Implementing Distributed Search Structures

Padmashree Krishna Theodore Johnson
pk@cis.ufl.edu ted@cis.ufl.edu
Computer and Information Sciences Department
University of Florida
Gainesville, FL 32611

March 26, 1994

Abstract

Distributed search structures are useful for parallel databases and in maintaining distributed storage systems. In this paper we discuss some issues in the design and implementation of distributed B-trees, such as methods for low-overhead synchronization of tree restructuring, node mobility and replication strategies. We have implemented a distributed B-tree that replicates the index and also performs data load balancing. Replication of the index provides for high throughput access, and data balancing allows for balanced processor and space utilization. For replication, we propose two different strategies, namely, *full replication* wherein the entire index is replicated at each processor and *path replication* where if a processor owns a leaf node, it also holds the index nodes on the path from the root to that leaf node. We present an algorithm for dynamic data-load balancing which uses node mobility mechanisms. We present some performance results of our algorithms. We find that path replication will create a scalable distributed B-tree.

Keywords: Data Structures, Distributed Databases, Load-balancing, Path Replication, Full Replication.

1 Introduction

Efficient search structures are needed for maintaining files and indices in conventional systems which have a small primary memory and a larger secondary memory. The normal operations carried out on an index are search, insert, and delete, range queries, and find-next member. Tree structures (in particular B-trees) are suitable for creating indices. The original B-tree algorithms were designed for sequential applications, where only one process accessed and manipulated the B-tree. The main concern of these algorithms was minimizing access latency. High performance systems need high throughput access, which requires parallelism. Distributing the B-tree can increase the efficiency and improve parallelism of the operations, thereby reducing transaction processing time.

In this paper, we look at the implementational issues and performance aspects of our approach of constructing distributed search structures. We have implemented the distributed B-tree on a network of SPARC stations (message-passing architecture). To study the portability and scalability of our system we ported it to the KSR, a shared memory multiprocessor system ([12]).

A common problem with distributed search structures is that they are often single-rooted. If the root node is not replicated, it becomes a bottleneck and overwhelms the node that stores it ([5]). In this paper we present two mechanisms, namely *full replication* and *path replication* for replicating the index of the B-tree. The theoretical framework for these algorithms has been developed in a previous paper [10]. Here, we discuss some issues in implementing the algorithms.

This paper also addresses the issue of data balancing, wherein every processor has a limited maximum load that it can hold. When the load at a processor increases beyond a threshold, some of the nodes are moved to another processor. Issues that arise from load balancing namely, mechanisms for node mobility and out-of-order information handling are also discussed in this paper. Finally, we present the performance results of our algorithms, where we find that the path-replication strategy for replicating performs better than full replication and is also scalable.

2 Previous Work

Several approaches to concurrent access of the B-tree have been proposed [2], [13], [15], [18]. Sagiv [17], and Lehman and Yao [13] use a link technique to reduce contention.

Parallel B-trees using multi-version memory have been proposed by Wang and Weihl [20]. The algorithm is designed for software cache management and is suitable for cache-coherent shared memory multiprocessors. Every processor has a copy of the leaf node and the updates to the copies are made in a 'lazy' manner. A multi-version memory allows a process to read an "old version" of data. Therefore, individual reads and writes appear no longer atomic.

Johnson and Colbrook [9] present a distributed B-tree suitable for message passing architectures. The interior nodes are replicated to improve parallelism and alleviate the bottleneck. Restructuring decisions are made locally thereby reducing the communication overhead and increasing parallelism. The paper also deals with the data balancing among processors. They suggest a way of reducing communication cost for data balancing by storing neighboring leaves on the same processor.

Colbrook, et al. [6] have proposed a pipelined distributed B-tree, where each level of the tree is maintained by a different processor. We used this approach as a preliminary design for our distributed B-tree. The parallelism achieved is limited by the height of the B-tree, and the processors are not data balanced since the processor holding the leaf nodes has more nodes than other processors.

In the context of node mobility, object mobility has been proposed in Emerald [11]. Objects keep forwarding information even after they have moved to another node and use a broadcast protocol if no forwarding information is available. Our algorithms require considerably less overhead, since we do not

broadcast, but look at relative nodes to find out where the node has moved.

A search structure node can be replicated using one of several well-known algorithms [4]. In what is called the TOTAL structure the entire data are replicated at each processor [16]. This increases the availability and fault tolerance but places a high demand on memory requirements. A compromise is to set up a balance between memory usage and cost considerations [1].

The multi-version memory algorithm proposed by Wang and Weihl [20] reduces the amount of synchronization and communication needed to maintain replicated copies. Several algorithms have been proposed for replicating a node [3]. *Lazy replication* has been proposed by Ladin et al., for replicating servers [14]. Replicas communicate information among themselves by lazily exchanging gossip messages. Several authors have explored the construction of non-blocking and wait-free concurrent data structures in a shared-memory environment [7, 19]. These algorithms enhance concurrency because a slow operation never blocks a fast operation.

In a previous paper, we have proposed fixed copy and variable copy algorithms for lazy updates on a distributed B-tree and provided a theoretical framework for replication [10]. Lazy update algorithms are similar to lazy replication algorithms because both use the semantics of an operation to reduce the cost of maintaining replicated copies. The effects of an operation can be lazily sent to the other servers, perhaps on piggybacked messages. The lazy replication algorithm, if necessary, blocks an operation until the local data is sufficiently up-to-date. In contrast, a non-blocking wait-free concurrent data structure never blocks an operation. The lazy update algorithms are similar in that the execution of a remote operation never blocks a local operation; hence they are a distributed analogue of non-blocking algorithms. 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.

Our contribution is to discuss implementational issues of distributed search structure algorithms. Two strategies for replication are proposed here, namely path replication and full replication. We find that path replication is better, with far lower message and space overhead. Path replication imposes only a small overhead on the number of messages required to search for a key. As a result, path replication permits highly scalable distributed B-trees. We also perform data balancing on a single copy B-tree, with the leaf nodes being allowed to migrate between processors. The performance results of our algorithm show that we can achieve a good data balance among processors with little node movement overhead.

3 Concurrent B-link tree

A B-tree is a multi-ary tree in which every path from the root to the leaf is the same length. The tree is kept in balance by adjusting the number of children in each leaf. In a B^+-tree, the keys are stored in the leaves and the non-leaf nodes serve as the index. A *B-link* tree is a B^+-tree in which every node contains a pointer to its right sibling. Concurrent B-link tree algorithms [13, 17] have been found to provide the highest concurrency of all concurrent B-tree algorithms [8]. In addition, operations on a B-link tree access one node at a time. A B-link tree's high performance and node independence makes it the most attractive starting point for constructing a distributed search structure.

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

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

4 Distributed B-tree

To distribute the B-tree over several processors, we migrate only the leaf level nodes, with the index level nodes being replicated. Currently we are performing only inserts and searches. Delete operations involve additional complications and the algorithms are still in the development stage.

The concurrent B-tree algorithms translate easily to our distributed one. A search operation can begin on any processor. During the search phase the B-tree is traversed downward. When a required node is not found on the processor (because it may have migrated), a request is sent to a remote processor based on the local information. Thus, a search operation may span several processors.

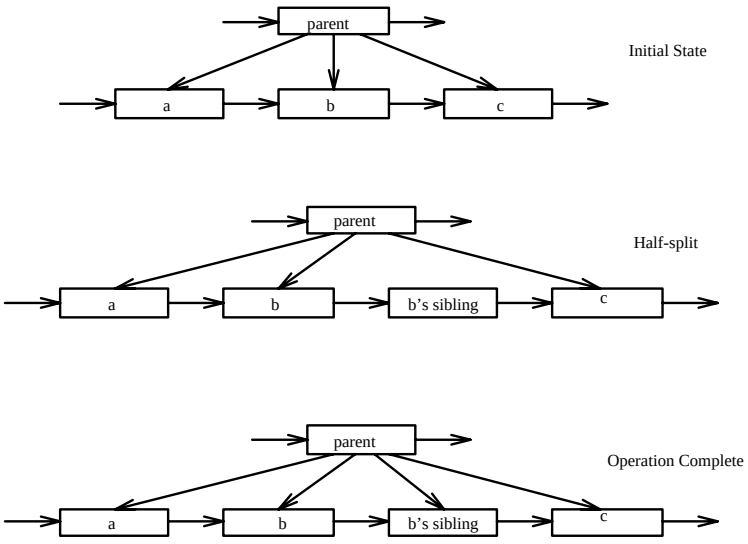

Figure 1: Half-split Operation

The insert operation works in two phases: the search phase and the restructuring phase. The search phase is the same as explained in Section 3. The restructuring phase is slightly more complicated. The half-split algorithm explained above translates to a distributed one easily. A new sibling is created on the same processor and half the keys are transferred to it. The siblings of the split node might not reside on the same processor, hence messages are sent to inform them of the split. The split node must now be inserted into the parent. In our replicated B-tree, since a copy of the parent is resident on the same processor, the node is inserted into the parent and messages are sent across to all other parent node copies informing them of the new insertion. If the insertion causes the parent to split, the parent's siblings are informed.

The following subsections give the details of the underlying architecture of our implementation, the typical node structure, the implemented algorithms for distribution and issues encountered.

4.1 Architecture

We implemented our algorithms on a LAN comprised of SPARC workstations. We have also ported our implementation to the KSR, a shared memory multiprocessor system. The network of processors communicate by BSD sockets, which provide a reliable link. An overall B-tree manager called the *anchor* overlooks the entire B-tree operations. On each processor we have a *queue manager* and a *node manager*. The queue manager enqueues all the messages from remote processors in a queue. The node manager actually handles the operations on the various nodes at that processor. This functional distinction of the process at a processor into the queue manager and the node manager enables the node manager to

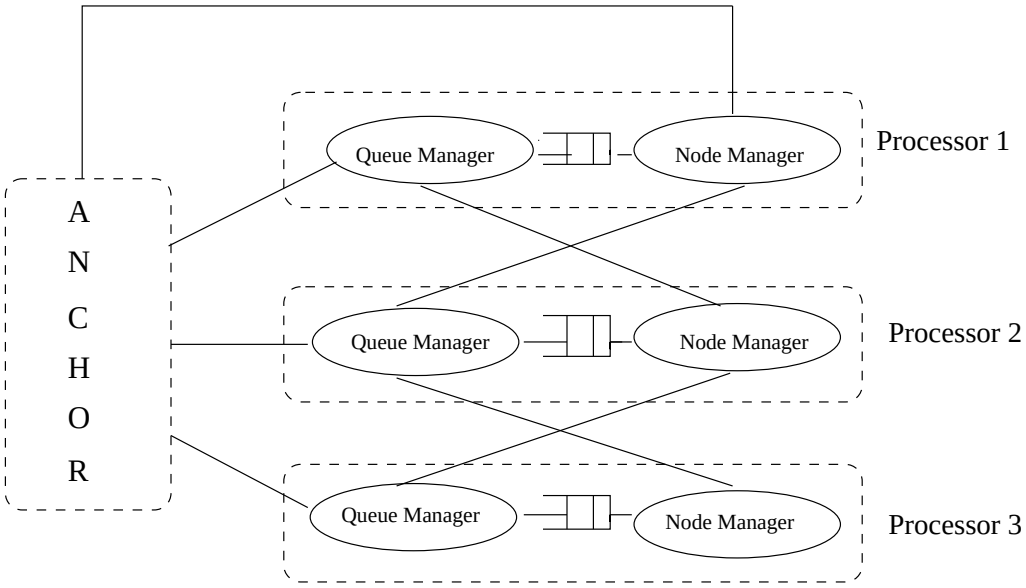

Figure 2: The Communication Channels

be independent of the message processing. The queue manager and the node manager at a processor communicate via the inter-process communication schemes supported by UNIX, namely message queues (Figure 2).

The number of processors over which our B-tree is distributed is a parameter and can be varied. Also, the number of keys inserted in the B-tree is a run-time parameter.

4.2 Node Structure

When distributing a B-tree, each node of the B-tree in addition to maintaining the indices, must also contain information that will help in the maintenance of the B-tree. The necessary information in a typical B-tree node is the following:

- Node Name: Every node in the B-tree has a name associated with it that is independent of the location of the node. This mechanism for naming nodes is known as *location independent naming of nodes*.

  Whenever a node is created, it is given a name that is unique among all processors. For instance, the node name may be a combination of the processor number that creates it and the node identifier within the processor. A hashing mechanism is used to translate between node names and physical node addresses. When a node **bob** moves from processor A to processor B, it retains its name. The advantage of this mode of naming nodes is that a parent, child or sibling node that references the node **bob** need not know the exact address of **bob** in processor B.

  A further advantage of location independent naming is when the nodes are replicated. All copies of a node on different processors have the same name. So, the primary and secondary copies of a

node can keep track of each other easily.

- Version Number: A node has a *version number* attached to it. The version number is used to produce ordered histories ([10]). A node gets the version number 1 when it is created for the first time. When a node splits the sibling gets the same version number as the node. When a node moves from one processor to another, the version number of the node is incremented by 1. This helps in the detection of obsolete messages in the system as explained in subsection 4.3.3. When a processor joins a set of node copies, the version number of the node copies in incremented. The purpose of this will be explained shortly in section 4.4.3.

- Level Number: Every node is associated with a *level number*. The level number, which indicates the distance to a leaf, is useful in recovery from misnavigation [10].

- Links to *Relatives*: The parent, siblings and children are the relatives of a node. A typical node would have a pointer to the primary parent, local parent, children, and siblings. A pointer to a node contains the node's name and version number. In addition to having the highest value in itself, the node must also keep the high value of it logical neighbors. This will enable a node to determine if the operation is meant for itself or destined for either of its siblings.

- Copies: Every node has information as to where (on which processors) the copies reside. This helps in maintaining replica coherency, when a node has to send messages to all its copies to keep node copies consistent.

4.3 Implementation Issues

Because of the dynamic movement of a node among processors, several issues were encountered. *Link Updates* are necessary to maintain the logical tree structure spanning over multiple processors. *Message Forwarding* is used to track a migrated node. Messages in the distributed environment may arrive out-of-order and we address this issue of *Obsolete Messages* by using version numbers in the nodes and messages.

4.3.1 Link Updates

An operation can be initiated at any processor. Consider an *insert* operation initiated at processor **P**. Several cases may arise concerning the node **n** for which the operation is intended:

- The node **n** is on the same processor **P**:

- **n** may have some space to insert the new key: the insert operation is performed.
- **n** may be full: In this case the insert operation will result in the node **n** being split into **n** and and a new node **s**. The node **s** gets the same version number as **n**. Half the number of keys are transferred to this new node **s**. After this transfer the parent, the sibling nodes, and the children of **n** must be informed of the split. These relative nodes might or might not reside on the same processor, **P**. If they do, then the appropriate link changes have to be performed on the relative nodes. If the relative nodes reside on some other processor **Q**, *link change* messages have to be sent to those processors, informing them of the changes. A link change message contains the node **n**'s version number. A node **r** at processor **Q** on receiving a message will act upon that message only if the link (to **n**) version number, is greater than or equal to the version number of the message.

- The node **n** is not the processor **P**. When does this happen and what we should do is explained below.

4.3.2 Forwarding Messages

A node expected to be on a processor may no longer be there when nodes are moved between processors for load balancing, that is explained in a later section. We have taken the approach of the *single-copy mobile nodes* algorithm proposed in a previous paper [10]. We will briefly discuss it here.

Every node has only a single-copy, but the nodes can migrate from processor to processor. When a node migrates, it informs its neighbors of the new address. It also leaves behind a *forwarding address*, by which a message that arrives for a migrated node can be rerouted. Two algorithms *eager* and *lazy* have been proposed.

The eager one ensures that a forwarding address exists until the processor is guaranteed that no other message will arrive for the migrated node. This approach is complex and requires too much message passing and synchronization.

In the lazy approach, if a message arrives for a migrated node, then obviously the message has misnavigated. Similar to the misnavigation in a concurrent B-tree protocol, we use pointers to follow where the node has migrated. We look for a node that is 'close' to the migrated node and take its pointer to determine where the migrated node currently resides.

In our implementation, only the leaves have a single copy, while the index level nodes may have several copies. Only the leaf level nodes are allowed to migrate. We have used a combination of both

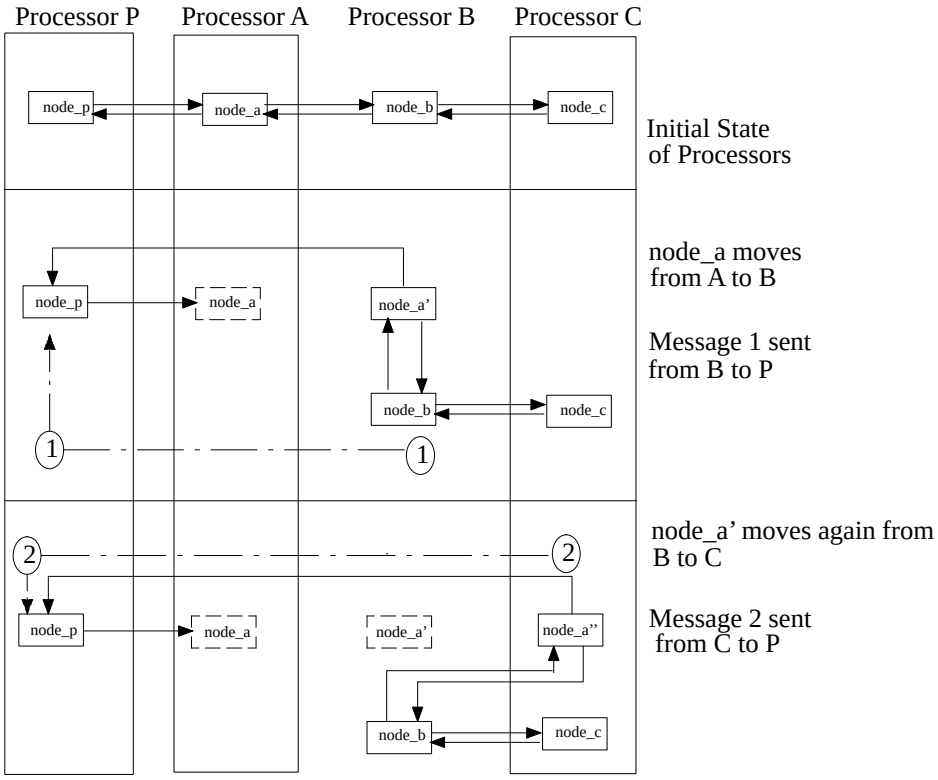

Figure 3: Node Migration

the eager and lazy approach. When a node **n** is moved, a forwarding address stub is left in the node **n** at processor **P**, that indicates its new location, say processor **R**. When a link change message arrives for the migrated node **n** at processor **P** , then that message is forwarded to the processor **R**.

Periodically, the migrated nodes are deleted and their storage reclaimed at processor **P**. Since we do not require acknowledgments for the link changes, it is possible that a message will arrive for the deleted node. What happens to this message? In this case the new location **R** is determined by the information in a relative node. The message then follows the B-link-tree search protocol to reach its destination. In our current implementation, we are guaranteed that the processor stores either a parent of the deleted node, or another node on the same level as the deleted node. The significance of this deleted node recovery protocol is that we can lazily inform neighboring nodes of a moved node's new address. The protocol is invoked rarely, since most messages for the transferred node are handled by the forwarding address.

4.3.3 Obsolete Messages

A message can get delayed in the system and may arrive eventually. This is know as out-of-order arrival of messages. How are these messages handled?

This scenario can be explained with an example (Figure 3). Suppose a node a moves from processor A to Processor B. Consider node p, which resides on processor P and contains a link to node a. When node a moves to processor B, a link update message is sent to node p at processor P. Before this message reaches P, processor B decides to move node a' to processor C and C sends a link update message to P. Suppose that the message from C reaches P before the message from B. If node p at P updates the node address to that of C and then to B, then node p at P has the wrong address for node a. How do we resolve this problem?

In our design we have a *version number* for every node of the tree. A node has a version number 1 when it gets created for the first time (unless it is the result of a split). Every time a node moves its version number is incremented, and when a node splits the sibling gets the same version number as the original node. Every pointer has a version number attached and each link update message contains the version of the sending node. When node r receives a link update message from s, r will update the link only if s's version number is equal to or greater than the link version number. In the above example, the version number of node a on processor A is initially 1. On moving to processor B the version number changes to 2. The update message to P from B contains the version number 2. The next update message sent to P from C has the version number 3. Now since this last message reaches P first, node p at processor P notes that its version number for node a is 1. Since $3 >= 1$, node p updates node a's address, version number and processor number. Now, the message from B arrives that contains version number 2. But now node p has version number 3 for node a, hence the version numbers do not match and the message is ignored. So delayed messages that arrive out of order at a processor are ignored.

Our numbering handles out-of-order link changes due to split actions also. Reliable communications guarantee that messages generated at a processor for the same destination arrive in the order generated, and when nodes move to different processors the version number of the nodes is incremented, so messages regarding link changes are processed in the order generated.

4.4 Distribution

The B-tree is distributed by having the leaf level nodes at different processors. Leaf level nodes are not replicated and only these nodes are allowed to migrate between processors. In a previous paper we ([10]) presented some algorithms for replication and provided a theoretical framework for them.

Two approaches for replication, i.e., **Fixed-Position copies**, and **Variable copies** are presented. Both these algorithms use one copy of a node to be a *primary copy* (PC). Our implementation of

the Fixed-Position copies algorithm is termed **Full Replication** and that of Variable copies is **Path Replication**. We will briefly discuss the algorithms and the implementational issues in sections 4.4.1 and 4.4.2.

When the nodes of the B-tree are replicated, an obvious concern is the consistency and coherency of the various replicated copies of a node. Section 4.4.3 will present the mechanism by which our implementation maintains coherent replicas.

4.4.1 Full Replication

The Fixed-position copies algorithm ([10]) assumes that every node has a fixed set of copies. An insert operation searches for a leaf node and performs the insert action. If the leaf becomes full, a half-split takes place. In this algorithm, the PC performs all initial half-splits. Two variations of this algorithm are the *synchronous splits* and *semi-synchronous splits*.

The semi-synchronous requires that the PC be consistent with other node copies. In this method, the PC performs the split and sends a *relayed split* to the other copies. Any *initial inserts* at a non-PC copy are kept in overflow buckets and adjusted after the *relayed split*.

In our implementation, the entire index levels of the tree are replicated at each processor. Whenever a leaf node migrates to a new processor, (one that currently stores no leaves) the index levels of the tree are replicated at that processor. Consistency among the replicated nodes is maintained by the primary copy of a node sending changes to all its copies.

Once the entire tree has been replicated, only consistency changes need to be propagated to this new processor.

- **Algorithm:** The decision to replicate the tree is made after a processor (sender) downloads some of its leaf level nodes to another processor (receiver). After the leaves are transferred, the sender checks to see if the receiver has received leaf nodes for the first time. If so, the receiver obviously does not have the index levels, so the tree has to be replicated at the receiver. The sender then transfers the tree (index levels) that it currently holds. Henceforth, only consistency maintenance messages are necessary to maintain the tree at this processor.

```
sender()
{
        transfer_leaf_nodes();
        if ( hastree(receiver) == false) {
                send await_tree_message();
                wait_for_ack();
                make_connection();
                transfer_tree_copy();
                close_connection();
        }
}
```

```
transfer_tree_copy(receiver)
{
        for (level = 2; level <= anchor.height; level++) {
                n = getlastnode(level);
                sendnode(n, receiver);
                n->copy[receiver] = PRESENT;
                proc = 1;
                while (proc <= numprocs && n->copy[proc] == PRESENT) {
                      send_copyjoin (proc, receiver, n);
                      proc++;
                }
        }
}

receiver()
{
        get leaf_nodes();
        if (next_message == await_tree_message) {
                send_acknowledgement();
                wait_for_connection();
                receive_tree_copy()();
        }
        else
                process next_message();
}
```

Alg 1. Algorithm for Full Replication.

4.4.2 Path Replication

In the Variable-copies algorithm ([10]), different nodes have different number of copies. A processor that holds a leaf node also holds a path from the root to that leaf node. Hence, index level nodes are replicated to different extents. A processor that acquires a new leaf node may also get new copies of index level nodes and such a processor then *joins* the set of node copies for the index level nodes. Similarly when a processor sends away some of its leaf level nodes, if some of the index nodes are left with no leaf nodes as children, then the processor has to *unjoin* the node copies.

In our path replication algorithm whenever a leaf node migrates to a different processor, entire path from the root to that leaf is replicated at this processor. However, if the processor holds a leaf and a new sibling migrates to that processor, only the parent nodes not already resident at this processor are replicated. All link changes are again handled by the primary copy of a node. When a new copy of a node is created the processor sends a 'join' message to all the copies of the node. In the interim of the node copy being created at the processor and the 'join' message reaching a processor, any messages about this node copy are forwarded by the primary copy of the node to this new copy. A processor that sends away all the leaf nodes of a parent will no longer be eligible to hold the path from the root to that leaf node. In this case, the processor has to do an 'unjoin' for all its nodes on the path from the root to the leaf.

Here, other than consistency messages, every time a leaf node migrates, 'join' and 'unjoin' messages have to be propagated. Further, broadcasting the consistency messages to all processors would be a

waste, rather the primary copy of a node has to check which processors hold the copies of a node and send the consistency messages to only those copies. Copies of a node at processors that have 'unjoined' (but not been deleted) do not receive any such messages.

- **Algorithm:** Our algorithm for path replication is asynchronous, based on a handshaking protocol. When two processors have interacted in the load balancing protocol, a decision has to be made concerning the path from the root to the migrated leaves. Either the sending or receiving processor can request that the path be sent to the receiver. In our algorithm, the receiver determines what ancestor nodes are needed after receiving new leaves. It then sends requests to the processors holding the primary copies of the ancestor to get the paths. As the receiving processor takes the responsibility of obtaining the path, the sending processor is free to continue. The receiving processor cannot do much anyway until it receives the path, so no time is wasted. Once the path is obtained, the receiving processor can handle operations (inserts and searches) on its own.

```
sender:
{
        distribute some of the leaves to the receiver processor;
}
any processor:
{
        if (next message == path request) {
                send_acknowledgement with port number where listening;
                wait for connection;
                send parent (p) node copy;
                send join_message to all processors that hold a copy of the parent;
        }
}
receiver:
{
        get leaf nodes from sender;
        for each leaf do
                if (parent not present) {
                        send message to parent's primary copy requesting path;
                        path_request_pending = TRUE;
                }
        if (next message == path acknowledgement) {
                make connection to primaryproc;
                get parent node copy;
        }
}
```

Alg 2. Algorithm for Path Replication.

4.4.3 Replica Coherency

The operations that the current implementation handles are searches and inserts. A **search** operation is the same as an **insert** operation, except for the key not being inserted. A search returns a true or false and does not cause any further relayed messages to be issued. An operation on the distributed B-tree can be initiated on any processor. Since the index levels are fully or partially replicated at all processors, a change in a node copy at any processor must be informed to all processors that hold a copy

of that node. Every processor need not be aware of a search operation performed at a processor, but must know of any insert operation that occurs. An insert operation in a node could result in a split, so all processors must be informed about the split. This is done in the following way:

- Insert An **insert** operation can be performed on any copy of a node. After performing the insert, the processor sends a **relayed insert** to all other processors that hold a copy of the node. When a processor receives a relayed insert, it performs the insert operation locally.
- Split A **split** operation is first performed at a leaf. If the local parent exists on the same processor the split is informed at the local parent. If the split at the any level results in a split at the parent level, then a **relayed split** is sent to all processors that hold a copy of the parent node. Otherwise, a **relayed insert** is sent.

The next concern is how does a node know of all its existing copies and when will the set of copies be informed if a new copy is made at a processor. The version number in every node is the clue to the solution here. Let's consider an example here.

- Example:

  Let node **n** with version number 5 (say), have copies **n1** at processor 1, **n3** at processor 3 and **n5** at processor 5, the **n5** being the primary copy. Suppose processor 4 acquires some leaf nodes from processor 1 and also some index nodes (that include **n**). Now processor 4 sends in a 'join' message to copies **n1**, **n3** and **n5**. Suppose that before the 'join' message arrives at processor 3, **n3** sends a 'relayed insert' (could be any other message) to copies **n1** and **n5** (that it already knows about). Node copy **n4** will not receive the 'relayed insert' from 3, hence **n4** will not be consistent with the other copies.

- Solution: When a new copy of a node is made the version number of that copy is incremented by 1. In the section on 4.4.2 we note that a processor that gets a new copy of a node, sends a 'join' message to all the existing node copies. When the node copies receive this 'join' message they increment their version number.

  In the example above, when processor 4 receives **n4** it increments the version number of **n4** to 6. When the primary copy, i.e., **n5** receives the relayed insert from **n3** with version number 5, it learns that processor 3 is yet unaware of processor 4, and hence forwards the 'relayed insert' to processor 4. Thus, the primary copy of the node is responsible for handling this consistency.

5 Data Balancing

Distributing the tree arbitrarily implies that some processors may have many nodes (due to splits) and hence run out of storage when there is plenty in the system. It is therefore necessary to balance the load among processors. One may select the nodes to be transferred to be the processors equally balanced in terms of capacity or work. Here, we choose to keep the processors capacity balanced.

The most fundamental issue in load balancing is the actual process of moving a node between processors. This is termed the *node migration* mechanism and is common to all of our algorithms for load balancing. For the following discussion on the node migration mechanism, let us assume that the node manager at a processor wishing to download its nodes has been notified of a recipient processor that is willing to accept nodes. The actual method by which this is done will be explained in section 5.1.

5.1 Node Migration

After the node manager is informed of a recipient for its excess nodes, it must decide which nodes to send. This may be based on various criteria of distribution. After selecting a node, the node manager begins the transfer.

The question we must consider here is: Should the sender and all other processors be locked up until all pointers to the node in transit get updated? If this is so, parallelism would be lost. How should we achieve maximum parallelism?

The solution to this problem is aimed at maintaining parallelism to the maximum extent possible by involving only the sender and the receiver during node movement. We have designed an *atomic handshake and negotiation protocol* for the node migration, so that the sender is aware of the forwarding address of a node before sending another node. After the node selection is done, the sending processor (henceforth called the *sender*) establishes a communication channel with the *receiver* and a negotiation protocol follows. In the negotiation protocol, the sender and the receiver come to an agreement as to how many nodes are to be transferred. After a decision has been reached the sender sends a node, awaits its acknowledgement and transfers the next node. The acknowledgement contains the new address of the node at the receiver processor. This forwarding address stub is left in the sender. A node that has been sent is tagged as in transit and no operations are performed on that node at the sender (5.1).

Once a node has migrated, all processors are informed of the new address. In the interim that node movement takes place and the other related processors are informed, any messages that arrive for this node are handled as explained in section 4.3.2.

When a node moves it sends a link update message to related processors. Suppose that the update

message for link change gets delayed and the node moves for a second time. The second link update message may arrive before the first one at a processor and will be handled as explained in section 4.3.3.

```
proc_alloc()
{
        get receiving processor id;
        establish communication channel with receiver;
        compute_nodes_to_send;
        while (nodes_to_send > 0) {
                n = select_a_node();
                current_capacity--;
                send_node(n);
                wait for acknowledgement;
                update pointers locally;
        }
        send close_connection to  receiver;
}

accept_nodes() {
        wait_for_connection;
        send_acceptable capacity;
        while (msg != close_connection) {
                accept a node;
                send_acknowledgement;
        }
        close_connection;
        update pointers related to new nodes locally;
        send messages to related nodes if not on sending or receiving processor;
}
```

Alg 3. Node Migration Algorithm.

5.1.1 Protocol for Node Migration

How one selects the nodes that have to be moved depends on factors that affect locality, availability, and fault tolerance. In our initial experiment we move nodes only at the leaf level, since all operations are eventually carried out at the leaf nodes. After choosing a criterion for load balancing, further steps are to decide where to download the excess nodes, and to determine who is responsible for decision making, node movement, and the address updates.

5.2 Decision Making

How does a processor choose which other processor to send its nodes to? Our current approach is a *semi-centralized* one, where, the anchor has been responsible for choosing the receiver. Another approach would be to leave the decision making to the individual processors. This is a *distributed data balancer* and we present some ideas on this in the section on future work.

In the current design, a limit is placed on the maximum number of nodes of the tree **threshold**, that a processor can hold. In addition each node has a **soft limit** (.75 * threshold) on the number of nodes. This represents a danger level indicating a need for distribution of the nodes. Whenever a node splits, the current number of nodes is checked against the soft limit. If the current number of nodes exceeds the soft limit, the processor must distribute some of the nodes it has to some other processor.

The processor sends a message to the anchor requesting a receiver. This message indicates how many nodes the processor wants to download.

When the anchor receives a node transfer request from processors, it goes through the following steps:

- Picks a receiver processor from the currently active ones with available capacity. A processor will have available capacity only if its current capacity plus the number of nodes that it may receive (if selected) does not exceed the **soft limit**.
- Selects a new processor, when none of the processors have sufficient capacity.
- Doubles the capacity of each processor. As explained earlier, the B-tree distribution is limited to some number of processors (PROC_LIMIT), by a run-time parameter. When this processor limit is reached, then a new processor cannot be added to the currently active ones. When this occurs, the threshold at each processor is doubled by the anchor. The processor that issued the node transfer request cancels its request as it now has capacity to store its nodes.

In order to make a decision the anchor must be aware of the current node capacity at each active processor. As no special messages are sent to the anchor when a node splits, the anchor has out-of-date information about the processor's capacity. The only time the anchor receives the node capacity is when the height increases or a node transfer request arrives from a processor. The anchor uses this inaccurate information to select a receiver, r, and sends a message to the selected processor. After the selected processor is ready to accept nodes from the sender, the anchor informs the sender of the processor r. The node manager at the processor s follows the node negotiation and migration technique already discussed. We see that the algorithm works even though the anchor has so little information because of the negotiation protocol.

The anchor's responsibility is over and it is free to continue with any other task.

5.2.1 Algorithm for Decision Making

```
Anchor()
{
        get next_message;
        if (next_message == node_transfer) {
                receiver =      NONE;
                n = number of nodes to be transferred; /* the message contains the # of nodes to be transferred */
                for (proc = 1; proc <= num_active_processors; proc++) {
                        /* Check the capacity of a processor if it takes the nodes, against the SOFT_LIMIT */
                        if (capacity[proc] + n < SOFT_LIMIT)
                                receiver = proc;
                }
                if (receiver == NONE) {
                        if (num_active_processors < PROC_LIMIT) {
                                start a new_processor;
                                receiver = new_processor;
                                num_active_processors++;
                        }
```

```
                        else {
                                THRESHOLD = 2 * THRESHOLD;
                                send new THRESHOLD to all processors;
                        }
                }
                if (receiver != NONE) {   /* a processor has been selected */
                        send are_you_willing message to receiver;
                        if (next_message ==     willing_acknowledgement)
                                send receiver_ready message to sender;
                        else
                                if (next_message == willing_nack)
                                        send receiver_nack to sender;
                }
        }
}

sender()
{
        if (capacity(self) >= SOFT_LIMIT) {
                n =     capacity(self) - SOFT_LIMIT + BUFFER;
                /* BUFFER is some fixed number of nodes, so that at least BUFFER+1 nodes are transferred at any time. */
                send node_transfer(n) to anchor;
                processor_request_pending = TRUE;
        }
        resume processing;
        if (next_message ==     receiver_ready) {
                get receiving processor id;
                perform node transfer;
                processor_request_pending = FALSE;
        }
        else
                if (next_message == receiver_nack)
                        processor_request_pending = FALSE;
}

receiver()
{
        if (next_message == are_you_willing)
                if (!path_request_pending) {
                        send willing_acknowledgement to anchor;
                        wait for sender to transfer nodes;
                }
                else
                        send willing_nack to anchor;
}
```

Alg 4. Algorithm for Decision Making.

5.2.2 Deadlock

The handshake protocol may in some cases lead to a deadlock. Deadlock may arise because both the load balancing algorithms and replication algorithms were asynchronous, requiring a handshake protocol. It may happen that two processors may wait for each other, one waiting for load distribution and the other for a path.

The deadlock may be resolved by having the processors not wait indefinitely but wake up and check for some important activity or by giving a priority to one of the processors. The first one would require a timer to wake up at regular intervals. We chose to resolve the deadlock by giving priority to the 'path request'.

The *path request* is given priority over the *load balancer* since, unless the processor has a path to the leaves no useful work can be done on those leaves. The load balancing request will get aborted and will be reissued again, so no harm is done, except for a few (4) wasted messages. When a processor is waiting

for a 'path', and if a load balancing request is received, it sends a **nack** to the processor. The processor then cancels its request and reissues it at a later time.

5.3 Experiments, Results and Discussion

In this section, we study the performance of the algorithms we have developed by performing some experiments.

5.3.1 Load Balancing

We have performed some experiments to study the performance of our load balancing algorithm for a non-replicated B-tree with a maximum node size of 16. We expect similar results for a replicated B-tree.

Experiment Description: To study the performance of our load-balancing algorithm, we compared the performance of a load-balancing system to one without load balancing. We chose to distribute the leaf nodes only, placing no restriction on the number of times that a node moves.

Initially the system uses load balancing to reach a stable state with 8 processors. A random distribution of keys is chosen to create the initial B-tree.

- *Experiment A:* In our first experiment we maintain the same random distribution of the keys as in the beginning and then either leave ON or turn OFF the load balancing, and observe the distribution of the leaf nodes. To study the load variation behavior under execution, we collect distributed snapshots of the processors at intervals, determined by the number of keys inserted in the B-tree.

- *Experiment B:* In the next experiment, we vary experiment A by changing the key distribution pattern dynamically. Since the B-tree property guarantees that the keys are nicely distributed among the nodes [8], a noticeable performance enhancement may not be observed in a uniformly distributed data pattern. So, to study the effect of our load balancing algorithm when the distribution changes, we have introduced *hot spots* in our key generation pattern. After the system reaches a stable state, the distribution is changed so that keys are generated with a probability of .6 in the range 0 to $2^{31} - 1$ and with a probability of .4 in the range 70000 to 80000. This concentrates the keys in a narrow range, thereby forcing about 40% of the messages to be processed at one or two 'hot' processors.

In each experiment we insert a total of 10,500 keys. The system is presumed to have reached a stable state after 500 keys have been inserted. From then on, we take a distributed snapshot of the system

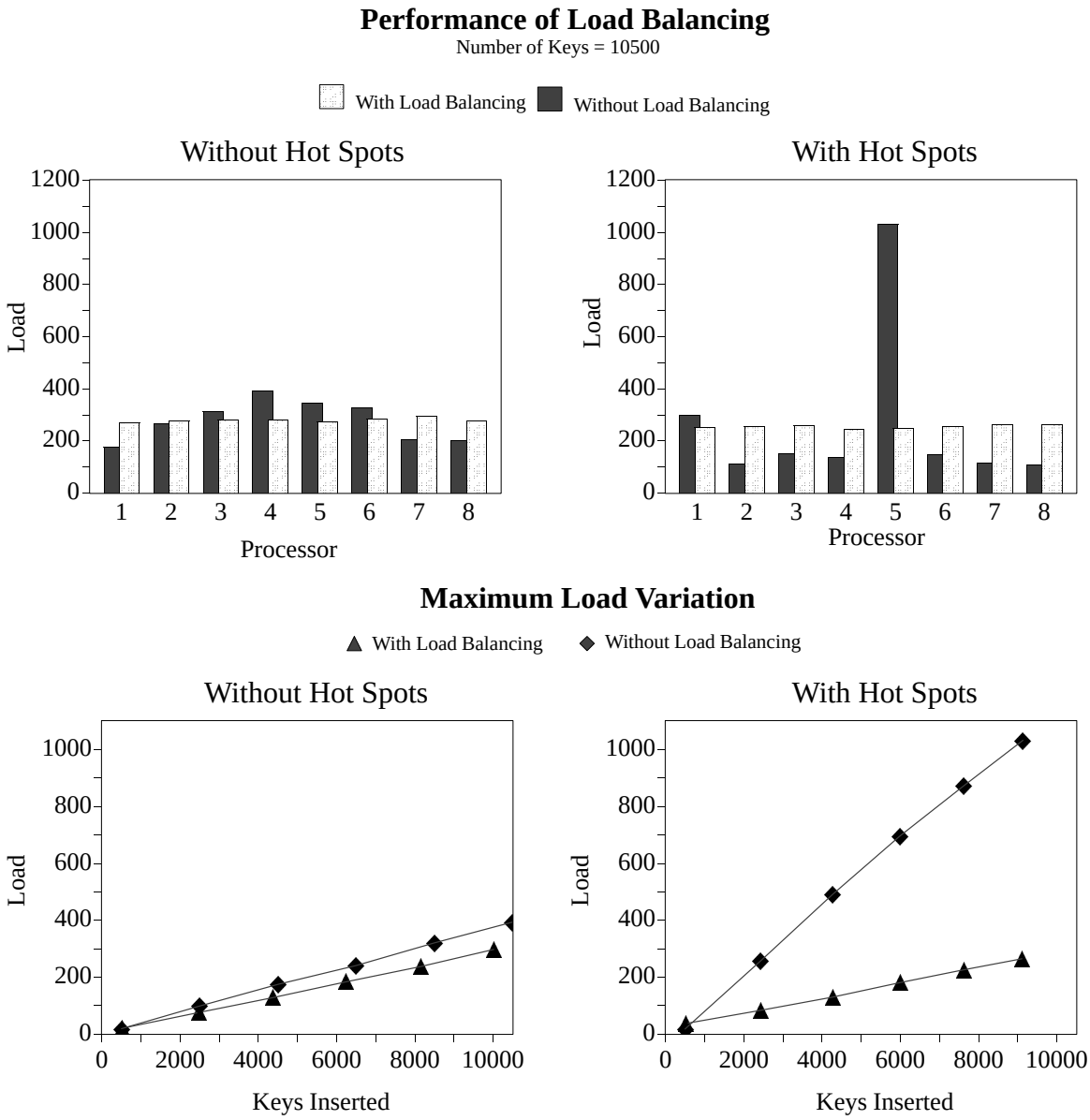

Table 1. Load Balancing Statistics

Without Hot Spots					With Hot Spots				
Keys	No Load Balancing		Load Balancing		Keys	No Load Balancing		Load Balancing	
	Avg. Moves	Coeff. of variation	Avg. Moves	Coeff. of Variation		Avg. Moves	Coeff. of variation	Avg. Moves	Coeff. of Variation
500	0.79	0.14	0.81	0.13	500	0.77	0.14	0.67	0.65
2500	0.17	0.25	0.20	0.04	2435	0.16	1.03	0.30	0.08
4500	0.10	0.25	0.13	0.02	4275	0.09	1.15	0.18	0.03
6500	0.07	0.24	0.09	0.03	6000	0.06	1.15	0.13	0.03
8500	0.05	0.25	0.08	0.02	7622	0.05	1.13	0.11	0.02
10500	0.04	0.26	0.06	0.03	9150	0.04	1.13	0.09	0.02

Figure 4:

every 2000 inserts. At each snapshot, we note the processors' capacity, the number of times a processor invokes the load balancing algorithm and the number of nodes that it transfers. We also calculate the average number of times a leaf node moves between processors (taken with respect to the nodes in the entire B-tree).

The *Performance* bar charts show the processors' capacity after the insertion of 10,500 keys, using both distributions. When the "hot-spots" distribution is used, processor 5 is the *hot processor*, and it receives a disproportionate number of inserts. Without load balancing, the processors vary greatly in load, with processor 5 having more than a 1000 nodes and processor 8 having only around 100 nodes. Our load balancing algorithm distributes the excess load at processor 5 among other processors, so that all processors contain about 250 nodes when all keys have been inserted. In the "no hot-spots" distribution, there is still a large imbalance in the data load on the processor, which load balancing corrects (Figure 4).

The *Maximum Load Variation* graphs show how the maximum load varies between snapshots. For this graph we pick up the maximum load value at each snapshot (perhaps at different processors at different snapshots). The plot without the hot spots indicates that the maximum load variation increases significantly (slope of the line is greater) indicating the need for load balancing even with uniform distribution. With hot spots the variation is much greater, indicating the nice effect load balancing has for smoothing the variation and reducing the gradient. (Figure 4).

Finally Table 1 shows the calculated average number of moves made by a node in the entire system, with and without hot spots and with and without load balancing, and the normalized variation of the capacity at each processor from the mean. The table shows that the load balancing reduces the coefficient of variation at the cost of a very small increase in the average moves in the system, indicating that load balancing is effective with low overhead (Figure 4).

5.3.2 Replication

Here, we compare the performance of full replication and path replication strategies for replicating the index nodes of a B-tree.

Experiment Description: In the experiment 15,000 keys were inserted, statistics being gathered at 5000 key intervals. The B-tree is distributed over 4 to 12 processors. We observed the number of times a path request has been made by a processor, the number of times that a load balancing request had to be reissued (to avoid deadlock) with priority being given to the path request. We collected statistics as to how many consistency messages are needed to maintain the distributed, replicated B-tree, how widely

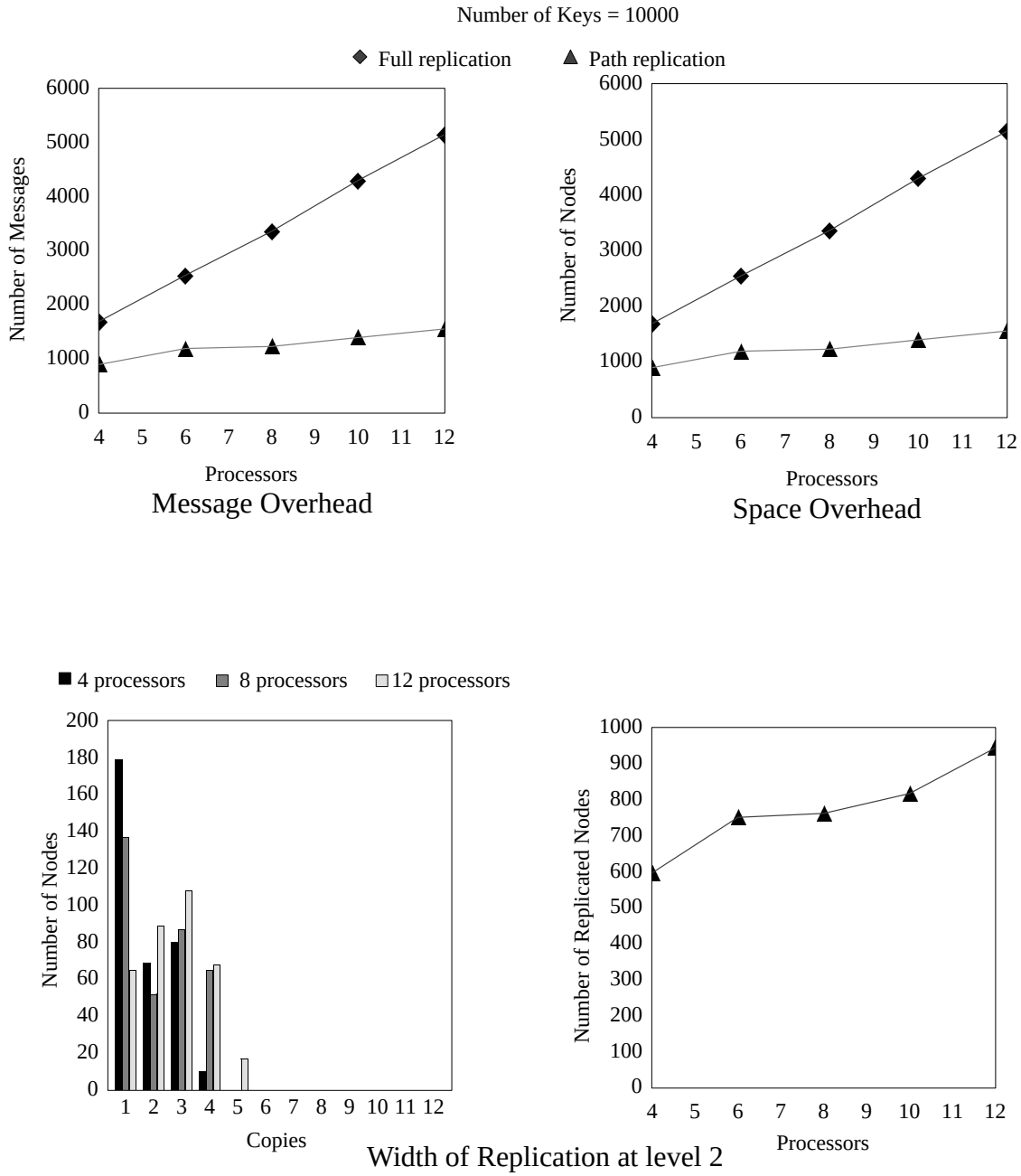
Number of Keys = 10000
Full replication
Path replication
Number of Messages
Processors
Message Overhead
Number of Nodes
Processors
Space Overhead
4 processors
8 processors
12 processors
Number of Nodes
Copies
Number of Replicated Nodes
Processors
Width of Replication at level 2

Figure 5:

the index nodes are replicated on each processor, and finally how many nodes each processor stores at the end of the run.

The *Message Overhead* graph shows the number of messages needed to maintain the replicated B-tree. We see that in case of full replication, the number of messages for a 4 processor B-tree is around 9000 and for 12 processors it is around 35000, i.e. the message overhead has increased linearly as the number of processors. However, for a path replicated B-tree, at 4 processors around 3800 messages are needed and for 12 processors only 9300 messages are needed, not even a linear increase (Figure 5).

The *Space Overhead* graph shows the number of nodes stored at all processors at the end of a run. The graph is similar in nature to the message overhead graph. In this graph we consider only the index nodes that account for the excess storage at each processor, the leaf nodes remaining nearly the same as the number of processors increase. For full replication, we see that for a 4 processor B-tree the number of index nodes stored are 1700, whereas for a 12 processor B-tree the number nodes are 5200, a nearly three-fold increase. In case of a path replicated B-tree, the number of index nodes stored over the entire tree for 4 processors is 900 and for 12 processors is 1550, not even a two-fold increase.

The *Width of replication at level 2* bar graph shows how widely the index nodes are replicated at each processor for a path replicated B-tree. We selected level 2 since activity takes place takes place at the leaf level (1), and affects mostly at level 2. It shows that even as we increase the number of processors, the level 2 index nodes are not widely replicated at all processors, with the concentration being around 4 processors.

Path replication causes low restructuring overhead, but can require a search to visit many processors for its execution. We measured the number of hops required for the search phase of the insert operation after 5000 inserts were requested in an 8 processor distributed B-tree. Full replication required an average of .88 messages per search, and path replication required 1.29 messages per search (additional overhead of .41 messages).

From the above observations, we see that a path replicated distributed B-tree performs better than a fully replicated one and is highly scalable (Figure 5).

6 Conclusions and Future Work

In this paper, we have examined the issues involved in implementing a distributed B-tree. These issues include synchronization, implementing data balancing, and replication strategies. We studied the performance of the data balancing algorithm, which is simple but does perform very well in achieving a good data balance among processors. We examined the overhead of the full and path replication of

the index nodes. We found that path replication imposes much less overhead than full replication and permits a scalable distributed B-tree.

We plan to extend the work to incorporate delete operations and a distributed data balancer. The load balancer discussed previously is a centralized one, with the *anchor* making the decisions. When a processor wants to distribute some of its nodes to some other processor, it requests the anchor for the target processor. This relieves all processors from having to maintain information about all other processors' status. We can extend the idea to a distributed one, where we leave the decision to each processor. One approach is to use a *probing* mechanism to find a receiver processor. Here, we can also include *locality* with the probing. A processor that wishes to download a node **q** will look at processors that hold any ancestors of **q**.

References

[1] Bal, H. E. and Tannenbaum, A. S. *Distributed Programming with Shared Data,* IEEE International Conference on Computer Languages, 1988, pp. 82-90.

[2] Bayer R. and McCreight E. *Concurrency of operations on B-trees,* Acta Informatica 1, 1972, pp. 173-189.

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

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

[5] F.B. Bastani and S.S. Iyengar and I-Ling Yen *Concurrent Maintenance of Data Structures in a Distributed Environment,* The Computer Journal, Vol. 21, No. 2, 1982, pp. 165-174.

[6] Colbrook A., Brewer A. E., Dellarocas C.N. and Weihl E. W. *An Algorithm for Concurrent Search Trees,* Proccedings of the 20th International Conference on Parallel Processing, 1991, pp. 38-41.

[7] Herlihy, M. *A Methodology for Implementing Highly Concurrent Data Structures,* Proceeding of the Second ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, ACM 1989, pp. 197-206.

[8] Johnson T. and Shasha D. *A Framework for the performance Analysis of Concurrent B-tree Algorithms,* Proceedings of the 9th ACM Symposium on Principles of Database Systems, April 1990.

[9] Johnson T. and Colbrook A. *A Distributed Data-Balanced Dictionary Based on the B-link Tree,* International Parallel Processing Symposium, March 1992, pp. 319-325.

[10] Johnson, T. and Krishna, P. *Lazy Updates for Distributed Search Structures* SIGMOD '93.

[11] Jul E., Levy H., Hutchinson N. and Black A. *Fine Grained Mobility in the Emerald System,* ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp. 109-133.

[12] *KSR1 Principles of Operation,* Copyright Kendall Research Corporation, 1991.

[13] Lehman P.L., and Yao S.B. *Efficient Locking for Concurrent Operations on B-trees,* ACM Transactions on Database Systems 6, December 1981, pp. 650-670.

[14] Ladin R., Liskov B., and Shira L. *Providing High Reliability Using Lazy Replication,* ACM Transactions on Computer Systems Vol. 10, No. 4, 1992, pp. 360-391.

[15] Miller R. and Snyder L. *Multiple Access to B-trees,* Proccedings of the 1978 Conference on Information Sciences and Systems, Johns Hopkins University, Baltimore, March 1978, pp. 400-408.

[16] Peleg D. *Distributed Data Structures: A Complexity-Oriented View,* Fourth International Workshop on Distributed Algorithms, 1990.

[17] Sagiv Y. *Concurrent Operations on B-Trees with Overtaking,* Journal of Computer and System Sciences, 33(2), October 1986, pp. 275-296.

[18] Samadi B. *B-trees in a system with multiple users,* Information Processing Letters, 5, 1976, pp. 107-112.

[19] Turek J., Shasha D., and Prakash S. *Locking without Blocking: Making Lock Based Concurrent Data Structure Algorithms Nonblocking,* ACM Symposium on Principles of Database Systems, 1992, pp. 212-222.

[20] Weihl E. W. and Wang P. *Multi-version Memory: Software cache Management for Concurrent B-Trees,* Proceedings of the 2nd IEEE Symposium on Parallel and Distributed Processing, 1990, pp. 650-655.