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

November 4, 1992

Abstract

Distributed search structures are useful for parallel databases and in maintaining distributed storage systems. Although a considerable amount of research has been done on developing parallel search structures on shared-memory multiprocessors, little has been done on the development of search structures for distributed-memory 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 and node mobility. One goal of this work is to implement a data-balanced dictionary which allows for balanced processor and space utilization. We present an algorithm for dynamic data-load balancing which uses node mobility mechanisms. We also study the effects that balancing and not balancing data have on the structure of a distributed B-tree. Finally, we demonstrate that our load-balancing algorithm distributes the nodes of a B-tree very well.

Keywords: Data Structures, Distributed Databases, Distributed Search Structures, Load-balancing.

1 Introduction

Current commercial and scientific database systems deal with vast amounts of data. Since the volume of data to be handled is so large, it may not be possible to store all the data at one place. Hence, distributed techniques are necessary to create large scale efficient distributed storage [7]. Larger amounts of data can be stored by partitioning the system, which also allows for parallel access to the data. The distributed data management's complexity depends strongly on the data structures used. One can improve locality, increase the availability, and make the system more fault tolerant by intelligently distributing a data structure.

Much research has been done on developing parallel search structures on shared-memory multiprocessors, but little has been done on developing search structures for distributed-memory systems. One such search structure is the B-tree. In this paper, we consider the design and implementation issues involved in distributing a B-tree on a message-passing architecture. A message-passing architecture is one in which no memory is shared. We chose the B-tree because of its flexibility and its practical use in indexing large volumes of data. Distributing the B-tree can increase the efficiency and improve parallelism of the operations, thereby reducing transaction processing time.

To provide efficient use of the resources of a distributed system, it's often necessary for all the processors to be utilized to the same extent. If not, problems can occur. For example, one processor might run out of storage space and cause an insert to fail, even though other processors are lightly loaded. The utilization can be evaluated in terms of the space/memory requirements and the load on each processor. In this paper we present some mechanisms for data balancing. Other issues that arise from load balancing are mechanisms for node mobility and out-of-order information handling.

1.1 Previous Work

Several approaches to the concurrent B-tree problem have been proposed [1], [10], [12], [16]. Each of these approaches uses some form of locking technique to ensure exclusive access to a node. All concurrent search tree algorithms require a concurrency control technique to keep two or more processes accessing the B-tree from interfering with one another. The contention is more pronounced at the higher levels of the tree. In shared-memory, this scenario occurs when more than one process contends for the same memory location. In a distributed architecture, contention occurs when one processor receives messages requesting access to a node from every other processor. Sagiv [15] and Lehman and Yao [10] use a link technique to reduce contention.

Several data structures suitable for shared memory parallel architectures have been described. A hash algorithm for massively parallel systems is proposed in [20]. This algorithm, called Two Phase Hashing, combines chaining and linear probing. Parker [13] describes a highly concurrent search structure for parallel shared memory that uses sibling tries. Parker uses links to increase concurrency.

Search structures based on LOLS family, Linear Ordinary-Leaves Structures (such as B^+-trees, K-D-B-trees, etc.) have been proposed in [11]. The paper addresses the problem of designing search structures to fit shared memory multiprocessor multidisk systems. Parallel B-trees using multi-version memory have been proposed by Wang and Weihl [19]. Here, every processor has a copy of the leaf node, and updates to the copies are made in a "lazy" manner. A distributed linear hashing method that's particularly useful for main memory databases is discussed in [17]. The paper addresses the problem of maintaining local copies of the centralized variables, and discusses various recovery mechanisms.

Distributed memory data structures have been proposed by Ellis [3], Severance [17], Peleg [14], Colbrook et al. [2] and Johnson and Colbrook [6]: Ellis [3] has proposed a distributed extendible hashing technique, that uses techniques similar to the ones we use here. Colbrook et al. [2] have proposed a pipelined distributed B-tree, where each level of the tree is maintained by a different processor. The parallelism achieved is limited by the height of the B-tree and the processors are not data balanced.

Johnson and Colbrook [6] present a distributed B-tree suitable for message-passing architectures. The interior nodes are replicated to improve parallelism and alleviate the bottleneck.

Data balancing among processors has been studied by Johnson and Colbrook [6]. Lee [9] et al. have discussed a fault-tolerant scheme for distributing queues. Here, the space required by each processor is balanced. Each processor can keep the same size segment of the queue. Peleg [14] has proposed several structures for implementing a distributed dictionary. The paper addresses message-passing complexity and data balancing.

In the context of node mobility, object mobility has been proposed in Emerald [8]. 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.

2 Concurrent B-tree Link Algorithm

A B-tree of order m is a tree that satisfies the following conditions:

1. Every node has no more than m children.
2. The root has at least two children and every other internal node has at least $m/2$ children.
3. The distance from the root to any leaf is the same.

A search for a key progresses recursively down from the root. If the root holds the key the search stops; otherwise it continues downward. An insert operation results in an insertion if the key is not already in the B-tree. If the node is full (i.e., an insertion would cause it to contain $m+1$ keys), the node splits and transfers half its keys ($m/2$) to the new sibling, and a pointer to the sibling is placed in the parent. If the insertion causes the parent to split, the split moves upward recursively. A delete searches for the key and removes it from the leaf node when found. If the node has less than $m/2$ keys, it is merged with either sibling. This technique is known as *merge-at-half* technique. A better option is to *free-at-empty* — delete nodes only when they are empty.

A variant of the B-tree known as the B^+-tree stores the data only at the leaf nodes. This structure is much easier to implement than a B-tree. A B-link-tree is a B^+-tree in which every node has a pointer to its right sibling at the same level. The link provides a means of reaching a node when a split has occurred, thereby helping the node to recover from misnavigated operations. The B-link-tree algorithms have been found to have the highest performance of all concurrent B-tree algorithms [5]. In the concurrent B-link-tree proposed by Sagiv [15], every node has a field that is the highest-valued key stored in the subtree.

A search operation starts at the root node and proceeds downwards. In this algorithm at most only one node is locked at any time. A search first places a **R** (read) lock on the root, then finds the correct child to follow. Next, the root node is unlocked and the child is **R** locked. Having reached a leaf node, the search finds the correct leaf node (i.e., the one whose highest value is greater than the key being searched for) by traversing the right links in a node. Search returns a success or failure depending on the presence of the key in the leaf node or not (Alg. 1).

```
search (v) {
        node = root;
        Rlock (node);
        while (not leaf) {
                child = findnextnode(node, v);
                unlock (node);
                node = child;
                Rlock(node);
        }                                                                    10
        / traverse right links till correct leaf node is found/
        sib = findsibling (node, v);
        unlock (node);
        success = findkey (sib, v);
        if (success) {
                n = sib;
                return (success);
        }
        else {
                n = NULL;                                                    20
                return (failure);
        }
}
```

Alg 1. Search Algorithm for a B-link Tree.

An insert operation works in two phases: a search phase and a restructuring phase [6]. The difference between the search phase of an insert operation and the search operation described above is that, here the **R** lock on the leaf nodes is replaced by a **W** (exclusive write) lock. The key is inserted if not already present in the appropriate leaf. If the insert causes a leaf node to become too full, a split occurs and the restructuring begins as in the usual B-tree algorithm. Since the operations hold at most only one lock at a time, restructuring must be separated into disjoint operations. The first phase is to perform a *half-split* operation (Figure 1). During this phase, a new node, the sibling, is created and half the keys from the original node are transferred into it. The sibling is put into the leaf list and the sibling pointers adjusted appropriately. The next phase is to inform the parent of the split. Now the lock on the leaf node is released, the parent node is locked, and a pointer to the sibling is inserted into the parent. During the time that the split occurs and the pointer is inserted into the parent, operations navigate to the sibling via the **link** and the **highest** fields in the node.

On-the-fly node deletion is not supported in shared-memory multiprocessors. Several alternatives to on-the-fly deletion exist, including never deleting nodes, performing garbage collection or leaving the deleted nodes as stubs without deallocating them physically. In our design we support deletes and

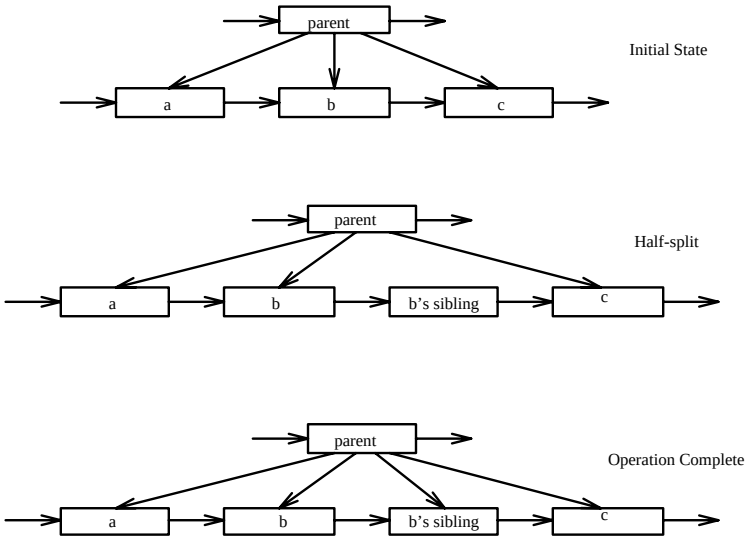

Figure 1: Half-split Operation

perform garbage collection. Currently only the pipelined B-tree supports deletes.

3 Translation to a Distributed B-tree

In the distributed B-tree proposed here, each node has a pointer to its parent and to both its siblings, not to its right sibling alone. Keeping both sibling pointers makes deletion and merging of nodes simpler and also helps with node mobility and address updates. The parent pointers provide an address for split (and merge) messages. The sibling pointers let "lost" messages navigate to the correct node.

As a first cut we implemented a pipelined B-tree by distributing each level of the B-tree to a different processor. We next extended the design to distribute a subset or a superset of a level of the tree to different processors.

3.1 Architecture of the underlying system

We implemented our algorithms on a LAN comprised of SPARC workstations. Later we will extend this to specific distributed architectures like the NCUBE. The processors in the system are assumed to be capable of communicating with any other processor and have sufficient amounts of local storage. Each processor acts as a server capable of responding to messages from others.

3.2 Design

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

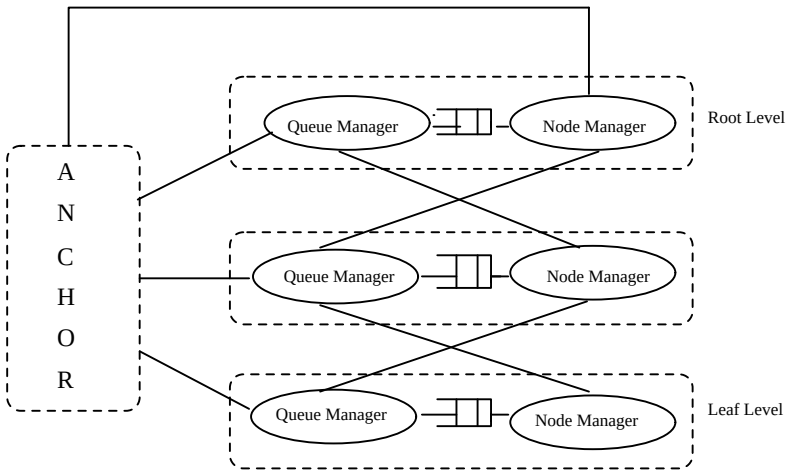

Figure 2: The Communication Channels

queue manager and a *node manager*. The queue manager manages the queue of messages from remote processors. 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 be independent of the message queue 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 node operations are completely local, and we assume that they cannot be interrupted by another message. As a result, locking is implicit and the search, split, and insert operations are atomic.

3.3 Anchor Process

The anchor is responsible for initializing the B-tree. In addition the anchor receives update messages from external applications and sends them to the appropriate processor. Each processor is responsible for the decision it makes concerning the tree structure it holds. In the current implementation the anchor makes the decision if two or more processors are involved. In order to do so, the anchor must have a picture of the global state of the system. The B-tree processing will continue while the anchor makes its decision, so the global picture will usually be somewhat out of date. Our algorithms (for load balancing, for example) take this fact into account.

4 Pipelined B-tree

The overall B-tree structure is monitored by the anchor process at processor zero, which is responsible for creating a *level server* process at every processor (Figure 3). The queue manager of each level server

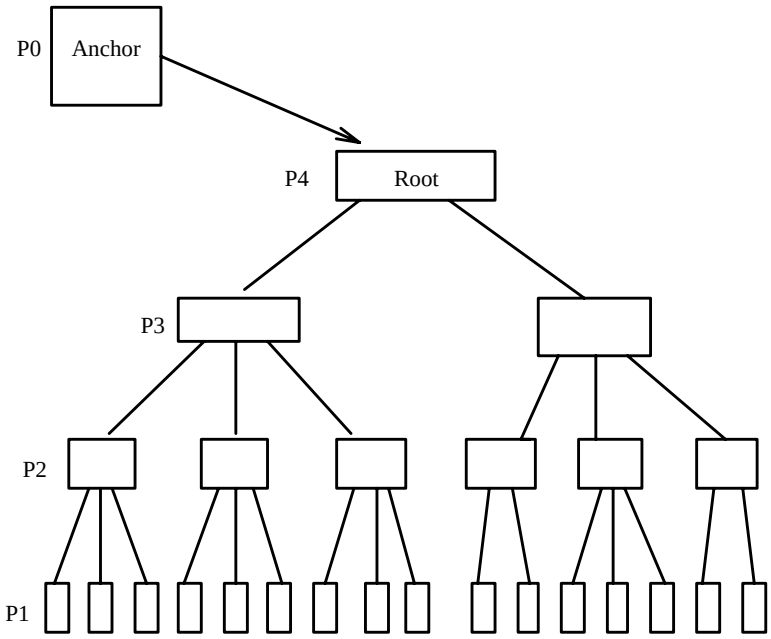

Figure 3: The Pipelined B-tree

and the anchor communicate by means of sockets. The anchor begins building the tree by selecting a processor (the *root processor*) to hold the root of the tree. The node manager at the root processor has a socket connection to the anchor. Update operations are passed to the root processor and percolate down to the leaf level, where the decisive action of the operation is performed. Since each level is maintained by a different processor, percolating an update operation involves passing a message to the processor holding the next lower level of the tree. The node manager at level **i** has a communication channel with the queue manager at levels **i+1** and **i-1**.

Our implementation is primarily concerned with update operations, insert and delete:

- **Insert**: An insert operation at a leaf processor inserts the key in the appropriate node. If the insertion of a key causes the node to become too full, the node splits by creating a new sibling and moving half the keys from the original node to the new sibling. The parent of this node is informed of this split by sending a message to the processor it resides on. To improve parallelism and reduce the number of messages in the system, the child processor does not wait for an acknowledgement from the parent processor; instead we use the B-link tree protocol.

  If the split message from the child causes the parent to split, the child will have an old parent pointer. The obsolete parent pointers are updated when the new parent sends a message to the child (who uses the source node information in the message). Our design can tolerate the "lazy" update of these pointers since a message to the old parent will find the correct (new) parent by

the sibling pointers at the parent processor.

If an insert causes the parent to split, the message percolates up towards the root node. In the event that the root node splits, a message is sent to the anchor. The tree grows by a level, so the anchor allocates a processor to hold the new root.

- **Delete**: A delete operation removes the key from a leaf level node, say node n. The restructuring actions that the algorithm takes depends on whether we have implemented a *free-at-empty* or *merge-at-half* B-tree:

  1. **Free-at-empty:**

     If the removal of a key makes the node n empty, an EMPTY message is sent to the parent node and n is marked as deleted. However, the node remains in the doubly-linked list with its siblings until an acknowledgement arrives from the parent. This ensures that no further updates to the node n will be received, so n is removed from the list and its space is reclaimed. In the interval before the acknowledgement is received any operations to the deleted node n are sent to its siblings (as appropriate).

  2. **Merge-at-half:** In addition to deleting nodes that are empty we have incorporated a *merge* protocol to implement *merge-at-half*. If the removal of a key reduces n to less than half its maximum capacity, the node shares its keys either to the right or to the left. The idea here is to keep the nodes equally full. If the right or left neighbor has more than half the keys, the excess are shared with the node n.

  When the highest value of a node at an index level is changed, a message is sent to inform the parent of the change. The parent is made aware of the key range in its child subtree, so that future updates would be directed properly.

When the parent node receives an EMPTY message, it removes the pointer to the child. On receiving a change in the high value message from a child, the parent changes the highest value of the child. A change in the parent may cause one of the above situations, so the algorithm is applied recursively.

If an EMPTY message reaches the root processor, it checks to see if it has only two children. If so, one of them is deleted and it is left with one child. A message is sent to the anchor to shrink the tree. The anchor makes the only child of the root the new root of the shorter tree, removes the old root node and deallocates the processor holding the old root node. Note that as the distribution is by levels, a

delete of one child and a split in its sibling cannot occur at the same time. The delete and the split will be processed in the sequence that they occur by the root processor.

5 Fully Distributed B-tree

The pipelined B-tree discussed in the previous section is work-balanced but is not data-balanced, since the leaf processor holds more nodes than other processors. In our next approach, a processor can hold more than one level, and a level may be distributed across many processors. Here too, the anchor is responsible for initializing the B-tree and for decision making involving two or more processors. The processors are no longer dependent on the tree structure; hence the connection among them could be arbitrary. In our implementation, however, we have a simple structure where all processors are connected to the anchor. The anchor begins building the tree by allocating the root node to a processor.

5.1 Node Structure

Unlike in the previous case, where all logically adjacent nodes on a level are also physically adjacent, here physically adjacent nodes may not be logically adjacent. Also, nodes cannot be uniquely identified by the local address. A node address consists of a combination of its node identifier and the processor number on which it resides. A typical node would have a parent pointer, the children pointers, and the sibling pointers. 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.

5.2 Updates

This implementation performs inserts only. The insert first searches for a leaf node in the appropriate range. If the key is inserted at a processor and the node becomes too full, it half-splits. After a node half-splits, it must inform its siblings and its parent. Nodes stored on the same processor are updated locally, and a message is sent to update remote nodes. As before, the B-link protocol routes messages that misnavigate.

If the root node splits, a parent has to be created. The processor holding the root node creates a new node and makes that the new root. It however informs all other processors that the tree height has increased (i.e., the tree has grown).

6 Load 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.

6.1 Criteria for Load Balancing

- Equal capacity: Here the objective is to keep all the processors equally balanced in terms of the number of nodes that they hold. That is, all the processors should utilize the same amount of memory space, so that no processor is short of space during execution.
- Equal work: The number of update operations performed on each processor can be a measure for data balancing. If each processor has the same amount of work, to be done, better processor utilization is obtained.
- Equal communication: The number of messages processed by each processor could be another criterion for load balancing. This is helpful in regulating the traffic in the system.

Having decided on the criteria for balancing, the next step is to decide when to perform the balancing. The nodes could be moved following a failure or recovery, or at certain intervals of time, or when the system reaches a particular state.

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 6.4.

6.1.1 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 actual process can be explained by addressing the following problems:

1. **Who is involved:** 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?

2. **When is everyone informed:** Once a node has been selected for migration, how and when is every other processor informed of its new address. In the interim that node movement takes place and the other related processors are informed, what happens to the updates that come for the node in transit? How do they get forwarded?

3. **Obsolete information:** When a node moves it sends an update link message to related processors. Suppose that the update message for link change gets delayed and the node moves for a second time. How is the problem caused by the second update message reaching a processor before the first one, handled?

Our design addresses these problems and provides solutions to them.

Who is involved: Our solution to the first 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 (Alg. 2).

```
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;                                    10
                update pointers locally;
        }
        send close_connection to  receiver;
}

accept_nodes() {
        wait_for_connection;
        send_acceptable capacity;
        while (msg != close_connection) {
                accept a node;                                               20
                send_acknowledgement;
        }
        close_connection;
        update pointers related to new nodes locally;
        send messages to related nodes if not on sending or receiving processor;
}
```

Alg 2. Node Migration Algorithm.

When is everyone informed: The sender and receiver update all locally stored pointers to the transferred nodes. If the related nodes are on different processors (other than the sender and receiver) the receiver sends link update messages to them. If in the meantime some messages arrive for this node at the sender (since all processors are not yet aware of the migration) the messages are forwarded to the new address. At specific intervals of time the nodes marked in transit are deleted and their storage reclaimed.

Since we do not require acknowledgements for the link changes, it is possible that a message will arrive for the deleted node. In this case, the node manager forwards the message to a local node that is "close" to the intended address. 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.

Obsolete information: The final problem is how to deal with out-of-order messages arriving at a processor. This can be explained with an example (Figure 4). 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, an 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 an 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.

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

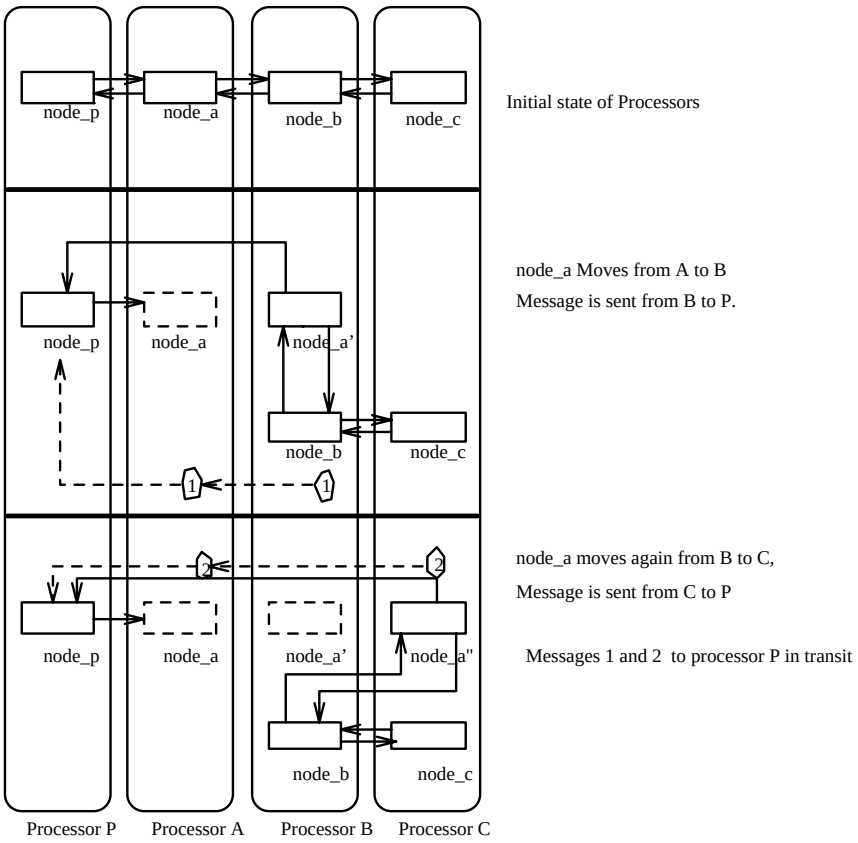

Figure 4: Node Migration

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.

6.2 Protocol for Node Migration

How one selects the nodes that have to be moved depends on factors that affect locality, availability, and fault tolerance. One may select a node at random, leaf nodes, or subtrees to be placed on one processor. Placing subtrees on a processor will reduce the number of sibling, children and parent updates. 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.

6.2.1 Decision Making

Having selected the criterion for load balancing, the next step is to select a processor to receive the nodes. As of now, the anchor has been responsible for choosing the receiver. Another approach would be to leave the decision making to the individual processors.

In our design, a limit is placed on the maximum number of nodes of the tree that a processor can hold. In addition each node has a soft limit 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 picks a receiver from the currently active ones, or selects a new processor. 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. The anchor's responsibility is over and it is free to continue with any other task.

6.3 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. As mentioned earlier, two factors are necessary for load balancing, namely, the criterion and the node selection process.

- Criteria for load balancing: In this experiment we chose the *equal capacity* rule for performing the dynamic load/data balancing.
- Node selection: 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 [5], 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.

6.4 Statistics, Results and Discussion

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 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). Finally, we also calculate the number of messages it would take to travel from the root to any leaf. As expected, the worst-case number of messages required is one (since only the leaves are distributed).

The *Performance* bar charts in Figures 5 and 6 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.

The *Maximum Load* graphs in Figures 7 and 8 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.

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.

7 Conclusion

We have designed and implemented a distributed search structure on a network of SPARC stations. We have studied the effect of dynamic data-load balancing on the B-tree and found that our algorithm performs nicely in achieving a data balance among processors, even with the creation of hot spots. We plan to introduce other variations of load balancing such as distributing subtrees and replicating nodes at various processors. This paper has dealt only with inserts; we intend to incorporate deletes and also provide correctness proofs for the algorithms designed and implemented. We will also extend this to distributed storage management and incorporate fault tolerance in the system.

8 Acknowledgements

We would like to thank Dennis Shasha and Johna Johnson for their valuable comments and suggestions.

References

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

[2] 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.

[3] Ellis S. C. *Distributed Data Structures: A Case Study,* IEEE Transactions on Computers, Vol. C-34, No. 12, December 1985.

[4] Fan Z. and Cheng K. *A Generalized Simultaneous Access Dictionary Machine,* IEEE Transactions on Parallel and Distributed Systems, Vol. 2, No. 2, April 1991.

[5] 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.

[6] 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.

[7] Johnson T. *Distributed Indices for Accessing Distributed Data,* 12th IEEE Mass Storage Symposium 1992. [To appear].

[8] 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.

[9] Lee P., Chen Y. and Holdman M. J. *DRISP: A Versatile Scheme For Distributed Fault-tolerant Queues,* IEEE 11th International Conference on Distributed Computing Systems, 1991.

[10] 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.

[11] Matsliach G. and Shmueli O. *An Efficient Method for Distributing Search Structures,* Proceedings of the 10th IEEE Symposium on Reliable Distributed Systems, 1991.

[12] 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.

[13] Parker D. J. *A Concurrent Search Structure,* Journal of Parallel and Distributed Computing 7, 1989, pp. 256-278.

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

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

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

[17] Severance C. and Pramanik S. *Distributed Linear Hashing for Main Memory Databases,* International Conference on Parallel Processing, 1990.

Uniform distribution					Hot spots distribution				
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

Table 1: Load Balancing Statistics

[18] Shasha D. and Goodman N. *Concurrent Search Tree Algorithms,* ACM Transactions on Database Systems, 13(1), 1988, pp. 53-90.

[19] 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.

[20] Yen I. and Bastani F. *Hash Table in Massively Parallel Systems,* Proceedings of the 1992 International Conferences on Computer Languages, April 20-23, 1992, pp. 660-664.

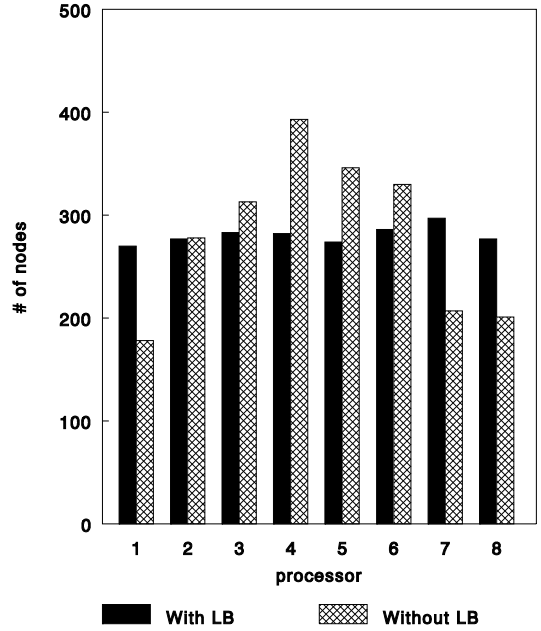

Figure 5: Distribution of nodes to processors, uniform distribution.

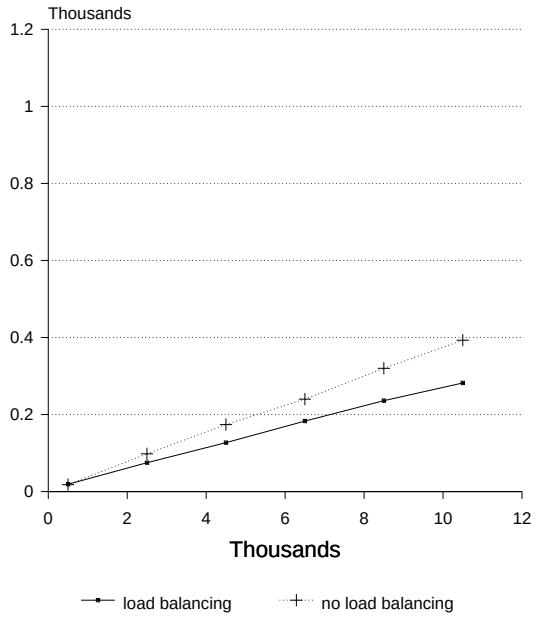

Figure 7: maximum number of nodes in a processor, uniform distribution.

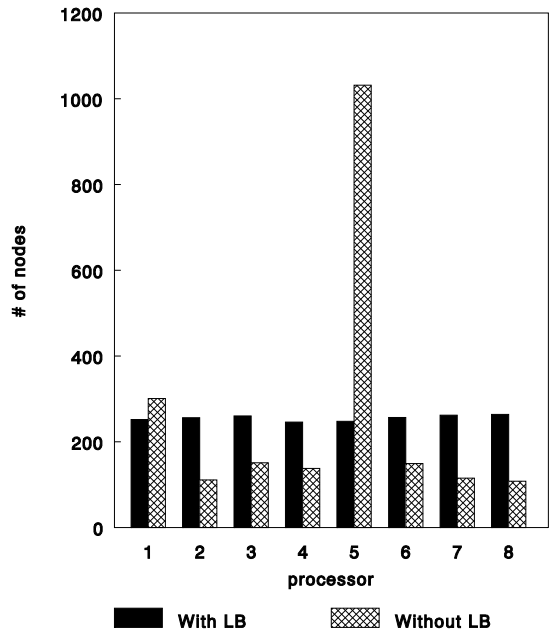

Figure 6: Distribution of nodes to processors, hot spot distribution.

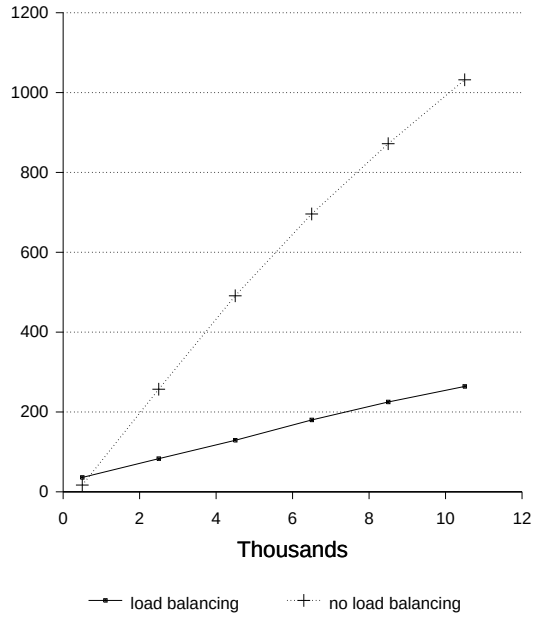

Figure 8: maximum number of nodes in a processor, hot spot distribution.