Highly Scalable Data Balanced Distributed B-trees

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

Abstract

Scalable distributed search structures are needed to maintain large volumes of data and for parallel databases. In this paper, we analyze the performance of two large scale data-balanced distributed search structures, the dB-tree and the dE-tree. The dB-tree is a distributed B-tree that replicates its interior nodes. The dE-tree is a dB-tree in which leaf nodes represent key ranges, and thus requires far fewer nodes to represent a distributed index. The performance of both algorithms depends on the method by which tree nodes are assigned to processors (i.e., the algorithm for performing data balancing). We present a simulation study of data balancing algorithms for the dB-tree and the dE-tree. We find that a simple distributed data balancing algorithm works well for the dB-tree, requiring only a small space and message passing overhead. We compare three algorithms for data balancing in a dE-tree, and find that the most aggressive of the algorithms makes the dE-tree scalable. Using the data from the simulation experiments, we present an analytical performance model of the dB-tree and the dE-tree. We find that both algorithms are scalable to large numbers of processors.

Keywords: Distributed Search Structures, Distributed Databases, Data-Balancing, Performance Analysis.

1 Introduction

Current commercial and scientific database systems deal with vast amounts of data. Since the volume of data to be handled is 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 [JK92]. Larger amounts of data can be stored by partitioning the data, which also allows for parallel access to the data. Managing and indexing large volumes of distributed and dynamically changing data require the use of distributed data structures.

The normal operations carried out on an index are search, insert, 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 examine two distributed search structures, the dB-tree and the dE-tree. The dB-tree is
a distributed $B_{link}$-tree that has replicated index nodes. The dE-tree is similar, but the leaf nodes represent key ranges, and are not limited in size (except by the processor’s storage capacity).

To provide efficient use of the resources of a distributed system, it is often necessary for all the processors to be utilized to the same degree. If not, the imbalanced use of resources at a site may become a bottleneck in the performance of the system. For example, one processor might run out of storage space and cause an insert to fail, even though other processors are lightly loaded. Thus, data balancing algorithms are required for the dB-tree and the dE-tree. The choice of data balancing algorithm is crucial, because the performance of the distributed search structure algorithms depends on their layout, which in turn depends on the data balancing strategy.

In this paper, we look at the performance aspects of our approach of constructing distributed search structures. We develop several data balancing algorithms for the dB-tree and for the dE-tree. We find that a simple distributed strategy works well for the dB-tree, but an aggressive strategy is required to make the dE-tree scalable. Based on our simulation results, we develop a model for large scale dB-tree and dE-tree characteristics. We use the model of the distributed search structures to develop a performance model.

## 2 Previous Work

Several approaches to concurrent access of the B-tree have been proposed [BS77, MR85]. Each of these approaches uses some form of locking technique to ensure exclusive access to a node. Lock contention is more pronounced at the higher levels of the tree. Sagiv [S86] and Lehman and Yao [LY81] use a link technique to reduce contention.

Ellis [E85] has proposed a distributed extendible hashing technique, that uses techniques similar to the ones we use here. A distributed linear hashing method that’s particularly useful for main memory databases is discussed in [S90].

Linear Hashing for distributed files, $LH^*$ has been proposed by Litwin et al. [LNS93]. In this work, Litwin et al. propose a general class of distributed data structures which they term scalable distributed data structures, or $SDDS$. An SDDS algorithm is similar to the algorithms discussed by Ellis [E85]. A significant difference is the manner in which modifications to the global state are distributed. An SDDS algorithm distributes updates passively, after a processor takes an incorrect action. After updating the processor’s state, the action is re-issued. The algorithms that underlie the distributed data structures actively distribute changes to the global state, but handle requests based on out-of-date information. Litwin et al. [LNS94] extend the SDDS family with the $RP^*$ hash table. The $RP^*$ hash table is order-preserving, and thus can
support range queries.

Distributed file organization for disk resident files has been discussed by Vingræck et al. [VBW94]. The focus of their work has been to achieve scalability (in terms of the number of servers) of the throughput and the file size while dynamically distributing data. Their results indicate that scalability is achieved at a controlled cost/performance. Matsliach and Shmueli [MS91] address the problem of designing search structures to fit shared memory multiprocessor multidisk systems. Other related works are multi-disk B-trees proposed by Seeger and Larson [SL91].

Johnson and Colbrook [JC92] 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. They discuss data balancing among processors and suggest a way of reducing communication cost by storing neighboring leaves on the same processor.

In a previous paper, we have proposed algorithms for efficiently replicating the index of a distributed hierarchical search structure [JK92b]. In contrast to the passive approach of the SDDS algorithms, we propose methods for actively distributing updates to the index nodes, but in a lazy manner. By taking advantage of the commutativity of the actions on index nodes, different operations can read and update the same node concurrently (though at different copies). The algorithms are also message efficient, requiring one message per copy per insert.

In [KJ94b] we discuss implementational issues of distributed search structure algorithms. In a previous paper [KJ94], we proposed two strategies for replication, namely path replication and full replication. We found 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. A good data balance is achieved with little node movement overhead.

3 Distributed Search Structure Algorithms

The algorithms for implementing dB-trees and dE-trees are well discussed in our previous work [JC92, JK92b, KJ94, KJ94b]. In this section, we briefly review the algorithms.

3.1 Concurrent B-link tree

The dB-tree is a distributed B\textsuperscript{link}-tree, which in turn 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\textsuperscript{link}-tree algorithms have been found to have the highest performance of all concurrent B-tree algorithms [JS90]. In the concurrent B\textsuperscript{link}-tree proposed by Sagiv [S86], every node has a field that is the highest-valued key stored in the subtree.

The reason for the high performance of the B\textsuperscript{link}-tree algorithms is the use of the half-split operation, shown in Figure 1. When a key is inserted into a full node, the node must split and a pointer to the new sibling inserted into the parent (the standard B-tree insertion algorithm). In a B\textsuperscript{link}-tree, this action takes place in two stages. First, a 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 upwards until the root.

In the interim between the half-split of the node and the completion of the split at the parent, operations traveling down to a leaf node may misnavigate, and search for a key in the half-split node, when that key has moved over to the sibling. The range information and the link to the sibling stored in the node help the operation recover from the misnavigation. This localizes all actions on the B\textsuperscript{link}-tree. 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.

3.2 The dB-tree

To distribute a B-tree over several processors, we migrate only the leaf level nodes, with the index level nodes being replicated. 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.

The insert operation works in two phases: the search phase and the restructuring phase. The search phase is the same as above. 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, a typical node structure, the implemented algorithms for distribution and issues encountered.

3.2.1 Design

Our design of the distributed B-tree consists of a queue manager and a node manager on each processor. The queue manager is solely designated to message handling. The node manager is responsible for the actual partial structure of the B-tree that the processor holds. It handles the processing of the operations on the various nodes at that processor. The queue manager and the node manager communicate by primitives supported by UNIX, namely message queues (Figure 2). An overall B-tree manager called the anchor overlooks the entire B-tree operations. (the responsibility of the anchor is minimal). The queue manager and the node manager communicate with the anchor and with the other processors by sockets. A more detailed discussion of the implementation appears in [KJ94, KJ94b].

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. A typical node would have a unique name, a version number that is used to produce ordered histories ([JK92b]) and pointers to the primary parent, local parent, children, and siblings, beside other fields.

3.2.2 Replication Control

In a previous paper we ([JK92b]) 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 designate one copy of a node to be the primary copy (PC). Our
implementation of the Fixed-Position copies algorithm is termed **Full Replication** and that of Variable copies is **Path Replication**. A previous work shows that path replication is better than full replication [KJ94], so we discuss only the variable copies algorithm here.

In the Variable-copies algorithm ([JK92b]), 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, a processor will ‘unjoin’ a node when it has no copies of the node’s children.

In the implementation [KJ94], whenever a leaf node migrates to a different processor, the 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 those parent nodes not already resident at this processor are replicated. All link changes are again handled by the primary copy of a node. Our approach requires few messages and takes advantage of the commutativity of the messages.

Coherency is maintained among the various copies of a node by having some consistency messages to keep the copies updated of the latest changes. Operations can be started on any processor and on any copy. For example, 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. If the insert does not cause a split, then only \( c - 1 \) messages are required, where \( c \) is the number of copies of the node.

A *split* operation is first performed at a leaf. If the local parent exists on the same processor the split is
performed at the local parent. If the split at 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. In most cases, a split requires only $c - 1$ messages.

### 3.2.3 Implementation and Performance

In [KJ94] we discussed the design issues in the implementation of a distributed B-tree, such as synchronization, implementing data balancing, and replication strategies. We briefly summarize the results obtained for our replicated B-trees here. We inserted a total of 15000 keys in a B-tree distributed over 4 to 12 processors. Observations were made on the number of times load balancing is done, number of consistency messages to keep the replicas coherent, width of replication and number of nodes stored at a processor at the end of the run. We found that path replication imposes much less space and message overhead than full replication and permits a scalable distributed B-tree.

#### 3.3 The dE-tree

To reduce the communication cost, Johnson and Colbrook suggest the dE-tree, where neighboring leaves are stored on the same processor. They define an extent to be a maximal length sequence of neighboring leaves that are owned by the same processor. When a processor decides that it owns too many leaves, it first looks at the processors who own neighboring extents. If the neighbor will accept the leaves, the processor transfers some of its leaves to the neighbor. If no neighboring processor is lightly loaded, the heavily loaded processor searches for a lightly loaded processor and creates a new extent.

Figure 3 shows a four processor dB-tree that is data balanced using the extents. The extents have the characteristics of a leaf in the dB-tree: they have an upper and lower range, are doubly linked, accept the dictionary operations, and are occasionally split or merged. The extent-balanced dB-tree can be treated as a $dE$-tree: the distributed extent tree. Each processor manages a number of extents. The keys stored in the extent are kept in some convenient data structure. Each extent is linked with its neighboring extent.

The extents are managed as the leaves in a dB-tree. When a processor decides that it is too heavily loaded, it first looks at the neighboring extents to take some of its keys. If all neighboring processors are heavily loaded, a new extent is created for a lightly loaded processor. The creation and deletion of extents, and the shifting of keys between extents in the dE-tree correspond to splitting and merging leaves in the dB-tree, and the index can be updated by using dB-tree algorithms.

Since a processor can store many keys, the index size is proportional to the number of processors. Also, index restructuring is greatly reduced as it takes place only after a large number of keys have been inserted.
Dynamic updates to the database mean that some processors will run out of storage while other processors have plenty of room (especially in the presence of hot spots). An advantage of using distributed search structures to manage storage is automatic data balancing. There are many possible data balancing strategies, with different performance implications. In this section, we will examine the performance of data balancing algorithms for dB-trees and dE-trees. We will also make observations that let us predict the performance of large scale trees.

4 Data Balancing

In our simulation, a limit is placed on the maximum number of nodes of the tree that a processor can hold, termed as the threshold. A processor’s threshold corresponds to attached storage. In addition each node has a soft limit (.75 * threshold) on the number of nodes it stores. If the number of nodes in a processor exceeds the soft limit (i.e., after a split), the processor will initiate the data balancing process.

Our algorithms are characterized by the method by which the receiver processor is selected. The simplest approach is centralized data balancing. The anchor processor stores a guess about the load and capacity at every processor. An overloaded processor contacts the anchor and asks for a receiver processor. The anchor updates its tables during these contacts.

A more scalable approach is to use distributed data balancing. When a processor determines that it is
overloaded, it probes the other processors to find a lightly loaded receiver (stopping when a good candidate is found). The probing can be *sequential*, where the probing works through a pre-determined list, or *random*, where the probe is determined by a coin flip.

After a receiver processor \( r \) has been selected, the sender \( s \) and the receiver \( r \) interact by a *negotiation protocol*. In this protocol, they decide exactly how many nodes are to be transferred from the sender \( s \) to the receiver \( r \). The negotiation protocol is necessary because in the interim that the receiver processor is selected and the actual node transfer takes place, the receiver or sender may experience more splits and hence a change in their capacities. Especially in the case of centralized load balancing, the anchor is likely to have poor information about the receiver’s load.

### 4.1.1 Performance Analysis

We were interested in answering several questions about the performance of data balancing algorithms and about dB-trees. A previous study [KJ94] showed us that the data balancing algorithms are effective in balancing the load, and at a low overhead. For this study, we are interested in determining their effect on the structure of the dB-tree — the storage overhead and the message passing overhead. In addition, we want to predict the structure of dB-trees that have very large nodes and are distributed over a large number of processors. We use these predictions in the performance model of the next section.

To determine the nature of a large scale dB-tree, we made a simulation study of data balancing. We computed the number of message *hops* required to complete an operation, and the *width of replication*, or average number of copies of a node. We are mainly concerned with the width of replication of level 2 nodes (which are most of the index nodes). The width of replication is a measure of the space overhead of maintaining a distributed index.

There are many non-algorithmic factors that can affect performance. The number of hops that an operation requires to find its data increases with the height of the tree. The width of replication increases with both increasing fanout and increasing numbers of processors that store the dB-tree. Finally, the manner in which additional storage is made available to the search structure affects the performance of the data balancing algorithm. To reduce the number of parameters that we need to examine, our experiments used the following two scenarios:

**Incremental Growth:** When the storage for the distributed index runs low, the system manager must add storage capacity to some of the processors, or allow the dB-tree to spread to more processors. Periodically, we perform incremental storage growth at the processors that store the dB-tree. This is equivalent to adding a disk to a site or creating a new storage site. When a processor wishes to share some of its nodes, and
all the currently active processors are near their threshold, either a new processor is started up, or (in the event that the processor limit is reached) a processor is selected randomly and its threshold is increased by a fraction of its current capacity. The overloaded processor then shares its nodes with this new processor with newly added capacity.

**Fixed Height Data Balancing:** To study the effect of large fanout on the width of replication, we fixed the height of the tree to 4 for all of the experiments.

### 4.1.2 Experimental Setup

We create an initial dB-tree with a uniform random distribution of keys. After the initial dB-tree is created we vary the key distribution pattern dynamically. To study the effect of our load balancing algorithm when the distribution changes, we have introduced **hot spots** in our key generation pattern, where we concentrate the keys in a narrow range, thereby forcing about 40% of the messages to be processed at one or two ‘hot’ processors.

We performed simulations on fixed height large dB-trees by inserting up to 2.5 million keys and varying the average fanout from 10 to 40 (average fanout is 69% of the maximum fanout [BY89]). When the root of the tree had the desired average fanout we collected statistics. We noted the processors’ capacity in terms of the number of leaves it has, the number of index level nodes, and the number of keys. We also noted the number of times a processor invokes the load balancing algorithm, the number of probes required, the number of nodes that it transfers and the average number of times a leaf node moves between processors (taken with respect to the nodes in the entire B-tree).

To calculate the number of message hops for a search, we simulated 10000 searches. A key to be searched is generated using a uniformly distributed random number. Since the path is replicated at each processor, every processor has a copy of the root of the tree. The search begins at the root of the tree on a randomly chosen processor. The search proceeds downward towards the leaves on the processor, and when a child has to be searched that is no longer on this processor, then a new processor is randomly chosen from among the processors that hold a copy of the child. This continues till a leaf node is reached. The message count is incremented each time a new processor is selected. We also noted at what level in the tree these processor boundaries are crossed. We finally calculated the average messages per search over all levels and over each level.
4.1.3 Results

The graphs in the Figure 4 show the width of replication at level 2 and the width of replication over all levels, plotted against an increasing fanout for a fixed number of processors.

The WOR (width of replication) at level 2 reaches a plateau around 2.1 for 10 processors (4a), around 2.8 for 30 processors (4b) and 3.2 for 50 processors (4c). Similarly, the width of replication over all levels shows that for 10 processors the plateau is about 2.2 (4d), for 30 processors it is around 3 (4e), and for 50 processors it is approximately 3.5 (4f).

The number of hops required to perform an operation shows a similar phenomena. Figures 5a through 5c plot the number of hops per operation against increasing fanout for a fixed number of processors. Again, the number of hops quickly reaches a plateau.

The data in charts 4a through 4f and 5a through 5c, lets us conclude that, for a dB-tree with a large fanout, the width of replication and the number of hops per operation depend on the number of processors only (and not the fanout). Therefore we can predict the number of hops and the width of replication by studying the increase in the plateau value with an increasing number of processors.

Figure 5d shows the effect of increasing the processors on the number of hops. Our results indicate that the hops do not increase significantly and reach a value of 1.9. We conclude that in a large scale dB-tree with 4 levels, only 2 hops are needed to complete an operation.

In figure 5e, we plot the plateau value of the width of replication against the fanout. Since the width of replication appears to be a linear function of the number of processors, we applied a linear regression model to the data. If \( P \) is the number of processors, and \( R(P) \) is the width of replication at level 2 under \( P \) processors, then:

\[
R_2(P) = 1.73 + 0.0295P \quad \text{sequential probing}
\]

\[
R_2(P) = 1.86 + 0.0248P \quad \text{random probing}
\]

If we have a 1000 processors and a fanout of 1000, then the WOR for level 2 nodes is about 27 for random probing, 31 for sequential probing.

In the path replication algorithm for the dB-tree, the width of replication for the root is the number of processors, and for the leaves is 1. We have just derived a model that predicts the width of replication of the level 2 nodes. We also examined the width of replication at level 3. For a height-4 tree with 50 processors and an average fanout of 40, the WOR at level 3 is 23.3. We plot the WOR for each level of the tree in Figure 5f. We find that a good estimate of the WOR at level 3 is \( P/2 \).
4.2 The dE-tree

The difference between the dB-tree and the dE-tree is that it is the load balancer that decides whether to split or merge an extent instead of a leaf. The load balancer is invoked when a key is inserted into an extent. If the load balancer decides that the processor holds too many keys, it decides to download some of its keys to a receiver processor. The balancer selects an extent and decides to either perform a merge or a split. The processor with which to merge or give away the split sibling is also selected based on certain criteria. Our algorithms differ in the manner of extent and processor selection.

In each of these algorithms, the load balancer decides if processor $P$ has an excess number of keys. Let the excess number of keys be $k$.

**Random:** As the name suggests the extent to be merged or split is selected randomly.

**Merge:** Here, we select an extent such that it can be merged with either its left or right neighbor. If there is no such extent then the largest extent is chosen for a split.

1. Let $n$ be the first extent in the list of extents owned by $P$.
2. If the extent $n$ has a right neighbor $r$ and $r$’s owner processor has available capacity then transfer the excess nodes and stop.
3. If the left neighbor $l$ of $n$ has available capacity, then transfer the excess nodes and stop.
4. If there is another extent on the list, let $n$ be the next extent and go to 2. Otherwise, continue.
5. Scan the entire list of extents again and get the largest extent $s$ owned by $P$.
6. See if any of the processors have free key space. If so, let the processor be $R$. Otherwise, randomly select a processor $R$ and increase its capacity.
7. If $R$ is either the right neighbor’s owner or the left neighbor’s owner, merge $s$ with $R$ by transferring the excess keys and stop.
8. Split the node $s$. Give the new sibling to processor $R$. Stop.

**Aggressive Merge:** In the above merge algorithm, we search through $P$’s extents for one such that a neighbor can take all keys offered. In the aggressive merge approach, we first search for an extent such that a neighbor can take all of the keys in the extent. Then, if we cannot find any neighbor that can take all
keys, we settle for sending lesser number of keys (than $k$). So, we search for a neighbor that can accept the largest number of keys. The strategy works because on the next insert, the processor will balance again.

1. Set $\text{merge\_node} = \text{NULL}$; Set $\text{maximum} = 0$; Let $n$ be the first extent in the list of extents owned by $P$.

2. Let $N$ be the neighbor processor with the greater amount of free space, $f$. If ($f > k$) set $\text{merge\_node} = n$, transfer $k$ keys from $n$ to $N$ and stop.

3. If $f > \text{maximum}$ set $\text{merge\_node} = s$ and $\text{maximum} = f$.

4. If there is another extent on the list, let $n$ be the next extent and go to 2. Otherwise, continue.

5. If $\text{maximum}$ is 0, then go to step 5 of the merge algorithm. Else $\text{merge\_node}$ gives the node that can be merged with its neighbor by giving away $\text{maximum}$ keys. Stop.

4.2.1 Experiments

We are interested in determining which of the three algorithms is best, and if the differences are significant. In addition, we are interested in the structure of the $dE$-tree – the number of extents that are created, the height of the tree, and the width of replication. To answer these questions we wrote a simulator.

The simulation of the $dE$-tree is similar to that of the $dB$-tree, except that the leaves hold key ranges (extents) and can hold an arbitrary number of keys. A uniform random distribution of keys is chosen to create the initial $dE$-tree. Initially each processor was given one extent with a range of keys. We inserted a total of 500,000 keys. To study the load variation behavior under execution, we collected distributed snapshots of the processors at intervals of every 50,000 keys inserted in the $dE$-tree. At each snapshot, we noted the processors’ capacity in terms of the number of extents it has, the number of index nodes at each level, the ratio of current number of keys to the maximum that the processor can hold, and the number of splits, merges, and deletes. We also noted the number of times a processor invokes the load balancing algorithm and the number of nodes that it transfers. Other important statistics are the number of message hops for a search, the width of replication and the number of probes required for load balancing. We also calculated the average number of times an extent moves between processors.

4.2.2 Results

We first compared the random and the merge algorithms. In this experiment we built a $dE$-tree of an average fanout of 10 in the interior nodes. We inserted 500,000 keys and used from 10 to 50 processors. We observed
that the two algorithms behaved quite similarly for certain statistics. Both algorithms did a good job at maintaining a data balance with the mean being around 74% of capacity, and the variance in load being 0.000001. The number of hops per message also varies similarly in both algorithms, ranging 1.18 to 2.04 while varying the processors from 10 to 50 in a tree of height 3. The width of replication varied between 5.8 to 7.13 for the algorithms.

The difference in the algorithms is reflected in the number of extents and the number of interior nodes that are stored at each processor. We see in Figure 6a that the random algorithm stores 4500 extents, whereas the merge algorithm stores 530 extents. Thus, the merge algorithm does a far superior job at reducing the storage overhead of the dE-tree. However, the number of merges that occur is about 1000 for the random algorithm and 1900 for the merge algorithm. The merge algorithm also incurs a larger restructuring cost, with 70 nodes and 346 copies being touched (i.e. involved in the restructuring) while only 16 nodes and 71 copies are touched by the random algorithm.

Next, we compare the merge and the aggressive merge algorithm. A comparison of the number of extents in the tree after 5,000,000 keys are inserted in a 30 processor tree is shown in Figure 6c. The number of extents for the merge algorithm are about 2048, whereas for the aggressive merge the number of extents is only 339. The plot shows us that the aggressive merge algorithm is significantly more efficient.

In Figure 7 we plot the number of extents versus the number of keys for different numbers of processors, varying them between 10 and 50, for the aggressive merge algorithm. It can be seen from the charts (7a and 7b) that the number of extents is flattening out, reaching a plateau for the plot of 10 and 20 processors. A good algorithm should have no more than about $n(n-1)/2$ extents (a processor is neighbors with every other one). Our aggressive merge algorithm achieves this as the number of extents flattens out with increasing numbers of keys for 10 and 20 processors. For 30 or more processors, the simulation did not execute long enough to reach a plateau value, as the final number of extents is less than $n(n-1)/2$ for $n > 20$.

We observed the width of replication at all levels, the height of the tree and the number of hops per message for a dE-tree with 5 million keys. We found that the height of the dE-tree is 3 for 10 processors and 4 for 20 to 50 processors, with the number of hops varying from 1.05 to 1.74 as we increase the processors from 10 to 50. The width of replication at level 2 varies between from 6.14 to 10.65. We thus see that our algorithm does not significantly increase the space and message overhead.
5 Performance Model

In this section, we present a simple analytical model that predicts operation response times and the maximum throughput of the distributed search structures described in this paper. The performance depends on the structure of the dB-tree or dE-tree. For example, the number of hops per operation and the degree of replication both affect the amount of overhead required to maintain the search structure. These values are very difficult to calculate, and they depend on the algorithm used to perform the data balancing. For this reason, we will use the estimates of the number of hops and the degree of replication developed in Section 4.1. The model described in this section is loosely based on the model presented in [JS93]. We assume that operations are generated uniformly at all processors, and the accesses are made to the data uniformly.

We first define the variables that we use in the analysis:

$L$: Number of levels in the search structure (level 1 is the leaf, level $L$ is the root).

$P$: Number of processors that maintain the search structure.

$H$: Average number of hops required to navigate to a leaf.

$R_i$: Degree of replication at level $i$, $i = 1, \ldots, L$. $R_1 = 1$ and $R_L = P$.

$F$: Maximum node fanout.

$q_i$: Probability that an operation is an insert operation.

$p_{rez}$: Probability that an operation causes restructuring (split or merge).

$t_s$: Message transmission time.

$t_a$: Time to process an action, search structure.

$t_m$: Processing time for sending and receiving a message.

$\lambda$: Arrival rate of operations to a processor.

$\lambda_{tot}$: Total arrival rate of operations to the distributed

$N_a$: Average number of actions generated by an operation.

$N_m$: Average number of messages generated by an operation.

$W$: Waiting time.
$T$: Response time of an operation.

$Th_{max}$: Maximum throughput.

We start by determining the number of messages and actions required to process an operation, $N_a$ and $N_m$. Since there are $L$ levels, $L$ search actions are required. Since each operation requires $H$ hops, $H + 1$ messages are required (a slightly pessimistic estimate). In addition, an operation might cause restructuring. If there are more inserts than deletes, then $p_{res} \approx 1/(0.68 * F)$ [JS93]. When a node splits, the sibling is created, its right and left neighbors must be informed, and all copies of the parent must be informed about the new sibling. In turn the parent might split, with probability $p_{res}$. Therefore,

$$N_a = L + q_i \sum_{i=1}^{L-1} p_{res}^i (3R_i + R_{i+1})$$  \hspace{1cm} (1)

$$N_m = H + q_i \sum_{i=1}^{L-1} p_{res}^i (2R_i + R_{i+1} - 1) + 1$$ \hspace{1cm} (2)

If $\lambda$ is the rate at which operations are generated at a node that helps to maintain the distributed search structure, then the total rate at which operations are generated is

$$\lambda_{tot} = P\lambda$$  \hspace{1cm} (3)

A processor that helps to maintain the distributed search structure will be required to process jobs that correspond to actions and jobs that correspond to message passing. The average time to process a job is:

$$t_{avg} = (N_at_a + N_mt_m)/(N_a + N_m)$$  \hspace{1cm} (4)

Since the root is fully replicated, it is not a bottleneck. If the data balancing distributes the nodes properly, then no leaf node is a bottleneck either. Therefore, the work to execute an operation is evenly spread among the processors in the system. As a result, the processor utilization due to search structure processing is

$$\rho = \lambda/(N_at_a + N_mt_m)$$  \hspace{1cm} (5)

The time that a job spends waiting for processor service can now be calculated by applying a queuing model. We use a simple $M/M/1$ queue, and find that

$$W = t_{avg} \frac{\rho}{1 - \rho}$$  \hspace{1cm} (6)

The time to get a response from an operation is the time to process all messages and actions associated with the operation.

$$T = L(W + t_a) + (H + 1)(W + t_s + t_m)$$  \hspace{1cm} (7)
The maximum throughput is the maximum rate at which every processor can execute the jobs associated with the search structure operations.

\[ Th_{max} = P / (N_a t_a + N_m t_m) \]

In a distributed search structure with a large number of processors, the overhead of maintaining the search structure is primarily due to the number of hops, \( H \), and the cost of maintaining the level 2 nodes. As we saw in Section 4.1, \( H \) approaches an asymptote for a fixed-height tree. The algorithms described in [JK92b] require \( R_2 \) actions for every split of a level 1 node. Fortunately, we found that \( R_2 \) grows very slowly with increasing \( P \). As a result, the overhead of maintaining a dB-tree does not increase as fast as the processing power of the system increases when processors are added. As result, the dB-tree algorithm is scalable to a very large number of processors.

5.1 An Application

Let us make an analysis of a large dB-tree, one \( P = 1000 \) and \( F = 1000 \). In Section 4.1, we saw that in a large-fanout dB-tree with 4 levels, the number of hops is about 2, and the width of replication on level 2 is about \( 1.908 + 0.0248 \times P \), where \( P \) is the number of processors. We have found that the level 3 nodes are almost fully replicated, so we will assume that \( R_3 = P \). We measured our current unoptimized implementation of a dB-tree, and found that \( t_a = .0045 \) seconds.

With these statistics in mind, we will use the following additional parameters as input to the model:

\[ t_m = .001 \]
\[ t_s = .001 \]
\[ q_i = .1 \]
\[ p_{res} = 1 / (0.69 + F) = .00145 \]

We use these parameters to determine the number of messages and actions that an operation generates.

\[ N_a = 4.005 \]
\[ N_m = 3.004 \]

We can use the the estimates of the number of actions and messages to compute the average execution time and the maximum throughput:

\[ t_{avg} = .0030 \]
\[ Th_{max} = 47560 \]
With a processing rate of 23780 operations per second, $\rho = 1/2$, and the response time for an operation is .035 seconds.

For a comparison, consider the performance of a centralized index server that has the same message passing cost, $t_m = .001$. Servicing each request requires the processing of two messages (the request and the response). We will assume that the actual index lookup requires $t_a = .0045$ seconds. Then, servicing and operation requires .0065 seconds, allowing a maximum throughput of 153.8 operations per second. If the processing rate is 77 operations per second, then the response time for an operation is .015 seconds. Therefore, at the cost of an doubled latency, the throughput is increased by a factor of 300 by using the distributed search structure.

6 Conclusions and Future Work

The focus of this paper has been to examine the performance of data balancing algorithms on large scale distributed B-trees. We simulated distributed B-trees and developed several strategies of data balancing. All the algorithms achieve a good data balance, so the factors that make some algorithms superior are the width of replication (storage overhead) and the number of message hops per operation (message passing overhead). We found that for dB-trees, a simple distributed sequential probing algorithm works well.

We performed a performance study to determine the characteristics of a large scale dB-tree (1000 processors and node fanout of 1000). We found that the overhead of maintaining a dB-tree is not significantly affected by the node fanout, as long as the number of processors is large. We then studied the effect of increasing the number of processors, and found that the overhead of maintaining the dB-tree grows very slowly. The number of message hops approaches a limit, and the width of replication is a slowly growing linear function of the number of processors. We conclude that the dB-tree scales well to a large number of processors.

Next, we examined three data balancing algorithms for the dE-tree, which stores extents of keys in its leaves. We found that the aggressive merge algorithm to have significantly better performance than the other algorithms we tested. The asymptotic number of leaves in a dE-tree using the aggressive merge algorithm approaches about $n(n-1)/2$. Typically this is much smaller than the number of leaves in a dB-tree.

We used our empirical model of the characteristics of the large scale dB-tree to develop a simple analytical performance model. We found that a distributed search structure permits a much larger throughput than a centralized index server, at the cost of a modestly increased response time.

Much work has been done lately on distributed hash tables [LNS93, LNS94, S90, E85]. Because of their
two level structure, a search operation typically requires two messages (the request and the reply), while we have found that a four level dB-tree requires 3 messages per search operation (two for the request and one for the reply). In spite of the additional message passing required for the dB-tree, the hierarchical structure has several advantages. In a very widely replicated hash table, every processor must store a copy of the index. This can impose an unacceptable storage overhead. In the dB-tree, the degree of replication of the second level nodes increases very slowly with the number of processors, making the dB-tree more scalable than a distributed hash table. In addition, it is easier for a processor to join the dB-tree than a distributed hash table, because only a few index nodes of limited size must be transferred, instead of the entire index. As in centralized storage, we can see that hash tables and hierarchical indices both have advantages and disadvantages, and areas of preferred application.

We intend to perform timing experiments on our implementation to collect processing time and message delays. We also intend to study multidimensional search structures and range queries.

References


Figure 4: Width of Replication of the Fixed Height dB-tree
Figure 5: Performance of the Fixed Height dB-tree
**dE–tree: Comparison of the Random vs. Merge Algorithms**

Number of Keys (500,000)

![Graph a. Random vs. Merge](image)

**dE–tree: Comparison of the Merge vs. Aggressive Merge Algorithms**

Number of Keys (5 Million)

![Graph c. Merge vs. Aggressive Merge](image)

**Figure 6**: Performance of the dE-tree
**dE-tree with 5 Million: Number of Leaves**
(Aggressive Merge)  +  Aggressive Merge

**Figure 7**: Performance of the Aggressive Merge Algorithm