1-1-1987

The Ubiquitous B-tree: Volume II

Sally E. Fischbeck

Rochester Institute of Technology
School of Computer Science and Technology

The Ubiquitous B-Tree

Volume II

by
Sally E. Fischbeck

September 20, 1987

A thesis, submitted to
The Faculty of the School of Computer Science and Technology,
in partial fullfillment of the requirements for the degree of
Master of Science in Computer Science

Approved by: _____ 10-6-87
Jeffrey A. Lasky, Chairperson

_____ 10/6/87
Henry A. Etlinger, Committee Member

_____ 6 Oct 87
Dr. Peter Anderson, Chairman, Graduate Computer Science

Date: October 6, 1987

Title of Thesis: The Ubiquitous B-Tree, Volume II

I, Sally E. Fischbeck hereby grant permission to the Wallace Library, of RIT, to reproduce my thesis in whole or part. Any reproduction will not be for commercial use or profit.

_____

Sally E. Fischbeck

Abstract

Major developments relating to the B-tree from early 1979 through the fall of 1986 are presented. This updates the well-known article, "The Ubiquitous B-Tree" by Douglas Comer (Computing Surveys, June 1979). After a basic overview of B and B^+ trees, recent research is cited as well as descriptions of nine B-tree variants developed since Comer's article. The advantages and disadvantages of each variant over the basic B-tree are emphasized. Also included are a discussion of concurrency control issues in B-trees and a speculation on the future of B-trees.

Table of Contents

Table of Figures

Chapter One - Statement of the Problem

1.1 Introduction

The B-tree is an indexing structure first introduced less than twenty years ago. In its short history, the B-tree has revolutionized indexing methods and has achieved widespread acceptance as the structure of choice for indexes that are too large to store entirely in main memory. The characteristics primarily responsible for the success of the B-tree include low cost for insertion, deletion and updating data, and structure preserving dynamic tree reorganization.

Interest in B-trees has been intense since they were first formally described by Bayer and McCreight in their 1972 classic paper "Organization and Maintenance of Large Order Indexes". In 1973, Donald Knuth included a description of B-trees in Volume Three of The Art of Computer Programming. Variants of B-trees were proposed even in these early writings.

By 1979 the classic report by Comer "The Ubiquitous B-tree" was published. Comer's article was, in part, a primer on B-trees, describing in detail their basic characteristics and algorithms. By surveying the literature available through 1978, he also presented an overview of the most recent directions of research being pursued at the time. Comer included descriptions of several B-tree variants, as well as a detailed description of IBM's B-tree based VSAM access method.

Interest in B-trees continues to be strong. Many new variants have been developed since Comer's article. The B-tree has turned out to be a very flexible structure that can be altered to fit the indexing needs in many diverse application areas. Also, new directions in research have been taken since the late 70's.

Apparently, there has been no comprehensive update of the work done in the B-tree area since Comer's 1979 article. The research reported here fills this void and so explains the title "The Ubiquitous B-tree, Volume II". We report on major developments relating to B-trees from early 1979 through the fall of 1986.

1.2 Comparison to Volume I

Comer begins his article by defining the basic terminology relating to trees, files, index files and operations on index files. The reader is assumed to be familiar with this terminology.

Both works devote a section to defining the structure, algorithms and costs of a basic B-tree. Volume II's definitions are more formal and include more details and generalizations. The present section on costs is less theoretical than Comer's, but contain a report on recent findings relating to costs. Finally, Volume II summarizes the major advantages and disadvantages of B-trees.

As in Comer, Volume II includes a section summarizing some of the recent advances and directions of research involving B-trees. Both authors gathered most of this information from journal articles and conference papers, as such information is not available in textbooks.

Volume I and II present some of the same variants, specifically the B^*, B^+ and Prefix-B^+ trees. Volume I discusses variants of B-trees which are only suitable for one-level store; the Binary B-tree and the 2-3 Tree. In II, the study has been limited to variants where it is assumed the data base is too large to reside in main memory.

New variants discussed include variants concerned with improving storage characteristics; the Compact B-tree, the dense multi-way tree and a hybrid called the H-tree. Other new variants handle complex data; the multi-dimensional data B-tree (MDBT) and the R-tree. Another variant discussed, the Write-Once B-tree (WOBT), takes advantage of new storage mediums. The Linked-B-tree (LB tree) can detect and correct mistakes.

Research work on concurrency issues related to the B-tree has increased in recent years. Comer has only one page on the topic of B-trees in a multiuser environment. Here we present more detail on this subject and report on recent advances and theories. Two B-tree variants developed to address concurrency issues are described in the current work,the B-link tree and the Preparatory Operations B-tree (PO-B tree).

The last section of this thesis will be a summary of the old and new variants of the B-tree. Included is material which contrasts and compares the variants to each other and to the basic B-tree, and a summary of what performance features of the original B-tree each variant claims to improve upon. It is hoped that a reader of Volume II will gain an appreciation for the variety of application areas the "simple" B-tree structure has been adapted to and will perhaps begin to imagine a new variant suited to some application area of interest.

1.3 Methodology

The extensive bibliography at the end of Comer's paper cites articles published as late as 1978. Therefore, a computer search was conducted to locate publications pertaining to B-trees from 1977 through 1986. Journal articles are the primary source of information for determining recent advances, research directions and B-tree variants. Recent for this research means since Comer's B-tree article. A summary of recent developments directly follows the description of the basic B-tree in Chapter 2.

The general format for presenting the B-tree variants will be to define the new structure, describe differences in the search, insert and delete algorithms from those of a basic B-tree, and then to summarize advantages and disadvantages of the variant. For most, a sample tree will be shown to help illustrate the structure and how it differs from the pure B-tree.

1.4 Summary

This section is included to provide an overview of the variants to be discussed. A short description of each follows.

B^*-tree

The B^*-tree is a variant which improves storage utilization. Where nodes of a B-tree are guaranteed to be at least half full, the B^*-tree's nodes are always at least two-thirds full. This is accomplished by a minor modification to the insertion algorithm which delays splitting nodes.

B^+-tree

The most common B-tree variant, the B^+-tree,allows for efficient sequential access to the keys of the tree. All keys are stored at the leaf level and are linked together by pointers as a sequential set. Some key values are also stored in the non-terminal levels where they serve as index values to facilitate traversing the tree to the leaf level. Many variants discussed in this paper are actually variants of the B^+-tree.

Prefix B-tree

The prefix B-tree is a variant of the B^+-tree which increases the number of entries per non-leaf node by using variable length prefixes of keys instead of actual key values. Full key values are required only at the leaf level, so entries in non-leaf nodes act only as search separators and can therefore be modified. This change increases the branching factor of the tree, reducing the height of the tree and thereby improving the retrieval time for keys.

Compact B-tree

Compact trees are a space optimal variant of the B-tree. A compact B-tree has the minimal number of nodes based on the total number of keys and the order of the tree. Compacting is done from the leaf level up to the root during the periodic reorganization required, as the tree does not dynamically reorganize to stay compact after insertions and deletions. The advantages are the high initial storage utilization and the low search costs.

Dense Multiway Tree

Dense multiway trees (DMT) are a B-tree variant which attempt to improve storage utilization by setting a parameter for the density of keys per node. Nodes of DMT's are split only after all other nodes on the same level are found to be full. Otherwise, shifting occurs and keys are redistributed among brother nodes. Unfortunately, this structure also fails to dynamically reorganize itself to preserve the proper denseness of nodes without costly insertion and deletion algorithms.

H-Trees

An H-tree is a B-tree with two additional parameters. The first parameter is the minimum number of grandsons, G, each non-root, non-leaf node must have. For large order trees, this value plays an important role in determining the height of the tree. The second parameter is the minimum number of leaves, D, per bottom node of the tree. This value effects storage utilization. As the user's needs change, these parameters can be altered to change the characteristics of the tree. Unlike the compact tree, insertions and deletions can be done with reasonable cost to preserve the H-tree structure.

Multi-dimensional B-Trees

Multi-dimensional B-Trees (MDBT) are designed to handle queries that specify more than one attribute (ie, multi-key retrieval). The MDBT is an hierarchy of B-trees with each level of the hierarchy corresponding to a different attribute of the data. In order to facilitate range searches and partial matches, additional pointers between levels, as well as in each level are required. All pointers to data records are found in the bottom level of the MDBT. The structure is dynamically defined.

R-Trees

R-Trees are a B^+-tree variant designed to efficiently handle spatial data in a pictorial database. Data objects can be retrieved based on their spatial location instead of on an alphanumeric encoding scheme. A pictorial query language, PSQL, is used to formulate queries. Associations between pictorial data and alphanumeric data is also facilitated. Pictorial databases are used in geo-data and computer aided design applications.

Write-Once B-Trees

The Write-Once B-Tree (WOBT) is a variant which preserves all old values of the data. Instead of erasing or overwriting data to update the database, new information is included in pages until an overflow occurs. At this point, only the most current data is transfered to a new page. The old page is not destroyed. Insertions are made on appropriate pages in the next available empty slot. Sorting occurs later when the data is transfered to new pages. This B^+-tree variant would be used when a high-density storage medium is available and when a built-in audit trail of historical information is important to have readily available. The cost of accessing keys in the WOBT is proportional to the logarithm of the number of current records in the tree. It is also possible to as easily access the database as it was at some previous point in time.

Linked-B Tree

The Linked-B Tree (LB tree) is a variant of the B^+-tree that can detect and correct some combinations of errors in the index portion of the tree. This is facilitated by additional header nodes at each level of the tree and many additional pointers that chain and thread nodes together. By performing local error detection algorithms, errors in pointers from one level to the next in the LB tree can be found and corrected. The cost of the detection algorithm is high, but in a system that has a low fault tolerance, the LB-tree is a structure to consider.

B-Link Tree

This B-tree variant is designed with concurrency control issues in mind. In order to require fewer locks on nodes in a B-tree in a multi-user environment, the B-link tree has additional link pointers joining nodes at a given level with their right brothers. These link pointers are essential in the situation when one user is trying to locate a key value in a node which has since been split by the actions of another user. When nodes split, some key values are redistributed to the newly created right brother node which can now be reached with the link pointer. Without this pointer, unsafe nodes that might split must be locked.

PO-B Tree

The preparatory operations B-tree (PO-B tree) is another B^+-tree variant designed to improve the level of concurrency allowed by reducing the number of locked nodes required. The strategy for this tree is to prematurely split unsafe nodes along an insertion search path (and similarly prematurely concatenate unsafe nodes along deletion paths). Unsafe nodes are ones in which one more key being inserted (or deleted) would cause the node to split (or be concatenated). Normally, these nodes and their descendants would have to be locked until the next safe node is encountered on the search path. This is because splits and concatenations can not propogate up the tree past a safe node. Only the nodes actually being changed must be locked with this B-tree variant.

Chapter #2 - The Basic B-Tree

2.1 History and Definiton

The B-tree was created by R. Bayer and E. McCreight at the Boeing Scientific Research Labs in the early 1970's in an effort to find an efficient, dynamic, external indexing method for large random access files. The "B" in the term B-tree was never defined by its creators, although others have thought it to stand for Bayer, Boeing or balanced.

A B-tree of order d is defined as a tree with the following properties:

1) Each nonroot node of the tree contains at least d keys and no more than 2d key values.
2) The root node has at least 1 and no more than 2d keys.
3) The leaf nodes are all at the same level of the tree and have no pointers.
4) A non-leaf node with s keys will have (s+1) pointers.
5) Keys in each node are sequentially ordered.

The first condition ensures that the B-tree is always at least half full, not counting the root node. The second condition allows the root node to be less full than other nodes to help delay splitting the root node which would increase the height of the tree. The path length from the root to all leaves of the B-tree are equal because of the uniform height enforced by condition three.

The fourth condition guarantees each node will always have one more subtree pointer than keys. If the node is thought to represent a range of possible key values, specifing "s" of the key values results in dividing the range into (s+1) subintervals. Each subinterval will contain a pointer to a node at the next lower level of the tree covering the range of values of the subinterval. Thus the natural ordering of the keys is preserved which facilitates range searching and sequential processing. This leads to conceptualizing the key values as separators. See Figure 1 for the structure of a typical node.

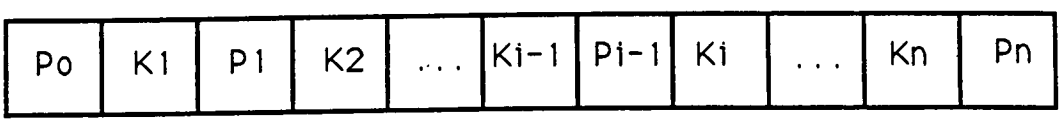


Figure 1 Typical Node of a B-Tree

P_i = pointer to i^{th} subtree, K_i = key values, $d \leq i \leq 2d$

There is not universal agreement over the meaning of the order of a B-tree. Order is defined in one of two ways. Bayer and McCreight refer to order, d, as being the minimum number of keys found in a non-leaf, non-root node. This notation facilitates using fewer fractions to represent many of the formulas and subscripts associated with B-trees.

Many others prefer to think of order, m, as the maximum number of subtrees a node is allowed to have. This definition follows from viewing the B-tree as a special case of a m-ary multiway tree with a maximum of m branches allowed at each level. An advantage of this definition of order is that it allows the number of subtrees of each node to be either even or odd. With the first definition, the number of subtrees is always odd (2d+1). Based on this alternate definition of order:

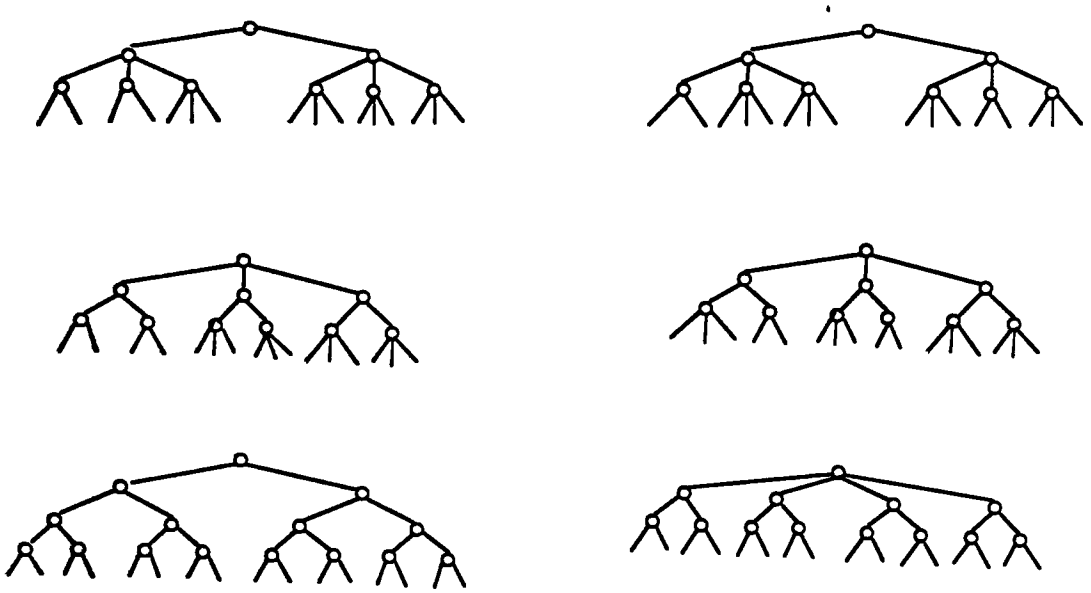
1) Each node contains at least $\lceil m/2 \rceil$ and no more than m pointers, except the root and leaf nodes.
2) Each node contains at least $\lceil m/2 \rceil - 1$ and no more than (m-1) keys, except the root node.

In this research both definitions of order are used, depending on which makes the notation and examples easier to understand. Whenever order means the minimum number of keys per node, the "d" notation is used. Using "m" implies that order is the maximum number of pointers per node.

The size of the B-tree nodes are determined by the order of the tree. Since B-trees are designed to be an indexing method for external file structures, it is generally assumed that the node size corresponds to the physical page size of secondary storage. Each node accessed (with the possible exception of the root node) must physically be read into main memory.

Even with seemingly many rules, the B-tree is underconstrained as it allows for many legal but structurally distinct B-trees of a given order and number of keys. See Figure 2 for six distinct B-trees with 15 keys of order m = 4. This allows flexibility in the dynamic reorganization to maintain the essential properties of B-trees.

Figure 2 Six Distinct B-Trees of Order m = 4 with 15 keys



For interested readers, the find, insert and delete algorithms and their related costs are presented in Appendix A.2, along with information on B-tree storage utilization.

2.2 Advantages over other Indexing Methods

The B-tree made numerous improvements over previous indexing methods available for large, random access files. One major advantage is the dynamic definition of the B-tree, which ensures the tree efficiently maintains its balance and occupancy properties in environments with multiple insertions and deletions. This is possible since reorganization occurs as needed from the bottom of the tree, up in a single path which only in the worst cases reaches the root. These changes are made at relatively low cost, as there is no opportunity for runaway overhead. Retrevials and updates are of complexity O(log N), where N is the number of keys in the tree.

Previous indexing methods, such as inverted files were statically defined. This necessitates expensive periodic reorganizations of the index as updates to the index cause performance degradation. In contrast, B-trees dynamically allocate and release storage as needed while preserving the structure of the tree. Therefore, there is no degradation of performance even if storage utilization is high.

B-trees preserve the natural order of the keys which facilitates range searching and sequential processing. This makes B-trees preferable to hashing techniques when the retrieving of records must be done in order of key values. In general, B-tree average access times are competitive with most other methods. The ordering of the keys also contributes to low-cost, high performance updating, since the prior and next keys are immediately available. This facilities the redistribution of the keys.

The only B-tree parameter relating the index to the physical device level is the order of the tree. This makes the B-tree a very flexible structure as the node size is easily adapted to the page size for input and output operations by specifing the order. Thus hardware requirements can easily be met without changing the B-tree algorithms.

When indexes reside in secondary storage it is important to minimize the number of disk accesses made. B-trees tend to be very short and bushy, especially for high orders, which means even worst-case searches for keys involve low numbers of nodes read in from secondary store. Find operations in B-trees never exceed $\log_d N+1$ while unbalanced trees may require N nodes being visited. (N is the number of keys in the tree and d is the minimum number of keys per node.)

In addition, because B-trees dynamically allocate and release storage as needed the number of disk accesses to retrieve a specific record depends only on the number of records currently in the file, not on the initial file size.

Other trees such as an optimal m-way search tree for a given collection of keys may have shorter search path lengths than B-trees of the same keys. However, the B-tree is still the better choice as it is easier and less costly to maintain.

2.3 Disadvantages

Disadvantages are few for the B-tree. The major drawbacks have all been overcome by variants described in later sections. Sequential searching is not done efficiently in the standard B-tree. This problem has been addressed in the design of the B^+-tree (see Section 2.6). The worst case storage utilization of 50% has been improved significantly with a simple modification of the insertion algorithm as described in the B^*-tree section (2.5).

2.4 Recent developments

2.4.1 Overview

Recent research trends with B-trees fall into four categories; concurrency issues, performance evaluations, the development of variants and advances due to new technology. Concurrency issues deal with establishing locking protocals for B-trees used in multi-user environments. These issues are discussed in detail in Chapter 6.

Performance evaluations fall into two categories; analytic and empirical. Although many aspects of the performance behavior of B-trees are considered well known and established, new analytic techniques and models have been developed in recent years to aid in obtaining a better understanding of the behavior. Often empirical testing is done to confirm analytic results. We will review three recent research efforts that fall in this category in Section 2.4.2. A fourth example of recent research attempting to improve performance of B-trees concerns finding an optimal page replacement strategy for main memory when the B-tree is stored in secondary memory. This work is described in section 2.4.4.

The third research trend is the development of B-tree variants. Some variants maintain the basic B-tree structure but use optimizing algorithms. An example of this type of research work will be discussed in Section 2.4.3.

Other variants add new constraints to the basic B-tree and thereby change the data structure. For example, the Compact B-tree and the dense multiway tree (DMT) add density contraints to improve on storage utilization. The H-tree has the additional constraints of number of grandsons a node must have and the minimum number of keys in the leaf nodes. The constraints necessitate the development of new constructing and updating algorithms. Analytic work to support the claims of the properties these variants display is an important part of the development of the new structures. Recent work with this type of variant is discussed in Chapters 3-5.

Advances in technology have openned new avenues for research with the B-tree. The Write-Once-B-tree (discussed in Chapter 5) represents a variant that takes advantage of new, dense storage mediums which are cheap enough not to require overwrites. Other research (discussed in section 2.4.5) focuses on implementing the basic operations of the B-tree using hardware instead of software.

2.4.2 Performance Evaluations

Recent research efforts by W. Wright at Southern Illinois University at Carbondale have centered on the development of a new analytic technique for assessing average performance measures of B-trees. He specifically studied B-trees with large orders and derived formulas for average storage utilization, height, number of splits per insertion, total number of nodes of each size and number of accesses for retrieval or insertions.

First, Wright derived a formula to determine the expected number of nodes of each possible size at the bottom level of the tree. From that, he formulated the probability of an insertion being made into bottom nodes of each size. If the node is of maximum size then a split will occur. Therefore, the probability of an insertion made into a full node is equivalent to the probability of a split occurring.

Wright distinquishes between a node insertion which adds a new key to the tree without causing a split versus a tree insertion which requires a new node be added to the tree. He shows that for B-trees with large orders and large numbers of keys, the average number of splits per tree insertion is 1/(2d*ln2) or approximately 72% as large as the worst case 1/d . The order d represents the minimum number of keys per node.

Based on the expected number of nodes of each size at the bottom level of the tree, he also shows the average storage utilization for the B-tree to be ln2, as is well known. Storage utilization at the bottom level is shown to have a very small range of possible values , remaining between 2/3 (.6667) and ln2 (.69315). The average height of the tree depends on the average storage. Wright derives formulas which show average height to be logarithmic with base ((2d*ln2)+1).

All the theoretical work presented was based on the formulas derived for the number of nodes of each size at the bottom level of the B-tree. In order to verify the accuracy of these formulas, empirical tests were run using 100 B-trees generated by inserting 20,000 random key values into an initially empty tree. Counting the various sized nodes at the bottom level demonstrated the derived formulas to be very accurate.

Much B-tree research has also been conducted in West Germany. Analytic methods for determining utilization and path length has been developed by K. Quitzow and M. Klopprogge at the University of Karlsruhe. They created a deterministic model in the form of a state vector and a system of differential equations describing the behavior of the state changes.

They assume that the keys are uniformily distributed between some minimum and maximum values. For insertions they assume all

intervals are equally likely and for deletion all keys are candidates with equal probability. Their model is sufficiently flexible to be used to evaluate B^*-trees as well as B-trees by inserting or omitting terms in the differential equations relating to overflow treatment.

Two cases were considered; trees where only insertions were performed and trees with an equal number of insertions and deletions. For each of these cases storage and path lengths where derived based on B-trees and B^*-trees (allowing for overflow without splitting).

They present data showing the results based on different page sizes, L, where L is the maximum number of keys allowed per page. Under pure insertion (versus insertion and deletion), storage utilization was higher and path lengths shorter for all trees at all page sizes. Specifically when L=100, $N=10^6$ the B-tree storage was 69.2% vs 59.2% and path length was 3.24 vs 3.35. For the B^*-tree storage was 91.8% vs 82.6% and path length was 3.04 vs 3.11. Note the B^* tree outperforms the standard B-tree for all measures.

A third example of a new analytical tool for B-tree analysis is based on work done by C. Leung from London University. He offers a delightfully simple and intuitive derivation of the well known average storage utilization of B-trees. The technique is general enough to be used for determining storage for some variants of B-trees.

Leung calculates the expected storage utilization by assuming the distribution of the number of nodes in the tree, n, can be approximated by a continuous rectangular distribution over the interval defined by the minimum and maximum number of nodes in the tree. To find the expected value of n, E(n), requires only elementary integration.

Leung's technique can be used to calculate the average storage of any B-tree variant based on a minimum fullness factor, f. For the standard B-tree, f is .50 since all nodes are required to be at least half full, while for B^*-trees f is set to .67. The value of f is used to find the maximum number of nodes possible in the tree, a value needed to determine the interval to calculate E(n) on. (See Appendix A.1 for the mathematical details.)

Calculations using Leung's formula verify ln2 is the average B-tree storage utilization. For the B^*-tree the value is 81%. If a B-tree was defined such that three brothers had to be full before a split could occur, the storage would be 86%, based on a fullness factor f = .75. Unfortunately, the overhead to transfer keys amoung three brothers would be much higher than the B^*-tree which only uses two brothers.

2.4.3 Memory Management For B-trees

A new memory management strategy to keep retrieval costs in B-trees low has been proposed by Spirn and Tsur. They measure the cost of key retrieval by the number of page faults that occur, instead of the number of pages accessed.

The page replacement policy they recommend is called LAP (least access probability). The page of the tree currently in main memory with the least access probability will be swapped out to allow another page to be copied in from secondary store.

The access probability of a node i, is the sum of the target probabilities of each node in the subtree of i. The target probability of a node is the probability that the key is found in the node. The root nodes access probability is always the maximum value of one. Thus, the root node will never be replaced. Nodes of the B-tree closer to the leaf level will have the lowest access probability.

This page replacement policy assumes that the sequence of key values being retrieved are independent and identically distributed from some set of possible key values. The well known LRU (least recently used) page replacement algorithm has an advantage over LAP only when there is a dependence between successive key retrievals.

LAP is an example of a priority paging algorithm which choses pages based on a priority list. In this case, the list consists of access probabilities. Such types of algorithms are believed to produce the lowest expected page fault rate. Spirn and Tsur analytically show that using the access probabilities to generate the priority list is not optimal, but is very close to the optimal achieved using an algorithm called LEC. LEC (least expensive cost) uses a priority list derived from a cost function.

Spirn and Tsur also simulated the two paging algorithms (LAP and LEC) and found their performance to be almost identical. They therefore recommend the more easily implemented LAP for B-trees that are not volatile. The overhead of maintaining the access probability information for nodes when frequent insertions or deletions are made outweighs the savings in cost for retrievals in databases that are volatile.

2.4.4 Optimization Variant

G. Diehr and B. Faaland at the University of Washington in Seattle have recently developed algorithms for the optimal organization of B-trees with variable length items. Their work is a continuation of earlier optimization work done by E. McCreight. In [McCR] McCreight presented results from using two different strategies for determining the pagination of B-trees; keeping pages approximately equal in size versus minimizing the sum of the key lengths of boundary items (items that get promoted to be separators used at higher levels when pages split). The theory is that if you promote shorter items toward the root and keep longer items toward the leaves the resulting tree will be shorter, thus resulting in fewer accesses.

McCreight posed the question of whether an algorithm exists that could "quickly" chose the boundary items whose sum of lengths is minimial such that the pagination is feasible. Diehr and Faaland found an algorithm that is O(NlogN) for doing this and also an algorithm $O(N^3 logN)$ to find the minimal depth tree.

In terms of practical applications, if the files being indexed were fairly static then the expense to optimize the B-tree using this global optimization algorithm might be worthwhile. Dynamic files would be better served by local optimization algorithms. For example, B^+ trees tend to be more static than B-trees as deletion of keys does not require their removal from the index since the actual keys are found at the lowest level. Thus certain B^+ trees might be worth optimizing using this technique.

2.9.5 B-tree Parallel Processor

Miller and Hurson [MILL] propose the architecture for a hardware system that implements the basic operations of search, insert and delete for B-trees. This design is made possible because of specific B-tree features. These features include the recursive simplicity of the basic algorithms and the dynamic nature of the B-tree that guarantees small heights which bound the search algorithm. Hardware replacing software results in the advantages of greater reliability and efficiency, increased speed and the freeing up of the main frame processor from operations on the indices of the tree.

The architectural structure consists of a controller and an array of identical and independent processors. The only supervision is by a front end processor which issues the query operation to the controller. The controller then communicates with and coordinates the actual implementation done by the array of processors.

The search for a key value in the B-tree is conducted in the following way. Each processor receives a copy of the search key

and the address of the root node from the controller. The processors simultaneously access the node, each accessing a different key value and pointer pair from the node. (It is assumed that the number of processors is greater than or equal to the maximum number of keys per node.)

If the search key is found, the controller is interupted by the processor that found it. Otherwise, each processor sets a binary flag. The flag's value is based on the comparison between the search key and the current key value (read into the processor). If the search key is greater than the key value in the processor, the flag is set to one. From this string of ones followed by zeros sent back to the controller, the processor containing the address (pointer) of the next node in the B-tree to search can easily be identified. This new address is then broadcast to the array of processors and the search continues recursively. An unsuccessful search is indicated when a nil pointer is returned to the controller.

For insertions, a search for the proper leaf node into which the new key will be placed is conducted. The controller maintains a stack of addresses of nodes on the search path which are used only if splits up the tree are required. Processors can shift their contents to left or right neighbors to make room in the leaf node in the proper position for the new key. (The position for insertion is determined by the strings of ones and zeros generated when searching a node.) The need to split a node can be detected by comparing the number of processors used to the order of the tree. If a split is required, the parent's address is taken off the controller's stack and the middle key is inserted into the parent node using the same procedure outlined above.

Hurson and Miller do not implement the usual deletion algorithm because of its complexity. Instead, they introduce a new deletion algorithm (the swap and mark algorithm) that they claim will be efficient as long as the number of insertions and deletions in the tree are uniform with respect to the range of key values.

The basic idea is to avoid underflow in non-leaf nodes which would require either rotation of key values or concatenation. They also want to avoid keys marked as deleted (but still occupying space) in the upper levels of the tree for space utilization and performance considerations.

They propose swapping the deleted key with the next larger key in the tree which for B-trees is always found at the leaf level. The deleted key will therefore always end up at the leaf level, marked as deleted, in a position where an insertion of a new key value can eventually replace it.

Hurson and Miller analytically evaluate the performance of their proposed system versus the usual software implementation and conclude that by a factor of m (the order of the tree), the proposed system had a better response time. Better execution time results in part from the parallelism amoung the processors in executing operations, and from the concurrency facilitated between the operations and the mainframe processor.

2.5 B^* Tree

2.5.1 Definition and Algorithms

Bayer and McCreight in their classic paper "Organization and Maintenence of Large Ordered Indexes" described an unnamed variant of the B-tree with improved storage utilization. This tree was later more formally defined by Knuth [KNUT] and named the B^*-tree. (Some refer to another B-tree variant, one with all keys in the leaf nodes, as a B^*-tree. Here, we use the name B^+ for that variant.)

The B^*-tree results from using an overflow technique to delay expensive page splitting during key insertions. Inserting a key into a node that is already full results in a split for a B-tree. For a B^*-tree, the fullness of adjacent brother nodes is considered. If there is room for the key in one of the two sibling nodes, then keys are locally redistributed and a new node is not needed. This guarantees that each node is at least 2/3 full as a split causes two full nodes to be redistributed to three nodes. Splits will be delayed until adjacent brother nodes are full.

Knuth defines this "superior breed of tree, say a B^*-tree of order m", as follows:

1. Every node except the root has at most m sons.
2. Every node, except the root and leaves, has at least $\lceil(2m-1)/3\rceil$ sons.
3. The root has at least 2 and at most $2\lfloor(2m-2)/3\rfloor+1$ sons.
4. All leaves appear at the same level.
5. A nonleaf node with k sons contains k-1 keys.

Notice that condition two requires that each node is at least 2/3 full. The root is allowed to have more than m sons by condition three since it is the only node without a brother. When the root splits it will produce two nodes, each approximately two-thirds full. For other non-root nodes, when two full brothers split into three nodes, the number of keys in the nodes are $\lceil(2m-2)/3\rceil$, $\lceil(2m-1)/3\rceil$, and $\lceil 2m/3\rceil$.

2.5.2 Advantages over B-trees

Worst case storage for B^*-trees (66%) is close to the average

case storage for B-trees (69%). Average storage for B*-trees is approximately 85%, a considerable improvement for such minor changes in the insertion algorithm. Improving storage utilization decreases the height of the tree for a given number of keys. The smaller height favorably effects retrieval time.

The cost of insertion using the overflow method is only marginally higher. For B*-trees, an average of 2 + 2/d more fetches and 2 more writes are required for a single key insertion when compared to the B-tree.

Other B-tree variants try to improve on storage utilization but have very expensive insertion/deletion algorithms to maintain the compactness of storage. The B*-tree efficiently retains the minimum two-thirds storage of each node using a slight variation of the standard B-tree algorithm for insertion.

2.6 B^+ Tree

2.6.1 Definition

The B^+-tree is an early variant of the basic B-tree which allows for efficient sequential searching for keys. Knuth presented the basic definition for the B^+-tree in [KNUT] in 1972. Work in the early seventies was carried out by researchers at IBM who use the B^+-tree as the basis for its commercial VSAM database system.

A B^+-tree has two parts; the index part and the sequence set. The index part consists of the interior nodes of the tree which contain routing information to the actual keys found in the sequence set. The sequence set is the leaf nodes of the tree which have additional links from left to right joining the leaves in sequential order.

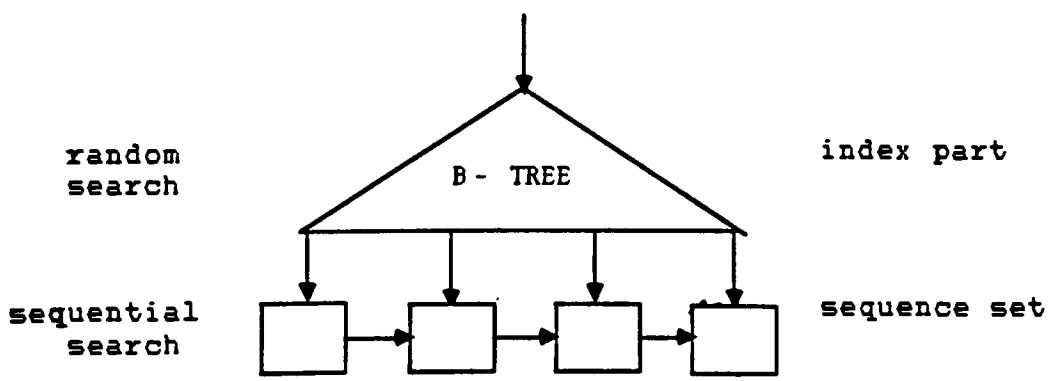


Figure 3: A B^+-Tree

The index part is structurally a B-tree. The index nodes, however, contain less information than the nodes of a B-tree. Instead of storing pointers, key values and either actual records or pointers to records corresponding to the key values, a node in a B^+-tree only contains pointers and separator values which may or may not be key values. All key values and actual records or pointers to records for the key values reside in the leaf nodes in the sequence set.

2.6.2 Differences in Algorithms

Inserting a new key value into a B^+-tree is similar to key insertion for a B-tree if no splits are required. In both cases, the insertion occurs at the leaf level of the trees.

If a split is required in a B^+-tree, a copy of the middle key of the node to be split is used as a separator instead of the actual key. In the example below, the key value of 16 is inserted into a B^+-tree. Note the duplication of the median key value of 15 at the leaf level.

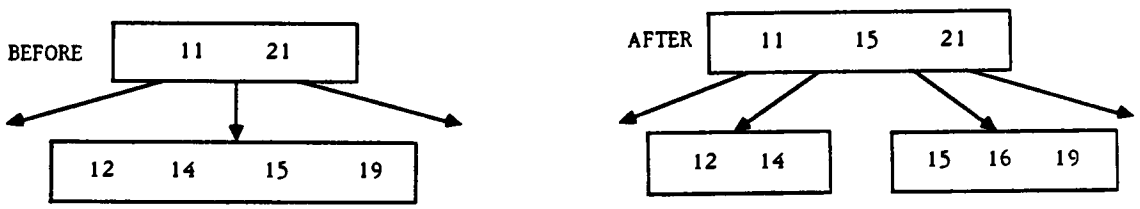


Figure 4: Insertion of 16 into a B^+-Tree of Order d=2

A similar insertion in a B-tree would be as follows. Notice the lack of duplication of the key value 15.

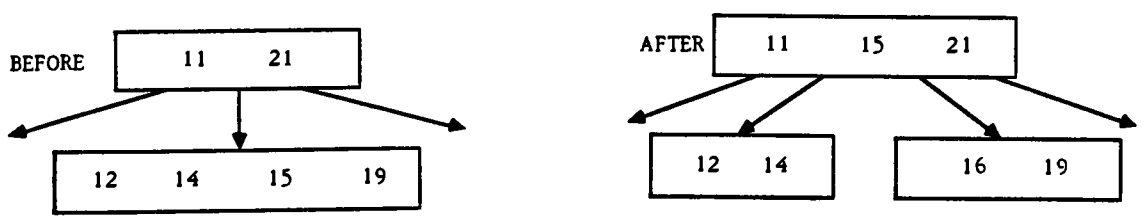


Figure 5: Insertion of 16 into a B-Tree of Order d=2

Deletion is actually easier in a B^+-tree than in a B-tree. Since all keys are stored at the leaf level, all deletions can take place at that level. A copy of the deleted key in the index

part can remain to act as a separator. Concatenation and reorganization is necessary only when the node in the sequence set is less than half full after a deletion. In this case, the copy of the deleted key found in the lowest level of the index part of the B^+-tree will also be deleted.

The find algorithm for a B^+-tree will **always** end at the sequence set level, unlike the B-tree which could end at an interior node. Since key values are found as separators in the index part of the B^+-tree, a "match" could be made at an interior node. However, to reach the information about the key value, one must continue to follow the nearest right pointers til the leaf level is reached.

2.6.3 Advantages over B-trees

For a standard B-tree there is no efficient "next" algorithm. The next sequential key value is found in the leftmost leaf of the right subtree of the previous key. To reach it may require many expensive accesses of nodes in secondary storage, resulting in a cost of $\log_d N$. For a B^+-tree, only one additional read is required since the keys are linked sequentially at the leaf level. The storage requirement in main memory for the "next" operation is therefore only one node.

Keys in a B^+-tree can be efficiently accessed both directly and sequentially. Alternative sequential indexing methods do not offer the dynamic allocation and release of storage, nor the guaranteed 50% storage utilization that a B^+-tree does. Expensive, periodic reorganization of the index part is never required for the B^+-tree even after numerous insertions and deletions.

Since only keys and pointers are stored in nodes in the index portion of the B^+-tree (not the entire records), more keys can be stored in a node of a fixed size. This increases the branching factor of the tree, reducing the height. The resulting shorter paths to the leaves will reduce search time as fewer seeks will be required to reach the keys in the leaves.

The independence between the index and sequence set allows for the compression of key values, unlike in a B-tree. The values in the nodes of the index portion only serve as separators. The correct uncompressed key values can be found in the leaves. This alternative is the prefix-B tree, discussed in the next section.

The fact that all searches for keys terminate at the leaf level, unlike for a B-tree where a key could be found from the root level on down, may seem like a large disadvantage. However, the higher branching factor in the index nodes reduces the heights of the B^+-trees. Thus, the efficiency of direct, random

searches are approximately the same for B and B^+-trees.

2.6.4 Prefix-B trees

2.6.4.1 Definition and Algorithms

Prefix B-trees, developed by Bayer and Unterauer [BAYEa] in the mid 1970's, are a variant of the B^+-tree. Like the B^+-tree, prefix B-trees consist of a B-tree index part and a linked list of leaves called the sequence set. Unlike the B^+-tree, nodes in the index part contain carefully selected prefixes of minimum length instead of entire key values. The separators are therefore of variable length. There are two types of such trees: simple prefix B-trees and prefix B-trees. In the latter type, the prefixes are reconstructed as the tree is searched, thus necessitating the storage of only partial prefixes in the index nodes.

When picking the shortest separator between similar keys such as mechanism and mechanic, little space would be saved by using the prefix "mechani". This is a common problem since keys in practical applications frequently have many terms that vary only in the last few letters. In situations like this, Bayer and Unterauer suggest "scanning a small neighborhood of keys" to obtain a good separating value. This neighborhood is refered to as the "split interval around the median key" of the node to be split. The larger the split interval, the smaller the average length of the separators. By not always using the median key as a separator, the node being split may subdivide into two unevenly loaded nodes. This is not a serious problem.

For example, if the leaf node to be split contained the keys machine, mechanic, mechanical, mechanics, mechanism, mule and nail, using "mu" as a separating prefix instead of "mechanics" saves seven characters. The node will be split into nodes with five and two keys. Note that the shortest separator is promoted to the father node only when splitting leaf nodes. When a index node is split, one of the current separators must be used.

The variable length separators in the index nodes makes the search algorithm only a bit more difficult. Chosing a minimum length separator adds overhead to the insertion algorithm too. However, the height of the tree is reduced which lessens the cost of insertion, deletion and retrieval.

In the prefix B-tree the length of the separators are further reduced by pruning off common prefixes which act as a front compression of keys. The common prefix for a node is stored once on the node. The storage of full keys at the leaf level is still desirable to facilitate sequential processing.

2.6.4.2 Advantages over B^+ trees

Using the shortest possible separators maximizes the number of separators per index node thereby increasing the branching factor and reducing the height of the B-tree index. This results in the dual advantage of decreased access time and a saving in storage space.

Bayer and Unterauer empirically tested the performance of B^+, prefix B and simple prefix B-trees. They found comparable computing times for basic insert, delete, and find algorithms for B^+ and simple prefix trees. The prefix B-tree required between 50 and 100% more time.

The number of disk accesses required for all three trees were comparable if the tree had no more than 200 pages. Simple prefix B-trees required 20-25% fewer disk accesses than the B^+ trees for trees having between 400 and 800 pages. Simple prefix B-trees in this situation required only 2 percent more disk accessses than prefix B-trees.

Chapter Three - Storage Oriented B-Tree Variants

3.1 Compact B-trees

3.1.1 Definition

Compact trees are defined as a "space-optimal" variant of the B-tree. This is achieved by having the minimal number of nodes for a given order and number of keys for a B-tree.

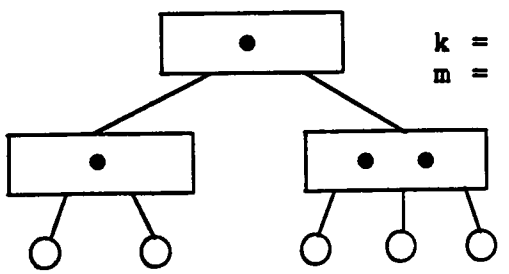
Rosenberg and Snyder [ROSE] define B-trees to be compact if the number of nodes at the i^{th} level of the tree is minimal based on the order of the tree and the number of nodes at the (i+1)st level. In this case, order refers to the maximum number of sons a node can have.

If v_i is the number of nodes at the i^{th} level of the compact tree of order m with k keys and depth d, then:

$$v_i = \lceil v_{i+1} / m \rceil \quad \text{for} \quad 0 \leq i < d$$

$$v_d = k + 1$$

Note that the number of leaves in the tree, v_d, is always one more than the number of keys in the tree. Adherence to the above formulas forces the compaction of the tree to be done from the lower levels up to the root node. See the example illustrated in Figure 8.



k = 4
m = 3

$v_2 = k + 1 = 4 + 1 = 5$

$v_1 = \lceil v_2/m \rceil = \lceil 5/3 \rceil = 2$

$v_0 = \lceil v_1/m \rceil = \lceil 2/3 \rceil = 1$

$d = \lceil \log_m v_d \rceil = \lceil \log_3 5 \rceil = 2$

Figure 6: Compact B-tree of Order 3

3.1.2 Differences in Algorithms

A compact tree is built by an algorithm which maximizes the number of keys per node at the lowest level and then repeats this process at succeeding levels of the tree. This maximizes both the branching factor and key density in the lower levels. This is in contrast to the method of creating a bushy (time-optimal) tree. Time is measured by the average number of nodes accessed to find a key in the tree, so bushy trees use a high branching factor in the nodes nearest the root in order to minimize the length of the path to the keys. Rosenberg and

Snyder point out that it is rare to find a tree that is both space and time optimal. They contend that space-optimal trees are nearly time-optimal, but the reverse is not true.

There is no known efficient insertion algorithm that will preserve the compactness of the tree. Therefore, the normal B-tree insertion (and deletion) algorithm is used and periodic re-compactification is done as part of back-up procedures. Compact trees are very fragile for insertions because their densest nodes are near the bottom leaf nodes where the insertions are done. As a result costly node splits are impossible to avoid.

3.1.3 Advantages

According to [ROSE] the search-time costs are near optimal regardless of the order of the tree for large data sets. Empirical study [ARNO] has shown that compact B-trees have substantially lower search costs than B-trees for low order. For large order trees this advantage of compact trees almost disappears.

Initial storage utilization (before insertions and deletions) is not quite 100% for all compact trees due to some wasted space needed to preserve the equal path lengths to all leaves for B-trees. The work done by Arnow and Tenenbaum [ARNO] showed the storage for compact trees to be 92-100% compared to 67-71% for standard B-trees.

The compact B-tree is a variant of the B-tree that should be considered when the database is non-volatile. If storage considerations are important or if searches greatly outnumber insertions, then compact B-trees are an attractive alternative due to their "space-optimal" and nearly "time-optimal" characteristics.

These advantages lessen with insertions which do not preserve compactness. According to [ROSE] performance remains reasonable if there is less than a 10% insertion rate between compactions of the tree done in conjunction with normal backups.

3.1.4 Disadvantages

The cost of insertions is a major problem for compact trees. An algorithm that would preserve the compactness of the tree is O(N), where N is the number of nodes in the tree. This is a serious disadvantage when B-tree insertion can be done in O(log N).

Empirical studies [ARNO] indicate insertion costs for a sufficiently large compact tree using the standard B-tree insertion algorithm, can be twice as high as with random

B-trees. This insertion inefficiency, relative to B-trees, persists after space advantages disappear. For example, a compact B-tree of order 41 with 5000 keys after a 2% growth will have an insertion cost 20% greater than that of a random B-tree even though storage has fallen to 64% (versus 68% for B-trees).

Initial storage utilization is not a problem for compact B-trees. However, storage and performance begins to degrade after a modest number of insertions which do not preserve compactness. Storage utilization decreases as the order of the compact tree increases according to [ARNO] and [KLON]. Most of Rosenberg and Snyders' empirical testing was done with compact trees of order 3.

According to [ROSE] for a compact tree of order 150 with 10,000 keys, the storage utilization will be just over 50% after only a 1% insertion rate. These results were analytically derived, based in part on the probability of nodes splitting.

Empirical results in [ARNO] support this finding. Arnow and Tenebaum found that for a given order, the storage utilization of compact B-trees shows a "dampening oscillation" about the average for B-trees of 69%. For a compact tree of order 41 with 5000 keys, a 1.4% insertion rate cancelled the initial storage advantage of the compact tree over a B-tree. With a 2% insertion rate, the compact tree was notably worse.

As indicated above, several problems must be overcome before compact B-trees will gain wide usage. Without efficient insertion and reorganization algorithms, compact trees cannot be considered as an option for files which dynamically shrink and grow. Static files are not as common as volatile ones. Even [ROSE] admit that the compact trees are only "modestly robust". Empirical evidence [ARNO] shows that performance and storage utilization degrades at a faster rate even with low insertion rates than first thought.

There are also problems with relying on periodic recompactification. As pointed out by Klonk in [KLON], the algorithm to reestablish space-optimality is an "in-place" reorganization of the file. To preserve the security of the file a backup would first be required. This makes the execution of this algorithm "during" the daily backup to be not as convenient as it first sounds.

Klonk also raises the question of the practicality and relevance of studying the problem of increasing storage utilization. He points out that using the simple overflow node-splitting algorithm for B-trees (resulting in the B^*-tree), space utilization of 85% can be expected. The potential for saving space is at most 15%.

3.2 H-trees

3.2.1 Basic Definition

While doing a PhD disertation at the University of Texas at Austin, S. Huang developed a restricted subclass of B-trees named H(beta, gamma, delta) trees. For ease of notation in this paper the notation H(B,G,D) will be used.

By definition, a Height-balanced tree (H-tree) of order beta (B), gamma (G), delta (D) is a B-tree of order beta (B) such that:

1) Every node, except the root and nodes at the bottom level, must have at least gamma (G) grandsons.
2) All nodes on the bottom level must have at least delta (D) leaves.

The order, beta (B), refers as usual to the maximum number of sons a node can have. Order also determines the minimum number of sons per node, alpha (A), which Huang assumes to always be B/2.

The parameter G is a relaxation of the brother condition of 2-3 trees. The brother condition requires that if a node has 2 sons, it must have a brother node with 3 sons. Huang instead requires that the parent node has 5 grandsons (G = 5) so more structural flexibility is allowed, especially for trees with larger orders. The possible range of values for G is from A^2 to B^2 inclusive. Since a G value of B^2 implies that all nodes are full, to ensure efficient insertion algorithms G is required by Huang to be less than or equal to $(B^2+1)/2$.

The parameter D plays an important role in the storage utilization of the entire tree as it controlls the number of leaves of the tree. Allowable values for D are $A \leq D \leq B - (B+1)/2$. At the upper limit, D is nearly equal to B. Note that if G and D are set to their minimal values the resulting H-tree is a B-tree of order B.

3.2.2 Differences in Algorithms

Huang presents an algorithm to build an H-tree that is dependent only on the number of keys and the order of the tree. His algorithm ensures that maximal values of the parameters G and D are satisfied. This results in an H-tree that is a subset of H-trees with lower values of G and D.

To preserve the properties of the H-tree, update algorithms more complex than the standard B-tree versions are necessary. Additional information must be contained in each node, specifically variables for the number of sons and grandsons.

Both the insertion and deletion algorithms are divided into two phases. The first phase deals with nodes at the bottom level while the second deals with higher levels. The basic search algorithm for a B-tree is used to locate where the insertion or deletion is to take place.

Three additional operations are needed. SHIFT(C,N,m) moves m sons from node C to a brother node N. PACK(N,m) reduces the number of sons of N to m. MERGE(C,N) will concatenate node C and its adjacent brother N. Pack is the most costly of the three and can sometimes be replaced by a SHIFT and MERGE. Huang claims that the insertion and deletion algorithms for H-trees are exactly those of B-trees when minimum values of D and G are chosen.

The height of the tree effects searching and update algorithms. Huang proves that the parameter G (minimum number of grandsons) dominates the calculation of maximum height for H-trees with a large number of keys N, while the minimal height only depends on B.

The average access cost for H-trees is of complexity h + O(1). Thus the height is used by Huang to represent cost. The worst case cost for insertion and deletion are O(h+B) and O(h*B) respectively. As the values of D and G increase, so does the cost of maintaining the trees, even though the height of the H-tree is decreased.

3.2.3 Advantages

H-trees provide a framework for comparing different classes of B-trees. Huang has shown that H-trees with properly chosen parameters can perform like other trees. He also shows that by varying the parameters, in many cases the H-tree can out perform the other trees.

The B-tree is an $H(B,A^2,A)$ tree, using minimal values of G and D. A $H(B,A^2,(2*B+1)/3)$ tree is a superset of B^*-trees, chosing the value of D to ensure minimum storage utilization of 2/3. The 2-3 brother trees are H(3,5,2) trees which specify a minimum of 5 grandsons. The H-trees with minimal G and maximal D values are the most compatible to both the compact B-tree and the dense multiway tree (DMT).

H-trees achieve a good tradeoff between insert/delete and access/storage costs. Both the compact B-tree and the DMT are designed to optimize storage, but their updating algorithms are O(N). H-trees with good storage (with high values of D and G) still can be maintained with updating algorithms O(B + log(N)).

Storage utilization in H-trees can be controlled by chosing different values of D, the minimal number of leaves for the

bottom level nodes. This is because most keys of the H-tree are on the bottom level. For H-trees storage utilization is at least D/ (B+1). Increasing the value of D improves storage. When D is minimal, storage is close to the 50% of B-trees.

The average storage utilization for a H(3,5,2) tree was computed to be the average of the best (100%) and worst (80%) cases. Huang speculates that although detailed analysis on H-trees of higher order is not possible in general, the average storage utilization of H-trees will be 90% .

The user of the H-tree can define values for the parameters B, G and D that reflect the current requirements of the system. As needs change, the parameter values can be altered. The H-tree will gradually restructure itself as more insertions and deletions are made, thus not requiring an expensive reorganization of the system.

For example, when initially setting up a system, storage is not usually a concern so D can be set small to make insertion and deletion costs lower. Later in the life of the system, as storage space becomes more critical, larger values of D can be used. Similarly the value of G can be modified as the importance of the height of the tree changes.

3.2.6 Disadvantages

H-trees are a relatively new structure. The only literature to date is written by its creator, S. Huang. Therefore, disadvantages have not been discussed in the literature yet.

Huang is very optimistic about H-trees. He glosses over the complexities of the updating algorithms for large values of G and D. It still remains to be seen how easily his algorithms are implemented. They are definitely more complex to understand than the easy, efficient insertion/deletion algorithms for the basic B-tree.

Huang asserts that the H-tree provides a framework for comparing various trees. It appears that he overstates the ability to pick parameter values such that the H-tree will behave like other trees. Similiarly, the framework seems limited for future use, as there are many other variants for which it is not appropriate.

3.3 Dense Multiway Trees

3.3.1 Basic Definition

Strictly speaking, a dense multiway tree (DMT or dense m-ary tree) is not a B-tree although there are many similarities. Both are multiway trees with all leaves at the same depth and both maintain keys in ordered form in nodes which contain one more pointer than keys. Both types of trees have a maximum number of keys possible per node which is one less than the order of the trees. The differences arise because of the density factor, r that is specified for dense m-ary trees. In order to guarantee a certain density, the tree is reorganized in such a way that it is possible for a node of the tree to contain one pointer and no key values. This violates the B-tree rule that ensures a minimum number of keys per node based on the order of the tree.

More formally, according to the creators; Culik, Ottman and Wood [CULI] a m-ary tree T of order m is said to be r-dense, where r is a natural number $1 \leq r \leq (m-1)$ if and only if all of the following are true:

1) The root of the tree is at least binary.
2) Each unsaturated node different from the root has either
   a) only saturated brothers and at least one such brother or
   b) at least r saturated brothers.
3) All leaves have the same depth.

A saturated node has the maximum allowable sons, m, the order of the tree.

If the density factor is the minimum (r=1) the tree is called weakly dense. If r= m-1 the tree is refered to as a strongly dense multiway tree. Since (m-1) equals 1 for binary trees, there is only one class of m-ary trees with density r = 1. For ternary trees (m=3) two classes exist, 2-dense (strongly dense) and 1-dense (weakly dense). See Figure 7. Notice that in the 2-dense tree each unsaturated brother has either all saturated brothers or at least two; while in the 1-dense tree each unsaturated brother has at least one saturated brother.

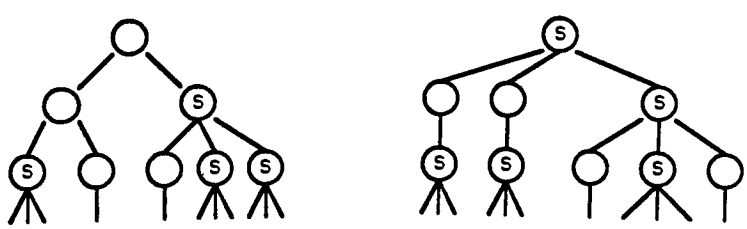


Figure 7: Strongly (r=2) and Weakly (r=1) Dense MDTs

The above is a static definition of r-dense m-ary trees. Static definitions require expensive reorganization in response to insertions and deletions while dynamic definitons do not. A dynamic definition of the r-dense m-ary tree is that a tree is dense if it is either the empty tree of height one with no keys or if it is obtained by one further insertion into a dense tree. Every tree that is thus dynamically defined is at least 1-dense (weakly dense) but the converse is not true. According to the creators of the DMT, the precise relationship between the dynamic and static class is still an open problem.

3.3.2 Differences in Algorithms

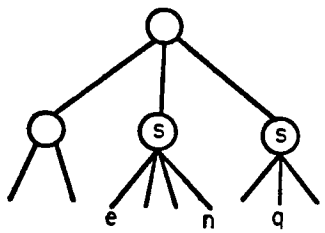
A slight modification of the B-tree retrieval algorithm can be used for dense m-ary trees (DMT). It is possible for a DMT to have unary nodes with only one son. Since the number of keys is one less than the number of sons, this requires that no keys be stored in these nodes. The find algorithm of B-trees must therefore be modified to allow for the possiblity of finding zero keys in a node.

As with a B-tree, inserting a key into a DMT first uses the Find algorithm to locate the proper position of the new leaf. If the parent of the new leaf is unsaturated, the insertion can be done immediately. If the parent p is saturated but has an unsaturated brother b, then a shift to the left (or right depending on the position of b relative to p) is made. If b is to the left of p, it shifts the left-most son from the overfilled node p to its left brother, then the left-most son of that node to its left brother and so on until the unsaturated brother is reached. See the example illustrated in Figure 8.

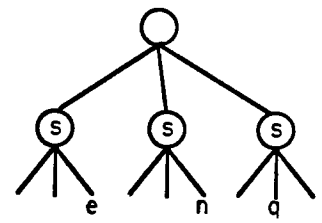
Figure 8: Shifting Nodes in a DMT of Order 3

a) before inserting "q" b) after, shift needed

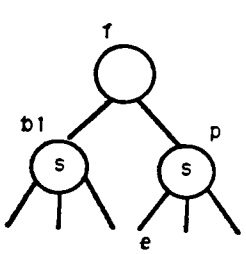
c) "n" is shifted to left



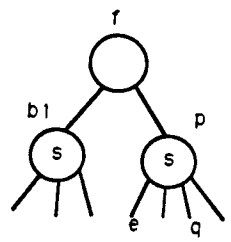
d) "e" is shifted to left



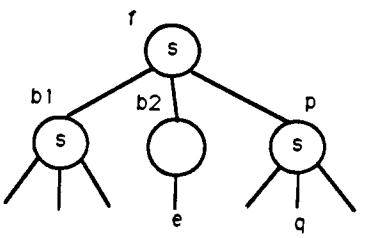
If the saturated parent p has only saturated brothers then the father, f of p is checked. If f is unsaturated then a new son is added to f containing just the left-most son of p. This is demonstrated in the example of Figure 9.



a) before insertion



b) after insertion of q



c) split p and get new son

Figure 9: Shifting Nodes Closer to the DMT Root

If f is saturated a new son is added to f and the problem is recursively moved up one level on the tree. Unsaturated brothers of f are either found or the reconstruction moves up yet another level closer to the root. In the worst case the recursion reaches the root node and causes the height of the tree to increase by one.

Unfortunately, this insertion algorithm does not guarantee that the new tree will be r-dense. Another recursive procedure is used that ensures only saturated nodes are shifted. This recursion and reconstruction moves down the tree and in the worst case reaches the leaves of the tree.

The basic B-tree does not check the degree of fullness of any brothers of nodes to be split. When splits do occur, half the keys are transfered to the new node. For the dense multiway tree the occupancy rate of all brother nodes are considered before splits occur. Only the single right-most or left-most son of a node being split is transfered to the new node. The B-tree and DMT thus represent the two extreme cases for how

splits are managed. Koesler and Ottman [KOES] have developed an insertion algorithm for different classes of DMTs. This (m,b,k) insertion scheme allows for the specification of the number of brothers to be considered,b and the number of keys to be transfered when nodes are split,k. The m refers to the order of the tree. Unfortunately their scheme does not preserve the r-density of the tree either.

Empirical tests were run comparing the DMT and the (m,b,k) insertion schemes using the number of input/output operations as the measurement of time. It was shown [KOES] that the number of i/o operations (excluding searching i/o's) was proportional to the order of the DMT, m with a factor of .6 . Most i/o operations resulted from lateral shifting to unsaturated brothers rather than from the upward or downward restructuring of the tree, especially for orders m > 30.

If the number of keys shifted during a split (k) was half the tree's order (as it is for the B-tree), the number of i/o's was reduced by .6 when compared with the DMT scheme (k=1). The number of brothers considered before splits also has a strong impact on efficiency. When no brothers are considered as in the B-tree (b=0), 1.4 i/o operations per insertion are needed. When all brothers are considered (b = m-1) as with the DMT scheme, this number jumped to 9.0 i/o operations per insertion. These calculations were done with trees of order m = 12.

3.3.3 Advantages

The creation of dense multiway trees was motivated by trying to improve upon the storage utilization of B-trees. This objective has been met. After some simplifying assumptions it has been shown [KOES] that storage utilization tends to 1 for DMTs with large order and number of keys. It was also experimentally shown [CULI] that storage utilization was above 95% (compared to 70% for B-trees) when using the DMT insertion scheme.

If (m^h-1) ordered keys are inserted into an initially empty DMT, a complete m-ary tree of height h is created. Trees at intermediate steps are also as dense as possible which maximizes storage utilization. If the same insertion of ordered keys were done into an empty B-tree, the result is a sparse tree (a tree of maximum depth). The storage utilization in this case is close to the worst-case of 50% for B-trees.

Therefore, the only advantage the dense multiway tree offers is excellent storage utilization. As with the compact B-tree, this improvement is accompanied by a high input/output cost. The DMT could only be considered for databases that are very stable with very few updates and many searches, where storage utilization is of primary concern.

3.3.6 Disadvantages

The disadvantage of the dense m-ary tree is the lack of efficient updating algorithms. This is the result of the undefined relationship betweeen the static and dynamic definitions of the DMT. Static definitions require expensive reorganizations in response to insertions and deletions, while dynamic reoganizations maintain the properties of the tree while handling updates.

An insertion algorithm $O(\log_2 N)$ is known for weakly dense multiway trees (r+1) that preserve the order of the keys and the weakly dense tree structure. No such algorithm is known to exist for dense m-ary trees with r values greater than one. Currently, the best deletion algorithm known is $O(\log_{m-1} N)$ while the worst case is O(N).

The choice of the density (r) effects storage utilization and the complexity of the updating algorithms. No choice of r results in both acceptable complexity of insertion and deletion algorithms and acceptable worst case storage utilization.

The worst case storage for DMT's is (r+1)/(m+1) according to [HUAN]. For small values of r, storage percentages far below the worst case of 50% for the B-trees are possible. This is a problem as efficient updating algorithms are only available if r equals one.

Chapter 4 - B-Tree Variants For Complex Data

4.1 Multi-dimensional B-trees

4.1.1 Basic Definition

The multi-dimensional B-tree (MDBT) was proposed in 1982 [SCHE] as an index structure for associative retrieval. Associative retrieval (also known as multi-key retrieval) responds to queries that specify n attributes ($n \geq 1$). The four main types of such queries are exact matches (specifying all n attributes), partial matches (specifying d attributes where $d < n$), range queries (specifying a range of values for each of the n attributes) and partial range queries (ranges specified for some of the attributes).

The MDBT organization is basically a hierarchy of B-trees. Each level of the hierarchy (except the bottom) corresponds to a different attribute. The values of the attributes at each level are organized using a B-tree structure. The MDBT in Figure 10 is a two attribute tree. The bottom level of the MDBT is a linked list of accession pages containing pointers to corresponding data records.

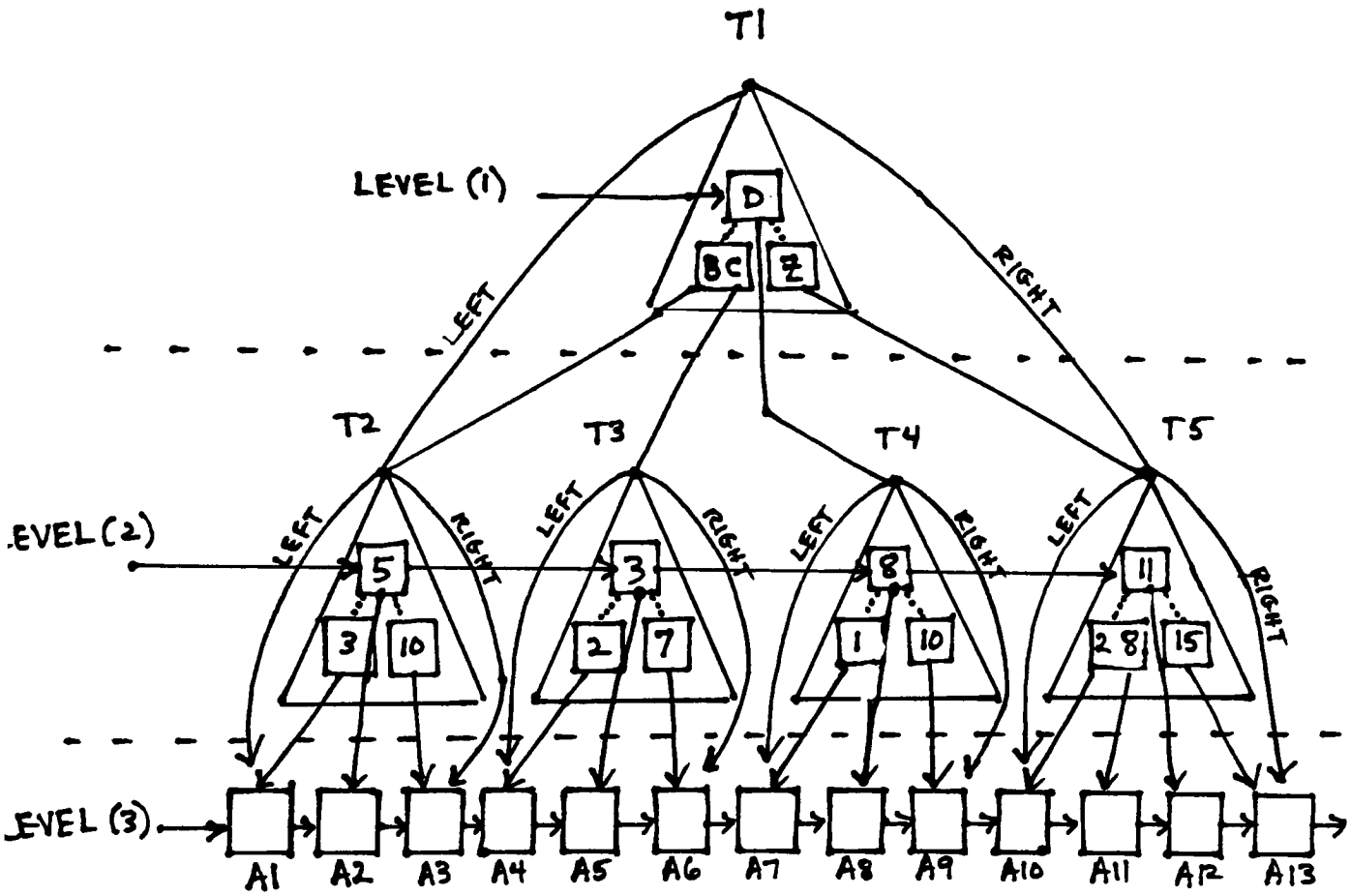
All B-trees on the same level of a MDBT have the same order and height, although from level to level the order and height may differ. This assumes that the values of an attribute are uniformly distributed and independent of other attributes values at other levels of the tree.

As is evident from the MDBT of Figure 10, many additional pointers are necessary to facilitate efficient multi-attribute searches. When the i^{th} attribute has been located in the B-tree at level i, an additional pointer is needed to level (i+1) so the search for the $(i+1)^{st}$ attribute can begin in the correct B-tree at level (i+1). These pointers make exact matches very efficient as only one B-tree per level is accessed. A unique path from root to leaf represents a distinct combination of the n attributes.

Since partial and range matches require sequential processing at each level, a linked list is maintained containing pointers to the root nodes of all B-trees at a given level. The entry point of this linked list for each level is maintained in an array LEVEL(i) which allows any level to be accessed directly.

ATTRIBUTES: B3,B5,B10,C2,C3,C7,D1,D8,D10,Z2,Z8,Z10,Z15

Figure 10: MDBT with two attributes



Each root node of the B-trees at level i has two additional pointers, LEFT and RIGHT. The LEFT pointer indicates the root of the B-tree at level (i+1) that contains the information if the smallest possible key at level i was used, while the RIGHT pointer locates the root corresponding to the largest value of the B-tree at level i. Note the pointers labeled LEFT and RIGHT in Figure 10 from B-tree T1 to B-trees T2 and T5.

LEFT/RIGHT pointers are used when partial matches leave some attributes unspecified. If a query states a value for attribute (i+1) but not for attribute (i) then the search at level (i+1) will begin at the B-tree indicated by the LEFT pointer at level i and end at the one specified by the RIGHT pointer. For the two attribute example in Figure 10, if attribute #1 is not specified it would result in looking at the level #2 trees T2 (pointed to by LEFT of T1) through T5 (pointed to by RIGHT of T1).

The general root node structure is illustrated in Figure 11. LEFT, RIGHT and M_k are all pointers to different B-trees at level (i+1). M_k indicates which B-tree to look for the (i+1)st attribute when attribute (i) equals the key value K_k. The pointers P_k refer to nodes of the B-tree at level i. The root node for the MDBT of Figure 10 is given in Figure 12.

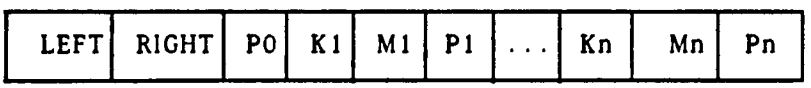


Figure 11: Structure for a MDBT Root Node at Level (i)

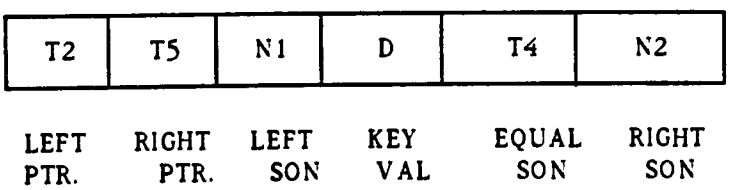


Figure 12: Root Node of the MBDT for Figure 10

LEFT and RIGHT pointers are also used for range queries. Assume in Figure 10 that attribute 1 must be in the range of A through D. The search would begin by locating the smallest key value at level 1 greater than or equal to A and the largest key value less than or equal to D. The level 1 values of B and D would be identified for this example, which point to B-trees T2 and T4 at level 2. If attribute #2 was not specified then the LEFT pointer of T2 to accession page A1 and the RIGHT pointer from T4 to accession page A9 would be used to narrow the search interval.

4.1.2 Advantages

Multi-dimensional B-trees were designed to provide efficient associative retrievals and to dynamically maintain the indexing structure in environments with frequent insertion and deletions. These features are accomplished by the use of the B-trees at each attribute level and by maintaining inter- and intra- level pointers.

Commerically available database systems often use inverted files for accomodating multi-attribute indexing. Inverted files require maintaining separate indexes for each attribute and performing costly logical operations to find intersections of the different attributes.

Experiments were run [KRIE] to compare the performances of inverted files (IF) and MDBTs. For insertions, the inverted file system required twice as much CPU time and ten times as many page accesses. For deletion over four times as many page accesses were required. Storage utilization was comparable (70% for IF and 77% for MDBT). Average retrieval time for exact, partial, range and partial range queries (measured by CPU-time and number of page accesses) for MDBTs was better than for IFs.

Storage utilization for MDBTs is generally close to 70%. This is determined in part by the average storage utilization of the B-trees used. The space complexity of the MBDT organization is O(N) which is comparable to other systems such as the k-d tree (a static system). Compression techniques can be used to decrease space required.

The MDBT has an advantage over other tree structures used for multiple attribute queries because it is dynamically defined. Other tree structures require expensive reorganization after insertions and deletions along with a preprocessing step to impose a clustering effect on the database. Such structures include the multi-attribute tree (MAT) and the key-discriminator tree (k-d tree, a multi-dimensional binary search tree). The MDBT's low maintainence cost and ability to efficiently handle all primary types of associative queries makes it an attractive choice over other static trees.

Dynamic insertion and deletion algorithms for the MDBT are presented in [SCHE] and [OUSK]. Average and worst case performance for these algorithms are shown to be O(log N) where N is the number of composite keys. The average retrieval time for exact match queries and small range queries is of complexity O(log N') where N' is the number of nodes in the directory.

4.1.3 Disadvantages

Most analysis of the MDBT assumes that it is a random tree. This

means that the values of an attribute at a given level of the tree are uniformly distributed and are independent of attribute values at other levels. This is to ensure that trees at the same level have the same height which results in a maximal height of log (N+k). Some evidence shows that this assumption may not hold for most real life databases, in which case the worst case height would be k*log N .

Kriegel [KRIE] has suggested a MDBT modification to reduce worst case height. The k-dimensional B-tree (kB-tree) allows biased B-trees (B-trees where leaves have different distances from the root) to improve distribution of the records in the file space. This modification results in worst case height of log(N+k). The kB-tree, a more complex structure,has worse update behavior but better retrieval times than the MDBT.

The worst case for partial matches is when the earliest attributes (found in the levels near the root) are not specified. This allows for a geometric explosion of trees at lower levels to be considered. This case has complexity of O(N'*logN"/N") where N" corresponds to the number of nodes in the levels of the MDBT where the attributes are not specified. N' is the number of nodes in the directory.

By ordering the attributes so that the ones most likely to be specified in a partial query occur at levels closest to the root, pruning will occur higher in the tree. This will greatly reduce the number of B-trees at lower levels to be considered. If the probabilities of specified attributes is not known, then the next best strategy is to organize the attributes according to the size of the B-trees at each level, placing the smaller order trees closer to the root.

Another factor which influences retrieval cost is the size of the B-trees at the levels of the MDBT nearest the leaves. Very small orders at these bottom levels create a degenerate structure with costs for partial match queries of O(N). The other cost estimates in the previous paragraphs assumed that this case would not occur.

Index storage utilization in a MDBT can be very low and the number of index pages very high if nodes are underfilled. If many MDBT nodes are allowed per physical page of storage this disadvantage disappears. Clustering of similar attributes into the same key value would also help.

4.2 R-Trees

4.2.1 Definition

First presented by Guttman [GUTT] in a 1984 article, R-trees are a B^+-tree variant designed to handle spatial searches for n-dimensional objects. Application areas include CAD and geo-databases.

Spatial objects are represented by pointers to data objects called tuple-identifiers. These pointers are found in the leaf nodes of R-trees. All examples below will use two dimensional data and rectangles.

Minimal bounding rectangles (MBR's) are used in the index and data nodes to facilitate the search for data objects. In leaf nodes, MBRs represent the smallest n-dimensional rectangle that contains the n-dimensional data object. (See R2 in Figure 13a). An MBR in a non-leaf node must physically cover all the area contained in the subtree identified by the MBR's pointer. In figure 13b the MBR indicated by R5 covers all the area enclosed by the MBRs in its subtree, namely R1, R3 and R4.

Non-leaf nodes consist of MBR and pointer pairs. Each pointer is an address of a lower node in the R-tree. Leaf nodes consist of pairs of MBRs and tuple-identifiers for the data.

Unlike B-trees, the number of pointers per node equals (instead of being one more than) the number of entries in the node. Each node is required to have between m and M pointers, where m can be set to any number less than or equal to M/2.

4.2.2 Differences in Algorithms

The R-trees in Figure 13a,b,c and their corresponding MBR maps will be used to illustrate the algorithms for search, insert and delete. The values of m and M are 1 and 3, so between 1 and 3 entries per node are allowed. The data is assumed to be two dimensional. The arrows in the data nodes are the tuple identifiers pointing to the data objects. The regions denoted R1, R2, etc. are the minimal bounding rectangles (MBRs).

The search algorithm is essentially the same as for a regular B-tree with the exception that more than one subtree of a given node might have to be searched. It is possible to have many search paths through the R-tree instead of just one. All MBRs in nodes on the search path(s) must be checked to see if overlap occurs with the search rectangle. Subtree pointers associated with all the MBRs containing part of the search object must be checked.

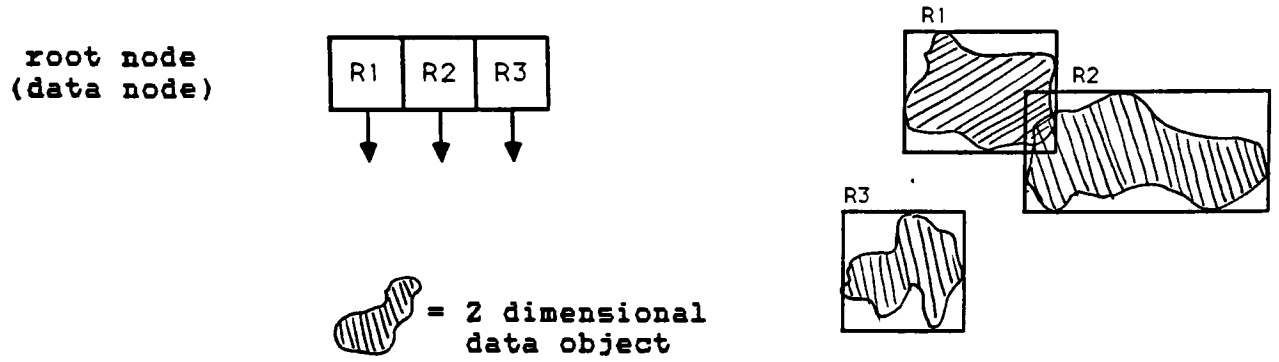


Figure 13a: R-tree Root Node and MBR Map

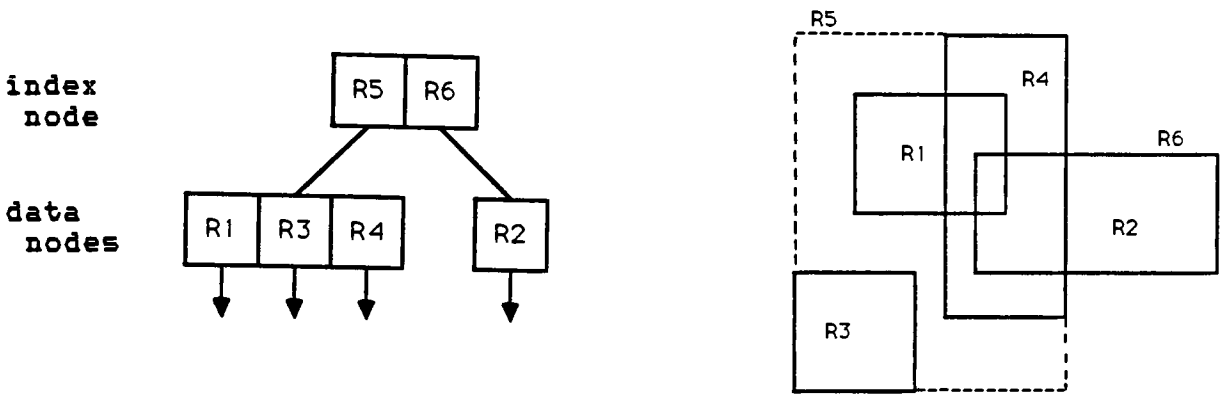


Figure 13b: R-tree after Insertion (of R4) and Split of Root Node

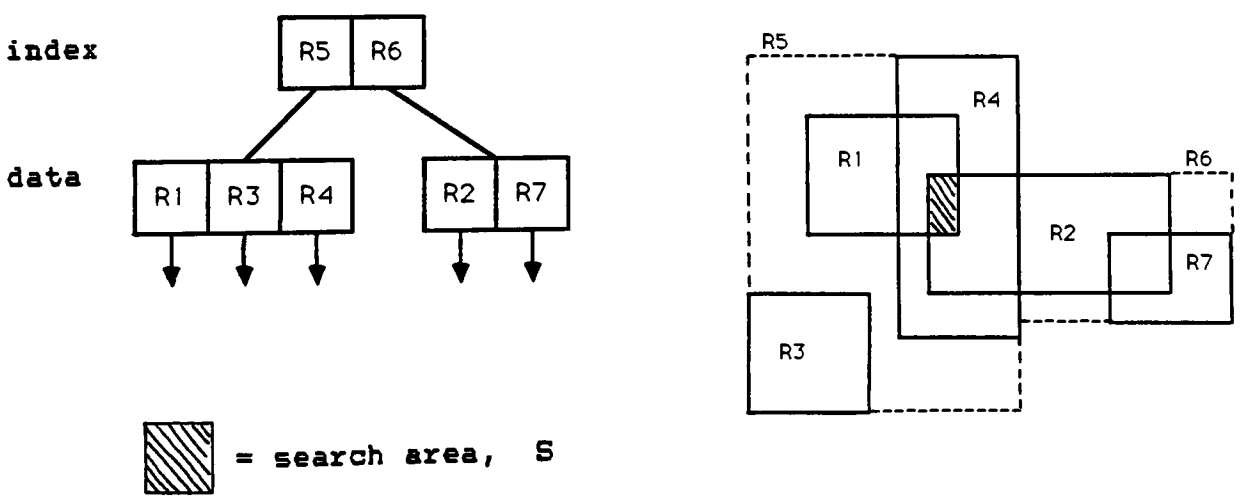


Figure 13c: R-tree after Insertion (R7) and Expansion of MBR (R6)

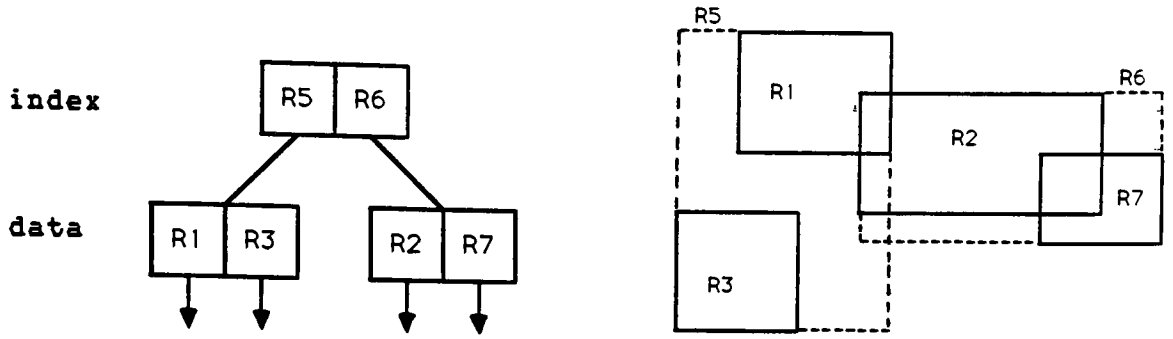


Figure 13d: R-tree after Deletion and Shrinking of MBR (R5)

In figure 13c, the search rectangle S, is shaded. Starting at the root node, both R5 and R6 are discovered to have some overlap with S. This means both subtrees of the root node must be searched. At the next level down (the leaf level) only the tuple identifiers for R1, R4 and R2 need be accessed. This search algorithm may at first seem inefficient but large, irrelevant regions are eliminated from consideration.

The insertion path generated when inserting a spatial object into a R-tree is formed by selecting the rectangle (MBR) at each level that would require the least enlargement to include the new object. Ties are resolved by choosing the MBR with the smallest total area. In figure 13c the MBR R6 was enlarged instead of R5 to include the new region R7.

This process is repeated until the leaf nodes are reached. After the spatial object is inserted into a leaf, an "adjust tree" algorithm must be applied to the parent node's MBR to enlarge it to tightly enclose all its sons including the newest one. This process backs up the tree, possibly to the root, until no further adjustment is required. Inserting R7 into figure 13b only required enlarging R6 in figure 13c.

As with B-trees, splits occur when nodes get too full (more than M entries per node). Inserting R4 will cause a split in figure 13a. New rectangles, R5 and R6, will be formed and inserted into the parent node (the root). Notice in Figure 13b that the split does not necessarily result in an equal number of MBRs per data node. Reorganization is based on trying to minimize the areas of the enclosing rectangles and to minimize the amount of overlap of regions.

The number of possibilities for reorganization is quite large and an exhaustive search for the optimal one would be too slow and costly. Guttman presents an algorithm with cost linear with

respect to M and the number of dimensions. He claims this algorithm produces nearly optimal reoganizations for splitting nodes.

Minimizing the coverage of the enclosing rectangles will reduce the amount of dead search space. Minimizing the overlap of regions will keep the number of distinct search paths low. In the worst case, a search region could overlap with every MBR which would require searching the entire R-tree. For performance considerations, minimizing overlap is more important than minimizing coverage.

Insertion of new data objects in R-trees causes the parent's minimal bounding rectangle (MBR) to expand. Deletions of data objects cause a corresponding shrinkage of coverage in the parent MBR. See the new size of R5 after R4 is deleted in Figure 13d.

When nodes become under-full during deletions (less than m entries), concatenation of nodes does not occur. This is because the necessary MBR to cover a node resulting from concatenating two unrelated nodes is likely to include a lot of dead space. Instead, the entries of the underfull nodes are reinserted in the R-tree. The advantage is that the entries find their "proper" place in the tree. The disadvantage is that costly splits may result.

There is some choice in picking the value of m, the minimum number of entries per node. It must be a number less than or equal to M/2. The smaller the value chosen, the fewer times nodes will be declared underfull. The tradeoff is that if m is too small the storage utilization for the nodes of the tree suffers.

4.2.3 Advantages

According to Guttman, the R-tree improves upon all previous methods of indexing spatial objects. Due to the multidimensional search space, the one-dimensional ordering of keys in a regular B-tree is not sufficient. Other indexing methods it improves upon include cell methods, quad trees, K-D-B trees, grid files and hash tables. The R-tree allows for high level object oriented searching rather than searching based on either low level elements of the spatial object or some artifical alphanumeric encoding of the picture. The R-tree is a dynamic structure. This permits efficient retrievals even after data objects have been inserted and deleted from the tree.

The R-tree allows pictorial and alphanumeric databases to be integrated, yet remain separate in terms of how data is represented and processed. Direct queries are allowed for pictorial data based on its spatial relationship with other

pictures in the database. An example of a direct query is "Find all buildings in a certain area". Indirect queries for spatial objects based on some non-spatial attribute are also accomodated. For example, "Find all buildings in a certain area with accessed values greater than or equal to $100,000".

The query language which permits these direct and indirect spatial searches is called PSQL (pictorial structured query language), an extension of SQL. PSQL has the R-tree as its data structure and has both pictorial and alphanumeric domains. Associations between the two types of domains are made with pointer identifiers.

There are many design features which make the R-tree a desirable data structure. First of all, the use of minimal bounding rectangles improves the efficiency of searches by keeping the amount of dead space to a minimum. MBRs also serve as an effective way to organize the tree and search spaces. Secondly, the structure of the leaf node in R-trees is flexible enough to handle many different "spatial" data objects. The leaves contain pointers to tuples which could represent points (cities) or regions (counties) or whatever.

4.2.4 Disadvantages

One major disadvantage of the R-tree is that it is not possible to guarantee a good search time. This results from the need to search the subtrees associated with all MBRs which overlap the search area. Multiple search paths are common.

Another problem is that new data objects must be inserted into pre-existing leaf nodes. Although the insertions are done so as to minimize additional area added, there will be times when excessive amounts of dead space will have to be included. This will occur if there is no good fit for the new data object in the available leaf nodes. This has a negative impact on search efficiency.

A solution to this problem is suggested by Roussopoulos and Leifker [ROUS]. They introduce the concept of packed R-trees. Assuming that pictorial databases are relatively static, they suggest preprocessing the initial data objects to pack them as tightly as possible, minimizing coverage and overlap.

The normal algorithms for insertion and deletion for R-trees can be used on the packed trees. A problem does occur however, because the leaf nodes containing the data are filled (packed). Inserting any new data will cause extensive and expensive splits which will force a reorganization of the tree. Packed R-trees are therefore most appropiate when insertions and deletions are not very frequent.

Chapter 5 - Miscellaneous B-Tree Variants

5.1 Write-Once B-Tree

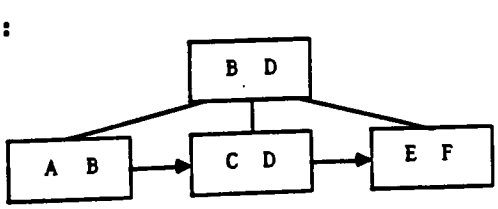
5.1.1 Definition

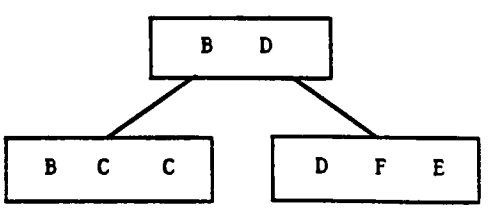
The Write-Once B-Tree (WOBT) is a B^+-tree variant created by Easton at IBM which uses indelible data [EAST]. An indelible data set is one where old data values are never erased during deletions or modified during updates. Instead, new data values are encorporated into the tree separately from old values. This concept is compatible with the properties of write-once storage mediums and with industrial accounting practice of never erasing data. A WOBT can provide rapid access to prior database states.

The structure of the WOBT, although similar to the B^+-tree, has many important differences. The following chart summarizes some of these differences.

	B^+-Tree	WOBT
Terminology:	node, page leaf page index page first key in node order of the tree (max # of ptrs/node)	bucket data bucket index bucket bucket header key capacity of the bucket
Node/Bucket Structure:	Each index node has one more pointer than separator value.	Each index bucket has the same number of ptrs as separator values.
	Keys are always in ascending order in nodes	First key is the smallest, but the rest are in insertion order.
	No duplicate key values allowed in a node.	Duplicate values allowed. The rightmost one is considered the current value.

Sample Trees:





A more detailed diagram of the WOBT in the chart is presented in Figure 14. Notice that the index bucket contains key->pointer pairs such as B -> 2. This indicates that key values greater than or equal to B but less than D are found in Bucket 2 in the next level of the WOBT.

The X;x entries in the data buckets correspond to a primary key X and associated data x. Two copies for the key C in bucket 2 indicates the key was updated (or deleted, depending on which flag bit is set). Each bucket has four sectors of fixed size. The index node contains the values found in the first position in the buckets at the next level down. Also note that the key values in bucket 3 are not in sorted order.

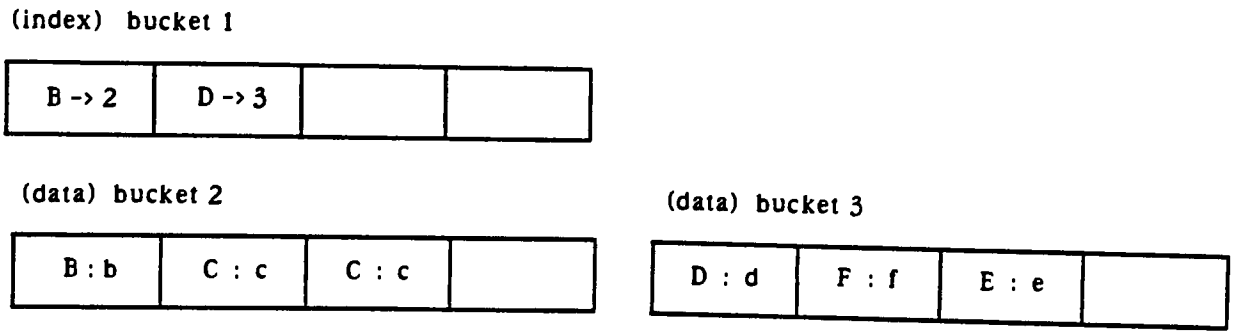


Figure 14: Sample WOBT

5.1.2 Differences in Algorithms

The find algorithm for a WOBT requires searching the root (index) bucket for the largest key that does not exceed the search key. An important difference is that the entire index bucket must be searched to locate the rightmost (most recent) copy of that key. After following the pointer associated with that key to the next level of the tree, the procedure is repeated until a data bucket is reached.

To delete a key in a WOBT the key is found in a data bucket and is flagged. It will not be recopied when the bucket is reorganized, unless it serves as the header key for the bucket. As with B^+-trees, keys are never deleted from index buckets where they remain to serve as separator values.

The insert algorithm for the WOBT finds the appropriate data bucket for insertion using the search algorithm. If there is room in the bucket for the new key, the data record is written into the next available space. There is no attempt to insert the keys in ascending order as is done with B^+-trees. If the bucket is full reorganization occurs.

The order of a B^+-tree determines whether an index or leaf page

is full. For a WOBT two distinct parameters, TI and TD, repectively determine if index and data buckets are overfull. TI values are typically 80-90% of the capacity of the bucket while TD values are between 50-75%. This will be discussed in more detail later.

Reorganization for a WOBT may or may not require buckets to split. First, the records (including the new one) are sorted by key. Only the most recent copy of each record is retained. Outdated records and records marked for deletion are ignored unless they serve as the bucket header key. If the number of remaining data records is less than the parameter TD then only one new bucket is written. Otherwise, two new buckets are used, splitting the records roughly in half. Adjustments in the parent (index) bucket must be made to include a reference to the new bucket(s). This may cause the parent bucket to overflow. Reorganizing the index node uses the parameter TI to determine how many new buckets are required.

When a B^+-tree reorganizes by splitting a node, the old node is always reused along with the newly created node. For a WOBT the old bucket is unchanged and never reused. During reorganization one or two new brand new buckets will be created to contain the current records. For both types of trees, splitting the root node increases the height of the tree by one.

Sequential access to key values in a WOBT is possible. Unlike the B^+-tree, the WOBT does not have horizontal pointers between data buckets. To retreive keys in sequential order thus requires more work. A list called the Search Sequence is created when searching for the data bucket of the first key of the sequential search. The list consists of pairs of the address of the last index bucket visited at each level and a key value. The key value is the smallest in the bucket that exceeds the search key.

The data bucket of the first search key and the index bucket one level up are found, sorted and saved. The keys in this data bucket are the start of the sequential list of keys. The list is continued by following the next pointer in the (sorted) index bucket. The Search Sequence list is used to backtrack up the tree. During this process all index and data buckets used are saved in sorted (ascending) order.

Since the root node also must be occasionally rewritten into a new index bucket, a method to locate the bucket containing the current root is reqired for a WOBT. Easton suggests keeping a time-stamped log of successive root buckets. This also facilitates finding the root as of some previous time, which is necessary if taking a historical look at the data.

5.1.3 Advantages

The main advantage of a WOBT is that it retains all past values of data records and yet it still efficiently finds, inserts, deletes and updates data. By time stamping each index and data entry, efficient searches can be conducted on old values of the database (as of a specified point in time) or current values. This eliminates the need to archive historical data. It also provides a built in audit trail for transactions.

The access time is proportional to the logarithm of the number of current records in the database. Easton estimates that the number of accesses required in a random search for a key in a WOBT is approximately equal to the number of accesses in a B^+-tree, if the fan out of the index buckets is high.

The storage required for a WOBT is greater than for a B^+-tree, but reasonable for the additional information available. If E is the number of inserts, updates and deletes, then the worst case storage is approximately 4E. Easton finds in practice that the actual storage to be closer to 2E or 3E.

Easton shows that at any time after the first bucket has been rewritten, the number of active buckets is slightly greater than the number of inactive buckets. Thus, the average WOBT requires twice the storage of a conventional B^+-tree.

The WOBT takes advantage of new optical disk technology which offers high-density, once write storage capabilities. It also can be used on erasable storage devices.

The storage utilization and depth of the tree can be controlled by setting the values of the parameters TI and TD. They represent the maximum number of entries allowed per index and data bucket respectively, in a reorganized bucket. The value of TI effects the fanout. It is advantageous to have high fan-out to reduce the height of the tree and improve storage utilization. Therefore the value of TI is optimally set to 80-90% of the capacity of the bucket. Since insertions and deletions occur only in the data buckets (unless reorganization requires splits), it is reasonable to keep the index buckets this full.

Easton suggests a TD value of 50-75% of capacity for the data buckets to prevent excessive storage consumption and at the same time allow space for insertions, updates and deletions. Storage can further be improved by using data compression or by sorting initial data records before entering them into the database.

Certain design decisions made by Easton have resulted in minimizing the amount of updating required in a WOBT. Index buckets never have to be changed when doing deletions since data

bucket's header keys are never removed. Also, by not having horizontal pointers between data buckets, expensive overhead is eliminated.

Having earlier versions of the database available might facilitate concurrent use of the WOBT. Easton suggests that concurrency controlls could be designed to allow readers to access earlier versions of the database, while writers are updating other portions of the tree.

As technological advances are made in write-once storage mediums, the WOBT will become an even more attractive variant of the B-tree. Also, if data management policies require quick access to historical data the WOBT is a good choice.

5.1.4 Diasdvantages

One disavantage of a WOBT is the greater search time within each bucket. Every value in each index and data bucket must be checked before confirming that the most recent occurance of a key value has been found. After a bucket has been copied (during reorganization) the initial entries are in sorted order. Easton suggests that a binary search be used on the old (sorted) entries and a linear search on the new (unsorted) entries.

There is added cost resulting from extra storage requirements and more complex algorithms such as the one for sequential searches. For applications where historical data is not important this added expense could not be justified.

5.2 Linked B-Tree

5.2.1 Definition

The standard B-tree is not a robust data structure. This means that errors in the structural data (as opposed to the user data) are not detectable and correctable. A B^+-tree variant called the Linked B-tree (LB tree) is a robust structure.

A storage system is called n-detectable (n-correctable) if it can detect (correct) errors in the structural data resulting from n or fewer errors. The normal B^+-tree is 0-detectable and 0-correctable. For example, each node has a count variable indicating the current number of entries per node. If this count number is just one too low, an entire subtree would be eliminated.

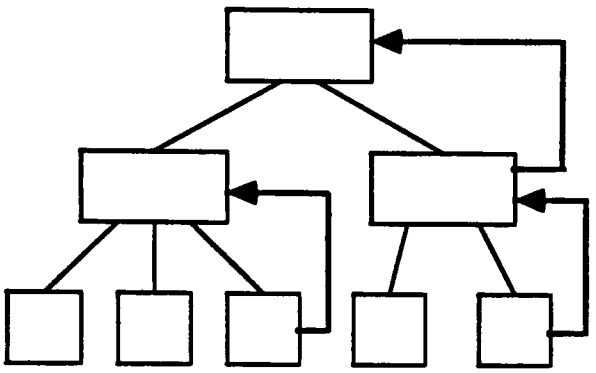
The LB tree is 2-detectable and 1-correctable. In addition, the LB tree is considered a locally correctable storage structure which can undo any number of errors in the tree, as long as any two errors are not "close" to each other. This feature is described in more detail later.

The LB tree is robust because of the pointer structure and key representation in each node. The robust key representation uses a difference field containing the difference of successive keys. The count variables in each node are similarly protected.

The pointer structure of the LB tree includes new nodes as well as new pointers. Each tree has a header node containing the count of the nodes in the tree and a pointer to the root node. Each level of the tree also has a separate header node. These level header nodes contain a node count for that level and a pointer to the first node in that level. The level headers form a linked list which shrinks and grows as the height of the tree changes.

In addition to the normal pointers of a B^+-tree, the LB tree has thread and chain pointers. The thread pointers are null for all nodes except the last son of a node which has a pointer back to its father node as shown in Figure 15. The header nodes and chain pointers are not shown in this diagram. The thread pointers facilitate entering the LB-tree at any level and being able to determine how the nodes at that level are assigned to nodes at the previous level.

Figure 15: Thread Pointers of a LB tree



The chain pointers connect the level header node and successive tree nodes at any given level as shown in Figure 16. The chain pointers form a circular list as there is a pointer from the last node of the level back to the header node for the level. These pointers facilitate reconstructing incorrect pointers from level to level in the tree.

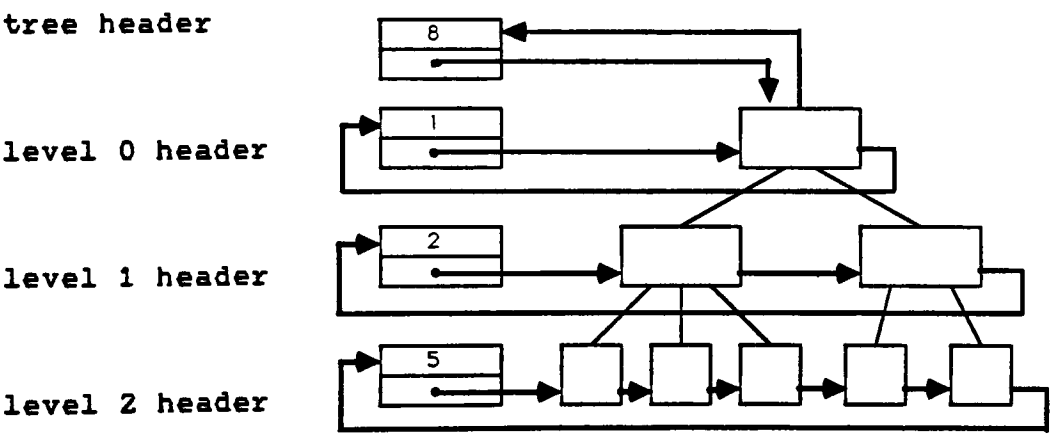


Figure 16: Chain Pointers for a LB tree

The pointer structure of the LB tree is 3-detectable. However, the list representation for the keys is only 2-detectable, so the overall detectability of the tree is 2. It is also possible to correct a large number of errors if there is no more than one error in a substructure of the tree containing four pointers. This is why the LB tree is called locally correctable.

For example, in the substructure in Figure 17, if pointer #1 is incorrect, pointer #2 can be used to correct it and visa versa. Likewise, pointer #3 can be corrected by #4, and #4 by #3. If more than one pointer error was in this substructure, correction would not be possible.

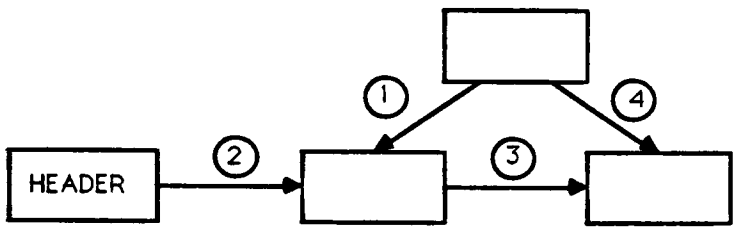


Figure 17: Locally Correctable Substructure

The algorithm to detect and correct errors in a LB tree simultaneously scans two adjacent levels, left to right. The upper level's chain pointers are assumed to be correct from the previous scan. The traditional B-tree pointers from the upper level to the lower level are checked concurrently with the chain pointers between nodes at the lower level. In Figure 17, pointers #1 and #2 are checked at the same time as are the pointers #3 and #4. If there is a pointer disagreement, the next node is checked. If there is agreement on the next node, then the error is in the B-tree pointer, as that type of error only will affect one node at the lower level. Otherwise, it must be a chain pointer error. Pointers are easily fixed by the redundancy of the pointer structure.

5.2.2 Advantages / Disadvantages

If a B-tree is used in an application area where a degree of fault tolerance is required, then the robust LB tree is an option. The LB tree is an improvement over previous robust B-trees including its predecessor the Chained and Threaded B-tree (CTB tree). The CTB tree does not have level header nodes and its thread pointers are only part of the leaf nodes.

The implementation of the updating algorithms for the LB tree compared to the CTB tree are easier. This is partly because of the more uniform use of the thread pointers. The error detection and correction algorithms for the LB tree are also superior. The CBT tree's error detection algorithm is only 2-detectable, one less than the LB's.

The CTB correction routine, which is longer and more complicated, can only correct a single error in the tree. The

LB correction algorithm can always correct a single error and can also correct any number of errors if the errors are all in distinct subtrees.

The CTB correction routine can not always determine which of two pointers is in error when there is a disagreement. It must guess and then verify to see if the quess was correct. This is not the case for the LB tree. Generally, the LB correction algorithm reads fewer nodes of the tree than does the CTB routine, although both require O(n) time to run (where n is the number of nodes in the tree.)

Although the update algorithms and error detection and correction algorithms are easier to implement than for previously defined robust B trees, they are still quite costly. For example, the correction algorithm for the LB tree is O(n), where n is the number of nodes in the tree. The LB tree also requires extra storage for the level headers and the extra pointers.

Another problem with the LB tree is the inability to have concurrent users. The error detection and correction algorithm assumes the entire tree to be locked because of the way it traverses the tree.

Chapter 6 - Concurrency Issues in B-Trees

6.1 Definition of Problem / History

In a multiuser environment, maintaining the integrity of an index for a large database is essential. In order for B-trees to be a feasible indexing method, the issue of controlling concurrent users must be addressed.

Basically there are three types of operations done to the B-tree; search, insert and delete. Readers search trees but do not modify them. Updaters insert or delete keys which always changes the tree. Users must be guaranteed that each operation on the B-tree can be completed correctly without interfering with simultaneous operations by others. Concurrency controls must address the problems of lost updates, unrepeatable reads, serializability of operations and potential deadlock.

The problem of the lost update occurs when two updaters try to change the same node of the B-tree simultaneously. If an updater changes a node that a reader is about to access, the key being searched for may erroneously thought to be missing.

To illustrate this problem consider two operations; inserting the key 10 and searching for the key 7 in the B-tree in Figure 18. Listed below is the timing sequence of the execution of the two operations. The difficulty arises because the reader accesses a node that a writer has just changed.

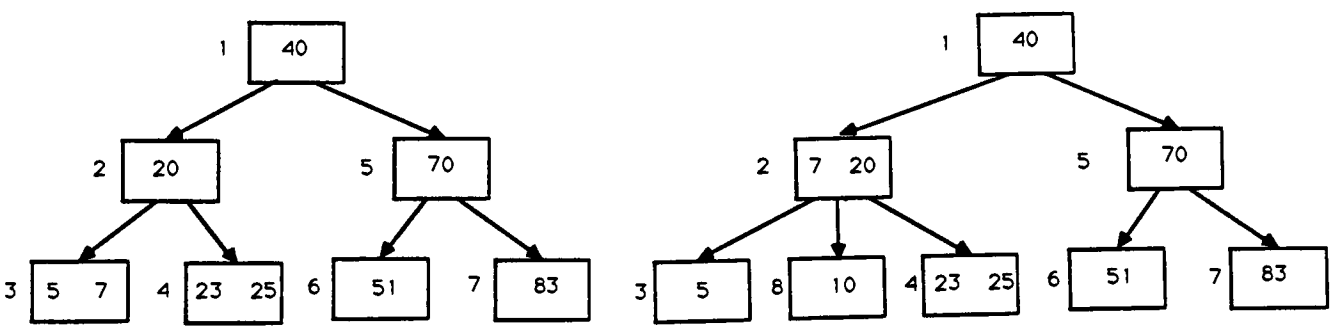


Figure 18: Lost Update B-tree

insert key value 10	search for key value 7
read node 1	
read node 2	
read node 3	
	read node 1
	read node 2
	(about to read node 3)
insert key 10	
split node 3, create node 8	
write node 3	
write node 8	
write node 2	
	read node 3 (key 7 is NOT found)

This example illustrates the non-serializability of these operations. This means that the outcome, if these two operations run concurrently is not guaranteed to be equivalent to the outcome if serial execution of the two operations is enforced.

Early concurrency work on B-trees centered around various locking schemes for nodes. Simple locking methods have been replaced by more complex ones. Other approaches include restructuring the nodes of the B-tree or preprocessing the B-tree to lessen the amount of locking necessary. These are exemplified by the B-tree variants called the B-link-tree and the PO-B-tree described in Sections 6.4 and 6.5. More recent work explores "optimistic" approaches which replaces locking by relying on restarting the conflicting operations.

6.2 Locking Concurrency Control Methods

Early concurrency control work on B-trees was done by Samadi [SAMA]. In his simple locking protocol design, each node of the tree had a semaphore associated with it. A queue of waiting processes is maintained.

To minimize the amount of locked nodes, Samadi noted that the restructuring associated with splits and concatenations in B-trees will never propogate past safe nodes. (The terminology of safe and unsafe nodes was first formally presented in Bayer [BAYEb]) A safe node is not full so an insertion of another key will not cause a split but is more than half-full so the deletion of a key will not require a concatenation of nodes. The number of keys, K in a safe node is $d < K < 2*d$, where d is the minimium number of keys per node.

Samadi uses only one type of lock. Two operations can not hold

a lock on the same node simultaneuously. Readers lock a node before scanning it and unlock it after locking the next node in the search path. Each reader will have one or two nodes locked in the tree at any one time.

Updaters hold locks on one or more nodes in the search path depending on whether nodes are safe or unsafe. If an unsafe node is encountered , then the parent of the node, the unsafe node itself and all subsequent nodes in the search path of the update will be locked until another safe node is found. The rationale for this is that changes caused by updaters will never propagate up the B-tree pass a safe node.

Updaters follow the algorithm:

```
while ( current page is not leaf ) do
     lock appropriate son of current node
     read son, make it current node
     if ( current node is safe ) then
         release all locks on ancestors of current node
insert or  delete
release all locks
```

To illustrate this algorithm, refer to the skeletal B^+-tree of order 2 in Figure 19. Assuming that node b is unsafe, the locking of nodes proceeds as follows:

```
node a is locked
node b is locked  (since b is unsafe, keep lock on node a)
node c is locked
release locks on nodes b and a   (since node c is safe)
node d is locked
release lock on node c (since node d is safe)
```

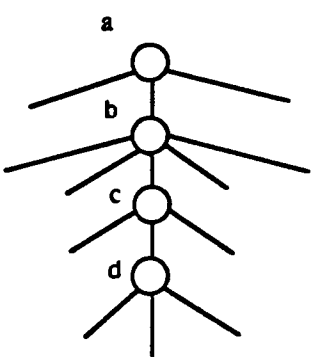


Figure 19: Skeletal B-tree of Order 2

The advantage of Samadi's algorithm is the allowance of concurrent use of the B-tree. The locked portions of the tree

are determined by the level of the deepest unsafe node for updaters and the current nodes being accessed by readers. Deadlock cannot occur since all locking is from the top-down such that one process cannot overtake another. The disadvantage of Samadi's algorithm is that the amount of concurrency allowed is severely restricted by having the same exclusive lock for all users.

More sophisticated locking protocols for B-trees are presented in [BAYEb]. In one solution described, three types of locks are used; R-lock (reader), A-lock (writer exculsion) and E-lock (exclusion lock). Nodes with an E lock cannot be accessed by another user until that lock is released. The R and A locks are compatible, that is a node can hold both concurrently. At any one time a node can have many R-locks but only one A-lock. This allows multiple readers but only one updater per node. The A-lock on a node can be upgraded to an exclusive lock (E) if a reader lock (R) is not currently on the node.

Readers lock and unlock nodes as in Samadi's solution, only now with their own (weak) R-type of lock. As an updater proceeds down the tree, it sets A-locks on nodes until a safe node is encountered. At this point A-locks on ancestors are released as before. This process continues until the updater reaches a leaf node. If the leaf is safe, the tree should have only one A-lock (on the leaf). If the leaf is unsafe, the tree will have two or more A-locks. Unsafe leaves only occur roughly once every d updates, where d is the minimum number of keys per node. Since most B-trees have large orders, d, unsafe leaf nodes occur infrequently.

At this point it is determined if an update will be successful. If nodes with A-locks do not also have reader (R) locks , then all the A-locks are upgraded to exclusive (E) locks. Otherwise this upgrade is delayed until the reader locks are released. A request to upgrade a nodes's lock to E has priority over any other pending locks on the node. After E-locks are obtained top-down in the tree, the update occurs and the locks are then released.

The advantage of this approach over Samadi's is that it increases the concurrency with readers and updaters, especially at the higher levels of the tree. Exclusive locks are usually restricted to the lower levels of the tree where most updates will occur, allowing readers to be restricted from fewer nodes.

Disadvantages include increased overhead for lock conversion and for checking lock compatability. According to Lausen [LAUS], these tests require global data and must be synchronized. As locks are requested several times for each operation, this reduces concurrency considerably. Also, since only one writer exclusion lock (A) is allowed per node at any given time,

updaters may temporarily block each other.

A more generalized and complicated solution is presented by Bayer and Schkolnick. An additional type of update lock is introduced as well as parameters for the maximum number of levels an updater can place locks on. Details are in [BAYEb].

Lehman and Yao simplified locking protocols by making the B-tree structure more complex. Nodes at each level are chained together by link pointers from left to right brothers. The link pointer in the right-most node is null. Each node also has a high key value indicating the upper bound of key values for all subtrees of the node. The resulting structure is called a B-link-tree.

The high key and link pointer remove the need to lock unsafe nodes. Now, if an unsafe node has split, one can use the link pointer to reach the new node (if necessary). There are no reader or writer-exclusive locks needed for this tree. Nodes being changed must still be exclusively locked before modified, but at most three nodes will be locked by a single updater. A more detailed explanation of B-link trees is given in section 6.4.

A hybrid solution between Lehman and Yao's B-link tree (L/Y) and Bayer and Schkolnick's R, A, E locking (B/S) is proposed by Ellis [ELLI]. She states that the major difference between the two approaches is whether pages of the B-tree brought in from secondary store are shared (B/S's) or private (L/Y's). Shared pages imply many users have access to the same local copy of a B-tree node, while private pages imply each user has their own local copy of the node in main memory.

Shared pages require more locks while private pages can result in multiple copies of a single page being stored concurrently in the limited main memory. In Ellis' hybrid solution, shared pages are used for the upper level of the B-tree and private pages for the lower levels. A parameter specifies at which level of the tree the transition takes place.

Upper level pages are most likely to be accessed by multiple operations as fan-out has not yet been significant. To avoid multiple private copies of these pages, shared copies are used. These upper pages are also the least likely to retain the update locks (from Bayer's shared pages) as most A-locks will occur at lower levels where splitting is more likely. The deeper in the tree, the less likely it will be to have two operations accessing the same page, each with a private copy in main memory. Using private copies at the lower levels will lessen the number of exclusive locks required.

One disadvantage of this hybrid model results from the fact that

the height of the B-tree may increase or decrease. If the transition level is fixed then nodes that previously were in the "shared" portion of the tree will now be in the "private" portion or visa versa. Ellis introduces a monitor to facilitate this changeover.

6.3 Optimistic Concurrency Control Methods

A basic assumption underlying locking schemes is that there will always be conflicting operations attempting to execute concurrently which must be controlled. Kung and Robinson [KUNG] suggest that an "optimistic" approach might be more appropriate, where conflicting operations are assumed to be the exceptional case. Under this assumption, locking with all of its expensive overhead is not needed. At the end of an update operation if a conflict is detected, the operation is undone and restarted. This will happen seldomly so it should not require excessive overhead.

With optimistic concurrency control, each operation has two phases: local and global. In the local phase readers and updaters use local copies of pages when performing searches and writes. The global phase begins with a validation step which determines if the execution is serializable. If it is, the local copies are made global and written to secondary memory. It not, the local copies are thrown away and the operation is restarted.

In order for the optimistic model to work, the global phases are serialized. This ensures updaters cannot execute concurrently. The validation and updating of the global phase must be executed quickly, relative to the time required for local phases, to obtain a high level of concurrency. Validation time is proportional to the degree of concurrency possible for the tree.

The execution is serializable as long as a proper version of the tree has been processed. For a given operation the tree will be correct if the locally read nodes do not include nodes for which local copies have been written by other concurrent operations.

If the features of the database are such that conflicting operations are likely, an optimistic control method should not be used. For example, if consecutive business orders are processed concurrently using the order number as a key value, conflicting insert operations would be quite likely. However, in a query intensive environment where little updating is done, an optimistic approach would result in very low overhead.

An integrated method of concurrency control is suggested by Lausen [LAUS]. All operations still use the local/global phases of the optimistic model, but operations are also designated as either o-type (optimistic) or l-type (locking). When conflict is

likely or when an operation has had to be restarted numerous times, the type can be switched from o (optimistic) to 1(locking). If conflict is unlikely, the overhead can be reduced by designating operations as o-types. A monitor is used to ensure serializability.

Kung and Robinson [KUNG] point out two features of B-trees that make them good candidates for the optimistic approach. First, the ratio of nodes accessed during an operation versus the total number of nodes in the tree is small. This is due to the large order and resulting small heights of most B-trees. This feature lessens the probability of concurrent users of the same node in the tree.

Secondly, the probability of modifying a congested node is small. All searches for B-trees start at the root and fan out, so the root and level one nodes are the most congested. Yet modifications rarely occur at the top levels as insertions and deletions are done from the bottom up. Only if there is extensive splitting will modifications propagate to the root.

For the optimistic method to be efficient, minimal time must be spent in the validation phase. Intersection of the read set of one operation and the write sets of concurrent operations must be null for a positive validation to occur. For B-trees the time required to do this check will be small since the largest size of a write or read set is the typically small height of the tree. Kung uses the example of a B-tree of order d = 199 with almost two million keys which will have a depth less than five. Therefore, each write/read set would have at most four elements.

Optimistic methods also require a short time spent in the global phase of validation/writing compared to the local phase of reading/writing. This condition is satisfied by B-trees. Since B-trees are implemented with a limited amount of main memory, most reads of nodes require page swapping between secondary and main memory, a lengthy process. Also reads are done more frequently than writes. Each operation requires up to h reads, where h is the height of the tree, while the number of nodes that need updating is commonly one.

Kung and Robinson derive a formula which conservatively estimates the probabilty of the read set of one insertion intersecting with the write set of another insertion. The probablity of conflict is based on the height of the tree, the order and the number of leaf pages. For example, for the B-tree of height 3, order 199 with 10^4 leaf nodes, the probability of conflict is less than .0007. They claim that restarts due to conflicts with insertions in B-trees is thus rare.

6.4 B-link tree

6.4.1 Definition

The B-link tree, introduced by Lehman and Yao [LEHM], is a variant designed to facilitate efficient locking of nodes in a multi-user environment. The B-link tree is a B^+-tree (all keys found at the leaf level) such that each branch node has an additional key,pointer pair. This extra "high value" key indicates the highest key value in the rightmost subtree of that node. The extra "link" pointer is to the right brother of the node thus forming a linked list of nodes at each (non-leaf, non-root)level of the tree.

This additional information in the node allows fewer locks to be used. As updaters search down a B-tree using conventional locking, unsafe nodes are locked unless a safe node is subsequently found in the search path. Readers also put locks on nodes as described in section 6.2. The above locks are now unnecessary. The only nodes that require locks are the actual nodes being changed.

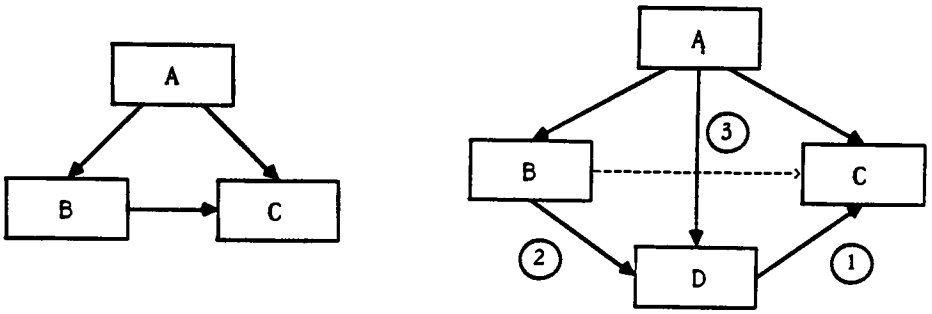
6.4.2 Differences in Algorithms

A stack of the right-most nodes examined at each level of the tree is created while an inserter goes down the tree. If a split is required, backtracking is possible using the stack values. In the example in section 6.1 a problem arose when an updater changed a node before a reader had a chance to scan it for a particular value. Since the node being searched had been split, the key value being sought was now either in the right-brother node or in the parent node.

With a B-link tree, if the high key value is less than the node being searched for, this indicates a split has occurred. The extra link pointer can take the searcher to the right brother node and the stack values can take the searcher to the parent node, whichever is required. Thus, searchers never require locks.

To split node B in the example below, a new node D is formed. Pointers are assigned in the order numbered in Figure 20. Before the link (3) between the new node and the parent (A) is established, nodes B and D are considered twin nodes and the values in node D can be correctly reached via node B instead of node A.

Figure 20: Splitting of a B-link Tree Node



In conventional locking schemes, nodes are locked or unlocked depending on whether they are safe or unsafe. With B-link trees the only nodes locked are ones being changed. While backtracking up the tree, if nodes being accessed have been split since putting them on the stack, the correct keys can still be found using the high key and link pointer.

To simplify the deletion algorithm leaf nodes with fewer than order d keys are allowed, under the assumption that this rarely occurs. If many leaves are less than minimally full then a batch reorganization of the B-link tree could be done at a convenient time or the whole tree could be locked while an underflow algorithm reoganizes the tree. As with all B^+-tree variants, keys in non-leaf nodes act as separators and never need to be deleted.

6.4.3 Advantages and Disadvantages

The main advantage of the B-link tree is the efficient use of locks. For any locking method a minimum of one lock is needed on the node being changed. With a B-link tree a maximum of three locks is required for an operation, and this is for the unusual situation described below.

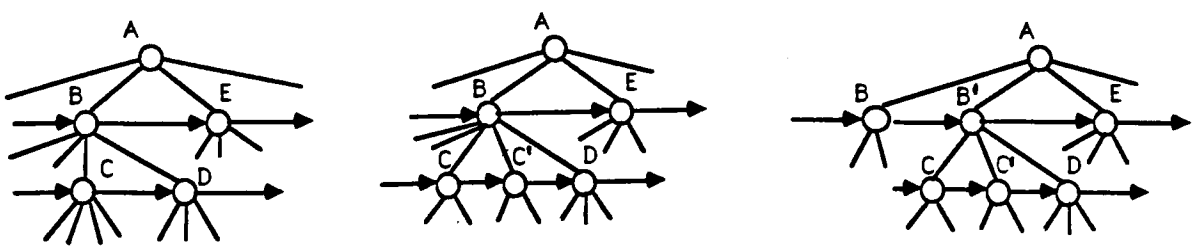


Figure 21: B-link Tree Reorganization

Consider the skeletonal B-link tree in Figure 21. Assume a key is inserted in node C after C is locked which causes a split. Meanwhile node B, the parent of C, has split from a different, concurrent operation so the parent of C is now B'. As C splits, keys are redistributed into C' and the separator key for C and C' must be inserted into their parent. According to the stack the parent of C is B, so B is locked. The high value key of B is checked and it is less than the separator key to be inserted so the third node B' is locked. This is the only situation where the maximal number of nodes (3) will be locked (nodes B, B', C). Normal splits require two locks and no-split insertions require one.

Another advantage of B-link trees is that the search and update algorithms are not very different from the sequential B^+-tree algorithms and are therefore easy to implement.

A disadvantage discussed by Ellis [ELLI] is the inefficient use of the limited main memory. To avoid having locks on readers or searching writers, each operation must have its own private copy of pages in main memory. Therefore it is possible for many copies of the same page to be present in main store simultaneously, even though at most one operation has a lock on it.

6.5 PO-B Trees

6.5.1 Definition

The preparatory operations B-tree (PO-B tree) is another B^+-tree variant designed to improve the level of concurrency allowed by reducing the number of locked nodes. The strategy is to prematurely split (concatenate) unsafe nodes along an insertion (deletion) search path. Changes to the structure of the tree required during the actual insertion or deletion of a key will therefore not propagate up the tree past the parent of the node(s) affected. This limits the number of nodes locked by any updater to two; the node being updated and its parent.

The naming and the implementation of the PO-B tree was done by Mond and Raz [MOND]. The idea behind this method of concurrency control was not original. Others (Guibas and Sedgewick, Keshet) suggested the idea of safe paths to minimize long chains of locked nodes as early as the late 1970's.

The structure of the PO-B tree differs from the B^+-tree only in the number of allowable entries per node. To accomodate premature splitting/concatenation, the number of keys per node may be no more than 2d+1 (instead of 2d) and no fewer than d-1

(instead of d). The value of d is the order of the tree in the B^+-tree meaning of the word (ie, the minimum number of keys per node).

When prematurely splitting a node with 2d entries, the "middle" entry gets promoted to the parent leaving the sons with d and (only) d-1 entries apiece. If premature concatenation of 2 nodes occurs,each with d entries, then the separator key from the parent node must be brought down for a total of 2d+1 entries in the new node.

6.5.2 Differences in Algorithms

The search, insert and delete algorithms for a PO-B tree are the same as for the standard B^+-tree with the exception of the premature splitting and concatenation of nodes. The only node which is not prematurely changed is the root.

Premature splitting will occur when a node of 2d or 2d+1 keys is found when descending the tree to find the proper leaf for inserting of a new key. Likewise, premature concatenation will occur if two adjacent nodes with d or d-1 keys are found during a deletion. The resulting node will have 2d-1, 2d or 2d+1 keys with the additional key coming from the parent node. No concatenation will occur if a node with d or d-1 keys has an adjacent brother node with more than d entries. As with the B^+-tree, in this case the keys will be shifted instead.

A sample skeletonal PO-B tree is shown in Figure 22a. From the portion of the tree shown it could also be considered a B^+-tree of order 2. Upon inserting the key value of 19, the unsafe node B is prematurely split. The new node D has only one entry, which is legal for a PO-B tree (d-1) but not for a B^+-tree. The value of 19 can be inserted with no further splits.

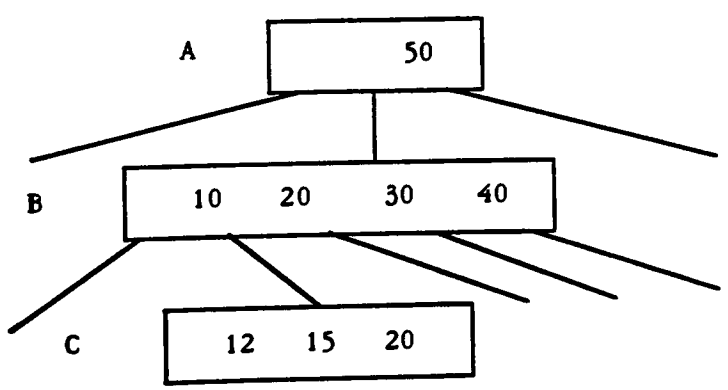
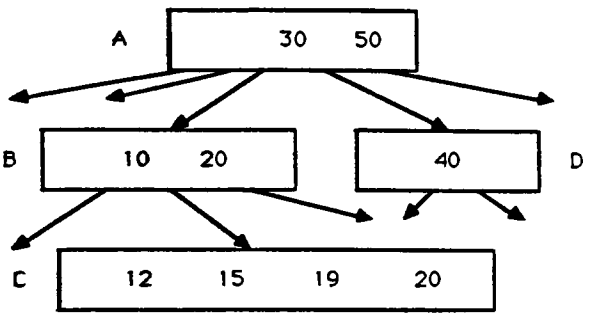


Figure 22a: Skeletonal PO- B tree of Order 2

After inserting 19 into the above PO-B tree :



Locks	Number of locked nodes
lock A	1
lock B	2
split B/ create D/ update A	2
unlock A	1
lock C / insert 19	2
unlock B	1
unlock C	0

Figure 22b: Locks and New PO-tree After Insertion

No more than two nodes were locked at any given time during the insertion of 19. If this had been a B^+-tree, concurrent locks on nodes A, B and C would have been required. Since node B is unsafe, the lock on A could not be released until it is determined that no splits will occur in nodes B and C. For a B^+-tree, the value of 19 would have been inserted without a split.

6.5.3 Advantages and Disadvantages

Encountering unsafe nodes in other locking schemes requires the the parent of the unsafe node, the unsafe node itself and all unsafe descendent nodes to be locked. This chain of locked nodes could potentially be h in length, where h is the height of the tree. The advantage of the PO-B-tree is that this chain of nodes has a maximum length of two.

Another advantage is that the standard algorithms for B^+-trees can be used for the PO-B tree. Minor modifications of when splits/concatenations occur and the size of the node would have to be made.

By reducing the number of locked nodes, the throughput of transactions on the data is improved. This is especially true since all data is stored at the leaf level which requires traversing the full height of the tree. Delays due to concurrency controls in the index portion of the tree could slow down the processing of the data.

For large order PO-B trees, the number of unsafe nodes relative to the overall number of nodes in the tree is small. The extra

overhead associated with the premature reorganization of the tree is therefore a smaller percentage of the overall operating expense for large order trees.

A disadvantage of the PO-B tree is illustrated by the example tree in figure 22a. Node B was split when 19 was inserted even though the value could have safely been inserted without a split. If the next operation is to delete the key 15, nodes B and D would be concatenated before 15 is removed from node C. Two unneccessary, expensive reorganizations of the tree would have taken place. The excessive overhead from unnecessary splits and concatenations is a disadvantage of this tree.

The previous example also illustrates a potential problem when insertions and deletions of similar valued keys are alternated. Premature splits result in at least one node with d or fewer entries. This node is safe for insertion but unsafe for deletion. Likewise, premature concatenation often result in nodes of 2d or 2d+1 keys. These nodes are safe for deletion and unsafe for insertion. This drawback could be lessened by avoiding alternating inserts and deletes of similar values.

Chapter #7 - Summary

7.1 Introduction

This final chapter is a summary of the evolution of B-tree research work done as of mid 1986 (section 7.1) and some speculations concerning the future direction of B-tree development (section 7.4). Also included are summary charts of the main advantages and disadvantages of the B and B^+ tree variants discussed in this thesis (sections 7.2, 7.3).

In the twenty years since the creation and early performance evaluation work of the basic B-tree by Bayer and McCreight, interest and research centered on the B-tree has steadily grown. Evidence of this are the many journal articles that cite B-trees. This thesis has been an attempt to outline the major developments and variants.

Much of the early work involved emperical and analytical evaluation of B-tree implementations. Researchers in many large corporations, including Honeywell, Boeing and IBM, designed file systems incorporating B^+-tree structures in the early 1970's.

Interest in improving storage utilization followed in the early 1980's. New variants such as the compact and dense multiway B-trees were created to address this goal. Much of the work with these variants apppears to have deadended as the improvements in storage utilization have been overshadowed by complex and costly algorithms to maintain these non-dynamic B-tree variants.

The flexiblilty of the B-tree structure has lead researchers to create variants for a variety of application areas. Two such examples discussed in detail in this report are the multi-dimensional B-tree and the R-tree. The first handles multikey retrievals and the later handles n-dimensional (pictorial) data. In both variants, the modifications preserve the dynamic quality of the trees and efficiency of updating algorithms, except in extreme cases.

The B-tree has been updated to take advantage of newly available hardware and storage mediums. The basic algorithms for searching, insertion and a modified deletion of keys in B-trees can now be implemented in hardware using parallel processors. A new variant, the Write-Once B-Tree developed at IBM, takes advantage of write-once storage mediums that are now available. It preserves all old values of the index and data which are as readily available to the user as current values. This variant eliminates the need to archive past versions of the database.

Concurrency issues were just beginning to be addressed at the

time of publication of "The Ubiquitous B-Tree" by Comer. Since then much research work has dealt with the modification of B-tree variants to ensure successful implementation in a multiuser environment.

Early concurrency solutions used primitive locking schemes for the tree nodes. More comlex locking schemes followed which allowed for more concurrent users in the tree.

Another approach to the concurrency problem has been to modify the B-tree structure to minimize the number of locked nodes required by each user. Extra pointers were used to accomplish this in the B-link tree, while premature splitting and concatenation algorithms are used in the PO-B tree.

Promising new concurrency work centered on the "optimistic approach" that conflicting operations are rare in B-trees, has lead to less expensive, less cumbersome concurrency control techniques. Many of the features of B-trees of large orders facilitate their use in a multi-user environment.

Since the completion of this report many journal articles and conference papers referencing B-trees have been published. Samples of such work are briefly described below.

New algorithms have been developed for concurrent B-trees. A novel, symmetric deletion algorithm only requires one locked node except in rare cases [LANI]. A compression algorithm to reorganize nodes that are too sparse after deletions in concurrent B-trees is described in [SAGI]. Other operations can run concurrently with the compression algorithm which only requires the locking of three nodes.

Another article presents a new algorithm for maintaining minimum heights in the B-tree and its variants [DRIS]. Many other articles refer to the use of B-trees for file organization in new data base implementations. A new high speed relational data base uses an extended B-tree index [OKAM]. B-trees are also used to allow for partial expansion of files [LOME].

7.2 Summary Chart Comparing B-tree Variants to B-trees

B-Tree Variant Name	Major Improvement Over Basic B-tree	Other Advantage
1. B^*-tree	storage utilization	uses similar algorithms as the B-tree
2. Compact B-Tree	storage utilization	low search costs
3. H-Tree	storage utilization and height controlled by parameters	dynamic structure, reasonable update costs
4. Dense Multiway Tree	storage utilization	low search costs

B-Tree Variant	Major Disadvantage versus B-tree	Other Disadvantage
1. B^*-tree	none	none
2. Compact B-Tree	costly insert/ delete algorithms	not a dynamic structure
3. H-trees	more complex algorithms	cost increases for some parameter values
4. Dense Multiway Tree	costly insert/ delete algorithms	not a dynamic structure

7.3 Summary Chart Comparing B^+-tree Variants to B^+-trees

B^+-Tree Variant Name	Major Improvement Over Basic B^+-tree	Other Advantage
1. Prefix B-Tree	storage utilization	reduced height,fewer disk accesses
2. Multi-Dimensional B-Tree	efficient associative retrievals	dynamic structure
3. R-Tree	dynamic structure, handles spatial searches for n-dimensional data	integrates pictorial and alphanumeric databases and searches
4. Write-Once B-Tree	past values of data records are retained and readily assessable	can effect storage utilization by user defined parameters
5. Linked B-tree	a robust tree, detects and corrects some pointer errors	easier to implement, more efficient than other robust trees
6. B-Link Tree	concurrent version, maximum of 3 locks	similar algorithms to basic B^+-tree
7. PO-B tree	concurrent version, maximum of 2 locks	similar algorithms to basic B^+-tree

B^+-Tree Variant	Major Disadvantage Versus B^+-Tree	Other Disadvantage
1. Prefix B-tree	costly algorithms to find optimal separator	none
2. Multi-Dimensional B-tree	performance and height affected by distribution of the attribute values	high retrieval costs for some partial matches
3. R-tree	no guaranteed worst-case search time	potential for dead spatial search space
4. Write-Once B-tree	larger storage requirement versus normal B^+-tree	longer search time within nodes since keys not ordered nor unique
5. Linked B-Tree	costly detection and correction algorithm	overhead from extra pointers
6. B-Link Tree	inefficient use of pages in main memory	overhead from additional pointers
7. PO-B tree	greater overhead from the more frequent splits and concatenations	splits and concatenations may be unnecessary

7.4 Thoughts on the future of B-trees

We forsee that new variants of the B-tree will continue to be created. The more successful ones will be those that retain the dynamic reorganization ability of the B-tree and the efficient updating algorithms. The added complexity of new variants will continue to be justifiable only by a true gain over some aspect of the basic B-tree.

Recent variants are primarily of the B^+-tree variety, having separate index versus data nodes. We forsee this trend to continue. In the case of the Write Once B-tree, this feature was used to maximize storage utilization and updating efficiency. One order was set for the index nodes and a smaller order was set for data nodes. (Order refers to the maximum number of entries per node.) This allowed the index portion of the tree to be denser than the data portion where most insertions take place. Other advantages for B^+-tree variants include easier sequential access to data and less complicated deletion algorithms.

We forsee that new concurrency work will follow the recent optimistic approach. New ways to monitor concurrent users without expensive locking still need to be developed.

We predict that eventually multidimensional B-trees will completely replace older structures such as inverted files. As older software system are replaced, we feel the dynamically defined Multi-Dimensional B-tree (or some future version of it) will replace inverted files for most applications.

With the advent of huge, reasonably priced external memory, storage utilization is not as high a priority. B-trees will continue to be valued with their 69% average storage utilization and for their small heights. We forsee research efforts to develop variants that improve storage utilization to lessen.

Huge internal memories are becoming available on newer computer systems. This will allow further work to be done on the optimistic concurrency model and on other algorithms for B-tree variants which require users to have private (local) copies of index pages in main memory. With limited internal memory, these private copies are expensive.

Most B-tree implementations store the root node in main memory and fetch other pages as needed. With large internal memories perhaps the root and first level nodes could permanently reside in main memory. Perhaps new page swapping strategies based on B-tree properties and expanded main memory can be developed in the future.

The new high density optical storage and advanced magnetic

recording will perhaps create a class of B-tree variants modeled in part after the Write-Once B-tree. The idea of having past versions of the database on line will become even more feasible and perhaps more standard as storage mediums improve.

We feel much future work will be done using B-trees to facilitate the efficient integration of graphics and text databases. The recent work with R-trees provides a promising start. This area is of growing importance as visual images are becoming easier to store, but not to categorize or retrieve.

An exciting feature of B-trees is that as new technologies and application areas arise, this structure is usually appropriate in some modified form. This has been true in the twenty years since the B-tree was originally defined and no major flaw has been discovered which would lead one to believe it will not continue to be true for decades to come.

A.1 Mathematical Details of B-Tree Average Storage

A.1.1 Symbol Table and Facts

Symbol Table : μ = storage utilization
r = number of keys (items) in the tree
n = number of nodes of the tree
d = order (minimum number of keys/node)

Facts:

$d \leq r/n \leq 2d$ — The average number of keys per node (r/n),

$$n = \frac{r}{(\text{ ave \# of keys/node})}$$ — The number of nodes in the tree (n) is approximated by the number of keys in the tree divided by the average number of keys per node.

$r/2d \leq n \leq r/d$ — The maximum number of nodes (n) in the tree will occur when all the nodes except the root are half full. For large r this is r/d. Likewise, the minimum value of n will occur if the nodes are full (r/2d).

$2dn = \text{total storage capacity}$ — The total storage capacity is the maximum number of keys per node (2d) times the number of nodes in the tree, n.

$\mu = r / 2dn$ — Storage (μ) is defined as the number of items in the tree (r) divided by the total storage capacity. $0 \leq \mu \leq 1$

$E(\mu) = E(r/2dn) = r/2d * E(1/n)$ — Assuming the number of keys (r) and the order (d) to be fixed, to complete the calculation of $E(\mu)$ only requires computing E(1/n).

$$E(y) = \frac{1}{b-a}\int_a^b y\, dy$$ — The expected (average) value of a continuous function over a closed interval [a,b] is computed by evaluating this definite integral.

For large orders and number of keys in the tree it is not an unreasonable assumption that the function 1/n is continuous over its defined interval [r/2d, r/d]. Therefore, the above average value formula can be used to find E(1/n).

$$E(1/n) = \frac{1}{r/d - r/2d}\int_{r/2d}^{r/d} (1/n)\ dn$$

A1.2 Calculations

$$E(1/n) = \frac{1}{r/d - r/2d} \int_{r/2d}^{r/d} (1/n)\, dn$$

$$= \frac{2d}{r} \ln n \Big|_{r/2d}^{r/d}$$

$$= \frac{2d}{r} \left[\ln(r/d) - \ln(r/2d) \right]$$

$$= \frac{2d}{r} \left[\ln(r/d) - \ln(r/d) + \ln(2) \right]$$

$$E(1/n) = \frac{2d}{r} * \ln 2$$

$$\text{So, } E(\mu) = \frac{r}{2d} E(1/n) = \frac{r}{2d} * \frac{2d}{r} * \ln 2 = \ln 2$$

A1.3 Generalized Calculations and Results

To compute E(1/n) it is necessary to know the interval on which n is defined . The values of n (the number of nodes in the tree) are determined by the order of the tree and the required minimum fullness factor f, of each node.

If all nodes are full, the minimal value of n is achieved, r/2d. If all nodes are at least f% full, then the maximal value of n is r/2fd. In general, the interval for which n is defined is [r/2d, r/2fd]. For B-Trees the minimal fullness factor is .50 and the interval is [r/2d, r/d] as was used in the earlier worked out problem.

So in general,

$$E(1/n) = \frac{1}{r/2fd - r/2d} \int_{r/2d}^{r/2fd} 1/n \; dn$$

$$= \frac{2fd}{n(1-f)} \left[\ln(r/2fd) - \ln(r/2d) \right]$$

$$= \frac{2fd}{n(1-f)} \left[\ln(r/2d) - \ln(f) - \ln(r/2d) \right]$$

$$E(1/n) = \frac{2fd}{n(1-f)} \ln(1/f)$$

So, $$E(\mu) = \frac{r}{2d} * \frac{2fd}{n(1-f)} * \ln(1/f)$$

$$= \frac{f}{(1-f)} * \ln(1/f)$$

Note: For a B*-Tree with each node at least 2/3 full, the value of f = .67 and $E(\mu)$ = .811 using the above formula.

A1.4 Weakness of this derivation

In a recent paper [GUPT] two weaknesses of Leung's inituitive derivation of approximate storage utilization in B-trees are discussed. The weaknesses are:

1) The number of nodes, n, is not a continuous variable as Leung says he can "safely" assume for large n.

2) The assumption of a uniform distribution for n is inappropriate.

To correct the first weakness, integration is replaced by summation. The more cumbersome calculations (outlined in the paper) result in an extra term in the expression for the expected storage utilization, $E(\mu)$.

$$E(\mu) = \frac{f}{1 - f} * \left[\ln (1/f) + (d/r)*(f+1) \right]$$

The extra term, $(d/r)*(f+1)$, is insignificant only when d (order) is much smaller than r (the number of keys in the tree). This extra term also implies that one could improve storage utilization by decreasing the number of keys in the tree (r); a fact that is not supported by empirical studies.

The second weakness is based on Leung's assumption that n, the number of nodes in the tree, is uniformily distributed between the minimum and maximum number of nodes possible in a tree of given order and number of keys. In practice, the value of n is almost always nearly equal to the expected value of n. A formula for the probability of B-trees with r keys having n_i nodes is given. A chart with the probabilities calculated for each n_i between the maximum and minimum values of n clearly demostrates a non-uniform distribution of n.

A.2 Basic B-Tree Primer

A.2.1 Find Algorithm

The searching technique for the B-tree is an extension of a binary search. Starting at the root node, the key value to be found, x, is compared with the keys in the node until the first key value, k_i greater than or equal to x has been located. If $x=k_i$ then the search has successfully terminated. If $x > k_i$ then the search continues in the root node of the left subtree of k_i. Notice that all values between k_{i-1} and k_i (including x, if x is in the tree) are found in the subtree pointed to by p_{i-1}. Refer to Figure 1 for clarification of the notation used above. The search is then continued recursively at lower levels in the tree until either the key value has been found or a leaf node is reached, in which case the search has been unsuccessful.

Most B-trees have large orders which result in many key values per node to be searched. Appropriate techniques can be employed within nodes to find key values such as a binary search.

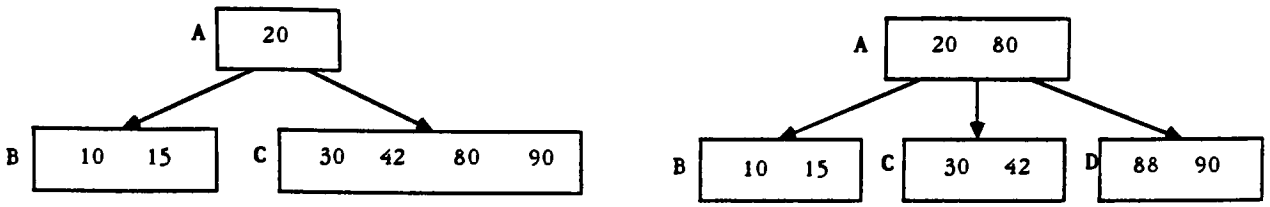
A.2.2 Insertion Algorithm

When inserting keys into a B-tree there is the potential for nodes to overflow, that is contain more than the maximum number of keys allowed per node. In this situation the B-tree will reorganize by splitting nodes.

To insert a key in a B-tree of order d (where d is the minimum number of keys per node) first use the find algorithm to locate the position in the leaf level the new key is to be inserted. If the leaf node has less than 2d keys (and is therefore not full) insert the new key in its proper position in the node. If the leaf node is full before the insertion, then a split is done involving the 2d keys from the full node and the one new key. A new node is allocated into which the highest d keys are placed. The lowest d keys remain in the old node and the middle key value is promoted to the next higher level as a separator for the new and old leaf nodes.

If this separator key causes an overflow in the parent node then this process is repeated at that level. In the worst case the splits will propagate to the root level causing the tree to increase in height by one. Detailed examples of insertion are found in many sources [BAYEb, CHATa, COME, KNUT, LOOM, WIRT].

Figure 23 shows a B-tree of order d = 2, before and after inserting the key value of 88. Placing 88 in node C causes an overflow as no more than 2d or 4 values are allowed. The middle value of 80 is identified and promoted to the root node A.



a) before insertion

b) after inserting 88

Figure 23: Insertion for a B-tree of Order d=2

A.2.3 Deletion Algorithm

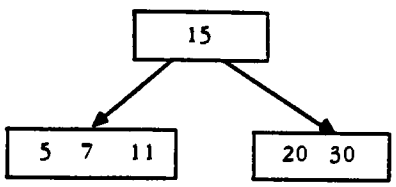
When deleting keys of B-trees there is the potential of a node underflowing, that is ending up with fewer than the minimum number of keys ,d, required by definition. In this situation either a redistribution of the keys in existing nodes will occur or the B-tree will reorganize by concatenating nodes. Concatenation is the inverse of the splitting operation for insertion.

To delete a key in a B-tree of order d (the minimum number of keys per node) first use the find algorithm to locate the node in which the key resides. If the node is a leaf that is more than minimally full (has d+1 or more keys) then the deletion can take place with no underflow.

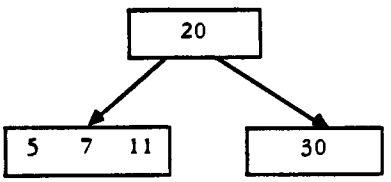
If the node is not a leaf then the key to be deleted functions as a separator for two subtrees at the next lower level. Key values must therefore be shifted even if underflow does not occur. This is accomplished by using the leftmost leaf of the right subtree of the key being deleted. This ensures that the next highest number is used as the new separator value.

When the deletion of a key has been completed, changed nodes must be checked for underflow. If underflow is detected in a node, keys are borrowed from sibling nodes on the left or right. This redistribution results in both nodes having roughly the same number of keys. If there are less than 2d keys to redistribute then the nodes are concatenated using (and thus removing) their separator key from the next higher level. In the worst case the concatenation propogates to the root. Detailed examples are found elsewhere [BAYEb, CHATa, COME, KNUT, LOOM, WIRT].

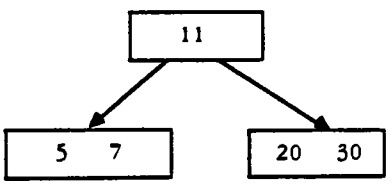
Figure 24 shows a B-tree of order 2 before, during and after deleting the key value of 15. Since 15 was a separator value in the root node A, the next largest key (20, found in node C) replaces it. This causes node C to underflow. Note that a single key value is only allowed in a root node. The left brother of node C is checked for an extra key. Since B has one more than the minimum required, the key values are redistributed by shifting them one to the right through the root node. In this situation, concatenation (resulting in a single node) would not work as there is one key too many in the three nodes.



a) before deletion of 15



b) underflow during deletion



c) after deletion

Figure 24: Deletion from a B-tree of Order d=2

A.2.4 Costs

A.2.4.1 Look-up Costs

The cost of retrieving a key depends on the number of nodes accessed in the tree during the search. Assuming the root node resides in main memory, in the worst-case a total of h nodes are read where h is the height of the tree. The minimum and maximum height values for a B-tree are $\log_{2d+1}(N+1)$ and $\log_{d+1}((N+1)/2)$ respectively where N is the number of keys in the tree and d is the minimum number of keys per node. The worst case complexity of the Find algorithm is therefore $O(\log_d N)$.

A derivation of the worst-case number of nodes to be accessed is as follows. Given a B-tree with N keys, at level L there will be N+1 leaves. At level one there must, by definition, be at least two pointers (unless it is a leaf node itself). Since each node after the root must have a minimum of (d+1) pointers,

at levels 2, 3,... L there are at least $2(d+1)$, $2(d+1)^2$....$2(d+1)^{L-1}$ nodes. Since at level L there is at least $2(d+1)^{L-1}$ nodes, the following inequality holds.

$$N + 1 \geq 2(d+1)^{L-1}$$

$$(N+1)/2 \geq (d+1)^{L-1}$$

$$L - 1 \leq \log_{(d+1)}((N+1)/2)$$

$$L \leq \log_{(d+1)}((N+1)/2) + 1$$

To illustrate how few accesses must be made in the worst-case to find a node, consider a B-tree of order d=100 with two million keys (N = 2,000,000). According to the above inequality:

$$L \leq \log_{101}(2,000,001/2) + 1$$

$$L \leq 3.9935$$

Therefore, at most 3 levels are acessed to find any key in this example B-tree.

Using the Find algorithm, up to $2d/(2d+1)$ of the remaining keys are no longer considered each time a pointer takes the search down one level in the B-tree. Using the example B-tree from above with two million keys and order d = 100, this means that a choice at any given level eliminates up to 99.5% (200/201) of the remaining keys in the tree.

The average search path length according to [WRIG] is approximately logarithmic to the base $(2d*\ln 2)$. The larger the order of the tree, the shorter the tree becomes and the less expensive the search.

A.2.4.2 Insertion Costs

Insertions are made at the leaf level of the B-tree so the full height of the tree must be traversed each time. In the best case no splits are needed and only the changed leaf node must be written back to secondary storage. In the worst case the splits reach the root page. This would necessitate $2h + 1$ writes of the changed nodes.

A more important number is the expected number of splits per insertion which is less than 1/d, where d is the minimum number of keys per node. More recent work [WRIG] shows that for large orders this number is approximately $1/(2d*\ln 2)$. For orders of 1000 and 1,000,000 the average number of splits per insertion are .00072 and .0000007 respectively. The major costs for insertions is thus dependent on the height and order of the tree. Insertion algorithms are of complexity proportional to

$\log_d(N)$.

A.2.4.3 Deletion Costs

As with insertions, the cost of deletions depends on the number of accesses made to the secondary storage where nodes are stored as pages. The deletion of a key in a leaf node, causing no underflow or concatenation causes the lowest number of fetches (h) and writes (1). The worst case occurs when all but the first two pages in the retrieval path are concatenated, the son of the root has underflowed and the root is changed.

Each concatenation causes an additional fetch (to get the brother node) and two additional writes (of the new combined node and the parent node with one less separator key value). A table of costs of a single retrieval, insertion and deletion under variations of insert/delete conditions, with/without overflow techniques is given in [BAYEb].

A.2.5 Storage Utilization

Although storage can be as low as 50% if all nodes of the B-tree are minimally full, this is not common. With a simple overflow technique described in Section 2.5, this worst case can be improved to 67%.

It has been proven using a variety of analytic techniques that the average storage utilization of B-trees is ln 2 (approximately 69%). An intuitive derivation of this fact is presented in the Appendix (A.1). This value has been confirmed empirically. According to experiments by Arnow and Tenebaum, the storage utilization of B-trees varied between 67-71% and showed no dependence on the order of the tree.

Bibliography

[ARNO] Arnow, D.; Tenenbaum, A., "An empirical comparison of B-Trees, Compact B-Trees and Multiway Trees", ACM SIGMOD Record, Vol. 14, No. 2, June 1984, p. 33-46.

[BAYEa] Bayer, R.; McCreight, E., "Organization and maintenance of large order indexes", Acta Informatica, 1972, p. 173-189.

[BAYEb] Bayer, R.; Schkolnick, M., "Concurrency of operations on B-trees", Acta Inf., Vol.9, No.1, 1977, p. 1-21.

[BAYEc] Bayer, R.; Unterauer, K., "Prefix B-Trees", ACM Trans. Database Systems, Vol. 1, No. 3, Sept 1976, p. 268-75.

[CHATa] Chaturvedi, A., "Tree structures I", PC Technical Journal, Vol. 3, No. 2, Feb. 1985, p. 78-87.

[CHATb] Chaturvedi, A., "Tree structures II", PC Technical Journal, Vol. 3, No. 3, Mar. 1985, p. 131-158.

[COME] Comer,D., "The ubiquitous B-tree", Computing Surveys, Vol. 11, No. 2, June 1979, p.122-137.

[CULI] Culik, K. , II; Ottmann, T.; Wood, D., "Dense multiway trees", ACM Trans. Database Systems, Vol. 6, No. 3, Sept. 1981, p. 486-512.

[DESA] Desai, B.C.; Goyal, P.; Sadri, F., "Use of composite index in DDBMS", ACM Computer Science Conference '86 Proceedings, 1986, p.251-60.

[DIEH] Diehr,G.; Faaland,B., "Optimal pagination of B-Trees with variable length items", Communications of the ACM, Vol. 27,No. 3, March 1984, p. 241-247.

[DRIS] Driscoll,J.R.; Sheau-Dong Lang; Bratman, S.M., "Achieving minimum height for block split tree structured files", Inf. Systems, Vol. 12, No. 1, 1987, p.115-124.

[EAST] Easton, M.C., "Key-sequence data sets on indelible storage", IBM J. of Research and Development, Vol. 30, No. 3, May 1986, p.230-41.

[ELLI] Ellis, C.S., "A hybrid solution for concurrent operations on B-trees", unpublished paper, March 1980.

[HUAN] Huang, Stephen, "Height-balanced trees of order (Beta, Gamma, Delta), ACM Transactions on Database Systems, Vol. 10, No. 2, June 1985, p. 261-284.

[HURS] Hurson, A.R.; Miller, L.L., "A B-tree parallel processor for information retrieval", Proceedings of the ISMM Inter. Symp: Mini and Microcomputers and their Appli., Acta Press, 1985, p. 39-43.

[GUPT] Gupta, G.K.; Srinivasan, B., "Approximate storage utilization of B-trees", Inf. Process. Letters, Vol. 22, No. 5, April 1986, p. 243-6.

[GUTT] Guttman, A., "R-Trees: a dynamic index structure for spatial searching", ACM SIGMOD, 1984, p.47-57.

[KERS] Kerstein, M.L.; Tebra, H., "Application of an optimistic concurrency control method", Software Practice and Experience, Vol. 14, No. 1, Feb. 1984, p. 153-68.

[KLON] Klonk, J., "Comments on optimality of B-Trees", ACM SIGMOD Rec., Vol. 13, No. 2, Jan. 1983, p. 35-8.

[KNUT] Knuth, D., The Art of Computer Programming, Vol 3: Sorting and Searching, Addison Wesley, Reading, Mass., 1973, p. 473-80.

[KOES] Koesler, P.; Ottmann, T., "An experimental study of insertion schemes for classes of multiway search trees", International Journal of Computing Math., Vol. 9, No. 3, 1981, p. 185-93.

[KRIE] Kriegel, H., "Performance comparison of index structures for multikey retrieval", ACM SIGMOD Rec., Vol. 14, No. 2, June 1984, p. 186-96.

[KUNG] Kung, H.T.; Robinson, J.T., "On optimistic methods for concurrency control", ACM Trans. on Database Systems, Vol. 6, No. 2, June 1981, p. 213-26.

[LANI] Lanin, V.; Shasha, D., "A symmetric concurrent B-tree algorithm", ACM 1986 Proceedings of the Fall Joint Computer Conference, 1986, p. 380-389.

[LAUS] Lausen, G., "Integrated concurrency control in shared B-trees", Computing (Austria), Vol. 33, No. 1, 1984, p. 13-26.

[LEHM] Lehman, P.L.; Yao, S.B., "Efficient locking for concurrent operations on B-trees", ACM Trans. Database Systems, Vol. 6, No. 4, Dec. 1981, p. 650-70.

[LEUN] Leung, C.H.C., "Approximate storage utilization of B-Trees: a simple derivation and generalizations", Inf. Process. Letters, Vol. 19, No. 4, Nov. 1984, p. 199-201.

[LIBE] Libera, F.D.; Gosen, F., "Using B-trees to solve geographic range queries", Comput. J., Vol. 29, No. 2, April 1986, p. 176-81.

[LOME] Lomet, D. B., "Prefix* B-Trees", IBM Technical Disclosure Bulletins, Vol. 24, No. 5, Oct. 1981, p.2492-6.

[LOME] Lomet, D.B., "Partial expansions foe file organizations with an index", ACM Trans. Database Sys., Vol. 12, No. 1, March 1987, p. 65-84.

[LOOM] Loomis, M., Data Management and File Processing, Prentice Hall, Englewood, NJ, 1983, p. 214- 241.

[McCR] McCreight, E. M., "Pagination of B*-trees with variable-length records", Communications of the ACM, Sept. 1977, p. 670-674.

[MOND] Mond, Y.; Raz, Y., "Concurrency control in B+-trees databases using preparatory operations", Very Large Data Bases. Proc. of the Eleventh Inter. Conf., 1985, p.331-4.

[OKAM] Okamura, Y.; Sato, T., "An interactive symbolic cell layout system with high speed relational data base", IEEE Inter. Conf. on Computer-Aided Design, 1986, p.486-9.

[OUKS] Ouksel, M.; Scheuermann, P., "Multidimensional B-Trees: analysis of dynamic behavior", BIT, Vol. 21, No. 4, 1981, p. 401-18.

[QUIT] Quitzow, K.H.; Klopprogge, M.R., "Space utilization and access path length in B-Trees", Information Systems, Vol. 5, No. 1, 1980, p. 7-16.

[ROSE] Rosenberg, A. L.; Snyder, L., "Time- and space- optimality in B-Trees", ACM Trans. Database Systems, Vol. 6, No. 1, Mar. 1981, p. 174-83.

[ROUS] Roussopoulos, N.; Leifker, D., "Direct spatial search on pictorial databases using packed R-trees", ACM SIGMOD Rec., Vol. 14, No. 4, Dec. 1985, p. 17-31.

[SAGI] Sagiv, Y., "Concurrent operations on B-trees with overtaking", J. Comput. & Syst. Sci., Vol. 33, No. 2, Oct. 1986, p. 275-96.

[SAMA] Samadi, B., "B-trees in a system with multiple users", Inf. Process. Letters, Vol. 5, No. 4, Oct. 1976, p. 107-12.

[SCHE] Scheuermann, P.; Ouksel, M., "Multidimensional B-Trees for associative searching in database systems", Information Systems, Vol. 7, No. 2, 1982, p.123-37.

[SPIR] Spirn, J.R.; Tsur, S., "Memory management for B-trees", Performance Evaluation, Vol. 5, No. 3, Aug. 1985, p. 159-74.

[TAYL] Taylor, D.J.; Black, J.P., "A locally correctable B-Tree implementation", The Computer Journal, Vol. 29, No.3, 1986, p. 269-276.

[WIRT] Wirth, N., Algorithms + Data Structures = Programs, Prentice Hall, Englewood, NJ, 1976.

[WRIG] Wright, W.E., "Some average performance measures for the B-Tree", Acta Informatica, Vol. 21, No. 6, 1985, p.541-57.