Lazy Updates for Distributed Search Structure

Theodore Johnson
ted@cis.ufl.edu

Padmashree Krishna
pk@cis.ufl.edu

Dept. of Computer and Information Science
University of Florida
Gainesville, Fl 32611-2024

Abstract

Very large database systems require distributed storage, which means that they need distributed search structures for fast and efficient access to the data. In this paper, we present an approach to maintaining distributed data structures that uses lazy updates, *which take advantage of the semantics of the search structure operations to allow for scalable and low-overhead replication. Lazy updates can be used to design distributed search structures that support very high levels of concurrency. The alternatives to lazy update algorithms (eager updates) use synchronization to ensure consistency, while lazy update algorithms avoid blocking. Since lazy updates avoid the use of synchronization, they are much easier to implement than eager update algorithms. We demonstrate the application of lazy updates to the dB-tree, which is a distributed* B^+ *tree that replicates its interior nodes for highly parallel access. We develop a correctness theory for lazy updates so that our algorithms can be applied to other distributed search structures.*

1 Introduction

A common problem with distributed search structures is that they are often single-rooted. If the root node is not replicated, it becomes a bottleneck and overwhelms the node that stores it [1]. A search structure node can be replicated by one of several well-known algorithms [2]. However, these algorithms synchronize operations, which reduces concurrency, and create a significant communications overhead.

Techniques exist to reduce the cost of maintaining replicated data and for increasing concurrency. Ladin, Liskov, and Shira propose *lazy replication* for maintaining replicated servers [12], making use of the dependencies in the operations to determine if a server's data is sufficiently up-to-date. Several authors have explored the construction of non-blocking and wait-free concurrent shared objects [6, 17]. These algorithms enhance concurrency because a slow operation never blocks a fast operation.

Lazy update algorithms are similar to lazy replication algorithms because both use the semantics of an operation to reduce the cost of maintaining replicated copies. The effects of an operation can be lazily sent to the other servers, perhaps on piggybacked messages. The lazy replication algorithm blocks an operation until the local data is sufficiently up-to-date. In contrast, a non-blocking or wait-free concurrent data structure never blocks an operation. The lazy update algorithms are similar to non-blocking algorithms in that the execution of a remote operation never blocks a local operation.

Lazy updates have a number of pragmatic advantages over more eager algorithms. They significantly reduce maintenance overhead. They are highly concurrent, since they permit concurrent reads, reads concurrent with updates, and concurrent updates (at different nodes). Finally, they are easy to implement because they avoid the use of synchronization.

We first present a framework for showing the correctness of lazy update algorithms. We next discuss lazy update algorithms for implementing a distributed B-tree, the dB-tree [8]. We present three algorithms, the last of which can implement a dB-tree that never merges nodes and performs data balancing on leaf nodes (we have previously found that never merging nodes results in little loss in space utilization [9], and data balancing on the leaf level is low-overhead and effective [11]). Our methods can be applied to other distributed search structures, such as hash tables [5].

1.1 The dB-tree

A B-tree is a multi-ary tree in which every path from the root to the leaf is the same length. The tree is kept in balance by adjusting the number of children in each leaf. In a B^+-tree, the keys are stored in the leaves and the non-leaf nodes serve as the index. A *B-link* tree is

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

SIGMOD /5/93/Washington, DC,USA

© 1993 ACM 0-89791-592-5/93/0005/0337...$1.50

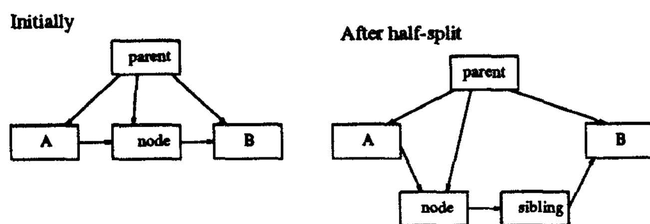

Figure 1: *Half-split operation*

a B^+-tree in which every node contains a pointer to its right sibling [4].

Previous work on parallel-access search structures (see [16] for a survey). has concentrated on concurrent or shared-memory implementations. Particularly notable are the B-link tree algorithms [14, 15, 13]. which we use as a base for the dB-tree. These algorithms have been found to provide the highest concurrency of all concurrent B-tree algorithms [10]. In addition, operations on a B-link tree access one node at a time. A B-link tree's high performance and node independence makes it the most attractive starting point for constructing a distributed search structure.

The key to the high performance of the B-link tree algorithms is the use of the *half-split* operation, illustrated in Figure 1. If a key is inserted into a full node, the node must be split and a pointer to the new sibling inserted into the parent (the standard B-tree insertion algorithm). In a B-link tree, this action is broken into two steps. First, the sibling is created and linked into the node list, and half the keys are moved from the node to the new sibling (the half-split). Second, the split is completed by inserting a pointer to the new sibling into the parent. If the parent overflows, the process continues recursively.

During the time between the half-split of the node and the completion of the split at the parent, an operation that is descending the tree can misnavigate and read the half-split node when it is searching for a key that moved to the sibling. In this case, it will detect the mistake using range information stored in the node and use the link to the sibling to recover from the error. As a result, all actions on the B-link tree index are completely local. A `search` operation examines one node at a time to find its key, and an `insert` operation searches for the node that contains its key, performs the insert, then restructures the tree from the bottom up.

Some work has been done to develop a distributed B-tree. Colbrook et al. [3] developed a pipelined algorithm. Wang and Weihl [18] use a special form of cache coherence to implement a parallel B-tree, so that it can be implemented on a shared-nothing architecture with the appropriate underlying software.

The dB-tree [8] implements the B-link tree algorithm as a distributed protocol (as in [1]). An *operation* on the index (search, insert, or delete) is performed as a sequence of *actions* on the nodes in the search structure, which are distributed among different processors. Each processor that maintains part of the search structure has two components: a *queue manager*, which maintains a queue of pending actions (the *message queue*); and a *node manager*, which repeatedly takes an action from the queue manager and performs the action on a node. The action execution typically generates a subsequent action on another node (for example, searching one index node leads to searching another node). If the next node to process is stored locally, then a new entry is put into the message queue. Otherwise, the node manager sends a message to the appropriate remote queue manager indicating the action to be taken. We assume that the processing of one action can't be interrupted by the processing of another action, so an action is implicitly atomic.

All operations start by accessing the root of the search structure. If there is only one copy of the root, then access to the index is serialized. Therefore, we want to replicate the root widely in order to improve parallelism. As we increase the degree of replication, however, the cost of maintaining coherent copies of a node increases. Since the root is rarely updated, maintaining coherence at the root isn't a problem. A leaf is rarely accessed, but a significant portion of the accesses are updates. As a result, wide replication of leaf nodes is prohibitively expensive.

In the dB-tree the leaf nodes are stored on a single processor. We apply the rule that if a processor stores a leaf node, it stores every node on the path from the root to that leaf. The dB-tree replication policy stores the root everywhere, the leaves at a single processor, and the intermediate nodes at a moderate level of replication. As a result, an operation can be initiated at every processor simultaneously, but the effects of updates are localized. As a side effect, an operation can perform much of its searching locally, reducing the number of messages passed.

The replication strategy for a dB-tree helps to reduce the cost of maintaining a distributed search structure, but the replication strategy alone is not enough. If every node update required the execution of an available-copies algorithm [2], the overhead of maintaining replicated copies could be prohibitive. Instead, we take advantage of the semantics of the actions on the search structure nodes and use *lazy updates* to maintain the replicated copies inexpensively.

We note that many of the actions on a dB-tree node commute. For example, consider the sequence of actions that occurs in Figure 2. Suppose that nodes A and B

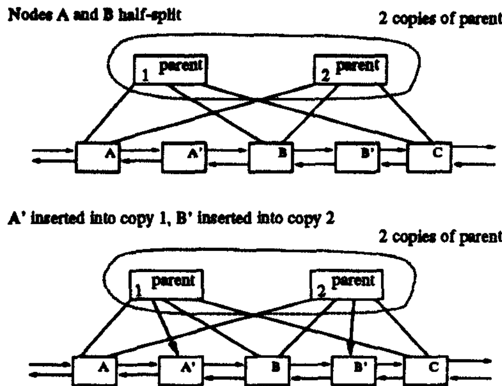

Figure 2: *Lazy inserts*

split at "about the same time". Pointers to the new siblings must be inserted into the parent, of which there are two copies. A pointer to A' is inserted into the first copy of the parent and a pointer to B' is inserted into the second copy of the parent. At this point, the search structure is inconsistent, since not only does the parent not contain a pointer to one of its children, but the two copies of the parent don't contain the same value.

The tree in Figure 2 is still usable, since no node has been made unavailable. Further, the copies of the parents will eventually converge to the same value. Therefore, there is no need for one insert action to synchronize with other insert actions on the node. The tree is always navigable, so the execution of an insert doesn't block a search action. We call node actions with such loose synchronization requirements *lazy updates*.

Before we terminate this introduction, we should mention some useful characteristics of lazy updates. First, when a lazy update is performed at one copy of a node, it must also be performed at the other copies. Since the lazy update commutes with other updates, there is no pressing need to inform the other copies of the update immediately. Instead, the lazy update can be piggybacked onto messages used for other purposes, greatly reducing the cost of replication management (this is similar to the *lazy replication* techniques [12]). Second, index node searches and updates commute, so that one copy of a node may be read while another copy is being updated. Further, two updates to the copies of a node may proceed at the same time. As a result, the dB-tree not only supports concurrent read actions on different copies of its nodes, it supports concurrent reads and updates, and also concurrent updates.

2 Correctness

Shasha and Goodman [16] provide a framework for showing that non-replicated concurrent data structures are serializable. Their work can be modified for application to replicated search structures. However, we would like the distributed search structure to satisfy additional correctness constraints. For example, when a distributed computation terminates, every copy of a node should have the same value. Performing concurrency control on the copies is the subject of this paper.

2.1 Copy Correctness

Intuitively, we want the replicated nodes of the search structure to contain the same value eventually. We can ensure the coherence of the copies by serializing the actions on the nodes (perhaps via an available-copies algorithm [2]). However, we want to be *lazy* about the maintenance. In this section, we describe a model of distributed search structure computation and establish correctness criteria for lazy updates.

A node of the logical search structure might be stored at several different processors. We say that the physically stored replicas of the logical node are *copies* of the logical node. We denote by $copies_t(n)$ the set of copies that correspond to node n at (global snapshot) time t.

An operation is performed by executing a sequence of actions on the copies of the nodes of the search structure. The specification of an action on a copy has two components: a final value c' and a subsequent action set SA. An action that modifies a node is performed on one of the copies first, then is relayed to the remaining copies. We distinguish between the *initial* action and the *relayed* actions. Thus, the specification of an action is:

$$a^s(p,c) = (c', SA)$$

When action a with parameter p is performed on copy c, copy c is replaced by c' and the subsequent actions in SA are scheduled for execution. Each subsequent action in SA is of the form (a_j, p_j, c_j), indicating that action a_j with parameter p_j should be performed on copy c_j. If copy c_j is stored locally, the processor it in the message queue. Otherwise, the action is sent to a processor that stores c_j. If the action is a *return value* action, a message containing the return value is sent to the processor that initiated the operation. If the final value of $a(p,c)$ is c for every valid p and c, then a is a *non-update* action; otherwise a is an *update* action. The superscript s is either *init* or *relay*, indicating an initial or a relayed action. We also distinguish initial actions by writing them in capitals, and relayed actions by writing them in lowercase (I and i for an insert, for an example).

In order to discuss the commutativity of actions, we will need to specify whether the order of two actions be exchanged. If action a^s with parameter p can be performed on c to produce subsequent action set SA, then the action is *valid*, otherwise the action is *invalid*.

We note that the validity of an action does not depend on the final value.

An algorithm might require that some actions must be performed on all copies of a node, or on all copies of several nodes, "simultaneously". Thus, we group some action sequences into ***atomic action sequences***, or AAS. The execution of an AAS at a copy is initiated by an ***AAS_start*** action and terminated by an ***AAS_finish*** action. A copy may have one or more AAS currently executing. An AAS will commute with some actions (possibly other AAS_start actions), and conflict with others. We assume that the node manager at each processor is aware of the AAS-action conflict relationships, and will block actions that conflict with currently executing AAS. The AAS is the distributed analogue of the shared memory lock, and can be used to implement a similar kind of synchronization. However, lazy updates are preferable.

2.1.1 Histories

In order to capture the conditions under which actions on a copy commute, we model the value of a copy by its history (as in [7]). Formally, the *total history* of copy $c \in copies_t(n)$ consists of the pair (I_c, A'_c), where I_c is the initial value of c and A'_c is a totally-ordered set of actions of c. We define correctness in terms of the update actions, since non-update actions should not be required to execute at every copy. The *(update) history* of a copy is a pair (I_c, A_c) where I_c is the same initial value as in the total history, and A_c is A'_c with the non-update actions deleted (and the order on the update actions preserved). To remove the distinction between initial and relayed actions, we define the ***uniform*** history, $U(H)$ to be the update history H with each action a^s replaced by a. Finally, we will write the history of copy c, (I_c, A_c) as $H_c = I_c \prod_{j=1}^{m} a_j^c$, where $A_c = (a_c^1, a_c^2, \ldots, a_c^m)$.

Suppose that $H_c = I_c \prod_{j=1}^{m} a_j$, and that I_c is the final value of $H' = I' \prod_{l=1}^{k} a'_l$. Then $H_c^* = (I' \prod_{l=1}^{k} a'_l) \prod_{j=1}^{m} a_j$ is the *backwards extension* of H_c by H'. It is easy to see that H_c and H_c^* have the same value, and the last m actions in H_c^* have the same subsequent action sets as the m actions in H_c. When a node is created, it has an initial value, I_n. When a copy of a node is created it is given an initial value, which we call the *original value* of the copy. This initial value should be chosen in some meaningful way, and will typically be equivalent to the history of the creating copy, or to a synthesis of the histories of the existing copies. In either case, the new copy will have a backwards extension that corresponds to the the history of update actions performed on the copy. If a copy of a node is deleted, then we no longer need to worry about the node contents. We denote set of all initial update actions performed on node n by M_n.

We recall that an action on a copy is valid if the action on the current value of the copy has its associated subsequent action. A history is ***valid*** if action a_j is valid on $I_c \prod_{k=1}^{j-1} a_k^c$ for every $j = 1, \ldots, m$. The ***final value*** of a history is the final value of the last action in the history. Two histories are ***compatible*** if they are valid, have the same final values, and have the same uniform updates. If H_1 and H_2 are compatible, then we write $H_1 \equiv H_2$.

Our correctness criteria for the replica maintenance algorithms are the following:

Compatible History Requirement: A node n with initial value I_n and update action set M_n has *compatible histories* if, at the end of computation C,

1 Every copy $c \in copies(n)$ with history H_c has a backwards extension B_c such that the update actions in $H'_c = B_c | H_c$ contains exactly the actions in M_n.

2 Every backwards extension H'_c can be rearranged to form H_c^* such that $U(H_c^*) = U(H_{c'}^*)$ for every $c, c' \in copies(n)$, and every H_c^* is valid.

If an algorithm guarantees that every node has a compatible history, then it meets the ***compatible history requirement.***

Complete History Requirement: If every subsequent action that is issued appears in some node's update action set, then the computation meets the *complete history requirement.* If every computation that an algorithm produces satisfies the complete history requirement, then the algorithm satisfies the *complete history requirement.*

Ordered History Requirement: We define an *ordered action* to be one that belongs to a class τ such that all actions of class τ are time-ordered with each other (we assume a total order exists). A history H is an *ordered history* if for any ordered actions $h_1, h_2 \in H$ of class τ, if $h_1 <_\tau h_2$ then $h_1 < h_2$ in H. An algorithm meets the *ordered history requirement* if every node has a compatible history that is an ordered history.

The compatible history requirement guarantees that every node is single-copy equivalent when the computation terminates. We note that our condition for rearranging uniform histories is a condition of the subsequent action sets rather than a condition of the intermediate values of the nodes. The copies need only to have the same value at the end of the computation, but the subsequent actions can't be posthumously issued or withdrawn without a special protocol.

The complete history requirement tells us that we must route every issued action to a copy. A deleted node is conceptually retained in the search structure to satisfy the complete history requirement. The ordered history requirement lets us remove explicit synchronization constraints on the equivalent parallel algorithm by shifting the constraints to the copy coherence algorithm.

2.1.2 Lazy Updates

An update action must be performed on all copies of a node. With no further information about the action, it must be performed via an AAS to ensure that the conflicting actions are ordered in the same way at all copies. However, some actions commute with other almost all other actions, removing the need for an AAS. In Figure 2, the final value of the node is the same at either copy, and the search structure is always in a good state. Therefore, there is no need to agree on the order of execution. We provide a rough taxonomy of the degree of synchronization that different updates require.

Lazy Update: We say that a search structure update is a *lazy update* if it commutes with all other lazy updates, so synchronization is not required.

Semi synchronous update: Other updates are almost lazy updates, but they conflict with some but not all other actions. For example, the actions may belong to a class of ordered actions. We call these *semi synchronous* updates. A semi synchronous action requires special treatment, but does not require the activation of an AAS.

Synchronous Update: A synchronous update requires an AAS for correct execution. We note that the AAS might block only a subclass of other actions, or might extend to the copies of several different nodes.

3 Algorithms

In this section, we describe algorithms for the lazy maintenance of several different dB-tree algorithms. We work from a simple fixed-copies distributed B-tree to a more complex variable-copies B-tree, and develop the tools and techniques that we need along the way. For all of the algorithms that we develop, we assume that only search and insert operations are performed on the dB-tree. In addition, we assume that the network is reliable, delivering every message exactly once in order.

3.1 Fixed-Position Copies

For this algorithm, we assume that every node has a fixed set of copies. This assumption lets us concentrate on specifying lazy updates. Every node contains pointers to its children, its parent, and its right sibling. When a node is created, its set of copies are also created, and copies of the node are neither created nor destroyed.

We use the dB-tree algorithm described in section 1.1. The first step in designing a distributed algorithm is to specify the commutativity relationships between actions.

1 Any two insert actions on a copy commute. As in Sagiv's algorithm [15], we need to take care to perform out-of-order inserts properly.

2 Half-split operations do not commute. Since a half-split action modifies the right-sibling pointer, the final value of a copy depends on the order in which the half-splits are processed.

3 Relayed half-split actions commute with relayed inserts, but not with initial inserts. Suppose that in history H_p, initial insert action $I(A)$ is performed before a half-split action s that removes A's range from p. Then, if the order of I and s are switched, I becomes an invalid action. A relayed insert action has no subsequent actions, and the final value of the node is the same in either ordering. Therefore, relayed half-splits and relayed inserts commute.

4 Initial half-split actions don't commute with relayed insert actions. One of the subsequent actions of an initial half-split action is to create the new sibling. The key that is inserted either will or won't appear in the sibling, depending on whether it occurs before or after the half-split.

By our classification methods, an insert is a lazy update and a half-split is a synchronous update. If the ordering between half-splits and inserts isn't maintained, the result is lost updates (see Figure 3). We next present two algorithms to manage fixed-copy nodes . To order the half-splits, both algorithms use a *primary copy* (PC), which executes all initial half-split actions. (non-PC copies never execute initial half-split actions, only relayed half-splits). The algorithms differ in how the insert and half-split actions are ordered. The synchronous algorithm uses the order of half-splits and inserts at the primary copy as the standard to which all other copies must adhere. The semisynchronous algorithm requires that the ordering at the primary copy be consistent with the ordering at all other nodes (see Figure 4).

We do not require that all initial insert actions are performed at the PC, so copies might find that they exceed their maximum capacity. However, since each copy is maintained serially, it is a simple matter to add overflow blocks.

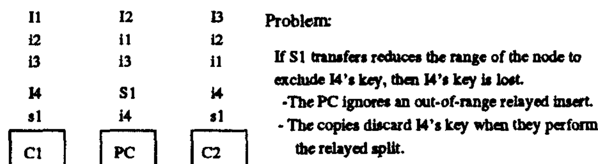

Figure 3: An example of the lost-insert problem

3.1.1 Synchronous Splits

Algorithm: An operation is executed by submitting an action, and each action generates subsequent actions

until the operation is completed. An operation is executed by performing its B-link tree actions, as discussed previously. Thus, all we need to do is specify the execution of an action at a copy. The synchronous split algorithm uses an AAS to ensure that splits and inserts are ordered the same way at the PC and at the non-PC copies (see Figure 4).

Half-split: Only the PC executes initial half-split actions. Non-PC copies execute relayed half-split actions. When the PC detects that it must half-split the node, it does the following:

1 Performs a split_start AAS locally. This AAS blocks all initial insert actions, but not relayed insert or search actions.
2 The PC sends a split_start AAS to all of the other copies.
3 The PC waits for acknowledgements from all of the copies of the AAS.
4 When the PC receives all of the acknowledgements, it performs the half-split, creating all copies of the new sibling and sending them the sibling's original value.
5 The PC sends a split_end AAS to all copies, and performs a split_end AAS on itself.

When a non-PC copy receives a split_start AAS, it blocks the execution of initial inserts and sends an acknowledgement to the PC. The executions of further initial insert actions on the copy are blocked until the PC sends a split_end AAS. When the copy processes the split_end AAS, it modifies the range of the copy, modifies the right-sibling pointer, discards pointers that are no longer in the node's range, and unblocks the initial insert actions.

Insert: When a copy receives an initial insert action it does the following:

1 Checks to see if the insert is in the copy's range. If not, the insert action is sent to the right sibling.
2 If the insert is in range, and the copy is performing a split AAS, the insert is blocked.
3 Otherwise, the insert is performed and relayed insert actions are sent to all of the other copies.

When a copy receives a relayed insert action, it checks to see if the insert is in the copy's range. If so, the copy performs the insert. Otherwise, the action is discarded.

Search: When a copy receives a search action, it examines the node's current state and issues the appropriate subsequent action.

We note that since non-PC copies can't initiate a half-split action, they may be required to perform an insert on a too-full node. Actions on a copy are performed on a single processor, so it is not a problem to attach a temporary overflow bucket. The PC will soon detect the overflow condition and issue a half-split, correcting the problem.

Theorem 1 *The synchronous split algorithm satisfies the complete, compatible, and ordered history requirements.*

Proof: We observe that the fourth link-algorithm guideline is satisfied, so that whenever an action arrives at a copy, its parameter is within the copy's inreach. Therefore, the synchronous split algorithm satisfies the complete history requirement.

Since there are no ordered actions, the synchronous split algorithm vacuously satisfies the ordered history requirement.

We show that the synchronous algorithm produces compatible histories by showing that the histories at each node are compatible with the uniform history at the primary copy. First, consider the ordering of the half-split actions (a half-split is performed at a node when the split_end AAS is executed). All initial half-split actions are performed at the PC, then are relayed to the other copies. Since we assume that messages are received in the order sent, all half-splits are processed in the same order at all nodes.

Consider an initial insert I and a relayed half-split s performed at non-PC copy c. If $I < s$ in H_c, then I must have been performed at c before the AAS_start for s arrived at c (because the AAS_start blocks initial inserts). Therefore, I's relayed insert i must have been sent to the PC before the acknowledgement of s was sent. By message ordering, i is received at the PC before S is performed at the PC, so $i < S$ in H_{PC}. If $s < I$ in H_c, then $S < i$ in H_{PC}, because $S < s$ and $I < i$ (due to message passing causality). □

We note that this algorithm makes good use of lazy updates. For example, only the PC needs an acknowledgement of the split AAS. If every communications channel between copies had to be flushed, a split action would require $O(|\text{copies}(n)|^2)$ messages instead of the $O(|\text{copies}(n)|)$ messages that this algorithm uses. Furthermore, search actions are never blocked.

3.1.2 Semi synchronous Splits

We can greatly improve on the synchronous-split algorithm. For example, the synchronous split algorithm blocks initial inserts when a split is being performed. Furthermore, $3 * |\text{copies}(n)|$ messages are required to perform the split. By the applying of the "trick" of *rewriting history*, we can obtain a simpler algorithm that never blocks insert actions and requires only $|\text{copies}(n)|$ messages per split (and therefore is optimal).

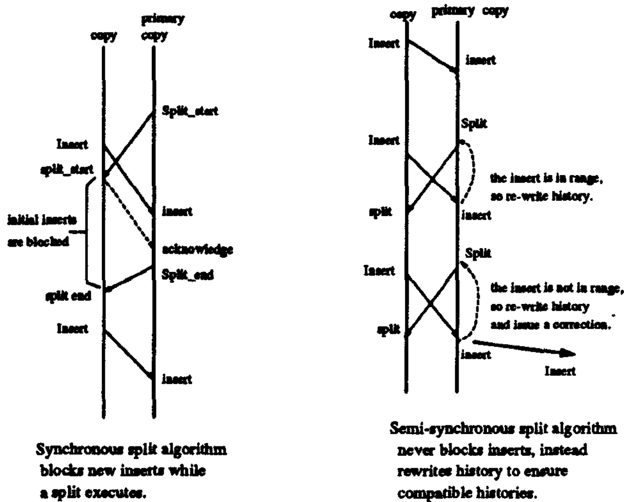

Figure 4: Synchronous and semi synchronous split ordering.

The synchronous-split algorithm ensures that an initial insert I and a relayed split s at a non-PC node are performed in the same order as the corresponding relayed insert i and initial split s are performed at the PC, with the ordering in the PC setting the standard. We can turn this requirement around and let the non-PC copies determine the ordering on initial inserts and relayed splits, and place the burden on the PC to comply with the ordering.

Suppose that the PC performs initial split S, then receives a relayed insert i_c from c, where I_c was performed before s at c (see Figure 4). We can keep H_{PC} compatible with H_c by rewriting H_{PC}, inserting i_c before S in H_{PC}. If i_c's key is in the PC's range, then H_{PC} can be rewritten by performing i_c on the PC. Otherwise, i_c's key should have been sent to the sibling that s created. Fortunately, the PC can correct its mistake by creating a new initial insert with i_c's key, and sending it to the sibling. This is the basis for the semi synchronous split algorithm.

Algorithm: The semi synchronous split algorithm is the same as the synchronous split algorithm, with the following exceptions:

1 When the PC detects that a split needs to occur, it performs the initial split (creates the copies of the new sibling, etc.), then sends relayed split actions to the other copies.
2 When a non-PC copy receives a relayed split action, it performs the relayed split.
3 If the PC receives a relayed insert and the insert is not in the range of the PC, the PC creates an initial insert action and sends it to the right neighbor

Theorem 2 *The semi synchronous split algorithm satisfies the complete, consistent, and ordered history requirements.*

Proof: The semi synchronous algorithm can be shown to produce complete and ordered histories in the same manner as in the proof of Theorem 1.

We need to show that all copies of a node have compatible histories. Since relayed inserts and relayed splits commute, we need only consider the cases when at least one of the actions is an initial action. Suppose that copy c performs initial insert I after relayed split s. Then, by message causality, the PC has already performed S, so the PC will perform i after S.

Suppose that c performs I before s and PC performs i after S. If i is in the range of PC after S, then i can be moved before S in H_{PC} without modifying any other actions. If i is no longer in the range of PC after S, then moving i before S in H_{PC} requires that S's subsequent action be modified to include sending i to the new sibling. This is exactly the action that the algorithm takes. □

Theorem 2 shows that we can take advantage of the semantics of the insert and split actions to lazily manage replicated copies of the interior nodes of the B-tree. In the next section, we observe a different type of lazy copy management that also simplifies implementation and improves performance.

3.2 Single-copy Mobile Nodes

In this section, we briefly examine the problem of lazy node mobility. We assume that there is only a single copy of each node, but that the nodes of the B-tree can migrate from processor to processor (typically, to perform load-balancing). When a node migrates, the host processor can broadcast its new location to every other processor that manages the node. However, this algorithm requires large amounts of wasted effort, and doesn't solve the garbage collection problems.

The algorithms that we propose inform the node's immediate neighbors of the new address. In order to find the neighbors, a node contains links to both its left and right sibling, as well as to its parent and its children. When a node migrates to a different processor, it leaves behind a *forwarding address*. If a message arrives for a node that has migrated, the message is routed by the forwarding address. We are left with the problem of garbage-collecting the forwarding addresses (when is it safe to reclaim the space used by a forwarding address). As with the fixed-copies scenario, we propose an eager and a lazy algorithm to satisfy the protocol. We have implemented the lazy protocol, and found it effectively supports data balancing [11].

The eager algorithm ensures that a forwarding address exists until the processor is guaranteed that no

message will arrive for it. Unfortunately, obtaining such a guarantee is complex and requires much message passing and synchronization. We omit the details of the eager algorithm to save space.

Suppose that a node migrates and doesn't leave behind a forwarding address. If a message arrives for the migrated node, then the message clearly has misnavigated. This situation is similar to the misnavigated operations in the concurrent B-link protocol, which suggests that we can use a similar mechanism to recover from the error. We need to find a pointer to follow. If the processor stores a tree node, then that node contains the first link on the path to the correct destination. So the error-recovery mechanism is to find a node that is 'close' to the destination, and follow that set of links.

The other issue to address is the ordering of the actions on the nodes (since there is only one copy, every node history is vacuously compatible). The possible actions are the following: insert, split, migrate, and link-change. The link-change actions are a new development in that they are issued from an external source, and need to be performed in the order issued.

Algorithm: Every node has two additional identifiers, a *version number* and a *level*. The version number allows us to lazily produce ordered histories. The level, which indicates the distance to a leaf, aids in recovery from misnavigation. An operation is executed by executing its B-link tree actions, so we only need to specify the execution of the actions.

Out-of-range: When a message arrives at a node, the processor first checks if the node is in range. This check includes testing to see if the node level and the message destination level match. If the message is out of range or on the wrong level, the node routes it in the appropriate direction.

Migration: When a node migrates,

1. All actions on the node are blocked until the migration terminates.
2. A copy of the node is made on a remote processor, and the copy is a duplicate (with the exception that the version number increments).
3. A link-change action is sent to all known neighbors of the node.
4. The original node is deleted.

Insert: Inserts are performed locally.

Half-split: Half-splits are performed locally by placing the sibling on the same processor and assigning the sibling a version number one greater than the half-split node's. An insert action is sent to the parent, and a link-change action is sent to the right neighbor.

Link-change: When a node receives a link-change action, it updates the indicated link only if the update's version number is greater than the link's current version number. If the update is performed, the new version number is recorded.

Missing Node: If a message arrives for a node at a processor, but the processor doesn't store the node, the processor performs the out-of-range action at a locally stored node. If the processor doesn't store a search structure node, the action is sent to the root.

Theorem 3 *The lazy algorithm satisfies the complete, compatible, and ordered history requirements.*

Proof: There is only a single copy of a node, so the histories are vacuously compatible. Each action takes a good state to a good state, so every action eventually finds its destination. Therefore, the algorithm produces complete histories.

The only ordered actions are the link-change actions. The node at the end of a link can only change due to a split or a migration. In both cases, the node's version number increments. When a link-change action arrives at the correct destination, it is performed only if the version number of the new node is larger than the version number of the current node. If the update not performed, the node's history is rewritten to insert the link change into its proper place. Let l be a link-change action that is not performed, and let l be an ordered action of class $\mathcal{L}$. Let a_j be the ordered action of class $\mathcal{L}$ in H_c that is ordered immediately after l (there is no a_k such that $l <_{\mathcal{L}} a_k <_{\mathcal{L}} a_j$). We rewrite H_c to be $H'_c = I_c \left(\prod_{v=1}^{j-1} a_v\right) \; l \; \left(\prod_{v=j}^{|H_c|} a_v\right)$. Thus, the history can be rewritten so that it remains valid.□

We note that an implementation of the lazy single-copy algorithm can use forwarding addresses to improve efficiency and reduce overhead. The forwarding addresses are not required for correctness, so they can be garbage-collected at convenient intervals.

3.3 Variable Copies

In this scenario, we assume that leaf level nodes can migrate, and that processors can join and leave the replication of the index nodes (so we can use this algorithm to implement a never-merge dB-tree). We assume that the leaf nodes are not replicated, and that the PC of a node never changes.

The lazy algorithm that we propose combines elements of the lazy fixed-copy and migrating-node algorithms by using lazy splits, version numbers, and message recovery.

To allow for data-balancing, we let the leaf level nodes migrate. The leaf level nodes aren't replicated, so we can manage them with the lazy algorithm for migrating nodes (section 3.2). We want to maintain the dB-tree property that if a processor owns a leaf node, it has

a copy of every node on the path from the root to the leaf. If a node obtains a new leaf node, it must *join* the set of copies for every node from the root to the leaf which it does not already help maintain. If the processor sends off the last child of a node, it *unjoins* the the set of processors that maintain the parent (applied recursively). When a processor joins or unjoins a node replication, the neighboring nodes are informed of the new cooperating processor with a *link-change* action. To facilitate link-change actions, we require that a node have pointers to both its left and right sibling. Therefore, a split action generates a link-change subsequent action for the right sibling, as well as an insert action for the parent.

We assume that every node has a PC that never changes (we can relax this assumption). The primary copy is responsible for performing all initial split actions for registering all join and unjoin actions. The join and unjoin actions are analogous to the migrate actions. Hence, every join or unjoin registration increments the version number of the node. The version number permits the correct execution of ordered actions, and also helps ensure that copies which join a replication obtain a complete history (see Figure 5). When a processor unjoins a replication, it will ignore all relayed actions on that node and perform error recovery on all initial action requests.

Algorithm:

Out-of-range: If a copy receives an initial action that is out-of-range the copy sends the action across the appropriate link. Relayed actions that are out of range are discarded.

Insert: 1 When a copy receives an initial insert action, it performs the insert and sends relayed-insert actions to the other node copies that it is aware of. The copy attaches its version number to the update.

2 When a non-PC copy receives a relayed insert, it performs the insert if it is in range, and discards it otherwise.

3 When the PC receives a relayed insert action, it tests to see if the relayed insert action is in range.

1 If the insert is in range, the PC performs the insert. The PC then relays the insert action to all copies that joined the replication at a later version than the version attached to the relayed update.

2 If the insert is not in range, the PC sends an initial insert action to the appropriate neighbor.

Split: 1 When the PC detects that its copy is too full, it performs a half-split action by creating a new sibling on several processors, designating one of them to be the PC, and transferring half of its keys to the copies of the new sibling. The PC sets the starting version number of the new sibling to be its own version number plus one. Finally, the PC sends an insert action to the parent, a link-change action to the PC of its old right sibling, and relayed-split actions to the other copies.

2 When a non-PC copy receives a relayed half-split action, it performs the half-split locally.

Join: When a processor joins a replication of a copy, it sends a join action to the PC of the node. The PC increments the version number of the node and sends a copy to the requester. The PC then informs every processor in the replication of the new member, and performs a link-change action on all of its neighbors.

Unjoin: When a processor unjoins a replication of a node, it sends an unjoin action to the PC and deletes its copy. The processor discards relayed actions on the node and performs error recovery on the initial actions. When the PC receives the unjoin action, it removes the processor from the list of copies, relays the unjoin to the other copies, and performs a link-change action on all of its neighbors.

Relayed join/unjoin: When a non-PC copy receives a join or an unjoin action, it updates its list of participants and its version number.

Link-change: A link-change action is executed using the migrating-node algorithm.

Missing-node: When a processor receives an initial action for a node that it doesn't manage, it submits the action to a 'close' node, or returns the action to the sender.

Theorem 4 *The variable-copies algorithm satisfies the complete, compatible, and ordered history requirements.*

Proof: We can show that the variable-copies algorithm produces complete and ordered histories by using the proof of theorem 3. If we can show that for every node n, the history of every copy $c \in copies(n)$ has a backwards extension H'_c whose uniform update actions are exactly M_n, then the proof of theorem 2 shows that the variable copies algorithm produces compatible histories.

For a node n with primary copy PC, let A_j be the set of update actions performed on PC when the PC has version number j. When copy c is created, the PC updates its version number to k and gives c an initial value $I_c = I_n B_k$, where B_k is the backwards extension of I_c to I_n and contains all uniform update actions in A_1 through A_{k-1}. The PC next informs all other copies of the new replication member. After a copy c' is informed of c, c' will send all of its updates to c. The copy c' might perform some initial updates concurrent with c's joining $copies(n)$. These concurrent updates are detected by the PC by the version number algorithm and are relayed to c. Therefore, at the end of a computation every copy $c \in copies(n)$ has every update in M_n in its uniform history. Thus, the variable copies algorithm produces compatible histories.□

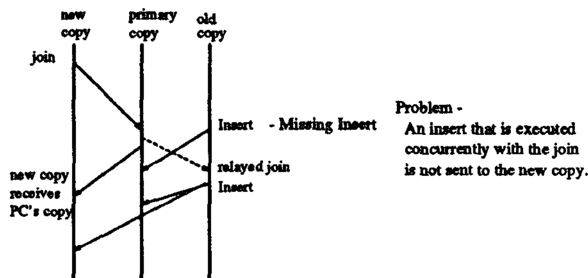

Figure 5: Incomplete histories due to concurrent joins and inserts.

4 Conclusion

We present algorithms for implementing lazy updates on a dB-tree, a distributed B-tree. These algorithms can be used to implement a dB-tree that never merges empty nodes and performs data-balancing on the leaves. We provide a correctness theory for lazy updates, so lazy update techniques can be used to implement lazy updates on other distributed and replicated search structures [5]. Lazy updates, like lazy replication, permit the efficient maintenance of the replicated index nodes. Since little synchronization is required, lazy updates permit concurrent search and modification of a node, and even concurrent modification of a node. Finally, distributed search structures that use lazy updates are easier to implement than more restrictive algorithms because lazy updates avoid the use of synchronization.

Our plans for future work include developing lazy updates algorithms that for node merging and node deletion (for a dE-tree [8]). We will apply lazy updates to other distributed data structures, such as hash tables, tries, and parallel file structures. Finally, we will investigate fault-tolerant lazy updates.

References

[1] F.B. Bastani, S.S. Iyengar, and I-Ling Yen. Concurrent maintenance of data structures in a distributed environment. *The Computer Journal*, 21(2):165–174, 1988.

[2] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[3] A. Colbrook, E.A. Brewer, C.N. Dellarocas, and W.E. Weihl. An algorithm for concurrent search trees. In *Proc. 20th Int'l Conf. on Parallel Processing*, pages III138–III141, 1991.

[4] D. Comer. The ubiquitous B-tree. *ACM Comp. Surveys*, 11:121–137, 1979.

[5] C.S. Ellis. Distributed data structures: A case study. *IEEE Transactions on Computing*, C-34(12):1178–1185, 1985.

[6] M. Herlihy. A methodology for implementing highly concurrent data structures. In *Proc. Symp. on Principles and Practice of Parallel Programming*, pages 197–206. ACM, 1989.

[7] M. Herlihy and J. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[8] T. Johnson and A. Colbrook. A distributed data-balanced dictionary based on the B-link tree. In *Proc. Int'l Parallel Processing Symp.*, pages 319–325, 1992.

[9] T. Johnson and D. Shasha. Utilization of B-trees with inserts, deletes and modifies. In *ACM Symp. on Principles of Database Systems*, pages 235–246, 1989.

[10] T. Johnson and D. Shasha. A framework for the performance analysis of concurrent B-tree algorithms. In *ACM Symp. on Principles of Database Systems*, pages 273–287, 1990.

[11] P. Krishna and T. Johnson. Implementing distributed search structures. Available at cis.ufl.edu:cis/tech-reports/tr92/tr92-032.ps.Z

[12] R. Ladin, B. Liskov, and L. Shira. Lazy replication: Exploiting the semantics of distibuted services. In *ACM Principles of Distributed Computing*, pages 43–57, 1990.

[13] V. Lanin and D. Shasha. A symmetric concurrent B-tree algorithm. In *1986 Fall Joint Computer Conference*, pages 380–389, 1986.

[14] P.L. Lehman and S.B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, 1981.

[15] Y. Sagiv. Concurrent operations on B^*-trees with overtaking. In *4th ACM Symp. Principles of Database Systems*, pages 28–37. ACM, 1985.

[16] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, 1988.

[17] J. Turek. *Resilient Computation in the Presence of Slowdowns*. PhD thesis, NYU Dept. of Computer Science, 1991.

[18] W.E. Weihl and P. Wang. Multi-version memory: Software cache management for concurrent B-trees. In *Proc. 2nd IEEE Symp. Parallel and Distributed Processing*, pages 650–655, 1990.