Designing Distributed Search Structures with Lazy Updates[*]

Theodore Johnson Padmashree Krishna
Dept. of Computer and Information Science
University of Florida
Gainesville, Fl 32611-2024

Abstract

Very large database systems require distributed storage for expansibility and high throughput, which means that they need distributed search structures for fast and efficient access to the data. In a highly parallel distributed search structure, parts of the index must be replicated to avoid serialization bottlenecks. Designing distributed and replicated search structures is made difficult by the complex interaction of the search structure concurrency control and the replica coherency algorithms. 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 develop a correctness theory for lazy update algorithms, then present lazy update algorithms to maintain a $dB-tree$, which is a distributed* B$^+$ *tree that replicates its interior nodes for highly parallel access. We show how the algorithms can be applied to the construction of other distributed search structures.*

1 Introduction

A distributed search structure is a search structure that is stored across several different processors that communicate through message passing. A distributed search structure is used to index and manage distributed storage, or to provide a location-independent name service. As distributed and parallel database techniques are used to improve transaction processing performance in a scalable manner, distributed search structures become a critical issue.

A common problem with distributed search structures is that they are single-rooted. If the root node is not replicated, it becomes a bottleneck and overwhelms the processor that stores it [2]. A search structure node can be replicated by one of several well-known algorithms [3, 8]. However, these algorithms synchronize operations, which reduces concurrency, and create a significant communications overhead.

While some distributed search structure algorithms have been proposed (see section 1.1.1 for references), most work in concurrent or parallel-access search structures have assumed a shared-memory implementation (see [34, 18] for a survey). One reason for the limited number of distributed search structures is the difficulty in designing them. A methodology for designing distributed search structures is needed before they can become popularly available.

[*]This work was partially supported by USRA grant number 5555-19. Part of the work was performed while Theodore Johnson was an ASEE summer faculty fellow at the NASA's National Space Science Data Center

Techniques exist to reduce the cost of maintaining replicated data and for increasing concurrency. Joseph and Birman [19] propose a method in which updates in a distributed database are piggybacked on synchronization messages. Ladin, Liskov, and Shira propose *lazy replication* for maintaining replicated servers [22], making use of the dependencies in the operations to determine if a server's data is sufficiently up-to-date. 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.

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 present in this paper general-purpose techniques for designing distributed search structures using lazy updates.

We first present a framework for showing the correctness of lazy update algorithms. We next discuss an increasingly general set of lazy update algorithms for implementing a distributed B-tree, the dB-tree [15, 16]. Finally, we show how some additional distributed search structures can be written.

1.1 Distributed B-trees

We initiated our study of distributed search structures by designing a distributed B-tree, the *dB-tree*. We use the dB-tree as a running example to illustrate techniques for designing distributed search structures.

1.1.1 Previous Work

A B-tree is a multi-ary tree in which every path from the root to a leaf is the same length. The tree is kept in balance by adjusting the number of children in each node. 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 a B^+-tree in which every node contains a pointer to its right sibling [7].

Previous work on parallel-access search structures (see [34] for a survey) has concentrated on concurrent or shared-memory implementations. Particularly notable are the B-link tree algorithms [24, 32, 23] 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 [17]. 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* action, illustrated

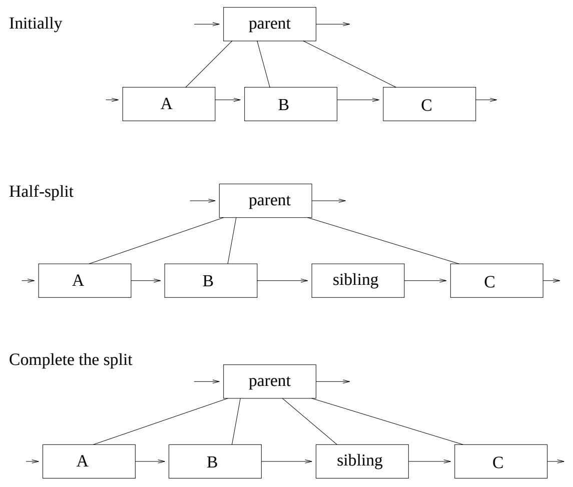

Figure 1: *Half-split action*

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. [6] developed a pipelined algorithm. Wang and Weihl [39, 41] have proposed that parallel B-trees be stored using *Multi-version Memory*, a special cache coherence algorithm for linked data structures. Multi-version Memory permits only a single update to occur on a replicated node at any point in time (analogous to *value logging* [40, 3] in transaction systems). Our algorithm permits concurrent updates on replicated nodes (analogous to *transition logging* [40, 3]).

Peleg [12, 30] has proposed several structures for implementing a distributed dictionary. The concern of these papers is the message complexity of access and data balancing. However, the issues of efficiency and concurrent access are not addressed. Matsliach and Shmueli [27] propose methods for distributing search structures in a way that has a high space utilization. The authors assume that the index is stored in shared memory, however, and don't address issues of concurrent restructuring.

Deitzfelbinger and Meyer auf der Hyde [9] give algorithms for implementing a hash table on a synchronous network. Ranade [31] gives algorithms and performance bounds for implementing a search tree in a synchronous butterfly or mesh network.

Some related work has been done to implement hash tables. Yen and Bastani [43] have developed algorithms for implementing a hash table on a SIMD parallel computer, such as a CM2. The authors examine the use of chaining, linear probing, and double hashing to handle bucket overflows.

Ellis [11] has proposed algorithms for a distributed hash table. The directories of the table are replicated among several sites, and the data buckets are distributed among several sites. Ellis' algorithm for maintaining the replicated directories is similar in many ways to our lazy update algorithms. Litwin, Neimat, and Schneider [26] propose a simple yet effective algorithm for a distributed linear hash table that uses a form of a replicated directory. They extended this work [38] to the order-preserving $RP*$ hash table. The algorithms in these works are similar to the algorithms discussed here, but the updates are performed reactively instead of proactively.

Other work on distributed search structures have primarily addressed distributing a search structure over several disks. These works include [37, 28, 33].

The contribution of this work is to present a method for constructing algorithms for distributed search structures. Unlike much of the previous work, our algorithms are explicitly designed for asynchronous distributed systems (or asynchronous parallel processors). While Ellis [11] and Litwin, Neimat, and Schneider [26, 38] have also proposed distributed search structure algorithms with a similar flavor, we present a structure for understanding, designing, and proving correct more complex distributed search structure algorithms.

1.1.2 The dB-tree

We use the dB-tree as a running example to demonstrate the application of lazy updates. In this section we briefly describe the dB-tree [15, 16].

The B-link tree protocol has two features that make it a promising starting point for designing a distributed search structure. First, all restructuring is performed as a sequence of local actions. Second, the B-link tree provides allows an operation to recover from a misnavigation. As a result, global co-ordination is not needed to restructure the index.

The dB-tree [15, 16] implements the B-link tree algorithm as a distributed protocol (as in [2, 11, 26]).

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 the 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. A search operation follows the link protocol [24, 32] until it reaches the leaf that can contain the key for which it is searching, then reports the result of the search. An insert or delete operation searches for the key that can contain the key to be inserted of deleted, then performs the operation. An insert might cause the node to become too full, in which case the node is half split. A delete might also cause similar restructuring [16].

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 (a previous work [1] shows that this is a good strategy). An example of a dB-tree that uses this replication policy is shown in Figure 2. 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. This paper focus on replication and design strategies, and a performance analysis of dB-tree implementation choices is beyond the scope of this work.

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 [3], 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

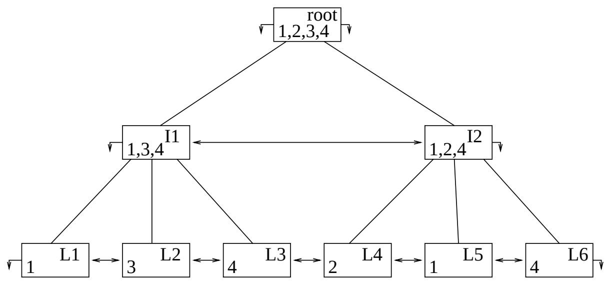

Figure 2: *A dB-tree*

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 3. Suppose that nodes A and B 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 3 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 (as in [19, 22]). 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.

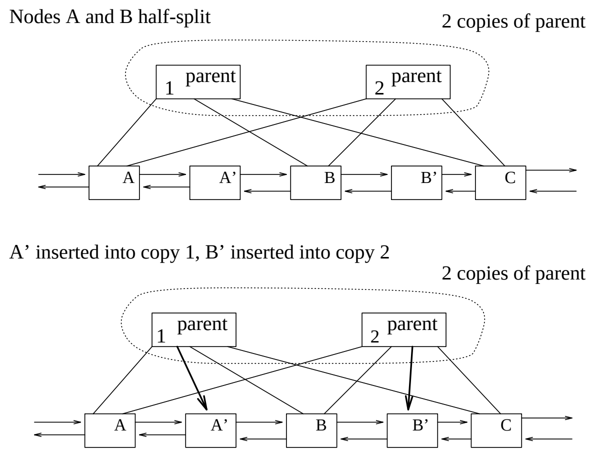

Figure 3: *Lazy inserts*

2 Correctness

Shasha and Goodman [34] provide a framework for showing that (non-replicated) concurrent search search structures are correct and serializable. We would like to show that when algorithms that have been validated by the methods of [34] are translated to replicated search structures, they are still correct. Here, we concentrate on the replica coherence protocols.

In this section, we present the theory for the case when replicated nodes never increase their key range. That is, replicated nodes never merge. This assumption makes the presentation of the correctness theory much simpler, because the set of actions on a node are well-defined. In Section 4, we extend the theory to handle merged nodes, and present algorithms that correctly handle actions on merged nodes.

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 [3]). However, we want to be *lazy* about the maintenance. In addition, the replica update protocol should support the design of distributed search structure algorithms. In this section, we describe a model of distributed search structure computation and establish correctness criteria for lazy updates.

Let $\mathcal{N}$ be the set of nodes that exist in the search structure during execution $\mathcal{E}$ (we will define $\mathcal{E}$ more precisely below). A node $n \in \mathcal{N}$ 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 time t.[1]

An operation (i.e., search for a key, insert a key) is performed by executing a sequence of actions (i.e., search this copy) 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

[1]By "time", we mean an ordering on the events that occur in $\mathcal{E}$ that is consistent with causality [5].

performed on one of the copies first, then is relayed to the remaining copies. The action that performs the first modification is an *initial* action, and the actions that are relayed to the other copies are the *relayed* actions. We distinguish between the initial and the relayed actions. Thus, the specification of an action is:

$$a^s(p, c) \rightarrow (c', SA)$$

When action a with parameter p is performed on copy c, copy c is replaced by c' (the *final value* of a) 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 (perhaps c and c_j are copies of different nodes). If copy c_j is stored locally, the processor puts the action in the message queue. Otherwise, the action is sent to a processor that stores c_j. If the subsequent action can be performed on any copy of a node (i.e., "search the next node"), we assume that the action chooses the copy. 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).

We need to make some assumptions about the execution of the actions. First, we assume that the execution of the actions is deterministic. Second, we assume that update actions on replicated nodes execute in two phases – the initial action and the relayed actions. The initial action might execute several times at different nodes because of navigation. When the initial action navigates to the correct node, it will *fire*, perform its modifications, and issue the relayed actions. Generally, relayed actions have empty subsequent action sets.

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. The intuition behind this definition is that we only care about the values in copies when the system is at rest, but it is hard to revise actions sent to other copies.

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 (as in [11]). However, lazy updates are preferable.

Example To make the concepts more clear, let us consider a simple example of a distributed B-tree and write the specifications of the possible actions. The algorithm that we specify will implement a dB-tree along the lines discussed in Section 1.1. To make the algorithm easier to specify, we assume that

1. The dB-tree has already been created.
2. No new copies of interior nodes are created.
3. Interior nodes are never split.
4. The only operations are to search for a key and to insert a key.

These assumptions are unrealistic, but they let us avoid some difficulties that are discussed in later sections. The algorithm is presented in Table 1. A search operation is initiated by calling the Search(k,c) action on a copy c of the root note, and an insert operation is initiated by calling the Insert_d(k,c) action on a copy of the root node. The execution of the action is specified by the modification made to the copy, and the subsequent action set. Typically, more than one execution is possible, so we list the conditions for each execution. In the algorithm, we make use of the function $\mathsf{keyrange}_{\mathsf{c}}(\mathsf{k}, \mathsf{c}')$. This function returns true if based on the information in c, key k can be found in the subtree rooted at c', and false otherwise.

The search operation returns a value by scheduling the Answer(t/f,O) action, where t/f is true or false, depending on whether k is in c, and O represents the task that issued the query. The Insert_d performs the downwards phase of the insert operation. When the proper leaf node is found, the insert is performed. If the leaf was full, it is half-split, creating the new leaf sibling. The restructuring phase of the insert is initiated by sending the Insert_u action to the parent of c. The exact parent of c might not be known, hence Insert_u must be able to navigate sideways. When the initial update Insert_u finally fires on copy c of node n, the relayed updates insert_u are sent to the other copies of n.

2.1 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 [3, 13, 42]). We note that our algorithms do not require that nodes be stored as history lists, we use histories only as a modeling technique. A typical algorithm will make use of a small amount of stored information to make its decisions.

Action	Conditions	Final value of c	Subsequent Actions
Search(k,c)	c is not a leaf, $\text{keyrange}_c(k,c)$ is true, c' is a child of c, $\text{keyrange}_c(k,c')$ is true	c	Search(k,c')
	$\text{keyrange}_c(k,c)$ is false, $\text{keyrange}_c(k,c')$ is true	c	Search(k,c')
	c is a leaf, $\text{keyrange}_c(k,c)$ is true	c	Answer(t/f,O)
Insert_d(k,c)	c is not a leaf, $\text{keyrange}_c(k,c)$ is true, c' is a child of c, $\text{keyrange}_c(k,c')$ is true	c	Insert_d(k,c')
	$\text{keyrange}_c(k,c)$ is false, $\text{keyrange}_c(k,c')$ is true	c	Insert_d(k,c')
	c is a leaf, $\text{keyrange}_c(k,c)$ is true, c is not full	$c \cup k$	Answer(true,O)
	c is a leaf, $\text{keyrange}_c(k,c)$ is true, c is full	(sib,sep)=halfsplit($c \cup k$)	Insert_u(sib,sep,parent(c))
Insert_u(k,n,c)	$\text{keyrange}_c(k,c)$ is false, $\text{keyrange}_c(k,c')$ is true	c	Insert_u(k,c')
	$\text{keyrange}_c(k,c)$ is true, $\text{keyrange}_c(k,c')$ is true	$c \cup (k,n)$	insert_u(k,n,c') for every other copy c' of n.
insert_u(k,n,c)		$c \cup (k,n)$	

Table 1: Actions for a simple dB-tree.

An *execution*, $\mathcal{E}_t$ is a record of all operations that have been initiated and all actions that have been executed up to time t (where t represents a consistent state). The record of an operation specifies its first action. The record of an action is discussed below.

The *total history* of copy $c \in copies_t(n)$ consists of the pair (V_c, A'_c), where V_c is the initial value of c and A'_c is a totally-ordered set of actions executed on c up to time t. 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 (V_c, A_c) where V_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 $a^s(p,c)$ replaced by $a(p,c)$.

The actions in $\mathcal{E}_t$ aren't necessarily the same as the actions in the copy histories, $\{H_c | c \in copies(n),\ n \in \mathcal{N}\}$. We will permit the algorithms to *re-write* their histories, to ensure that all copy histories meet some correctness requirements, discussed below.

Since a history (whether total, update, or uniform) is totally ordered, we can assign index numbers to the actions in the history. We will need to keep track of the subsequent action sets in the history, so we will denote an action in as (a, p, c, SA), where a is the action, p is the parameter of the action, c is the copy, and SA is the subsequent action set that was generated. So, we can write the history of copy c, (V_c, A_c)

as $H_c = V_c \prod_{j=1}^{m}(a_j, p_j, c, SA_j)$, where $A_c = ((a_1, p_1, c, SA_1), (a_2, p_2, c, SA_2), \ldots, (a_m, p_m, c, SA_m))$. The product sign $\prod$ denotes the application of actions to the copy, and lets us specify boundaries, but should not be understood as implying properties such as associativity and commutativity. Where there is no confusion, we will abbreviate $\prod_{j=1}^{m}(a_j, p_j, c, SA_j)$ as $\prod_{j=1}^{m} a_j$.

The *final value* of a history is the final value of the last action in the history. Suppose that $H_c = V_c \prod_{j=1}^{m} a_j$, and that V_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, V_n. When a copy of a node is created it is given an initial value, which we call the *initial value* of the copy. The 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 node. If a copy is deleted, we will retain its history to capture backwards extensions, but no further actions will be performed on it.

When we compare copy histories, we assume that the execution has reached a quiescent state. That is, every action that is specified in an operation or a subsequent action set in $\mathcal{E}_t$ has been executed. In the presentation, we do not explicitly tie our observations to a particular point in time and instead implicitly assume a quiescent point in time t. We denote set of all initial update actions performed on a copy of node n by time t as M_n, the *update history* of node n. Similarly, we denote the set of all actions performed on a copy of node n as M_n^*, which we call the *total history* of node 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 H is *valid* if update action a_j is valid on $V_c \prod_{k=1}^{j-1} a_k$ for every update action in H. We need a definition by which we can compare the histories of two different copies and say that they are the same. We say that history H is *compatible* with history $\hat{H}$ if

1. H and $\hat{H}$ have the same initial value.
2. The actions in H can be rearranged to form a valid history H' such that $U(H') = U(\hat{H})$.
3. The final values of H, H', and $\hat{H}$ are the same.

Let $\mathcal{N}$ be the set of all nodes accessed in $\mathcal{E}$. Our correctness criteria for the replica maintenance algorithms are the following:

Compatible History Requirement: A node $n \in \mathcal{N}$ with initial value V_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 $H'_c = B_c|H_c$ such that the initial value of H'_c is V_n, and the update actions in H' are exactly the actions in M_n.
2. There is a valid history $\hat{H}_n$ (the *standard history*) with initial value V_n and update actions M_n such that every H'_c is compatible with $\hat{H}_n$.

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

Complete History Requirement: We will say that an action a is *issued* if a is an initial action that appears in the subsequent action set of a', $a' \in \mathcal{E}_t$, such that a and a' have different actions and parameters. If every action that is issued appears as a fired action in some node's total history, 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 causal and 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 for every node, its copies meet the compatible history requirement with a standard ordered history $\hat{H}_n$.

The compatible history requirement guarantees that every node is single-copy equivalent when the computation terminates. We note that our condition for rearranging 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. However, we let search actions misnavigate.

The complete history requirement tells us that we must route every issued action to a copy. While the compatible history requirement applies to update actions only, the complete history requirement applies to all actions, including searches and relayed actions.

The ordered history requirement lets us remove explicit synchronization constraints on the equivalent concurrent algorithm by shifting the constraints to the copy coherence algorithm. The power of this constraint is explored in later sections.

2.1.1 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 almost all other actions, removing the need for an AAS. In Figure 3, 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 dB-trees, under increasingly relaxed assumptions about structure. 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. The algorithms and proofs developed in this section demonstrate techniques for designing distributed search structures.

We assume that the network is reliable, delivering every message exactly once in order. In addition, we initially assume that only search and insert operations are performed on the dB-tree. In Section 4, we discuss how nodes can be merged or deleted.

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 actions in this algorithm are the same as those in Table 1, with the exception that an Insert_u action will half-split an interior node if it is too full. Since the half-split is being executed on a replicated node, it becomes its own action (i.e., not combined with the insert, as in Table 1), and have an initial and a relayed version. The initial half-split creates the copies of the sibling node, removes the keys of the local copy c, and deletes the keys (pointers) that are no longer in the key range of c, and adjusts the sibling pointers. The relayed half-split adjusts the key range, removes keys (pointers), and adjusts sibling pointers.

By design, search actions can always execute, so we do not explicitly discuss them. Since the leaves are not replicated, we do not discuss actions on leaves either. So, the only actions of interest are the insert and half-split actions on the interior nodes. We abbreviate the initial and relayed insert_u actions as I and i, respectively, and the initial and relayed half-split actions as S and s. The first step in designing a replication algorithm is to specify the commutativity relationships between actions.

1. Any two insert actions on a copy commute. As in Sagiv's algorithm [32], we need to take care to perform out-of-order inserts properly.

2. Half-split operations do not commute. As an example, 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. To see why this occurs, examine Figure 4. In this scenario, there are three copies of a node n, copy 1, copy 2, and copy 3. Copy 1 performs an initial insert of key k concurrent with copy 2 processing a half-split that removes k from the node. When copy 1 performs the relayed split, it throws away key k, and copy 2 ignores the relayed insert because the key is out of range. The end result is that k does not appear in any copy of any node.

We 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 (that is, the PC generates the standard history, $\hat{H}_n$). The semisynchronous algorithm requires that the ordering at the primary copy be consistent with the ordering at all other nodes (see Figure 5).

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.

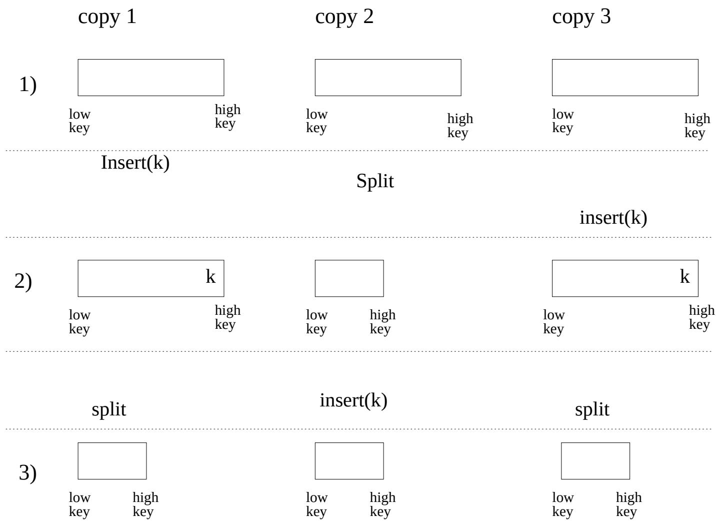

Figure 4: An example of the lost-insert problem

3.1.1 Synchronous Splits

Algorithm: To specify the algorithm, we only need to specify the execution of the insert and half-split actions at the interior nodes. A half-split action will invoke an AAS, and so is composed of several further actions. To simplify the presentation, we describe the action executions by listing the steps taken, instead of using a table table. The synchronous split algorithm uses ensures that splits and inserts are ordered the same way at the PC and at the non-PC copies. Figure 5 illustrates an execution.

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 initial 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. That is, the split takes effect then the split_end is processed.

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.

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 nodes always split to the right, so if an action misnavigates, it can always recover its path (that is, Shasha and Goodman's fourth link-algorithm guideline is satisfied [34]). 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). $\square$

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.

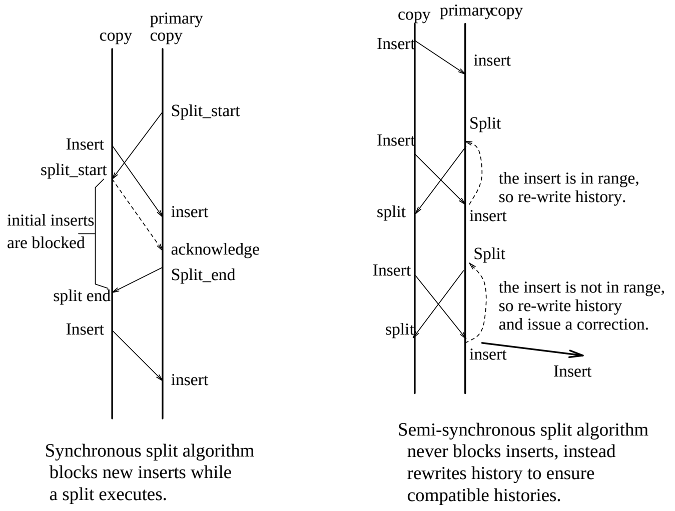
copy
primary copy
Split_start
Insert
split_start
initial inserts are blocked
insert
acknowledge
Split_end
split end
Insert
insert
Synchronous split algorithm blocks new inserts while a split executes.
copy
primary copy
Insert
insert
Split
Insert
the insert is in range, so re-write history.
split
insert
Split
Insert
the insert is not in range, so re-write history and issue a correction.
split
insert
Insert
Semi-synchronous split algorithm never blocks inserts, instead rewrites history to ensure compatible histories.

Figure 5: Synchronous and semi synchronous split ordering.

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, $3 * (|\text{copies}(n)| - 1)$ 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 $|\text{copies}(n)| - 1$ messages per split (and therefore is optimal).

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 5). 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. So, we re-write H_{PC} to make it compatible with the standard history. $\Box$

Discussion: 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. The key trick here is to examine the copy histories and issue a 'correction' if an incompatibility is discovered. Note that while the algorithms motivated by an examination of copy histories, the actual algorithms use only a small amount of saved state to make decisions.

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. The solution requires an algorithm to enforce ordered actions. 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 [20].

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. The problem that can arise is shown in figure 6. In this scenario, nodes A and B are neighbors. Node A half-splits to produce node sib 1. Sib 1 points to A and B, and a link-change action is sent to node B to inform node B of its new left neighbor. Before node B can process this link-change operation, sib 1 half-splits to produce sib 2, and another link-change action is sent to B. If node B processes the link-change for sib 2 before the link-change for sib1, the backwards list will be disordered for an indefinite period of time.

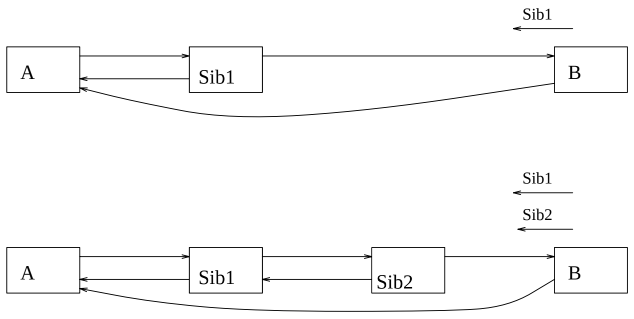

Figure 6: Possible Inconsistency if back links are modified out-of-order.

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 (equivalently, migration is atomic).
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 at the same or higher level. If the processor doesn't store such a 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. 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 $H'_c = V_c \left(\prod_{v=1}^{j-1} a_v\right)\ l\ \left(\prod_{v=j}^{|H_c|} a_v\right)$. Thus, the history can be rewritten so that it remains valid.

Since the lists are maintained properly, every action is eventually able to find its destination. Thus, the algorithm produces complete histories.

Each action takes a good state to a good state, so every action eventually finds its destination. Therefore, the algorithm produces complete histories. □

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.

Discussion This section shows two techniques for implementing distributed search structures. The first technique takes advantage of the fact that an action does not directly access a node, but must request access from the node manager. As a result, an action can try to access a non-existent node, as long as a recovery path exists.

The second technique discussed in this section shows how to make the node management protocol respect ordered actions. Since we assume that a natural time ordering exists between ordered actions, it is possible to assign timestamps to the actions. The particular ordered actions in the single-copy algorithm are *overwriting* actions (i.e, modifying link pointers). Since the result of executing an overwriting ordered action doesn't depend on the previous ordered actions, an overwriting action doesn't need to wait for the preceding ordered actions to execute. Instead, late actions are discarded. If an ordered action is not overwriting, then the later action must block until the earlier actions are executed. For example, if nodes can be deleted (as in [16]), then the insertion and the deletion of a pointer to a child are non-overwriting ordered actions. If the delete action arrives first, it must be held to kill the insert action when it arrives (as discussed in Section 4.

3.3 Variable Copies

In this section, we discuss how to allow processors to 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 remove this assumption later). 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 7). 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.
   - (a) 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 insert.
   - (b) 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 (as in the mobile nodes algorithm).

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 $n \in \mathcal{N}$, 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 $V_c = V_n B_k$, where B_k is the backwards extension of V_c to V_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.$\square$

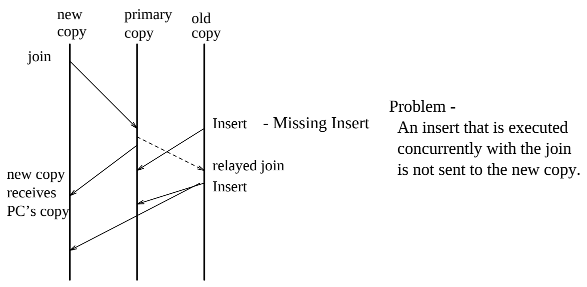
new copy
primary copy
old copy
join
Insert
- Missing Insert
Problem -
An insert that is executed concurrently with the join is not sent to the new copy.
relayed join
Insert
new copy receives PC's copy

Figure 7: Incomplete histories due to concurrent joins and inserts.

Discussion This section synthesizes the techniques presented in the previous two sections, contributing a method for ensuring that every copy has a complete history. If the PC wishes to give up its responsibilities, it contacts a non-PC copy c' to transfer information and responsibility for being the PC. At this point c' becomes the PC. The old PC then informs all other non-PC copies of the new PC, and waits for their acknowledgement. If an unjoin request arrives, it is relayed to the new PC, c'.

4 Handling Deletes and Merged Nodes

The discussion of the previous section assumed that all operations are search or insert operations. In this section, we discuss how to handle the deletion of nodes, and more generally the shifting of responsibility for a key range between two nodes.

If only leaf nodes may be deleted, then the insert and the delete of the pointer to the leaf node are ordered actions – the delete must come after the insert. Unfortunately, the insert and delete are not overwriting actions (as in Section 3.2). If a node is asked to delete a pointer that does not exist (but is in its key range), the delete action is delayed until the corresponding insert action arrives. The *delayed delete* action is remembered at the copy, and is executed immediately after the corresponding insert action is executed.

When a node is deleted, some care must be taken to preserve the double linked list in which it exists. That algorithm is described in [16]. One leaf node might transfer some of its key range to an adjacent leaf, in order to perform rebalancing [7]. The parent must be informed of the shift in the key range. This action is a link-change action, and is an ordered action.

If the key ranges of the interior nodes can increase (due to the merging or rebalancing of an interior node), then some new problems involving the synchronization of insert or delete actions with split and merge actions can occur. In particular, a relayed insert (delete) might arrive at a copy of a node after the node has split off and then merged back a portion of its key range. If the anti-action occurred while keyspace of the action existed at a different node, then the action might be performed twice.

For example, consider the execution illustrated in Figure 8. The initial action $I(k)$ is performed at copy c_1 and relayed to the other copies. The key range containing k is split off from the node, then later merged back in. A relayed insert $i(k)$ is performed at the PC before the merge, and at copy c_2 after the merge. If key k is deleted at the sibling before the merge, then $i(k)$ at c_2 inserts k after all $d(k)$ actions are quiescent.

4.1 Merged Histories

In Section 2, a replicated node can only lose key space (i.e., split into two nodes). Because the histories of two nodes are never shared, we avoided the issue of specifying the initial values and backwards extensions of

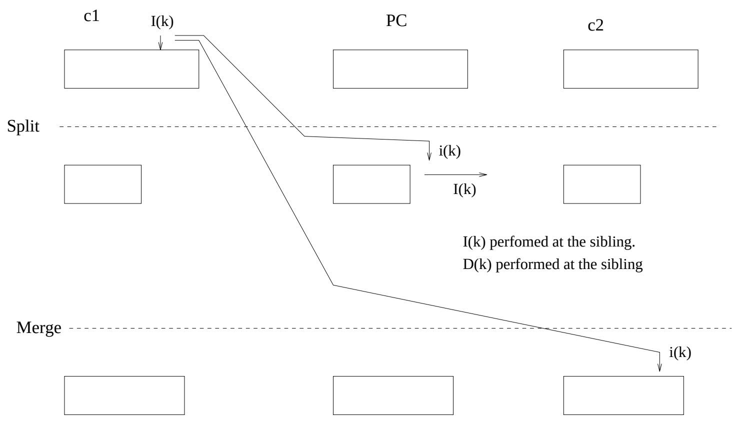

Figure 8: Duplicated operations due to merged histories.

split nodes. When a node gains a new key space (due to a merge or a rebalancing), it also gains the history of the actions that occurred on that key range. In this section, we discuss how the history of a merged node can be constructed.

We need to distinguish between two kinds of actions – actions on the node (*n-actions*), and actions on entries that a copy of the node contains (*r-actions*). An n-action specifies a node as its target, whereas a r-action specifies a key value as its target.

Given a copy of a node, c, let $range(c)$ be the key range of the entries that can exist in c. Given an action, let $range(a)$ be the key range that is the target of a. Let R be a key range. We define $H(R)$ the be the set of all r-actions a that were fired on some copy c and such that $range(a)$ is in R. That is,

$$H(R) = \{a | a \text{ was fired on } c \text{ and } range(a) \in R\}$$

When a replicated node engages in restructuring, it can split and create a new node, delete itself and give its key range to a neighboring node, send a portion of its key range to a neighboring node, or accept a portion of the key range of a neighboring node. Each of these n-actions changes the key range of a node, either by increasing it or by decreasing it. We call a n-action that decreases the node range a *split* action, and one that increases the key range a *merge* action. Note that a split action might send its key range to another node that executes a merge action, or it might create a new node. If a node is deleted, it splits off its entire key range, and then becomes dormant.

When a node n splits off a portion R' of its key range, it sends the entries (i.e., key range and pointer pairs) in R' to node n'. These entries correspond to a history $H(R')$ of r-actions such that $a \in H(R') \Rightarrow range(a) \in R'$. This history is received by n' and is incorporated into H_c for every $c \in copies(n')$ when c executes the corresponding merge action. We denote the incorporated actions due to a merge as *m-action*.

An m-action modifies the value of a copy, and does not instigate subsequent actions.

Given a history of a copy, the record of a r-action, a, might appear several times. The first appearance might be due to an initial or a relayed action (i.e, not an m-action), the subsequent appearances are due to m-actions. Yet, there might be no guarantee that the effects of the action are reflected in the final value of the copy. To account for this possibility, we need to make the following definition.

Def: An action a *appears in* history H_c of copy c if there is an a_i in H_c such that a_i is the same action as a, and there is no a_j in H_c, $j > i$ that removes $range(a)$ from $range(c)$.

Next we need to replace the complete history requirement with one that better expresses the way histories are transferred between nodes.

Merge-Complete History Requirement: If every subsequent action that is issued appears in the history H_c of a node copy c once, then the computation meets the *merge-complete history requirement.* If every computation that an algorithm produces satisfies the complete history requirement, then the algorithm satisfies the *merge-complete history requirement.*

The merge-complete history requirement only requires that there is some copy of a node such that an issued action appears in the history, and appears in the history only once. The compatible history requirement ensures that all copies of the node are also merge-complete.

4.2 Algorithm

The problem that can occur when nodes can merge as well as split is that actions can be executed twice at some copies. This problem can occur if an action occurs at the PC before the split, and at a non-PC copy after the merge.

To detect a potential problem, we divide a node's lifetime into *epochs.* When a node is created, it has epoch number 0. During an epoch, the node may execute a number of split actions. A new epoch begins when the node executes a merge, and the node has executed at least one split during the epoch. Each copy of a node keeps track of its current epoch, and also the number of split actions it has executed. These numbers are attached to every relayed action that a copy relays to the other copies of a node.

When a relayed r-action a arrives at a copy c, and has an epoch number smaller than the c's epoch, c next checks a's split number. If a's split number is smaller than c's split number, then the a might have been lost or duplicated during the change in epochs. Let e be the node pointer in the parameter p of a. The copy c performs the following protocol:

1. If c is the PC, then c executes a and transmits its state with respect to e to all copies.

2. If c is not the PC, then c asks the PC for its value of the state with respect to e. Copy c uses the reply from the PC to makes its state consistent with respect to e.

Theorem 5 *The algorithm for handling node merges is correct.*

Proof: Suppose a copy of a node executes an initial action A, and all relayed actions arrive at all other copies during the same epoch. Then, the previous correctness arguments still apply. So, suppose that a relayed action a arrives at a copy c when c is in an epoch later than the one when A was executed. In this case, the copy's history is made consistent with the PC's history. By the split protocol, all actions are executed once. The PC will execute the actions only once, so all copies will execute the action only once. □

5 Applications to Other Structures

The techniques described in this paper are not limited to the implementation of the dB-tree. For example, there is no requirement that the nodes of the distributed search structure be of a limited size. The distributed search structure might be limited to having three levels, If an average node stores 10,000 pointers, this is sufficient to index one trillion data items. In addition, the storage overhead per processor is limited by the shallow tree depth. We can also apply the techniques to search-structures that support multi-attribute range queries (such as the hB-tree [25]), provided that they support the link technique.

To illustrate the application of these methods to other distributed search structures, we analyze a distributed hash table due to Ellis. In [11], Ellis describes a distributed and concurrent extendible hash table. An *extendible hash table* is one type of hash table that increases the number of buckets available in the table in response to increasing storage demand (such hash tables are called *dynamic hash tables*). The extendible hash table contains a set of data buckets, and a directory that contains pointers to the data buckets. Initially, the hash table contains a single bucket, and the hash directory contains a single entry pointing to that bucket. When this bucket fills, it splits into two buckets. The original bucket is labeled '0' and the new bucket is labeled '1'. All data items whose hash function's least significant bit is a '0' are put in the bucket labeled '0', and the remaining keys are placed in the bucket marked '1'. The directory doubles in size to accommodate the new pointer. In general, when a bucket labeled 'tag' becomes too full, it splits into two buckets, the original marked '$0tag$' and the new bucket marked '$1tag$'. The keys from the original bucket are distributed among the original and the new bucket based on which of the labels the least significant bits of their hash functions match.

The directory size is 2^l, where l is the number of bits in the longest label. The directory is indexed by the least significant l bits of the hash function of the input key. If there is no bucket that is labeled with the directory index, the directory entry points to the bucket with the longest matching suffix. If a bucket splits to produce buckets with longer labels than any other in the directory, the directory doubles in length to accommodate the new labels. If a bucket with label $x\ tag$ becomes empty, it can merge with the bucket labeled $!x\ tag$, if it exists.

Ellis observed that the extendible hashing algorithm can be made into a highly concurrent hash table by linking together the buckets [10]. Suppose that a bucket with label tag is ready to split, and tag is not one of the longest labels in the hash table (that is, $|tag| < l$). Then there are directory entries with suffixes $0tag$ and $1tag$, both of which point to the bucket labeled tag. When the bucket splits, it is relabeled $0tag$ and the new bucket is labeled $1tag$. The bucket labeled $0tag$ stores a pointer to the bucket labeled $1tag$. If an operation read the a directory entry labeled with a suffix of $1tag$ and is directed to the bucket labeled $0tag$, it can detect the misnavigation and determine the correct bucket to access. As a result, bucket splitting can be performed concurrently with directory accesses.

Ellis further observed that the concurrent extendible hash table can be made into a distributed extendible hash table in which multiple copies of the hash directories exist [11]. The key observation is that since operations can recover from misnavigation, the directory copies can contain outdated information. In this section we analyze the distributed algorithm using lazy updates.

A picture (taken from [11]) of a distributed extendible hash table is shown in Figure 9. There are two copies of the directory, three active buckets, and one deleted bucket. The buckets are double linked. The forwards link is used to recover from outdated directory information, and is generated according to the rules discussed above. The backwards link is used to maintain the list. There are two incorrect links in the two directories. Directory 1 has not yet been informed that bucket '0' has split into buckets '00' and '01', so its entry for '10' points to the wrong bucket. Directory 2 has not yet been informed that bucket '11' was merged into bucket '01', so its entry for '11' points to a deleted bucket.

The three types of operations on the hash table: search, insert, and delete. A search operation reads a local copy of the directory, determines a bucket to visit, then sends itself to the processor that manages the bucket. When the operation reaches the bucket, it traverses the links until it finds the correct bucket, then performs the search on the bucket. The insert and delete operations similarly navigate to the correct bucket, then perform their operations. If the bucket becomes too full (or empty) it is split (or merged). The directories are informed of the restructuring, and they update their directory entries accordingly.

Ellis' algorithm maintains the doubly-linked list of buckets using a locking-style algorithm. We do not consider the bucket management further. The directories process three types of actions: search, split update, and merge update. The search structure has been set up so that search actions on the directories are never

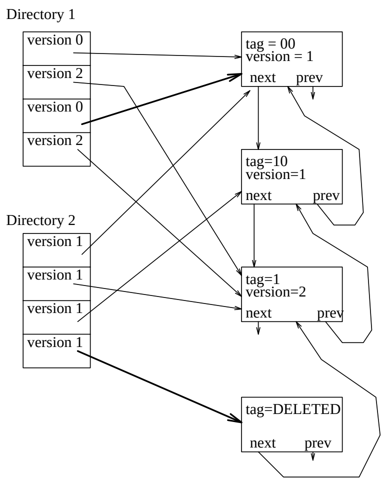

Figure 9: A distributed extendible hash table (figure from Ellis).

blocked. In addition, an action on one directory entry does not affect, and therefore does not need to block, another directory entry. However, actions on the same directory entry must be performed in the order issued (since they are link-change actions).

In Ellis' algorithm, each bucket and each directory entry contains a version number. Whenever a bucket splits or merges, all buckets involved increase their version number so that it is one greater than the previous version numbers. Since directory entries refer to buckets, these version numbers can be used to satisfy the ordered history requirement.

There is one complication, namely that a split update or a merge update action can affect many directory entries. If the length of the longest bucket label is l, and the label of the bucket undergoing restructuring is t bits, then the update affects 2^{l-t} bits. Ellis blocks an update until it is the succeeding update for all directory entries involved. We note that at any particular directory entry, the updates are overwriting updates. Thus the same effect can be had by applying the ordered overwriting update algorithm at each entry individually.

Suppose that the set of directories is fixed. Then since no operation blocks any other and the ordered history requirement is satisfied, our modification of Ellis' algorithm is a correct lazy-update algorithm on the directories. Furthermore, no PC is necessary. If directories can join and unjoin the replication, an agreement protocol is necessary (such as the one discussed in section 3.3).

We note one final detail, which involves garbage-collecting the deleted nodes. As is shown in Figure 9, a deleted node drains into an (currently) live node. In Ellis' algorithm, deleted nodes are garbage-collected only after all directories can guarantee that no operation will access that node. Since a lost operation can recover its path by searching a directory, there is no need to synchronize garbage collection.

Discussion As this section shows, the techniques described in this paper can be used to design a distributed search structure starting from a concurrent search structure. The implementor can take the following steps:

1. Select a concurrent implementation of the search structure that uses the link technique.
2. Translate the concurrent algorithm into a distributed algorithm.
3. Determine the ordered actions and the commutativity between actions.
4. Apply the algorithms in section 2 to manage the node copies.

In addition to hash tables and B-link trees, additional concurrent search structures that use the link technique have been proposed. For example, Parker [29] gives an a link-type algorithm for a concurrent trie. A multiattribute range query search structure, such as the hB-tree [25] can also be modified to serve as a distributed search structure.

6 Failures and Recovery

The algorithms in this paper have not explicitly accounted for processor failures and recovery. However, because lazy updates require little synchronization, a message recovery strategy (such as that discussed in [22, 36, 14, 35, 4] can be applied. When a server recovers, it recovers its message state, and applies the actions it received.

7 Conclusion

We present algorithms for implementing lazy updates for distributed search structures. Lazy updates avoid the need for synchronization between copies, and permit concurrent searches and concurrent updates. In other works [1, 21] we discuss implementation and performance issues, while this work concentrates on algorithms and correctness proofs. The application of lazy updates is demonstrated by their application the the dB-tree, a distributed B-tree. After presenting algorithms and proof for lazy updates, we discuss methods for designing distributed search structures.

References

[1] Krishna P. A. and Johnson T. Index replication in a distributed b-tree. In *Conference on Management of Data*, pages 207–224, 1994.

[2] 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.

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

[4] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. Fault tolerance under UNIX. *ACM Transactions on Computer Systems*, 7(1):1–24, 1989.

[5] K.M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, 1985.

[6] A. Colbrook, E.A. Brewer, C.N. Dellarocas, and W.E. Weihl. An algorithm for concurrent search trees. In *Proceedings of the 20th International Conference on Parallel Processing*, pages III138–III141, 1991.

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

[8] S.B. Davidson, H. Garcia-Molina, and D. Skeen. Consistency in partitioned networks. *Computing Surveys*, 17(3):342–370, 1985.

[9] M. Dietzfelbinger and F. Meyer auf der Hyde. An optimal parallel dictionary. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 360–368, 1989.

[10] C.S. Ellis. Extendible hashing for concurrent operations and distributed data. In *Proceedings of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 106–116, Portland, OR, 1983.

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

[12] K.Z Gilon and D. Peleg. Compact deterministic distributed dictionaries. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, pages 81–94. ACM, 1991.

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

[14] D.B. Johnson and W. Zwaenepoel. Recovery in distributed systems using optimistic message logging and checkpointing. *Journal of Algorithms*, 11:462–491, 1990.

[15] 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.

[16] T. Johnson and A. Colbrook. A distributed, replicated, data-balanced search structure. To appear in the Int'l Journal of High-Speed Computing. Available at ftp.cis.ufl.edu:cis/tech-reports/tr93/tr93-028.ps.Z, 1992.

[17] 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.

[18] T. Johnson and D. Shasha. The performance of concurrent data structure algorithms. *Transactions on Database Systems*, pages 51–101, March 1993.

[19] T.A. Joseph and K.P. Birman. Low cost management of replicated data in fault-tolerant distributed systems. *ACM Trans. on Computer Systems*, 4(1):54–70, 1986.

[20] P. Krishna and T. Johnson. Implementing distributed search structures. Technical Report UF CIS TR92-032, Availiable at anonymous ftp site cis.ufl.edu, University of Florida, Dept. of CIS, 1992.

[21] P.A. Krishna and T. Johnson. Highly scalable data balanced distributed b-trees. Technical Report 95-015, University of Florida, Dept. of CISE, 1995. Available at ftp.cis.ufl.edu:cis/tech-reports.

[22] R. Ladin, B. Liskov, L. Shira, and S. Ghemewat. Providing high reliability using lazy replication. *ACM Trans. Computer Systems*, 10(4):360–391, 1992.

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

[24] 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.

[25] D.B. Litwin and B. Salzberg. The hB-tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. Database Systems*, 14(4):625–658, 1990.

[26] W. Litwin, M. Neimat, and D.A Schneider. LH* – linear hashing for distributed files. In *Proc. 1993 ACM SIGMOD*, pages 327–336, 1993.

[27] G. Matsliach and O. Shmueli. An efficient method for distributing search structures. In *Symposium on Parallel and Distributed Information Systems*, pages 159–166, 1991.

[28] G. Matsliach and O. Shmueli. An efficient method for distributing search structures. In *Proceedings of the First International Conference on Parallel and Distributed Information Systems*, pages 159–166, 1991.

[29] J.D. Parker. A concurrent search structure. *Journal of Parallel and Distributed Computing*, 7, 1989.

[30] D. Peleg. Distributed data structures: A complexity oriented view. In *Fourth Int'l Workshop on Distributed Algorithms*, pages 71–89, Bari, Italy, 1990.

[31] A. Ranade. Maintaining dynamic ordered sets on processor networks. In *Proc. ACM Symp. on Parallel Algorithms and Architectures*, pages 127–137, 1992.

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

[33] B. Seeger and Larson P. Multi-disk b-trees. In *Proceedings of the 1991 ACM SIGMOD Conference*, pages 436–445, 1991.

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

[35] A.P. Sistla and J.L. Welch. Efficient distributed recovery using message logging. In *Proc. Symp. on Principles of Distributed Computing*, pages 223–238, 1989.

[36] R.E. Strom and S. Yemeni. Optimistic recovery in distribtued systems. *ACM Transactions on Computer Systems*, 3(3):204–226, 1985.

[37] R. Vingraek, Y. Breitbart, and G. Weikum. Distributed file organization with scalable cost/performance. In *Proceedings of the 1994 ACM SIGMOD conference*, pages 253–264, 1994.

[38] Litwin W., Neimat M., and Schneider D. A. Rp* - a family of order-preserving scalable distributed data structures. In *Proceedings of the 20th VLDB Conference*, pages 342–353, 1994.

[39] P. Wang. An in-depth analysis of concurrent b-tree algorithms. Technical Report MIT/LCS/TR-496, MIT Laboratory for Computer Science, 1991.

[40] W.E. Weihl. The impact or recovery on concurrency control. Technical Report MIT/LCS/TM-382b, MIT Laboratory for Computer Science, 1989.

[41] 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.

[42] M.H. Wong and D. Agrawal. Context-based synchroniczation: An approach beyond semantics for concurrency control. In *Proc. 1993 ACM SIGMOD*, pages 278–287, 1993.

[43] I.L. Yen and F. Bastani. Hash tables in massively parallel systems. In *Int'l Parallel Processing Symposium*, pages 660–664, 1992.