Distributing a Search Tree Among a Growing Number of Processors

Brigitte Kröll and Peter Widmayer

Institut für Theoretische Informatik
ETH Zentrum, CH-8092 Zürich
Switzerland

Abstract

Databases are growing steadily, and distributed computer systems are more and more easily available. This provides an opportunity to satisfy the increasingly tighter efficiency requirements by means of distributed data structures. The design and analysis of these structures under efficiency aspects, however, has not yet been studied sufficiently. To our knowledge, a single scalable, distributed data structure has been proposed so far. It is a distributed variant of linear hashing with uncontrolled splits, and, as a consequence, performs efficiently for data distributions that are close to uniform, but not necessarily for others. In addition, it does not support queries that refer to the linear order of keys, such as nearest neighbor or range queries. We propose a distributed search tree that avoids these problems, since it inherits desirable properties from non-distributed trees. Our experiments show that our structure does indeed combine a guarantee for good storage space utilization with high query efficiency. Nevertheless, we feel that further research in the area of scalable, distributed data structures is dearly needed; it should eventually lead to a body of knowledge that is comparable with the non-distributed, classical data structures field.

1. Introduction

As databases become larger and larger, and distributed computer systems are more and more easily available, the need for distributed data structures increases constantly. In the past years, not too many investigations on distributed databases have taken distributed data structures into account. For a fixed number of processors, several different data structures have been proposed (Matsliach et al. 1990, 1991, Pramanik et al. 1990). The focus of the few who concentrated on dynamic distributed data structures was not on efficient access to the data alone, but on a combination with other relevant aspects such as, e.g., concurrency control (Ellis 1985, Gudes et al. 1989, Johnson et al. 1993, Weihl et al. 1990) or other aspects (Bastani et al. 1988, Ladin et al. 1991), with one notable exception that generalized linear hashing to a distributed environment (Litwin et al. 1993), calling the result LH*. The latter took an isolated view on efficiency considerations, disregarding other aspects completely. We believe that LH* should be an important initiator of a whole track of research, namely the design and analysis of distributed data structures under efficiency aspects only. In our view, it is worthwile to start with such a seemingly narrow-minded approach, in order to clearly and fully understand the implications of distributed data structure design decisions on efficiency, just as it has been the case with classical data structures for a single processor.

This paper reports on a step in this direction. Just as with single processor data structures, where address computation in hashing techniques and key comparisons in tree structures coexist and have their respective merits for different purposes and goals, more than one paradigm is expected to lead to useful distributed data structures. As a step in this direction for distributed data structures, here we study the efficiency of distributed search trees. The merit of search trees against hashing techniques lies - in the distributed case as well as in the classical one - in the fact that trees are good in the worst case, i.e., adapt to any data distribution, and that they preserve the order of the keys, thereby supporting range queries and nearest neighbor queries.

The problem in designing an efficient distributed search tree lies in the need to maintain a collection of replicated parts of the tree at the various processors. If no part of the tree would be replicated, the root (or, more generally, the entry nodes into the tree) would become a bottleneck. If an up-to-date replication of the entire tree would be available at all processors at all times, maintaining the replications would

SIGMOD 94- 5/94 Minneapolis, Minnesota, USA
© 1994 ACM 0-89791-639-5/94/0005..$3.50

overwhelm the system. Lazy updates are a good way out of the maintenance problem (Johnson et al. 1993). LH* can be characterized, in fact, as linear hashing with full replication of the (constant size) directory information at all processors, where replicas are allowed to become obsolete, together with a technique for lazy updates. In the distributed random tree DRT method we propose in this paper, lazy updates also play an important role. They are, however, not the only constituent: In addition, we propose not to replicate the entire tree at all nodes and let it become gradually obsolete, but to replicate only some appropriate part of the tree for each node, and to do this in such a way that together with lazy updates, the node can (barely) perform its part of the operation of the DRT method.

The important question here is how to properly decide on two interrelated techniques: First, which parts of the tree are to be replicated at which nodes, and how should this be done, and second, to which extent should the replicas be allowed to become obsolete, while still guaranteeing correctness and overall efficiency. We will propose a technique that maintains as replicas potentially large parts of the tree and lets them become obsolete gradually, based on a simpler technique that does not even replicate many times large parts of the tree, but instead distributes the tree such that only one half of the nodes are replicated. Our technique is applicable to tree structures that grow and guide the search like random trees, such as the standard random search trees as well as all kinds of digital trees (tries). An analysis and performance experiments compare the DRT method with LH*, indicating that it is indeed useful to have both structures available. Especially, whereas in LH* scalability and high storage space utilization are to some extent contradictory goals, the DRT method can achieve both of them to a high degree. As expected, the DRT structure is more robust against skewed data distributions than LH*, just as linear hashing in the single processor case is better suited for smooth data distributions close to uniform, whereas search trees are better in the worst case.

In the next section, we give a precise problem definition. Section 3 describes the solution we suggest, the distributed random tree structure DRT. In Section 4, we prove the correctness of the DRT method for operations that are carried out simultaneously. Section 5 argues on the efficiency of the DRT method and compares it experimentally with LH*. In Section 6, we discuss a number of variants that suggest themselves on the basis of the DRT structure. The purpose of this section is to show that the DRT structure can be tuned to some extent to the particular needs of the user. Section 7 offers a conclusion.

2. The model and the problem

We assume that a given finite set of *sites* (processors) is connected by a message passing network (following Litwin et al. 1993). Each site in the network is either a *server* that stores data or a *client* that initiates requests. Each site (server or client) can communicate with each other site by sending and receiving messages (communication between clients will not be needed in our algorithms, though). Each site has a unique identifier, realized by its unique address in the network. A site can send a message to any other site, given an identifier of the destination site. We therefore use two notions interchangeably, the *identifier of a site* and a *pointer to* (network address of) *a site*. With each site, a *message queue* is associated that handles all incoming messages in the order in which they arrive. The network communication is free of errors. All clients together operate on a common file that is to be distributed over the servers.We have the freedom of distributing the data in an arbitrary way; that is, we do not consider the case in which the distribution of data has to follow some pre-specified pattern. As to the data on the file, we restrict our view to a set of linearly ordered *keys* that are stored on the file, representing arbitrary objects whose nature is irrelevant for access purposes. In this set, we want to efficiently *insert* a given key, *delete* a key, *search* for a key, search for the *nearest neighbor* of a key, and answer *range queries*, that is, search for all the keys in a given interval.

Each server can store at most a fixed, constant number b of keys. The storage space that a server provides will be treated as a single *block of capacity b*, because for our purposes, the maintenance of the keys within each server's storage space is irrelevant. This implies that it does not matter whether the keys at a server are stored in main memory or on external storage. As an aside, note that the distributed tree that we propose in the next section will very naturally allow to have sites with different capacities, because we might just as well associate more than one block with a site, thus keeping the block capacity constant. Efficiency in our model is measured solely in the *number of messages* exchanged between source and destination sites. That is, the length of a message and the network topology do not influence the cost of a message transmission; even if a message from one site to another travels along a path in the network, it has unit cost (this is in line with Litwin et al. 1993, because we want to abstract from the network topology for the time being).

Litwin et al. (1993) impose the following requirements on a *scalable, distributed data structure* (*SDDS*):

1. It should be *dynamic*, in the sense that the number of servers sharing the file should adapt gracefully to the number of keys to be stored. This requirement reflects the

expectation that the file may grow substantially during its lifetime (and perhaps also shrink).
2. It should be *scalable*, in the sense that there is no distinguished server that might become a bottleneck when the number of sites grows substantially. We will not be able to fully cope with this requirement (but almost), just as LH* cannot fully cope with it, because both methods make use of a distinguished server (in LH* the split coordinator site).
3. *No large update* should ever be necessary, in the sense that at any point in time, not too many sites should be forced to handle messages that inform about (past) changes of the data structure, and to adapt themselves to these changes. This requirement is motivated by the overall goal to avoid all kinds of bottleneck.

The track of research that is initiated by LH*, therefore, is the search for SDDSs. Our proposal for a SDDS that is worst-case efficient in storage space utilization and maintains the order of keys is the *distributed random tree* structure *DRT*. In addition to requirements 1 through 3, the DRT structure guarantees a satisfactory *storage space utilization*. More specifically, we request that utilization is at least a fixed, constant percentage at all times. The DRT method will turn out to guarantee 50% storage space utilization if no deletions occur, with an expected value of *ln* 2 for uniformly distributed keys (just like B-trees). Also, the DRT structure supports *nearest neighbor queries* and *range queries*, in addition to the usual dictionary operations (*exact search*, *insert* and *delete*). The motivation for supporting these queries is manyfold, depending on the application of the data structure; for instance, for an unsuccessful exact search one might wish to know the key that is closest to the one for which the search failed. Furthermore, *no overflow blocks* are allowed in the DRT. The motivation is that the search should be guided equally well to all keys. In other words, overflow blocks imply that there is only one path in the data structure guiding the search to more than one site, and we feel that the lack of a distinguished search path to every single site should not be tolerated. Clearly, this requirement aims at good worst case (rather than average case) performance.

3. The distributed random tree structure DRT

To solve the described data organization problem, we propose to generalize a random binary leaf search tree (Knuth 1973) to the distributed case. Each leaf of the tree represents a block (that is, a server); it actually stores the unique identifier (pointer) for that block (for simplicity, we identify a block with its server). Interior nodes of the tree store routing information that guides the search. An insert, delete or search operation starts at the root of the tree and proceeds to a leaf, according to the routing information. Then, the requested key is inserted, deleted or searched in the block that the leaf represents (that is, points to). Tree T in Figure 1 visualizes a distributed random binary tree.

Since we are free what to choose as routing information, this scheme embeds both, the usual search trees with arbitrary keys, and digital search trees (tries), in which a prefix of each key in a block is represented as a sequence of digits that are assembled along the path from the root to the block's leaf. This offers an interesting spectrum of performances, with the latter tending to behave similar to adaptive hashing schemes. In the sequel, we restrict the description to the usual search trees based on key comparisons.

3.1 Conceptual tree growth

The problem we face lies in the fact that we do not want all clients or even all sites to know this current *global tree* T at all times, since informing all clients or sites about each change of T would violate our requirement 3 that forbids large updates. On the other hand, maintaining T at some distinguished server and consulting it there whenever necessary would violate our requirement 2 that asks for scalability. We solve the problem by breaking up the global tree T and distributing appropriate information over the sites.

Even though the global tree T is merely a concept and therefore not necessarily a data structure stored at a site, it is useful to reason about global aspects of the behavior of the DRT structure in terms of T, regardless of where information on T is stored. Especially, this is the case for overall tree growth. Since the file must adapt itself dynamically to a changing number of blocks, we adjust the global tree to a changing number of leaves. Whenever a block B overflows, due to an insertion, the server of B determines the address (identifier) of an additional block (server) B' that starts to participate in storing the file. A simple method for identifying the new server consists of consulting an administration site that keeps track of all participating servers; other methods are possible, including a randomized choice. The leaf in the tree pointing to B (the leaf of B, for short) is replaced by an interior node with two leaves as its children. One of the leaves points to B, the other points to B' (it does not matter which of the leaves points to B and which to B'). The keys of the former block B are distributed among B and B', and the routing information is adjusted accordingly. We choose the distribution so as to balance the space utilization of both blocks, just as in a B-tree split. The server of B informs the server of B' by sending it a wakeup and initialization message, together with the keys for B', and it waits for an acknowledgement before doing anything else. This acknowledgement as well as the response from the administration site are not buffered as usual in the message queue at B; instead, they enter a separate acknowledgement message queue for B, and they

are processed by B with higher priority than the other messages. It will turn out that the acknowledgement message from B' serves to make sure that the initialization message is the first message that B' processes.

In this report, we refrain from discussing merge operations triggered by deletions; they make the problem more complicated, and they are also less relevant in practice (see also Litwin et al. 1993).

3.2 Tree distribution and growth of local trees of servers

Instead of informing all clients about the global tree T at all times, we distribute T by maintaining at each server s some *local* part $T(s)$ of tree T. Roughly speaking, $T(s)$ essentially keeps track of that part of T in whose creation s was involved. An example for such a set of trees for servers $s_0,...,s_6$ is shown in Figure 1. Let us now explain how these trees are created by a sequence of insertions of keys into a tree consisting of one server initially.

Let T_0 be the initial tree, consisting of one leaf only that points to the only server s_0. (In more generality, T_0 may be a large tree, with each leaf of T_0 pointing to a server.) Whenever a server s start participating in the method, it initializes $T(s)$ as a leaf that points to s.

The servers keep track of the growth of T in a number of ways; one of them is the following. Whenever a server s splits its block, an additional server s' is included in the set of servers storing the file. Server s' initializes its local tree $T(s')$ as a leaf that points to s', just like all other servers at the beginning. Server s keeps track of the split by replacing its single leaf by an interior node with two leaves, pointing at s and s'. The interior node stores the router that distinguishes between the keys at the two servers. In our example, let the sequence of splits that is triggered by a sequence of insertions of keys, starting with s_0 as the only server, be the following. First, s_0 splits into s_0 and s_1; then s_0 splits into s_2 and s_0, yielding tree $T(s_0)$ shown in Figure 1. Now s_2 splits into s_3 and s_2; then s_1 splits into s_1 and s_4. Then s_2 splits into s_2 and s_5, yielding $T(s_2)$ in Figure 1; s_1 splits into s_6 and s_1, yielding $T(s_1)$ in Figure 1. Note that $T(s_3),...,T(s_6)$ are still in their initial state.

To see in which way T is distributed among the servers, let us call the node in T where s started its participation in storing the file, being the new server involved in a split, the *birthnode birth(s) of s in T*. For example, in Figure 1, node v is the birthnode of s_2. In this terminology, $T(s)$ stores the path in T from *birth(s)* to the leaf pointing to server s, plus a sibling of each node (except the birthnode) on this path. Sibling nodes of path nodes come into existence whenever the block of s splits; hence, they are the birthnodes of the corresponding new servers. The path turns left or right, depending on whether the subset of lower or higher keys is stored at s (the other subset is stored at the new server).

In this way, T is partitioned into local trees associated with servers, except for a redundancy that adds an extra half of the nodes of T. This redundancy lies in the fact that each birthnode is stored twice: Once, it is the root of a local tree, and once it is a leaf in a local tree.

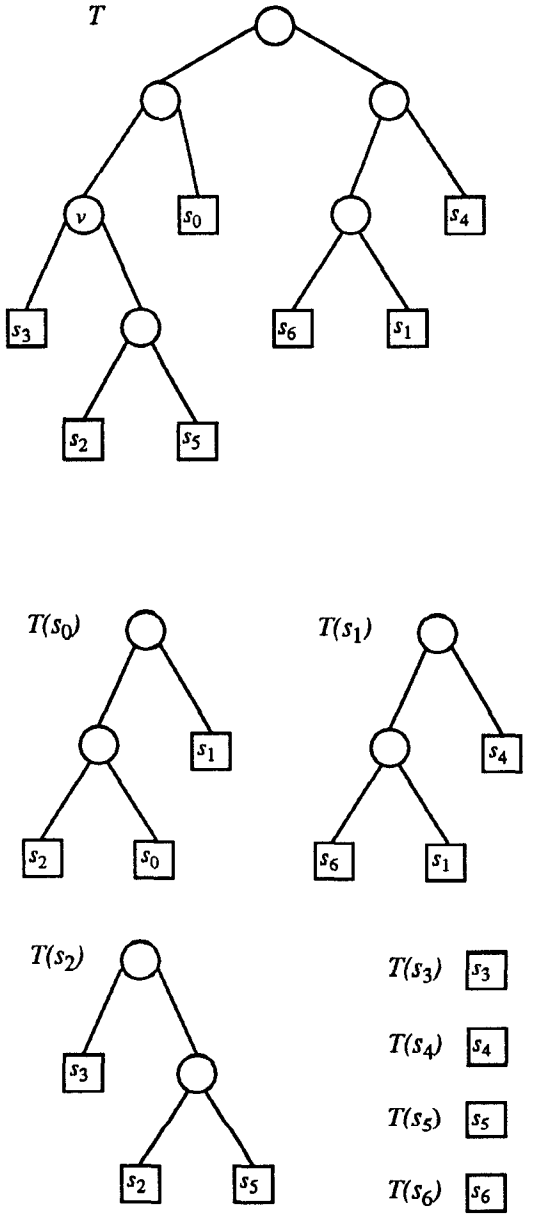

Figure 1: A global tree T, and local trees $T(s_0),...,T(s_6)$ of servers.

3.3 Processing requests of clients

Since not all clients know the current global tree T at all times, let us explain how to process the requests of clients. Each client c holds a *local client's tree* $T(c)$ representing the knowledge of c about T. In general, $T(c)$ is a part of T containing at least T_0; initially, $T(c)=T_0$ for all clients c. Figure 2 shows an example client's tree $T(c)$, representing the knowledge of c about T in Figure 1; here, c is aware of $T(s_0)$ and $T(s_1)$, but not of $T(s_2)$. We will later explain how to maintain clients' trees in general. Because only a part of T is available at a client, it is in general impossible for a client to determine from $T(c)$ alone the server that stores a given key. We therefore have to tolerate *referencing errors* due to obsolete local client's trees. To argue on the procedure of handling referencing errors, let us introduce some terminology. For each node v in T, we say that v *represents* the set of all keys whose search path in T goes through v. If v is a leaf of T, we say that the server s to which v points *owns* the set of keys that v represents. We assume that for each key, each server can determine whether it owns the key; this can be realized easily by storing with each server the corresponding key interval.

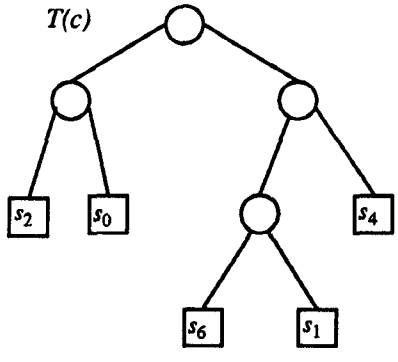

Figure 2: A local tree of a client.

To search for a key k or to insert it, a client c determines the server s that should own k, according to the local tree $T(c)$. It then sends the operation request (search or insert) with k to s. In this situation, s may or may not actually own k. If s determines that it owns k, it performs the requested operation. For an insertion, this may entail a block overflow and a split, to be carried out as described previously; for a search, this entails a lookup in the server's block for k, and a reply message to the client. If, however, s does not own k, a *referencing error* is said to occur. In that case, s searches its local tree $T(s)$ to figure out which server should own k. The search ends in a leaf of $T(s)$ pointing to some server s' different from s; then, s sends the operation request with k to s'. This forwarding of key k from server to server continues after each referencing error, until k finally reaches the server that currently owns k (we will prove later that this is guaranteed to happen eventually) and performs the requested operation. This process defines a *request forwarding chain of servers, chain (c,k)*, for key k coming from client c.

We mentioned above that in this paper, we do not discuss restructuring operations (such as merge) due to deletions. As long as there are not too many deletions, we can simply remove a key that needs to be deleted from the block that stores it, without affecting efficiency too much.

3.4 Lazy updates of local trees of clients

In order to achieve good performance, it must be our goal to keep the number of referencing errors low. Let us look at an example, to see where the problem lies. Assume that initially, T_0 consists of a single leaf pointing to s_0 only, but after some insertions of keys, we arrive at the situation shown in Figure 1. Let us now assume that a client c that has not been active yet wants to insert a key k currently owned by server s_5. From $T(c)=T_0$, c determines that s_0 should own k and forwards the operation request with k to s_0. When s_0 processes that message, it determines that s_2 should own k, and it forwards k to s_2; s_2 in turn finds that s_5 should own k (which it actually does), and it forwards k to s_5. Therefore, s_5 carries out the requested operation. (Note that in case there is a change in the situation, and therefore some local trees change, while the client's request is being forwarded, it will still reach the correct server, although this need not be s_5, and/or the forwarding chain might have changed. We will prove the correctness of the forwarding operation later in this paper.) In our example, the insertion of key k costs three messages (if the block of s_5 does not overflow), two of them due to referencing errors. If now, we would not inform the client c of the new situation and change the local tree $T(c)$ correspondingly, a subsequent search for key k would again cost two referencing errors (that is, two extra messages due to errors).

We aim at reducing the number of messages by updating the local tree $T(c)$ whenever client c commits a referencing error while requesting an operation with key k. To this end, servers do not only forward the request along *chain(c,k)* to the server that owns k, but they also assemble correction information backwards on *chain(c,k)* and pass it on to c. A *correction message* from a server s' to its predecessor in *chain(c,k)* consists of some part of the global tree T, the *correction tree*, containing at least a path from the root of $T(s')$ to the leaf in whose server k was stored. In effect, therefore, the correction tree that client c gets can be used to advance the knowledge of $T(c)$, as far as the server at the end of *chain(c,k)* is concerned.

The described behavior can be achieved as follows (see

Figure 3). A server s' receiving from site s an operation request with key k checks whether it owns k. If so (Case 1), we distinguish two cases, depending on whether s is a client or a server. If s is a client (Case 1a), then s' locally performs the requested operation. If the operation is a search, s' sends a reply message to s, after looking for k in its block; since a message is sent from s' to s anyway, s' sends with it its local tree $T(s')$, in order to reduce the subsequent error chances for s. If the operation is an insert, s' inserts k into its block; if this insertion causes the block to overflow, s' splits the block together with key k among itself and a new server, as described above. Of the two choices, namely to let or not to let s' send a correction message to client s, informing about the new local tree $T(s')$, we decide not to send such a message. The reason is that we prefer to save the message, but run a higher risk of a subsequent referencing error of s that could otherwise be reduced. If s is a server (Case 1b), s' performs the requested operation. After that, s' sends its local tree $T(s')$ as correction tree to s. The reason for the latter is that since s' must send s a message in any case, at least to trigger the transmission of a correction chain, that unavoidable message should carry as much information for the client c as possible. In addition, if the operation is a search, the reply is piggybacked on the correction message. If s' does not own k (Case 2), it determines the server s'' that should own k according to $T(s')$ and sends the request to s''. When s' receives the correction message for the request from s'', it combines the received correction tree with a copy of its own local tree $T(s')$ into a new correction tree (we will describe later how this can be done) which it in turn sends to s. A client that receives a correction tree combines it with its local tree as we will describe in 3.5; this completes the chain of corrections.

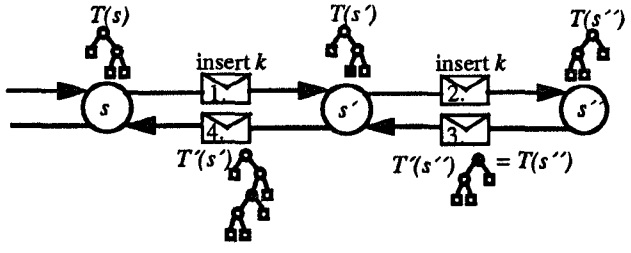

Figure 3: Request forwarding and correction message passing.

3.5 Assembling a correction tree and updating a local tree of a client

In our example, server s_5 sends a correction message consisting of a copy of $T(s_5)$ to s_2. Then, s_2 combines $T(s_5)$ with $T(s_2)$, but this does not change $T(s_2)$; it therefore sends a correction message consisting of $T(s_2)$ to s_0. After that, s_0 combines a copy of $T(s_0)$ with $T(s_2)$ by replacing the leaf in $T(s_0)$ pointing to s_2 with $T(s_2)$, yielding a tree $T'(s_0)$. Then, it sends $T'(s_0)$ in a correction message to client c who, in turn, combines his local tree $T(c)$ with $T'(s_0)$, yielding $T'(s_0)$ as $T(c)$ in our example. If client c now searches for that same key k, it correctly determines from its local tree that s_5 is the server who owns k, without any referencing error (unless s_5 has split in the meantime, of course).

In general, a correction tree $T'(s')$ is built at a server s' from the correction tree $T'(s'')$ it receives from its successor s'' in the operation request forwarding chain, in the following way. $T'(s')$ is a copy of $T(s')$ in which the leaf pointing to s'' is replaced by $T'(s'')$. In the same spirit, a client c updates its local tree $T(c)$ upon receipt of a correction tree $T'(s')$ by replacing the leaf in $T(c)$ that points to s' by some subtree of $T'(s')$ - not necessarily $T'(s')$ in its entirety, because c may already know some part of $T'(s')$. The proper subtree of $T'(s')$ can be identified as follows. First, c searches for the birthnode $birth(s')$ in $T(c)$. This can be realized, for instance, by marking a new leaf that points to a new server r with a *birthmark* r at the time of its creation, by keeping that mark when the leaf is replaced by an interior node, and by searching for the mark for server s' on the path from the root to the leaf pointing to s' (which can be accomplished by searching for the key k of the operation request that triggered the correction). Then, c simultaneously proceeds on the search path for k in $T(c)$ starting at $birth(s')$ and in $T'(s')$ starting at the root, until it reaches the leaf in $T(c)$ pointing to s'. Now, c replaces in $T(c)$ the leaf pointing to s' by the subtree of $T'(s')$ rooted at the node reached in the simultaneous search. This terminates the correction. For an illustration, see Figure 4.

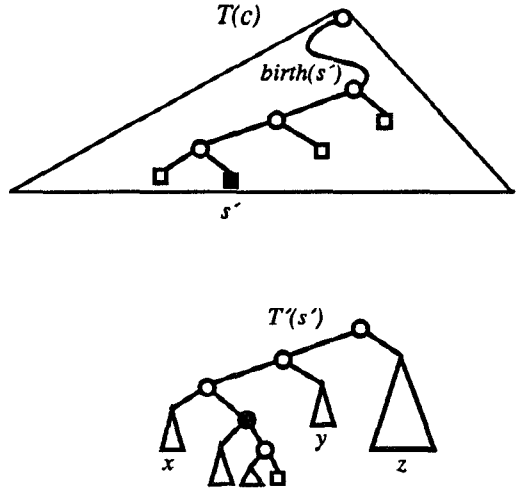

Figure 4: Correcting a local tree of a client.

As to the effect of this correction on performance, we know that it increases the knowledge of c about T. Roughly speaking, for a highly active client, i.e., a client with frequent requests, the local tree comes close to the global

tree in the queried parts of T, whereas the local tree of a less active client may differ considerably from the global tree in all its parts, reflecting the fact that the former has a more up-to-date knowledge of the state of the file than the latter. Hence, there are fewer referencing errors for highly active clients than for less active clients, and since active clients trigger more requests, most requests lead to few errors or none at all.

3.6 Lazy updates of local trees of servers

In the above description, servers pass correction trees along, without making use of the corresponding information. We selected this description as an intermediate step to show the essence of how the global tree T is distributed - with little redundancy - among the servers. In the DRT, however, we certainly make use of all information that comes free of charge. That is, each server corrects its local tree whenever it receives a correction message. Then, the combination of a server's local tree and a correction tree is no longer as simple as we described it; instead, it can be carried out in exactly the same way as the correction of a client's tree. The result will be that each server no longer stores only one path of the global tree T, together with sibling nodes of path nodes, but it now stores a larger part of T. With respect to potential extra corrections of local trees of servers (see section 6), note that for forwarding operation requests, a server s can make use only of the subtree of T rooted at *birth(s)*; therefore, it would be useless to correct $T(s)$ with information beyond that.

Let us look at a particular situation to see just how large the performance improvement due to this design choice might be. Assume that a large number of clients and a small number of servers operate in the system. Then we can expect that on average, each of the servers performs more operations than each of the clients. Hence, the local trees of the clients are significantly more obsolete than those of the servers. Even if now a less active client initiates an operation and, most likely, sends its message not to the server who owns the requested key, the server receiving the message can be expected to start quite a short forwarding chain to the key's destination.

4. Correctness of the DRT Method

So far, we have only considered a single operation in isolation. Let us now show why it is correct to carry out a number of operations concurrently; that is, many operations can be active in the network at the same time. Correctness here refers to the effect of any single one of the required search and update operations; it does not refer to the overall effect of sequences of these operations (viewed as transactions). At each site, one message after the other is removed from the queue; each message leads to a sequence of action steps local to the site, and perhaps a message to some other site. The local sequence of steps is an atomic action in the sense that the site does not take a (further) message from its queue before the sequence of steps has been completed; that is, no concurrent requests can interfere with the atomic action. Hence, let server s perform an insertion with a split of a leaf as an atomic action: More precisely, when the block of server s overflows due to an insertion of a key k, the atomic action of s consists of adding k to the block of server s, requesting a new server s', distributing the keys among s and s', waiting for the acknowledgement from s', and adjusting $T(s)$ by creating a new interior node that refers to the leaf for s and a new leaf for s'. In the same way as explained above for insertion requests causing a split, the initialization, the handling of search requests, of insertion requests that cause no split, and of correction messages are considered to be atomic actions.

Let us now show that each key of a search or an insert operation will eventually arrive at a server that owns it at the time of its arrival. We need not distinguish whether the block of the server s at which key k arrives and which owns k is full or not: If it is full and the requested operation is an insertion, s will trigger a split and distribute the keys, without any possibility of interference of some other operation. In our arguments, we make use of the invariant that whenever key k arrives at server s, either s owns k at present, or s owned k at some point in time in the past. The reason is that all local trees of sites are parts of the global tree T, and in addition, local trees of clients contain the initial tree T_0. This is true because T only grows (it does not shrink), and therefore a split as well as a combination of a local tree with a correction tree keep the local tree a part of T. Hence, whenever a client c determines a server s' to be the one who owns key k, then the search for k must have ended in a leaf v of $T(c)$ pointing to s'. Since $T(c)$ is a part of T containing T_0, and since T only changes by growing leaves, v is or has been a leaf in T pointing to s', and hence s' is or has been owning k. Now, the operation request with key k enters the message queue of s', and s' takes it from there after a finite number of steps (assuming that it takes at least a fixed, positive amount of time to transmit a message or to perform a step of a computation, and therefore the number of messages in the network after a finite time is finite). That is, when s' takes the message from the queue in order to perform the requested operation, the condition that s' is or has been owning k is still satisfied. Unless s' owns k, s' will now determine by a search for a leaf of $T(s')$ a pointer to a server s'' that is or has been owning k (and is different from s'). Since each server handles each operation request at most once and the set of servers is finite, we conclude that after a finite number of steps, an operation request arrives at a server s that does not forward it. But this implies that s owns k, as claimed.

5. Properties and efficiency of the DRT method

First, let us make sure that the DRT method indeed satisfies all of our requirements and let us compare it with LH*.

5.1 Satisfying the requirements

Clearly, the DRT structure is a dynamic data structure with controlled growth (just like LH*).

Scalability is more difficult to assess, for both the DRT method and LH*. To understand some aspects of it, let us pick the particular situation in which the methods start with only one server s. Each user will send his first request to s, no matter when it occurs. Therefore, s wil have to sustain a heavier load than other servers that participate later. This is true for both, the DRT method and LH*. In case all clients know the current tree in a later stage of the processing (for instance, after the majority of insertions has been performed, and queries dominate), all servers share the overall load roughly equally (under appropriate probability assumptions). In general, a client in the DRT method stores a part of the current global tree T, whereas in LH* it stores the indication of a file that is smaller than the current one (no deletions allowed). In effect, a server s whose node *birth(s)* is closer to the root in the global tree T in DRT or whose bucket has a lower address in LH* tends to get a heavier share of the overall load. In addition, if we use an administration site for identifying new servers, scalability suffers to some extent, but not more than scalability in LH* suffers from the use of the split coordinator. Furthermore, if the controlled split (split whenever the average storage space utilization of the file exceeds a fixed threshold) is chosen for LH*, with the goal of getting high storage space utilization, then the split coordinator site is consulted quite often, since it needs to be always informed about the average storage space utilization, and thus adversely affects scalability.

No large updates occur in the DRT method. To see this, note that any operation request message triggers at most one correction message, and out of those operation request messages and correction messages that an operation request induces, at most one is present in the network at any given time (because they are sent and received in sequence). Hence, the number of update messages changes just as the number of operation requests changes. This is also true for LH*.

Storage space utilization in the DRT method without deletions can be guaranteed to be at least 50%; on average for a uniform key distribution, its expected value is $ln\ 2$, that is, roughly 69%, just like in B-trees. For LH* with uncontrolled splits, the storage space utilization can be quite low in the worst case. This is no surprise, since hashing structures even in the single processor case do not guarantee good storage space utilization. It is simply an indication that both, hashing and tree structures, have their respective advantages and disadvantages also in a distributed situation.

Obviously, the DRT structure has no overflow blocks; a few thoughts reveal that it supports nearest neighbor and range queries. The same is not true for LH*.

Unfortunately, we are not in the position to analytically evaluate the average case performance of the DRT method. This is partly due to the lack of our knowledge on practically relevant probability distributions for the sequence of operations to be carried out by the DRT method: We do not know how many users request which operations with which frequencies. Therefore, we will simply point out the performance of the DRT method for a few particular situations, in order to shed some light on the way the DRT method operates. Since LH* is the only data structure that shares a respectable number of goals with DRT, we will compare both structures in a number of performance experiments.

5.2 Efficiency considerations for particular situations

To get a feeling for the efficiency of the DRT method, let us look at a few examples; the performance experiments will support the impression we get from these rather narrowly focussed arguments.

In the worst case, T_0, the global tree T, and the trees of the servers look as shown in Figure 5.

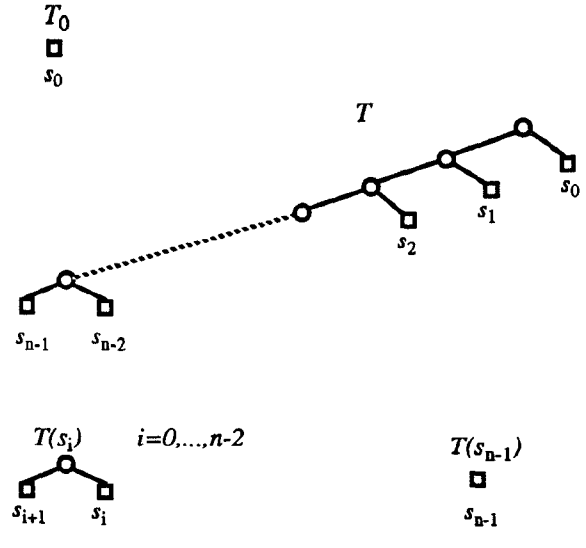

Figure 5: A global tree and trees of servers in the worst case.

This situation is the worst possible for a client whose first

operation is the insertion of a key that server s_{n-1} owns. In this case, the operation request takes n messages, and the correction of the client's tree takes another n messages. That is, a single insertion triggers up to $2n$ messages, where n is the number of servers.

In spite of this pessimistic worst-case bound, we expect a good average case performance from the DRT method. To this end, consider the situation in which the clients operate slowly: Assume that the time interval between two consecutive requests that a client puts on the network is much larger than the time needed for transmitting a message or performing a few local steps on a server. As a likely consequence, let us also assume that a client gets the correction message due to a referencing error before he submits his next request. Then, for l clients inserting a total of m keys into an initially empty structure, altogether involving n servers, we have $n \leq 2m/b$ and therefore, the number of nodes of T is at most $4m/b$. For each client, the local tree grows by at least two nodes, whenever a correction message arrives due to a referencing error. That is, each client causes at most $2m/b$ referencing errors and at most $2m/b$ correction messages. The total cost for building the DRT structure, consisting of the cost for insertion requests, referencing errors, correction messages and splits is therefore at most $m+2ml/b+2ml/b+4n$. In this calculation, we charged four messages for the initialization of a new server, following the strategy that a server s that splits will send a message to the administration site, wait for the response in which it is informed of the address (identifier) of the new server s', send an initialization message to s', and wait for the acknowledgement from s'. The average cost per insertion in such a sequence is then at most $1+(4l+8)/b$. Whenever b is much larger than l, then the average cost per insertion is close to one (which would be optimal); hence, the DRT method is very efficient in this setting.

Similarly, for a sequence of j consecutive search requests of l clients, performed after a set of keys has been inserted involving n servers, each client triggers at most n referencing errors. By a calculation similar to the above, we get an average cost per search of at most $2+nl/j$. Whenever the product of the numbers of clients and servers is fairly small as compared with the number of search requests, the average cost per search is close to two (which would be optimal). From the performance figures given in Litwin et al. (1993), we conclude that these are the situations tested for LH*. For comparison, we will now report on our experiments for the same situations and others.

5.3 An experimental performance comparison

In order to compare the performances of both, LH* and the DRT method, we have implemented both structures with the simulation software package CSIM (Schwetman 1992), just as Litwin et al. (1993) had done before for LH*. The DRT is implemented as proposed above, with none of the variations described in the next section; LH* is implemented as described in Litwin et al. (1993), with the understanding that a block overflow is defined as the situation in which a key is to be inserted in a home block that is already full. In our experiments, we start with one server (for both structures, of course). We let messages arrive without any delay at their destination, and we handle them there without delay.

Two of the three tables on experiments report on the performance of insertions, and one reports on the performance of search operations. For insertions, Tables 1 and 2 describe the results for a sequence of 10,000 insertions into an empty file. One parameter of our experiments is the capacity of a block, where the first two values in Table 1 were chosen to hook our results into those given in Litwin et al. (1993); as expected, our figures for LH* turned out to be the same as those given in Litwin et al. (1993). Other parameters are the number of clients, the number of keys inserted by each client, and the time interval between any two consecutive insert requests of the same client. For instance, in the last test shown in Table 1, the first client requests an operation once per time unit, whereas the second client requests an operation every three time units. As the result of the experiments, Tables 1 and 2 show the average number of messages per insert operation and the storage space utilization, for both LH* and the DRT method. In Table 1, we have drawn integer keys independently from a uniform distribution in the range from 0 to 10^6; we dropped a key that was to be inserted, whenever it was already in the file. To get a stable result, we repeated each experiment five times, with results that barely deviated from the mean value given in the table. We carried out further experiments, varying the number of keys to be inserted as well as the block capacity; it turned out that storage space utilization in LH* varies substantially, whereas it is always close to 70% in the DRT method. In Table 2 we choose a key distribution that implies that the number of keys belonging to one block and the corresponding overflow blocks in LH* follows a Gaussian distribution; here, ten repetitions of each experiment showed that the deviation from the mean is a little higher, but the mean is still meaningful. The evaluation of the performance of search operations is shown in Table 3. The parameters of the setup are as in Table 1. Before carrying out the sequence of 10,000 uniformly distributed search operations by clients as shown in the table, a different client inserted 10,000 uniformly distributed keys into the empty file.

The performance experiments show that storage space utilization is constantly good for the DRT method, and that it is significantly higher than that of LH* under a non-

uniform key distribution. Both structures do not differ by too much in the number of messages needed for insertions or search operations, with the odds being on one side or the other, depending on the setting.

6. Variants of the DRT method

The DRT structure, together with its operations, as described above, can be seen as a generic method that allows for a number of variations. For instance, we could vary the number of corrections, in order to exploit the tradeoff between the correction activity and the number of referencing errors: The more corrections we make, the fewer errors we can expect.

First, consider a correction for each operation request message, even without a referencing error. Recall that whenever a server s receives from client c a key that it owns, and the requested operation is a search, s sends $T(s)$ with its reply to c. In order to avoid subsequent referencing errors, it might be desirable that s informs c about $T(s)$ also in case of an insertion, even though this costs an extra message.

Then, let us see how we can make heavier use of correction information. If we correct trees as described above, a site may infer considerably less information than what would be possible. For an illustration, see Figure 4: If client c receives tree $T'(s')$ in a correction message, it should not only replace its leaf pointing to s' by the corresponding subtree of $T'(s')$. In order to make better use of $T'(s')$, it should instead merge the largest possible part of $T'(s')$ (including subtrees x, y, and z in Figure 4) into its own local tree.

Now, consider the possibility of uncoupling operation requests and corrections. The correctness of the DRT method does not depend on the existence or the timeliness of corrections, and it is unaffected if more correction messages than those suggested above are sent through the network. It is therefore possible to decouple operation requests and their corresponding correction messages. In times of heavy load of the system, one might, for instance, decide to defer some selected (or all) correction messages to more quiet times (a practitioner might say, corrections can be done at night).

Finally, note that block capacities need not be uniform. Correctness of the DRT method is unaffected by giving up the assumption of equal block capacity. In this case, we need not even choose a fixed capacity and allocate more than one block to sites with high capacity, but we can directly reference blocks of different capacities. The simplicity of this approach comes at the disadvantage of a potentially bad storage space utilization: Whenever a small block splits and involves a new, large one, utilization may become arbitrarily bad in the worst case. It may, nevertheless, be a useful concept whenever capacities do vary, but not to an extreme extent.

7. Conclusion and further work

In this paper, we have proposed an efficient, scalable, distributed, dynamic search tree structure, the distributed random tree structure DRT. There are two essential ingredients of this data structure: First, we distribute a global search tree over sites whose number varies with the amount of data to be stored, keeping the number of replicated nodes under control. Second, we lazily update the replicated parts of trees. The replication of parts of the global tree is not harmful: Since each leaf points to a block representing a large storage space, the number of tree nodes will be quite small in most applications. Therefore, replicating these nodes is not a major disadvantage against LH*, where each site keeps (a potentially obsolete version of) the complete directory information of constant size. The DRT method without deletions is currently the only structure with a guarantee on the storage space utilization, even for non-uniform key distributions, and offers efficient support not only for insertions and exact searches, but also for nearest neighbor queries and range queries.

Within the DRT method, several issues remain to be settled. Our next steps in making the DRT method more mature are the consideration of merge operations and more thoughts on the initialization of new servers. Note that the split coordinator in LH* solves a more difficult problem than our administrator, since it need not only identify a new server, but it must also find out which block is to be split. Hence, it has a stronger synchronizing function. We therefore hope that an efficient, distributed solution for initializing a new server can be found.

We feel that research that focusses on the efficiency of distributed data structures is only at its beginning. Since it is of increasing practical relevance, and it is interesting from a theoretical point of view, we expect that a certain body of knowledge, just as the one that has been assembled in the classical data structures field, should emerge over the next several years.

Acknowledgement

We want to thank Gerhard Weikum for inspiring discussions and pointers into the literature on distributed data structures. We gratefully acknowledge the support of this work by the Swiss National Science Foundation SNF.

References

F.B. Bastani, S.S. Iyengar, I-Ling Yen: Concurrent maintenance of data structures in a distributed environment. The Computer Journal, Vol. 21, No. 2, 1988, 165-174.

C.S. Ellis: Distributed data structures: A case study. IEEE Transactions on Computing, Vol. C-34, No. 12, 1985, 1178-1185.

E. Gudes, E. Shapiro: A parallel B-tree process structure, The Weizmann Institute, Rehovot, Israel, Technical Report CS 89-06, 1989.

M. Herlihy: A methodology for implementing highly concurrent data structures. Proc. ACM Symp. on Principles and Practice of Parallel Programming, 1989, 197-206.

T. Johnson, P. Krishna: Lazy updates for distributed search structure. Proc. ACM SIGMOD Conference on the Management of Data, 1993, 337-346.

D. Knuth: The art of computer programming, Vol. 3: Sorting and searching, Addison-Wesley, 1973.

R.Ladin, B. Liskov, L. Shira: Lazy replication: Exploiting the semantics of distributed services. ACM Principles of Distributed Computing, 1990, 43-57.

W. Litwin, M.-A. Neimat, D.A. Schneider: LH* - Linear hashing for distributed files. Proc. ACM SIGMOD Conference on the Management of Data, 1993, 327-336.

G. Matsliach, O. Shmueli: Maintaining bounded disorder files in multi-processor multi-disk environments, Proc. Int'l Conf. on Database Theory, Springer Lecture Notes in Computer Science, 1990.

G. Matsliach, O. Shmueli: An efficient method for distributing search structures, Proc. IEEE First Int'l Conf. on Parallel and Distributed Information Systems, 159-166, 1991.

S. Pramanik, M.H. Kim: Parallel processing of large node B-trees, IEEE Trans. Comp., Vol. 39, No. 11, 1208-1212, 1990.

H. Schwetman: CSIM Reference Manual, Revision 16, Technical Report, MCC, Austin, Texas, USA. 1992

W. E. Weihl, P. Wang: Multi-version memory: Software cache management for concurrent B-trees, Proc. 2nd IEEE Symp. Parallel and Distributed Processing, 650

Table 1

	block capa-city	no. of cl.	number of insert operations per client		time interval	average number of messages per insert		storage space utilization	
						LH*	DRT	LH*	DRT
1.	17	1	first client:	10,000	1	1.4265	1.2932	0.570	0.707
2.	33	1	first client:	10,000	1	1.2301	1.1578	0.566	0.701
3.	40	1	first client:	10,000	1	1.2121	1.1371	0.532	0.697
4.	40	2	first client: second client:	5,000 5,000	1 1	1.2262	1.2315	0.531	0.697
5.	40	3	first client: second client: third client:	3,333 3,333 3,333	1 1 1	1.2389	1.3183	0.532	0.697
6.	40	3	first client: second client: third client:	6,977 2,326 697	1 3 10	1.2329	1.2874	0.532	0.697

Table 2

	block capa-city	no. of cl.	number of insert operations per client		time interval	average number of messages per insert		storage space utilization	
						LH*	DRT	LH*	DRT
1.	40	1	first client:	10,000	1	1.4534	1.1394	0.264	0.697
2.	40	2	first client: second client:	5,000 5,000	1 1	1.5351	1.2401	0.260	0.694
3.	40	3	first client: second client: third client:	3,333 3,333 3,333	1 1 1	1.5999	1.3384	0.250	0.697
4.	40	3	first client: second client: third client:	6,977 2,326 697	1 3 10	1.5634	1.3188	0.257	0.695

Table 3

	block capa-city	no. of cl.	number of search operations per client		time interval	average number of messages per search	
						LH*	DRT
1.	17	1	first client:	10,000	1	2.0006	2.1127
2.	33	1	first client:	10,000	1	2.0002	2.0587
3.	40	1	first client:	10,000	1	2.0004	2.0500
4.	40	2	first client: second client:	5,000 5,000	1 1	2.0008	2.0873
5.	40	3	first client: second client: third client:	3,333 3,333 3,333	1 1 1	2.0012	2.1200
6.	40	3	first client: second client: third client:	6,977 2,326 697	1 3 10	2.0012	2.1160