest paper detailing B-tree concurrency with recovery.[1] Their method, which exploits the ARIES recovery method [14], globally serializes structure modifications (B-tree splits and node deletions). It has a structure

---

[1] The analyses in [6,7] deal briefly with the impact of recovery on B-tree concurrency, but do not describe a complete method.

modification bit in each page to guard a sub-tree whose structure is being modified. And it has a "delete" bit in each page that needs to be tested to handle deletes correctly. These bits and latches are encountered during normal operations on the B-tree, with the result that these operations are made substantially more complex. Unwinding of B-tree traversals can result, with the need for subsequent re-traversals. Furthermore, serialized structure modifications reduce concurrency.

Concurrency is higher when using a method based on B^{link}-trees [6,7,18]. Lomet and Salzberg derive their method [12,13] from B^{link}-trees. Their method has two variants, one without node deletion and another with it. In both cases, concurrency is very high. Further, when node deletes are not supported, the algorithms are quite simple. Unfortunately, node deletes add substantial complexity.

B^{link}-tree complexity arises with node deletes because there are periods when no latches are held on the tree. Hence, a program may acquire a valid node reference, release its latch, and when it accesses the node, this node has been deleted during the unlatched period. Two problems arise because of this:

1. Node splits propagate from data leaf upward toward the root. This upward traversal cannot latch couple since latch coupling is used in the downward traversal. So there is a period in which no latches are held. If node deletes are supported, it is uncertain whether the remembered parent of a node being split exists. Guarding against this previously entailed a tree re-traversal.

2. When posting a new index term for a split node, it is uncertain whether the new node resulting from the split continues to exist. It may have been deleted, again in a period when no latches are held. To guard against this previously required that the existence of the node be verified. This meant re-visiting the old node and checking whether its side pointer still referenced the new node.

Dealing with these problems leads to decreased performance, decreased concurrency, and increased complexity, including the re-traversal of the tree and the re-visiting of split nodes.

1.3. Simpler Proposal for Node Delete

With node delete, nodes can disappear unexpectedly. We want to use node addresses from the remembered path down the tree to post index terms for splits to the parents of the splitting nodes. However, with node deletes, our remembered path node pointers may be dangling.

One way to deal with this is to require that the node being deleted "live" until no pointers to it exist [16], called the "drain approach" in [19]. It entails waiting for the page to be empty, waiting for pointers to the page to be discarded, and only then deleting the page.

There are, we believe serious reasons why this approach, while "simple", is not robust.

1. It requires waiting until a node is empty before deleting it. This can sometimes be acceptable, e.g., if it is known that most deletes are part of a modify involving a delete followed by an insert, as would occur when an indexed field of a base tuple is updated [5]. But this becomes unacceptable when deletes occur with any regularity, and especially if their distribution is skewed. This may be caused, e.g., by purging out-of-date information, or dropping a set of products from an inventory database. Then many under-utilized pages may exist for extended periods, compromising utilization. The method of [15] also requires pages to be empty.

2. It can require that we update a page to mark it as empty prior to finally deleting it. Extra updates lead to extra logging, and potentially extra writes of the page back to disk.

However, we need not worry about dangling references across crashes since B-tree node references don't span crashes. Thus, in effect, a system crash does drain any delete state that we need to track whether references are dangling.

1.4. Our Contribution

Unlike prior approaches, we do not focus on revalidating references via tree re-traversal to check whether nodes have been deleted.[2] Instead, we remember how many nodes have been deleted, this number being maintained as "delete state". When we want to directly access a node of the tree in a structure modification action, we check whether delete state has changed since we discovered the need for the action. If not (the high frequency case), we use the remembered node reference since no node has been deleted in the interval of interest.

We use latch coupling to ensure that ordinary B-tree traversals do not need to check delete state. Thus, delete state is checked only during B-tree structure modifications. Since a characteristic of B^{link}-trees is that the tree is search correct even when such structure modifications are deferred, we choose to abandon the modifications if we discover that delete state has changed. This avoids re-traversals even in this case.

We describe this new approach in the remainder of the paper. In section 2, we provide an overview of the approach. Section 3 describes in some detail the specifics of the operations that the B^{link}-tree needs to support, and how the "delete state" approach simplifies these operations. In section 4, we describe how we separately track leaf level deletes and internal node deletes, and explain why this is advantageous. We very briefly discuss how this technique generalizes to handle multi-attribute indexing in section 5. Finally, an appendix provides pseudo-code for some of operations that index trees need to support.

2. Overview of Our Approach

2.1. Our Starting Point: B^{link}-Trees

We want to deal with node deletion robustly, while preserving the simplicity and high concurrency of the B^{link}-tree approach. The problem with node deletion as a general operation is that we need to know when nodes

---

[2] Previous methods [12,13,15] optimize the re-traversal by remembering node LSN's. But the complexity of re-traversal is not avoided. Further, performance and concurrency are impacted since each node on the remembered path has to be re-latched and re-accessed. Extra accesses involve extra memory fetches that are likely to be cache misses, which are very expensive on modern processors.

whose addresses we remember continue to exist, even when we have permitted the possibility that they may be deleted.

There are additional issues to deal with. Any method has to permit the desired locking so as to serialize transactions correctly. Sorting out transactional locking is not trivial, but can be made largely independent of B^{link}-tree latching. We will discuss briefly how transactional locking interacts with tree maintenance.

We must also ensure that B^{link}-tree mechanisms do not interfere with transactional operations during crash recovery. This problem was solved in [13] using multi-level recovery [11], and we assume multi-level recovery is used here. Thus, we needn't worry about partially completed structure modifications resulting from system crashes. Structure modifications are recovered first, restoring the B^{link}-tree to a well-formed state prior to the recovery of transactional operations that require a well-formed B^{link}-tree.

We want to support a framework that provides concurrency control and recovery, leaving the details of search and update within the nodes of the index tree to be determined by the specific data being indexed. This independence of B^{link}-tree support mechanisms from the index operations for specific forms of data is essential for a general indexing framework [8,13].

2.1. B^{link}-tree Fundamentals

A big plus for the B^{link}-tree is that it is well formed after a "half split" [19]. A half split allocates a new node and divides the full node's data between the full node and a new node. This half split **must** be done to accommodate the addition of data to the B^{link}-tree.

Figures 1, 2 and 3 show the node splitting process in a B^{link}-tree. Figure 1 shows the tree, with its side pointers, prior to the splitting of node F. Figure 2 shows the tree after the new node G is allocated and the contents of node F are divided between F and G. Note that node G is not referenced by an index term in Parent. Data in node G is found by means of a side traversal from node F. Finally, in Figure 3, the index term for G is posted to the parent. The important essential characteristic of the B^{link}-tree is that data in G is accessible even when it is not referenced by an index term in Parent.
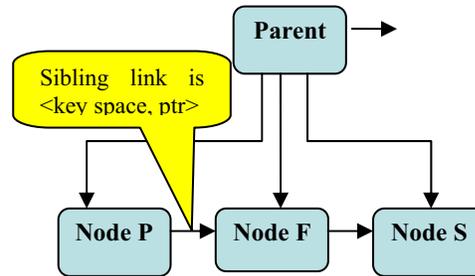

Figure 1: B^{link}-tree before split. Node F is full.
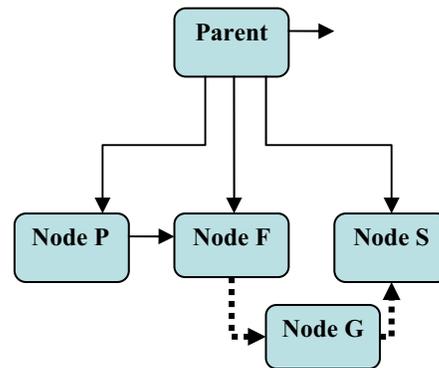

Figure 2: First "half split": contents of node F are divided between F and G.
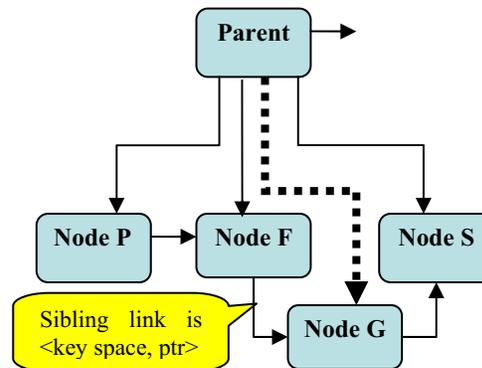

Figure 3: Second "half split": new index term for G is posted to parent.

With side traversals, we know both the node address and the key space description for an index term because side links contain this information, exactly as with a child link (see Figure 3). Thus, our B^{link}-tree is what was called a Pi-tree in [12,13]. We use the term "B^{link}-tree" in the sequel because it is more widely recognized, but the reader should not forget that the sibling links contain space descriptions as well as node addresses. The key space

descriptions are exploited below to enable our lazy structure modification approach.

2.2. Tracking Deletes

We track "delete state" so that we can be sure when a node has not been deleted. We introduce two delete states to deal separately with the two complications resulting from node deletes: (i) a parent to which we want to post an index term may have been deleted; (ii) a new leaf node for which we want to post an index term may have been deleted.

Index delete state (D_X) indicates whether it is safe to directly access a parent node (hence an index (internal) node, not a data (leaf) node) without re-traversing the B-tree. Index nodes are deleted in less than 1% of the node deletes. D_X has to answer for all nodes above the leaf level; and it must be maintained outside of the tree since any index node may be deleted. Testing D_X will return "yes" if an index node may have been deleted and "no if an index node cannot have been deleted.

Data delete state (D_D) indicates whether it is safe to post an index term for a new data node that resulted from a data node split. Since we access the parent of the new data node in any event, keeping D_D state tightly associated with the parent causes no extra page accesses. Further, data node deletes are much more common than index node deletes, and so there is real value to localizing data node deletes to a sub-tree. Hence we maintained D_D state in each parent of a leaf. Testing $D_D(nodeA)$ returns "yes" if data $nodeA$ may have been deleted, and "no" if data $nodeA$ cannot have been deleted. Higher up in the tree, D_X is used for this verification.

For both delete state tests, we can be conservative, returning "yes" even when the node we are asking about has not been deleted. Our algorithms will still work correctly, simply triggering a delay in propagating index terms up the tree. Because we are using a B^{link}-tree, the tree remains search correct even when index terms are missing. Since we only need to check delete state during structure modifications, normal B^{link}-tree operations can be almost completely unaffected.

2.3. Exploiting Laziness for Simplicity

Our goal is simplicity, including for structure modifications. When we know that nodes have not been deleted, we have a very simple structure modification mechanism. We simply promptly post the index term for a node split, and promptly delete under-utilized nodes. However, what happens if an index term has not been posted or a node has not been deleted?

The B^{link}-tree enables correct search when index terms are not posted, and when under-utilized nodes continue to exist. There is thus never a case where index posting or node deletion changes to the B^{link}-tree are required for continued operation of the tree. Normal reading and updating of indexed data can continue as the B^{link}-tree is always well-formed, even after a half split with an index term that has not been posted. Search is slower, however, because the side-traversal adds an extra node access to the search path. Hence, we want to make these structure changes as soon as it is "convenient".

B^{link}-trees thus enable lazy structure modification to be our strategy. Whenever we need to access a node further up the tree, e.g. as we would to post an index term, this access will start a new atomic action. We exploit a volatile *to-do* queue of structure modification actions that can be acted upon independently of mainline processing. This queue does not survive system failures.

1. We enqueue an index posting action on the *to-do* queue whenever we split a node. We do the first half-split "in line" because this must be done to accommodate the newly entered data.

2. We enqueue a node delete action on the *to-do* queue whenever we encounter an under-utilized node. The node does not need to be empty. We can set any utilization lower bound that we wish.

We abort enqueued actions should we detect a node delete that might impact the action. Such a node delete can lead to a substantial complication in what the action needs to do. This abort can result, e.g., in index terms not being posted. When a structure modification is aborted, we eventually re-discover the need for it and re-enqueue the action. We exploit this mechanism as well to deal with lost structure modifications that result from system crashes.

We re-discover the need for an index term posting during a B^{link}–tree side traversal; such a side traversal can only occur if we were directed to a predecessor page because an index term was not posted for the target page. Because our sibling links contain the key space description as well as the sibling's node address, we have the complete index term and so can fully describe the index term to be posted.

2.4. Latches and Locks

Latches are light-weight "locks" which provide exclusion without the need to go to the lock manager. Hence, the overhead for setting and releasing a latch is an order of magnitude smaller than for locks. "Real" B-tree implementations all use latches because of this performance gain. All users of latches are, however, required to acquire latches in the same consistent partial order so that deadlocks among latches are impossible.[3]

Like locks, latches come in multiple modes, *share*, *exclusive*, and *update* [3]. We do not need multi-granularity modes as the resource being latched is almost always a single node, and in any event does not impose a resource hierarchy. *Share* latches are compatible with each other and with *update* latches. *Update* latches are not compatible with each other. *Exclusive* latches are not compatible with any other latches.

Downward traversals of the tree exploit "latch coupling" in which, from a latched precursor node n_1, a node n_2 referenced by n_1 is latched prior to the latch on n_1 being released. This ensures that n_2 cannot have been deleted between acquiring its reference and accessing it. Note that latch coupling does not increase the number of latches acquired, though it does increase the time that they are held. Usually this will be minimal as most internal nodes are in the database's main memory cache.

The use of latch coupling in downward traversals imposes the partial order on latches, an ordering that is down the tree and to the right following side pointers. Thus, upward traversals, e.g. to post index terms, cannot use latch coupling because of the risk of undetected deadlocks with downward traversals.

The lock manager knows nothing about latches. Thus, a lock wait while holding a latch can produce undetected deadlocks. This is a

---

[3] The lock manager detects deadlocks among locks. Users are required to order latching to avoid deadlock.

problem whenever latches are used, including in other data organizations. The usual solution is to ask for the lock in "no wait" mode [15]. Almost always this request succeeds and normal program execution continues since lock conflicts are the exception. Otherwise, if the lock is held, the request returns immediately with an indication that the lock was not granted, instead of blocking and waiting for the lock,

When a lock is denied, the program gives up its latch, and re-requests the lock, this time blocking until the lock is granted. Once the lock is acquired, the latch is re-requested. It is here that the B-tree complication occurs, because the node that needs to be latched may have changed as a result of a structure modification. That is, the data that a program wants to access may no longer be in the original node. Hence, we need to once again find the node that contains the data that we wish to access. This is a problem for all B-tree methods that use latches. And it requires that we re-traverse the tree (to some extent) to find the node that needs to be latched.

A re-traversal is required in our method as well. We can optimize the re-traversal however. During the original downward tree traversal, we remember the path. There are two cases. Both check D_X to see if an index node delete has occurred. If it has not, then:

1. Re-latching a non-leaf node: we re-latch the original node, and traverse to a sibling if a node split has occurred, using latch coupling.

2. Re-latching a leaf node: we re-latch the parent of our leaf node and traverse to the leaf node currently containing the data that we just locked, using latch coupling. Finding the correct leaf can be immediate if D_D indicates that the remembered leaf node still exists.

In either case, if D_X indicates a node delete has occurred, we can abort the transaction. Such aborts are rare. The "no-wait" lock request will almost always succeed because lock conflicts are few. So re-traversals are rare. Furthermore, D_X changes are extremely rare. So, re-traversals are very fast, accessing only one or two nodes. We can bury the complexity of the re-traversal inside a *re-latch* procedure, which we call should our "no-wait" lock request fail. We provide *re-latch* with a path, the level of the desired page to latch, and a key.

3. Operations on the B^{link}-tree

In this section, we give an overview description of how our B^{link}-tree approach supports the operations a database index needs to provide.

3.1. Normal Operations

B-trees support a number of operations, not all of which we will fully explore here. The principal operations are:

1. Read a record: tree **traverse** followed by an access to the found page to **read node** the record requested.
2. Range read: a tree **traverse** followed by the accessing of potentially multiple pages to return records in the range. This is done via a series of **fetch** operations.
3. Insert, update, or delete a record: tree **traverse** followed by an access to the found node, and an update, insert, or delete of the record in an **update node** operation.

Below, we provide an overview of the preceding operations. The step by step descriptions of these operations are more fully described in the appendix.

3.1.1. Traverse Tree. Given a key, a node of the B-tree, and the level of the node that is desired in the tree, traverse returns the address of the node containing the key at the level desired. Latch coupling is performed both down the tree and for sibling traversals. The latch ordering is a partial order proceeding down the tree and to the "right", hence preventing deadlocks.

Latch coupling prevents a node deleter from accessing and deleting node n_2 in the tree traversal between the time the index term for n_2 is found in node n_1 and n_2 is accessed. We do not have to check delete state as n_2 cannot have been deleted without a prior access to n_1, which latch coupling prevents. Latch coupling isn't required if node deletes cannot occur. Tree traversal via latch coupling is one of two places outside of **access parent** (see below) in which support for node deletes has an impact.

Tree traversal latches differ depending upon the type of operation using them. But this difference only manifests itself when we reach the level of the tree that has been requested. Nodes higher in the tree than the requested level are latched in *share* mode. Latches for nodes at the level requested are *share* mode for reads and *update* mode for updates. The *update* latch on the final node in the traversal for updates is then promoted to *exclusive* before exiting **traverse**. We use an *update* latch for updaters to avoid the deadlock that may arise when two updaters both trying to promote their latches to *exclusive*.[4]

3.1.2. Read Node. We assume that a *share* latch is held on the node being read when this starts. Read node is not impacted by our handling of node delete. We are at the correct node, so we find the correct entry and return it. We enqueue a delete node action if we find the node is under utilized.

3.1.3. Update Node. We assume an *exclusive* latch is held on the node being updated when update node starts. Update node is not impacted by our handling of node delete. We are at the correct node, so we find the correct place in the node and do the update, be it an insert, delete or record update. If the update does not fit in the node, we do a split node action and try again. We enqueue a delete node action if we find the node is under utilized.

3.1.4. Fetch Next (Previous). Reading a key range of records involves more. We make only a comment about range reads. We use side pointers for B-tree concurrency control, not for enabling range reads via side pointer traversals. One might use them for range reads, but of course, they only are effective in a single direction.

Instead, we describe a technique that does not require side pointers to do range reads. Without side pointers, one avoids doing additional tree traversals by remembering the path down the tree. A "cursor" is maintained for the range. This cursor contains the path information. It shifts forward or backward as fetching proceeds.

For good range search performance, we want to avoid continually re-traversing the tree from the root. Our delete information makes this possible, even though we cannot maintain page latches continuously on the leaf nodes in the range. Hence, remembered nodes can be deleted during a range search. We use the re-latch procedure of 2.4 to deal with this.

---

[4] An exclusive latch must be held on a page before it can be changed. Update latches permit some sharing until it is sure that the page to be changed is identified.

3.2. Structure Modifications

Structure modification operations labeled as atomic actions are those we designate as "low level" transactions in the multi-level transaction hierarchy. These are actions that must be recovered prior to the recovery of "user" transactional level operations. Atomic actions [10] are like transactions in their atomicity and isolation, but durability is not required.

3.2.1. 1st Half Split: Split Node (atomic action). We split a node using the usual B^{link}-tree method involving two "half splits" (Figs. 1, 2, 3).

- Move old node high range contents and old side pointer to new node, and update side pointer in old node to point to new node. [5]
- Post index term for new node in parent.

Here we describe only the first half-split. The posting of the index term describing this split to the parent node is done as an update of the parent in a separate enqueued action. The impact of our approach for handling node deletion is all encapsulated in our **access parent** routine during the posting of the index term to the parent node.

This first half-split is the one tree modification action that must be done promptly. Otherwise, we would have to abort transactions whose updates caused nodes to become overly full. The important property of B^{link}-trees is that this can always be done under a latch that we are already holding. And the updating is confined to original node and new node. This atomic operation will typically involve two log records, one for each node, followed by a commit log record (which could be included in the second record). Only the contents of the original node are blocked from concurrent activity elsewhere.

3.2.2. Access Parent. Other B^{link}-tree modifications are performed lazily. Actions are enqueued on our *to-do* queue and all use the access parent routine.

Access parent accesses the parent of the splitting node (or node to be deleted) so that the index term can be inserted (deleted). It is given the remembered parent node address and returns with the current parent node latched if it determines that the parent has not been deleted. This parent may, because of concurrent splitting, be a sibling of the original parent. An error is returned if the parent may have been deleted. No
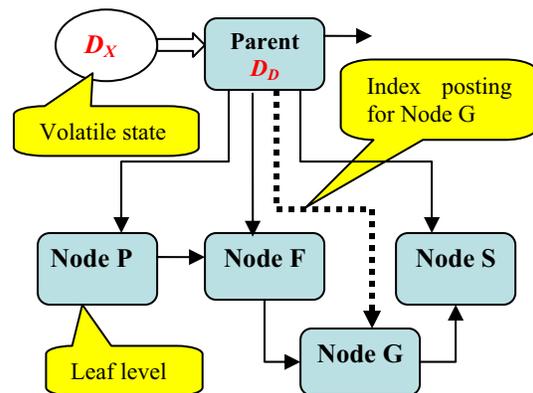
---

[5] We can leave the moved contents in the old node to avoid needing an undo log record for it.

latches can be held when executing **access parent** so that latch deadlocks cannot occur.

As a parent node must be an index node, we check D_X state to ensure that we can safely access "up the tree". The parent is guaranteed to exist when D_X has not changed. This is the key test that makes it possible for us to deal with node deletion while avoiding re-traversal of the tree. Once the parent is latched, it cannot be deleted until it is unlatched.

Within **access parent**, we use delete state to avoid having to verify that index term posting is still required, and that the descendent node has not been deleted. We use D_X for this purpose if the descendent is an index node. Since data node deletes are not captured in the D_X state, we use our separate D_D state for data node descendents. This is illustrated in Figure 4. We use the double arrow in Figure 4 to indicate latch ordering, where we must latch D_X prior to latching Parent, and the D_X latch is not released until the Parent is latched, hence latch coupling. D_D is protected by the Parent latch.

Within **access parent**, we also do all updating of delete state. **Access parent** is called indicating whether it is handling a delete or an index posting. In the case of delete, it determines whether an index node or a data node is the target of the delete. If **access parent** returns normally, it will have updated the appropriate delete state, either D_D or D_X. Because delete state information can be conservative, we can safely assume the completion of the node deletion action that is to follow.

Figure 4: For index term posting for leaf nodes, "access parent" checks D_x to ensure that Parent exists, and D_D in Parent to ensure that G exists.

Thus, **access parent** *encapsulates* all updating and testing of both forms of delete state, and for both parent node existence validation and for index term posting verification.

3.2.3. 2nd Half Split: Post Index Term (atomic action).

Our access parent routine determines whether posting an index term will be successful. If access parent returns normally with the parent node identified and latched, we use update node to post the index term. This can, of course, lead to a split of this parent node. But this is a separate atomic action, fully decoupled from the preceding split that triggered this index posting.

If **access parent** returns an exception, then rather than continuing to try to post the index term, we simply abort the index term posting and return. Deletes should be sufficiently rare so that this is not an issue.

3.2.4. Delete Node (atomic action).

Our delete node action permits us to consolidate all data nodes with low occupancy except for the left most sibling of a parent node. However, this consolidation will permit us to eventually consolidate the parent, and parent consolidation will permit the eventual consolidation of what were originally left most siblings.

We burden **delete node** so that dealing with splits is easy. Hence, **delete node** maintains the delete state information D_X and D_D. **Delete node** uses the **access parent** routine to access the parent node of a **delete node** candidate. **Access parent** updates the delete state information as it accesses the parent, and latches it. At this point it removes the index term for the candidate.

Delete node then *exclusive* latches and accesses the left sibling of the candidate, and the candidate itself, and releases the parent node latch, in that order. The contents of the deletion candidate are moved to the left sibling and the deletion candidate is then de-allocated.

4. Tracking Node Deletion

Here we describe how we track node deletions, and what the nature of the node deletion test might be when we attempt to perform a B^{link}-tree structure modification.

4.1. Delete State

The requirements for index and data node deletes are sufficiently different, both in usage and frequency, that we represent their node deletion states separately.

There are several ways to represent information about node deletes. We present just one, which has the benefit of simplicity by exploiting only approximate information which is conservative. It is possible to maintain precise information about deleted nodes, but we doubt its necessity.

4.1.1. Index Delete State.

Index node delete state D_X determines whether we can directly access a parent node for posting an index term or performing a node deletion, which requires the removal of the deleted node's index term in its parent. It is updated whenever an index node is deleted. It is checked when we want to access the parent. *During an access to the parent, if D_X has changed, the parent may have been deleted, so we return an exception.*

We maintain D_X as a counter that is incremented whenever an index node is deleted. Before we enqueue an action on our work queue we must have accessed and remembered the D_X value. When we enqueue the action, we enter the remembered D_X value with the action on the queue.

When we go to access the parent, if D_X has changed, we treat this as if the parent has been deleted. This conservative method should work fine if deletes are not common. Because we track data node deletes separately from index node deletes, D_X rarely changes. Over 99% of node deletes will be for data nodes. This means that almost always, parent access will be successful.

During an unsuccessful parent access, we remember the new D_X counter value so that when the need for the index posting is detected again, we will enter the more recent D_X value with the enqueued action, hence making it possible for this later action to complete successfully.

4.1.2. Data Delete State.

We use D_D state to determine whether leaf (data) nodes may have been deleted. If not, then we know that a new node resulting from a split will not have been deleted, and hence we can, without further checking, post its index term in its parent node.

It is natural to store the D_D state describing node deletes among a sub-tree of data nodes in their parent index node. That is, each such parent node contains D_D state that tracks the deletes of its descendents. We are accessing the parent index node in any event during the posting operation for any of these descendent

nodes. Thus, unlike with D_X, we can check D_D after we have accessed the parent, not before.

There are two additional reasons why it is useful (though not essential) for D_D state to be in these "parent of leaves" index nodes.

1. If D_D state persists across periods when the index node is not in the cache, fewer index postings will be aborted. With D_D in the node, we will retrieve it when this index node is again fetched into the cache.[6]

2. We protect D_D with the same latch used to access the "parent of leaves" node that contains it. Thus we avoid any need to protect D_D with a separate latch.

As with D_X state, we maintain D_D as a counter. Whenever we delete a data node in the sub-tree of this parent node, we need to have the parent latched and accessed. Hence, we update D_D during data node delete with little overhead.

If D_D for the parent node has changed when we attempt to post an index term for a new data node split, then the new node may already have been deleted, and hence no index term posting is required. In that case, we abort the posting. We may subsequently find that we have not yet posted an index term for this node, in which case, we place the posting action on our *to-do* queue again.

To make this "optimistic" approach work, we remember the prior value for D_D when we visit the node on the way to a leaf node. No additional latching is required. An enqueued action will also include the remembered D_D.

4.1.3. Volatile Delete State. Neither D_X nor D_D need be stable as (i) we are only interested in changed delete state since an action was enqueued; and (ii) no enqueued actions cross system crash boundaries. Thus no logging is needed to make them persistent.

4.2. Node Delete Impact

The following summarizes the incremental work needed to support node deletion over that needed when node deletion is not supported.

4.2.1. Impact outside structure modifications:
a) read and remember D_X state prior to accessing B$^{\text{link}}$-tree

---

b) read and remember D_D state in parent of leaf before accessing a data node
c) include remembered D_D or D_X on enqueued structure modifications
d) latch coupling during **traverse** instead of holding only single latch at a time.

4.2.2. Impact during structure modifications: all these are within **access parent**.
a) set D_X state when deleting index node
b) set D_D state when deleting data node
c) compare D_X to remembered D_X before accessing parent
d) compare D_D state to remembered D_D state to verify that new data node from split still exists
e) abandon structure modifications should delete states D_X or D_D indicate node delete

The enqueuing of structure modification actions in order to optimize the index is already required to deal with system crashes. We now exploit this lazy mechanism more frequently, i.e. when we detect possible node deletes.

5. Generalized Indexing Methods

This paper has focused on B$^{\text{link}}$-trees, but the approach described can be generalized to work with multi-attribute methods as well. Previous papers [2,12,13] describe the general approach. How we handle deletes here is directly applicable to the more general case. Note that the approach cannot be used directly with R-trees [4] because the space descriptions for R-tree index terms can change.

6. Acknowledgment

Phil Bernstein made many useful comments that greatly improved this presentation. Remaining difficulties are the responsibility of the author.

7. References

[1] R. Bayer, M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica* 9,1 (1977) 1-21.
[2] G, Evangelidis, D. Lomet B. Salzberg. The hB$^\pi$-tree: a multi-attribute index supporting concurrency, recovery, and node consolidation. *VLDB J.* 6,1 (1997) 1-25.
[3] J. Gray, A. Reuter. *Transaction Processing: Techniques and Concepts*, Morgan Kaufmann (1993).
[4] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. *SIGMOD* (1984) 47-57.

---

[6] We do not need to log changes to D_D as its exact value is not of interest, only the changes in value.

[5] T. Johnson, D. Shasha. Utilization of B-trees with Inserts, Deletes, and Modifies. *PODS Conf.* (1989) 235-246.

[6] T. Johnson, D. Shasha. A Framework for the Performance Analysis of Concurrent B-tree Algorithms. *PODS Conf.* (1990) 273-287.

[7] T. Johnson, D. Shasha. Performance of Concurrent B-tree Algorithms. *ACM TODS* 18,1 (1993) 51-101.

[8] M. Kornacker, C. Mohan, J. Hellerstein. Concurrency and Recovery in Generalized Search Trees. *SIGMOD* (1997) 62-72.

[9] P. Lehman, S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM TODS* 6, 4 (1981) 650-670.

[10] D. Lomet. Process structuring, synchronization, and recovery using atomic actions. *ACM Conf. on Language Design for Reliable Software* (1977) 128-137.

[11] D. Lomet. MLR: A Recovery Method for Multi-level Systems. *SIGMOD Conf.* (1992) 185-194.

[12] D. Lomet, B. Salzberg. Access Method Concurrency with Recovery. *SIGMOD* (1992) 351-360.

[13] D. Lomet, B. Salzberg. Concurrency and Recovery for Index Trees. *VLDB J.* 6,3 (1997) 224-240.

[14] C. Mohan, D, Haderle, B. Lindsay, H. Pirahesh, P. Schwarz. ARIES: a transaction recovery method supporting fine granularity locking and partial rollbacks using write-ahead logging. *ACM TODS* 17, 1 (1992) 94-162.

[15] C. Mohan, F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. *SIGMOD* (1992) 371-380.

[16] Y. Sagiv. Concurrent operations on B* trees with overtaking. *J Computer and System Sciences* 33,2 (1986) 275-296.

[17] D. Shasha, N. Goodman. Concurrent search structure algorithms. *ACM TODS* 13,1 (1988) 53-90.

[18] V. Srinivasan, M. Carey. Performance of B-tree concurrency control algorithms. *SIGMOD* (1991) 416-425.

[19] G. Weikum, G. Vossen. *Transactional Information Systems.* Morgan Kaufmann (2002).

Appendix A: B^{link}-tree Operations

We elaborate here on some of the common operations involved with concurrency control and recovery and its interaction with index tree structure modifications. Our descriptions are intended to reveal some of the difficulties that need to be addressed when using our method, not as full descriptions of all operation intricacies. Also, how to apply our technique to any particular database system needs to take into account the specifics of the target system.

A.1. Traverse tree

We assume starting node, *nodeA*, is latched in the appropriate mode. The found node is left latched when **traverse** returns. The search proceeds down to the requested level reqlevel. **Traverse** does the following:

1. Search *nodeA* for correct entry *entryA*.
2. Latch the node referenced by *entryA*, and call it *nodeB*.
3. Release the latch on *nodeA*.
4. If *entryA* points to a sibling node, enqueue **post index term** action for *<entryA,nodeB>* at parent(*nodeA*).
5. If *nodeA* is under-utilized, enqueue a **node deletion** action for *nodeA* at parent(*nodeA*)
6. If level(*nodeB*) > reqlevel or *nodeB* is sibling
   a. then return(**traverse**(*nodeB*, reqlevel))
      i. note that *nodeB* becomes the *nodeA* for the new invocation, and that *nodeB* is latched as required
   b. else return(*nodeB*)

A.2. Split node

We assume a latch is held on the "original" node to be split. Then **split node** does the following:

1. Allocate new node. (No latch is required as the node is invisible to the rest of the tree).
2. Split data between original and new node.
3. Assign to new node sibling pointer the original node sibling pointer.
4. sibling_ptr(new) gets sibling_ptr(original) and sibling_ptr(original) gets a pointer to the new node, with new node's space descriptor.
5. Enqueue a **post index term** operation for the parent of the original node on TODO queue.

A.3. Access Parent

Access parent does the following:
1. Latch D_X in share mode if the call is for **post index term**, in exclusive for **delete node.**
2. If test of D_X shows delete has occurred), release D_X latch and return error.
3. If the **parent access** is for an index node deletion, update D_X.
4. Latch node requested and release D_X latch.
5. Use **traverse**(node, level(node)) to find parent. This checks if node continues as the parent or whether node has split and real parent is a sibling of remembered node.
6. If the parent access is for a data node deletion, then update D_D state.
7. Else if **access parent** is to post index term for
   a. data node: if $D_D(node)$ has changed then release the node latch and return error.
   b. index node: if D_X has changed, then release node latch and return error.
8. Return.

A.4. Post Index Term

We assume that no latches are held when we start the posting of the index term. **Post index term** is very simple:
1. Access parent of split node via **access parent.** This will check the "delete states". If error returned, abort.
2. **Update node** atomic action posts index term
3. Return

A.5. Delete Node (atomic action)

The steps in **delete node** are as follows:
1. Perform **access parent**. If an error is returned, abort.
2. Retain the latch on the parent while latching the left sibling of the original node. If the parent node has no left sibling for our node to be consolidated, abort.
3. Latch the node to be consolidated via a side traversal from its left sibling. If the left sibling's pointer does not equal the node to be consolidated, abort.
4. Check whether original node remains under-utilized, and whether its contents will fit into its left sibling. If so, it will be consolidated. Otherwise return without consolidating.
5. Remove the index term for the deleted node. This will cause subsequent searches to access the left sibling instead.
6. If parent is under-utilized, enqueue a **delete node** action for the parent node.
7. Release the latch on the parent. The latch on left sibling and original page will protect the consolidation.
8. Delete the original node,
   a. copy its data and sibling pointer to the left sibling, replacing the left sibling's sibling pointer
   b. de-allocate the node.
9. Return