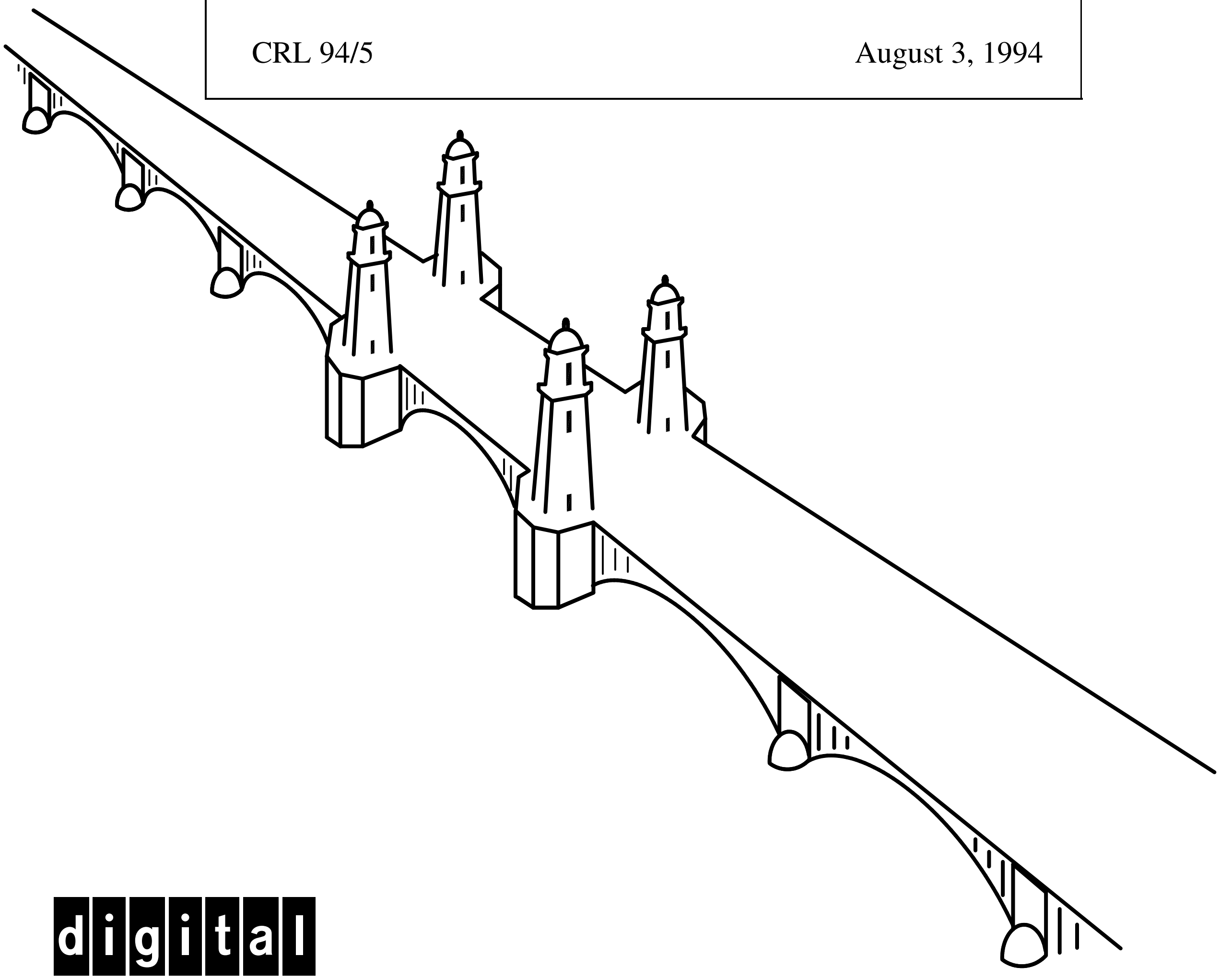
The hB$^{\Pi}$-tree: A Concurrent and Recoverable Multi-attribute Access Method

Georgios Evangelidis David Lomet
Betty Salzberg
Digital Equipment Corporation
Cambridge Research Lab

CRL 94/5 August 3, 1994



digital

CAMBRIDGE RESEARCH LABORATORY
Technical Report Series

Digital Equipment Corporation has four research facilities: the Systems Research Center and the Western Research Laboratory, both in Palo Alto, California; the Paris Research Laboratory, in Paris; and the Cambridge Research Laboratory, in Cambridge, Massachusetts.

The Cambridge laboratory became operational in 1988 and is located at One Kendall Square, near MIT. CRL engages in computing research to extend the state of the computing art in areas likely to be important to Digital and its customers in future years. CRL's main focus is applications technology; that is, the creation of knowledge and tools useful for the preparation of important classes of applications.

CRL Technical Reports can be ordered by electronic mail. To receive instructions, send a message to one of the following addresses, with the word **help** in the Subject line:

On Digital's EASYnet:	CRL::TECHREPORTS
On the Internet:	techreports@crl.dec.com

Cambridge Research Laboratory
One Kendall Square
Cambridge, Massachusetts 02139

The hB$^{\Pi}$-tree: A Concurrent and Recoverable Multi-attribute Access Method[1]

Georgios Evangelidis [2] David Lomet
Betty Salzberg [3]
Digital Equipment Corporation
Cambridge Research Lab

CRL 94/5 August 3, 1994

Abstract

We describe a new access method, the hB$^{\Pi}$-tree, an adaptation of the Lomet and Salzberg hB-tree index to the constraints of their Π-tree . The Π-trees, a generalization of the B^{link}-trees of Lehman and Yao, provide very high concurrency with recovery, hence permitting highly parallel access to data. The hB-tree is a multi-attribute index which is highly insensitive to dimensionality. The combination makes the hB$^{\Pi}$-tree suitable for inclusion in a general purpose database management system supporting multi-attribute and spatial queries.

Keywords: concurrency, recovery, indexing, access methods, B-trees

[1]This work was partially supported by NSF grant IRI-91-02821 and IRI-93-03403.
[2]College of Computer Science, Northeastern University, Boston, MA 02115
[3]College of Computer Science, Northeastern University, Boston, MA 02115

1 Introduction

Current DBMSs efficiently organize, access, and manipulate enormous quantities of data for traditional applications in banks, airlines, government agencies, hospitals, and other large organizations. Almost all of them implement some variation of the B$^+$-tree [BM72, Com79] which is the best choice for ordered single-attribute indexing.

However, today new non-traditional applications, with growing mountains of data, require innovative solutions to storage and access problems. These include scientific applications such as those proposed for the terabytes of meteorological, astronomical and geographic data streaming in daily from satellites. This data must be organized spatially in terms of latitude and longitude and height above the earth, for example, rather than linearly in terms of one attribute.

Unfortunately, current general purpose DBMSs do not support multi-attribute and spatial indexing. Spatial organization of large databases is a largely unsolved problem. Even the research literature does not offer many viable solutions which would stand up to arbitrary collections of data. There have been a number of proposals for multi-attribute and spatial indexing in the past 15 years (for example, [Gut84, NHS84, Gue89, SRF87, LS90]), but none of them has been integrated into a commercial general purpose DBMS. Two of the most important reasons for this are:

1. lack of performance guarantees, and

2. very complicated or no concurrency and recovery methods for them.

Concurrency in B$^+$-trees has been the subject of many papers [BS77, LY81, SG88, ML89, LS92]. Most of these papers, with the exception of [ML89, LS92], have not addressed the problem of system crashes during structure changes.

We introduce a new access method for multi-attribute point or spatial data that is based on the hB-tree [LS90] and can be used in a general purpose DBMS since a robust concurrency and recovery algorithm is available for it [LS92]. This algorithm is applicable to an abstract index tree structure, the Π-tree, which is a generalization of the B^{link}-tree [LY81]. A recent study [SC91] compared the performance of various concurrency control algorithms. Its conclusion was that algorithms using the link technique provide the most concurrency and the best overall performance.

We modify the hB-tree so that it becomes a special case of the Π-tree. This involves structural changes to the hB-tree, and invention of new node splitting, index term posting, and node deletion algorithms. Once this is accomplished, the concurrency and recovery algorithm of [LS92] is applicable on the resulting method, which we call the hB^{Π}-tree.

We have implemented the hB^{Π}-tree and have tested it in extensive experiments with computer-generated skewed point data and with point data from the Sequoia 2000 Storage Benchmark [SFGM93]. We also show that the hB^{Π}-tree is fairly insensitive to dimension. This property makes it suitable for k-dimensional spatial data that is mapped to $2k$-dimensional point data.

This paper is organized as follows. Section 2 briefly reviews the Π-tree. Section 3 introduces the hB^{Π}-tree, which is a combination of the hB-tree and the Π-tree. Finally, Section 4 reports performance results and explains how and why the hB^{Π}-tree can be used as a spatial data index.

2 Concurrency and Recovery: The Π-tree

An abstract index, the Π-tree, and a well-understood and robust concurrency and recovery algorithm for it are presented in [LS92]. In this section we review the Π-tree and show why it achieves a high degree of concurrency and meshes well with various recovery schemes. These are essential properties an index must have in order to be suitable for general purpose Database Management Systems.

2.1 Π-tree Structure

As a generalization of the B^{link}-tree [LY81], the Π-tree is a rooted DAG. It consists of **index** and **data** nodes. Each node is responsible for a specific part of the key space. A Π-tree node:

- Can be **directly responsible** for some part of the space. In an index node, this space is distributed among its children nodes and is described by **index terms**. In a data node, existing and potential data points lie in this space.

- Can also **delegate responsibility** for part of the space to sibling nodes. This space is described by **sibling terms**.

The index and sibling terms include pointers to Π-tree nodes. The pointers to sibling nodes are the links that make the Π-tree a generalization of the B^{link}-tree.

In the Π-tree it is possible for a node to be referred to by more than one parent, unlike the B^{link}-tree. This happens whenever the boundary of a parent split cuts across a child boundary. This child is called a **multi-parent** node. Nodes with only one parent are **single-parent** nodes.

2.2 Searching

For exact match searches, a unique path, that may include sibling-pointers, is followed down to the leaf (data) level where the point in question will reside if it exists at all. That is, exact match searches are precisely analogous to those in the B^{link}-tree.

2.3 Node Splitting and Index Term Posting

A Π-tree node is split when an insertion causes it to overflow. Part of the node's contents go to a new sibling node and an index term that describes the resulting space decomposition is posted to the parent of the split node. Π-tree node splitting is analogous to B-tree node splitting.

In the Π-tree **node splitting** and the **index term posting** are performed by **separate recoverable atomic actions** [1], as follows:

Node splitting: A Π-tree node is full and cannot accommodate an update. This node, called the **container node**, is split and part of its contents are moved to a newly created node, called the **extracted node**. Node splitting concludes by storing a sibling term in the container node (see Figures 1a and 1b).

Index term posting: An index term, that describes the space that was extracted from the container, is posted to the parent of the container in the current search path (see Figure 1c). An index term posting atomic action always posts to a single parent. When the container node is a multi-parent node, index posting may consist of several separate index term posting atomic actions (one for each parent).

[1] Like transactions, atomic actions are atomic, and isolated. Unlike transactions, since there is no communication of the changes to users, they need not be durable. Typically, these are made durable by the commitment of transactions that use their results.

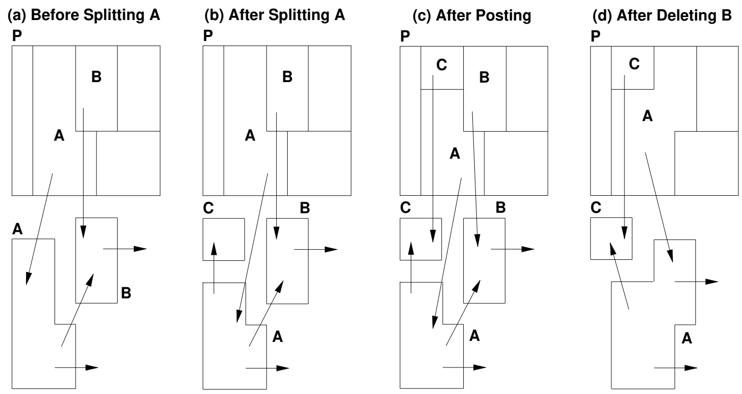


Figure 1: Splitting, Posting, and Consolidating in the Π-tree.

When Π-tree restructuring is interrupted by system failures, the Π-tree is left in a consistent state. Searchers can always traverse or visit an extracted node by going through its container node and following the sibling pointer. The same is true when, in the case of a multi-parent container node, only some of the index term posting atomic actions have been performed. That is, two instances of the Π-tree can be *structurally different* but *semantically equivalent*. An index term posting atomic action for the missing index term is scheduled whenever a sibling pointer is traversed.

2.4 Node Consolidation

A Π-tree node whose space utilization drops below a pre-specified threshold should be consolidated with another node (its container node or one of its extracted nodes) in order to improve the overall storage utilization.

In the Π-tree we always move the contents of an extracted node to its container node and we deallocate the extracted node. Also, all references to the deallocated node must be removed from its parent(s) by the same atomic action. This is in contrast to index term posting which can be performed using multiple independent atomic actions.

Two conditions are required for node consolidation: (a) both the container and the extracted nodes must be children of the same parent, and (b) the extracted node must be a single-parent node. These conditions simplify node consolidation and increase the degree of concurrency since only one parent node needs updating during a consolidation. They also ensure that no other tree traversal operation will reference the node being deleted via a different path.

In Figure 1c, we assume that node B is sparse and can be consolidated with node A since both A and B have P as parent. A absorbs B's contents, the reference to B is removed from P, and the index term for A is adjusted (Figure 1d).

2.5 Recovery Issues

The above approach for concurrency works very well with various recovery schemes as long as they provide two essential requirements. First, the write-ahead logging protocol (WAL protocol) must be used to ensure that the atomic actions are actually atomic, i.e., they have the all-or-nothing property. Second, the transaction manager must know about atomic actions in the sense it knows about database or system transactions. This is necessary because the transaction manager has to provide the atomicity property by possibly rolling back (undoing) incomplete executions of atomic actions [LS92].

3 The hB-tree as a Π-tree

A new access method which is suited for multi-attribute data cannot be used in a general purpose DBMSs unless it uses well-understood and robust concurrency and recovery methods. We have modified the hB-tree so that it becomes a special case of the Π-tree [LS92], which we call an hB^{Π}-tree. Then, the efficient algorithm for concurrency and recovery of [LS92] is applicable on it.

3.1 Multi-attribute Indexing with the hB-tree

The hB-tree [LS90] is a multi-attribute point data indexing method with good storage utilization. Its inter-node and growth processes are analogous to the corresponding processes in B-trees. In this section, we describe the structural modifications that transform the hB-tree into the hB^{Π}-tree. We present simple and efficient tree restructuring algorithms in the next section.

The nodes of an hB-tree represent bricks (i.e., multi-dimensional rectangles), or "holey" bricks, that is, bricks from which smaller bricks have been removed. An hB-tree node stores index and sibling terms in a unified way using kd-trees [Ben79].

In Figures 2a and 2b we can see the intra-node organization of an index hB-tree node Q and the corresponding space decomposition. Each path (from the root to a leaf) in the kd-tree of node Q describes either the space of a sibling node or part of the space of a child node. For example, the path ($x1$-left, $y1$-left) describes space that has been extracted from Q (shaded region of Figure 2b). The remaining four paths in the kd-tree of Q describe the decomposition of the space node Q is directly responsible for among its children, namely nodes K, L, and M (white regions of Figure 2b).

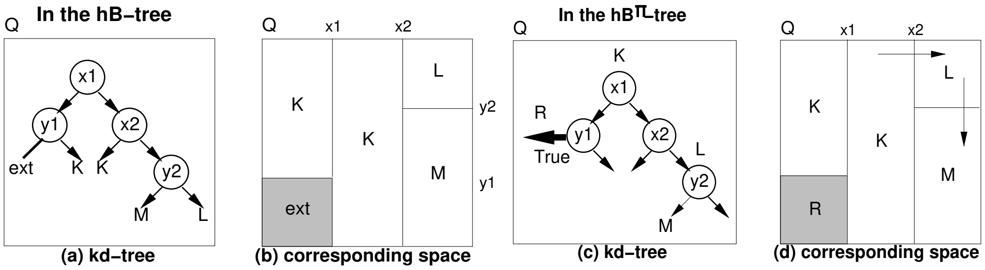


Figure 2: Intra-node organization of hB-tree and hB^{Π}-tree nodes using kd-trees.

In order to transform the hB-tree into a case of the Π-tree we need to: (a) know the actual address of an extracted node, (b) be able to determine the containment order of the children of an index node (this simplifies the deletion algorithm), and (c) have a means to detect whether a node is multi-parent or not by examining the kd-tree of its current parent. We describe four important structural modifications of the hB-tree which transform it into the hB^{Π}-tree.

3.2 Side-pointers

We replace the external markers by pointers to the extracted nodes, called **side-pointers**. This is an important modification and transforms the hB-tree into a subcase of the Π-tree. In Figure 2c the thick arrow with the address of node R represents the side-pointer that is now used in the place of the external marker. The address of R (the node that was extracted from Q) is known at the time of Q's splitting.

3.3 Splitting a Node at its kd-tree root

Another important modification is the way node splitting is done, when the kd-tree of the node is split at its root. In the hB-tree, one of the subtrees remains in the node that is split, and the other one becomes the kd-tree of the newly allocated node. The original kd-tree root disappears and, as has been described earlier, there is no pointer to the extracted node. For example, in Figure 2a, if hB-tree node Q is split at its kd-tree root $x1$ and the kd-subtree rooted at $x2$ is extracted, kd-tree node $x1$ is eliminated.

In the hB$^{\Pi}$-tree we keep the kd-tree root in the original node and we simply extract the appropriate kd-subtree which again becomes the kd-tree of the new hB$^{\Pi}$-tree node. The new node is now the extracted node, whereas the original node is the container node. This modification is necessary, because in the Π-tree a node that is split (container node) continues to keep information that describes the key space it is responsible for. For example, in Figure 2c, if the kd-subtree rooted at $x2$ is extracted, kd-tree node $x1$ remains in Q and its right child becomes a side-pointer to the extracted node.

3.4 Decorations

The third modification deals with the way the addresses (pointers) of child hB$^{\Pi}$-tree nodes are stored in the kd-tree of their parent. In the hB-tree we may have multiple references to a child in a node's kd-tree (for example, in Figure 2a K's address is stored twice). Every leaf node of the parent's kd-tree that refers to data directly contained in a child node contains a pointer to the child.

In the hB$^{\Pi}$-tree, we instead identify the node within the kd-tree that is the root of the subtree that describes the space for which a child node is responsible, both directly and via delegation. That subtree root is then tagged with the address of the child node. We use the term **decoration** for this child address. We call this subtree the **decorated subtree**. Specific instances of decorated subtrees are referred to using their decorations. For example, in Figure 2c, the decorated subtree rooted at $x1$ is referred to as the K-subtree. The decorated subtree rootted at $y2$ is the L-subtree. Decorated subtrees are nested.

Leaf nodes now contain one of three kinds of information:

1. a child node address: no index term has been posted for any extracted siblings of this child and the path to this kd-tree leaf node describes the

space for which the child is responsible. In this case the subtree for the child is degenerate.

2. a sibling node address: the index node itself has been split and the path to this kd-tree leaf node describes the space delegated to the sibling index node.

3. a null address: the path to this kd-tree leaf describes space for which the child node that decorates the smallest subtree including this node is directly responsible (as of the time that this information was posted).

In Figure 2c, the right child of kd-tree node $y1$ and the left child of kd-tree node $x2$ have null pointers. They share the child node described by the decoration K at the subtree rooted at kd-tree node $x1$. Similarly, the right child of kd-tree node $y2$ is null and describes the space that the decoration L at $y2$ is directly responsible for. Note that decorations on non-leaf nodes are relevant to index nodes only. Data nodes do not have children. Any sibling node decorations within data nodes are at kd-tree leaves.

The collection of kd-tree nodes sharing a child node decoration C form a **decorated fragment** that describes the partitioning of C. These are all the nodes in the C-subtree which are not nodes in a smaller nested decorated subtree.

One can determine the containment order of the children of a node by just examining the kd-tree in that node. For example, the kd-tree of node Q in Figure 2c indicates that, first node L was extracted from K, and later node M was extracted from L. The K-fragment consists of kd-tree nodes $x1$, $y1$, and $x2$, the L-fragment of kd-tree node $y2$, and the M-fragment is empty, indicating that either M has not been split yet, or that no splitting of M has been posted yet. The containment order of the children of node Q is indicated by the arrows in the space decomposition of Figure 2d.

3.5 Continuation flags

Multi-parent nodes are created when an index hB^{Π}-tree node is split and a kd-subtree that is not decorated, i.e., its root does not carry a decoration, is extracted. The extracted kd-subtree is decorated with the same decoration as the split kd-subtree. It is obvious that, after the completion of the split, the child hB^{Π}-tree node that appears as decoration will be a **multi-parent** node. It will be pointed to by both the original node and the newly created node.

Node consolidation in the Π-tree requires that the node being deleted be referenced only by a single parent. So, we have to be able to detect whether a node is multi-parent or not by examining its current parent. This is accomplished by our fourth modification.

We keep a special **continues-to** flag with every side-pointer. The continues-to flag of a side-pointer is true or false indicating whether the kd-tree fragment that contains that side-pointer is continued to the sibling node or not. This is a way to determine if the child node that appears as the decoration is multi-parent or not. The continues-to flag of the side-pointer to R in Figure 2c being set to TRUE indicates that the child node K of node Q is multi-parent, and that node R is its other parent.

In addition, R must contain an indication that K, the decoration at its kd-tree root, is multi-parent. Each new sibling node contains a **continues-from** flag which determines whether the child decorating its kd-tree root is multi-parent.

3.6 Terminology

Table 1 summarizes the terminology that we will be using in the sections that follow.

Term	Description
decorated fragment	collection of kd-tree nodes with common decoration
decorated subtree	kd-subtree rooted at a decorated kd-tree node
P, C, X	Parent, Container and eXtracted nodes
Cto-Ppath	from P's C-subtree root to null leaf in P's C-fragment
CtoX-path	from C's kd-tree root to X's sibling address

Table 1: Terminology.

In Figure 2c, the K-subtree is the whole kd-tree, the L-subtree is the same as the L-fragment, and the M-subtree is empty. If we consider Q to be the parent node (P) and K to be the container node (C), then we have two Cto-Ppath's: ($x1$-left, $y1$-right) and ($x1$-right, $x2$-left). Also, if we consider Q to be the container node (C) and R to be the extracted node (X), then the CtoX-path is ($x1$-left, $y1$-left).

In the figures and examples of the following section we use the notation X and C for an extracted node and its container node respectively, and P for the parent of the container node where the index term that describes the split is posted.

4 hB^{Π}-tree Restructuring

4.1 Data Space Boundaries

The directly contained space of a data hB^{Π}-tree node, i.e., the space that does not include subspaces which have been delegated to sibling nodes, can be viewed as a union of disjoint rectangular regions corresponding to the record-lists that reside in the node. We call the boundaries of these disjoint spaces at the data level, or of contiguous collections of them **Data Space Boundaries** or **DSBs**. We have found that if index nodes are split in such a way that the extracted space and the remaining directly contained space of the nodes do not each correspond to a union of DSBs, there will be search correctness problems. In [ELS94] we show that the splitting/posting algorithm presented in [LS90] is flawed. Figure 3 shows a data level space decomposition and three possible space boundaries for this decomposition. Two of them, namely (a) and (b), are DSBs, whereas (c) is not.

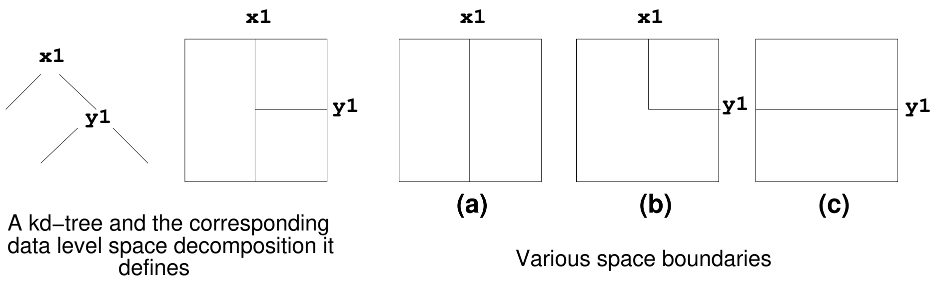


Figure 3: Data level space decomposition and various data and non-data space boundaries.

What is needed to correct the hB-tree flaw is to perform index hB-tree node splittings only at DSBs. This can be accomplished by imposing minor restrictions on splitting and/or posting. In this section we consider the simplest algorithm that accomplishes this. This algorithm splits index nodes only by extracting decorated subtrees (*D*) and we post the full CtoX-path (*fp*). We call the resulting algorithm *D/fp*. Two other algorithms that also split index nodes at data space boundaries are briefly described in Section 4.4.

It is important to notice that since data nodes do not have decorated subtrees, if their kd-tree has to be split, any subtree can be extracted (as in the hB-tree). Any place in a data node's kd-tree defines data space boundaries. Thus, the restriction we impose on splitting applies only to index nodes.

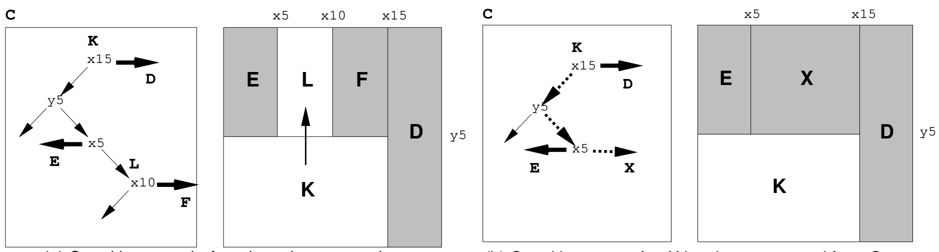
4.2 Splitting at Decorations (*D*)

When an hB-tree node had to be split, one had to find and extract a subtree whose size was between one and two thirds of the size of an index hB-tree node [LS90].

When splitting is done at decorations this is still true for data hB$^{\Pi}$-tree nodes. However, for index hB$^{\Pi}$-tree nodes, the extracted subtree must be a decorated subtree, which may preclude it satisfying the size requirement. Fortunately, this splitting convention simplifies the index term posting phase by not allowing multi-parent nodes.

In general, the kd-tree of an index node must be exhaustively searched in order to find the best possible decorated subtree, i.e., the one whose size is closest to half the hB$^{\Pi}$-tree node size.

Finally, in the highly improbable situation when no decorated subtree exists one can defer the splitting action (that would be the case in Figure 4a if kd-tree node $x10$ were missing). Remember that index hB$^{\Pi}$-tree nodes are split when they have not enough space to accommodate an index term posted by a posting action. By deferring a splitting action in an index node we actually defer a posting action. This is acceptable, since the hB$^{\Pi}$-tree remains well formed.



(a) C and its space before the subtree rooted at x10 is extracted to create node X.

(b) C and its space after X has been extracted from C. NOTE: the dotted path is the CtoX–path.

Figure 4: Node X is extracted from node C. kd-tree nodes from the CtoX-path will have to be posted to the parent of C.

Here is the algorithm for splitting an index hB$^{\Pi}$-tree node C by extracting a decorated subtree of its kd-tree:

SPLITTING AT DECORATIONS (D)

1. *find a decorated subtree in C whose size is closest to half a node's size, else EXIT*
2. *create a new node X, extract the decorated subtree from C, and move it to X*
3. *in C, replace the extracted subtree with a pointer to X (this is the side-pointer)*

In the next section we will show how to post the CtoX-path of Figure 4b to the parent P of C.

4.3 Posting the Full Path (*fp*)

Index term posting has to "correctly" insert the posted kd-tree nodes into an existing kd-tree. Intuitively, a kd-tree in an hB^{Π}-tree index node is well-formed if its kd-tree nodes appear in the same order as their corresponding kd-tree nodes at the level below. Two kd-tree nodes are corresponding if (a) they are located in consecutive hB^{Π}-tree levels, and (b) the one at the higher hB^{Π}-tree level has been created as a copy of the one at the lower hB^{Π}-tree level during an index term posting atomic action.

In the hB-tree one posted the **condensed** CtoX-path, i.e., only the kd-tree nodes that were necessary to describe the extracted region, and had not already been posted (see discussion in Section 4.4). One had to keep track of the kd-tree nodes that have already been posted by marking them. Also, kd-tree nodes that were ancestors of posted kd-tree nodes had to be marked.

In the fp variation of the hB^{Π}-tree we simplify the index term posting process by posting the full CtoX-path, that is, all kd-tree nodes from the CtoX-path that have not already been posted. Since we split at decorations, all hB^{Π}-tree nodes have exactly one parent, so one need only post the index term for a split to one hB^{Π}-tree node. Also, since the full paths are being posted, posting reduces to bringing the Cto-Ppath leading to or including X up-to-date so that it reflects the space decomposition described by the CtoX-path.

By posting the full CtoX-paths we preserve DSBs across the levels of the hB^{Π}-tree. Any subtree (decorated or not) of an hB^{Π}-tree node's kd-tree can be extracted because it describes a region which is defined by DSBs.

The resulting posting algorithm is straightforward. All we have to do is compare the Cto-Ppath for X against the CtoX-path. If it is equal or longer than the CtoX-path, we simply X-decorate part of it. If it is shorter than the CtoX-path, we append the extra kd-tree nodes of the CtoX-path to it, including a pointer to X.

In Figures 5 through 7 we demonstrate the three cases discussed above. Node P corresponds to the parent of node C of Figure 4. In each case, P and the space it is responsible for are shown before and after the posting takes place. The dotted lines indicate the Cto-Ppath leading to or including X that in each case is compared to the CtoX-path of Figure 4b. The algorithm is the following:

1. **len(Cto-Ppath for X) $>$ len(CtoX-path)**: this indicates that all kd-tree nodes of the CtoX-path had already been posted by other posting actions. All we have to do is X-decorate the first node of the Cto-Ppath which no longer refers to space in C (see Figure 5).

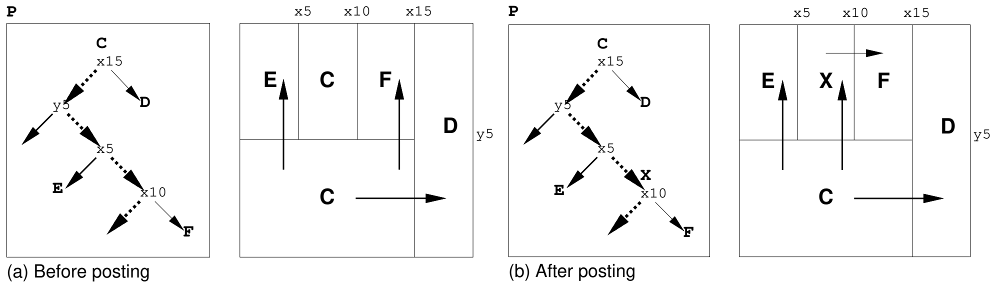


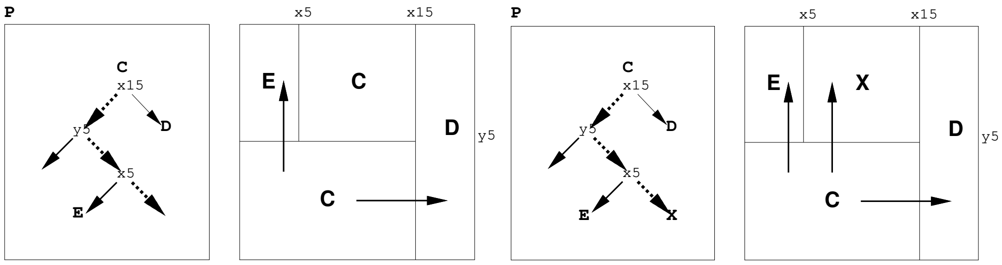
Figure 5: len(Cto-Ppath for X) $>$ len(CtoX-path): X-decorate the first kd-tree node of the Cto-Ppath which no longer refers to space in C.

2. **len(Cto-Ppath for X) $=$ len(CtoX-path)**: again, all kd-tree nodes of the CtoX-path had already been posted. We make the last kd-tree node of the Cto-Ppath for X point to X (see Figure 6).

3. **len(Cto-Ppath for X) $<$ len(CtoX-path)**: that is, the Cto-Ppath is a prefix of the CtoX-path. We append copies of the extra CtoX-path nodes to the Cto-Ppath, with the last posted node pointing to X (see Figure 7).

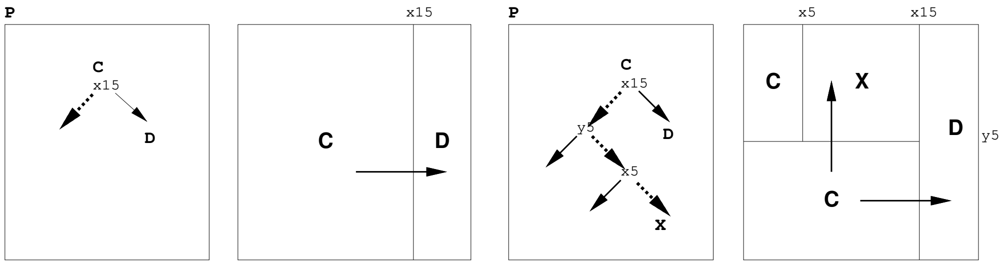
4.4 Other Splitting/Posting Algorithms

Posting the full path (*fp*) may increase the size of the index terms posted. Especially when the data is skewed, we may end up posting long CtoX-paths. Splitting at



(a) Before posting (b) After posting

Figure 6: len(Cto-Ppath for X) = len(CtoX-path): make the last kd-tree node of the Cto-Ppath point to X.



(a) Before posting (b) After posting

Figure 7: len(Cto-Ppath for X) $<$ len(CtoX-path): append the extra kd-tree nodes of the CtoX-path to the Cto-Ppath.

decorations (*D*) requires an exhaustive search of the whole kd-tree of the index hB$^{\Pi}$-tree node to find the largest decorated subtree whose size is between one and two thirds of the contents of the node. It may be the case that such a subtree does not exist at all. In this case we will have a bad quality split. If bad splits are too frequent, the utilization of the index nodes will decrease, and the size of the index will increase. Despite the lack of worst case guarantees for index term size or index node storage utilization, when algorithm *D/fp* is used, the hB$^{\Pi}$-tree has demonstrated good performance in our study(see section 5.1).

Algorithm *D/fp* is but one of several alternatives one can use. There are other splitting and posting strategies that also result in splitting only at DSBs. We briefly describe two strategies for hB$^{\Pi}$-tree node splitting and two strategies for index term posting.

Our two splitting strategies are: (a) split a kd-tree at arbitrary places (strategy *A* = Arbitrary), or (b) split a kd-tree only at decorated subtrees (strategy *D* = Decorated). In the example of figure 2c node Q can be split anywhere if we use strategy *A*. On the other hand if we use strategy *D* it can be split only at y_2.

Our two posting strategies are: (a) post the condensed path as in the hB-tree, that is, only the kd-tree nodes of the CtoX-path that are necessary to describe the extracted space (strategy *cp* = condensed path), or (b) post all kd-tree nodes, or the full CtoX-path (strategy *fp* = full path). In the example of figure 2c let us assume that the subtree decorated with L is extracted. Then, if we use strategy *cp* during posting we must post only kd-tree node x_2 since x_1 is redundant (the extracted space contains all points with $x > x_2$). If we use strategy *fp* both x_1 and x_2 must be posted.

Depending on the splitting and posting strategy that is used, there are four splitting/posting algorithms: *D/fp*, *A/fp*, *D/cp*, and *A/cp*.

Like *D/fp*, algorithms *A/fp* and *D/cp* also split only at DSBs. *A/fp* posts full paths, so any extracted subtree describes a region which is defined by DSBs (exactly as *D/fp* does). In the case of *D/cp*, although condensed paths are posted, the index term that we post when we extract a decorated subtree describes a region which is defined by DSBs.

Algorithm *A/cp* corresponds to the splitting and posting algorithm for the hB-tree described in [LS90]. The strengths and weaknesses of the other algorithms are summarized in Table 2.

Property / Algorithm	Splitting algorithm	Posting algorithm	Worst case split	Worst case index term size	Multiple parents	Concurrency
D/fp	Restrictive	Append	Skewed	Large	No	High
A/fp	Flexible	Append	Balanced	Large	Yes	High
D/cp	Restrictive	Merge	Skewed	Small	No	High

Table 2: Comparison of the various splitting/posting algorithms.

4.5 Node Consolidation

Following the Π-tree algorithm for node consolidation, a sparse hB$^{\Pi}$-tree node is consolidated with a sibling node and the parent of the deleted node is modified to reflect the change. For reasons of simplicity and efficiency we always choose to consolidate a sparse hB$^{\Pi}$-tree node with its container node. So, the two conditions for hB$^{\Pi}$-tree node consolidation are: (1) the sparse (extracted) node shares the same parent with its container, and (2) it is also a single-parent node.

Condition (1) is not true when the sparse (extracted) node's decoration appears at the root of its parent's kd-tree, that is, the sparse node is the container of all the children of that parent. This is not very common, since in the worst case there are as many such nodes at a given level as parent nodes at the level above. Similarly, condition (2) is not very restrictive either. There is a limited number of nodes that are multi-parent when the splitting strategy allows multi-parent of nodes. Since at most one kd-tree fragment is split per hB$^{\Pi}$-tree node split, at most one multi-parent is introduced per split. In the worst case there will be as many multi-parent nodes at a given level of the hB$^{\Pi}$-tree as parent nodes at the level above. Our continuation flags are used to detect when a node is multi-parent by checking only a single path to the node.

Since an hB$^{\Pi}$-tree node uses a kd-tree for its intra-node organization, we also have to reorganize the kd-trees of the parent and container nodes of the extracted node. In [ELS94], we show how one can determine whether a node can be deleted by examining the kd-tree of its parent, how deletion is performed, and how kd-tree node pruning is used to reorganize kd-trees.

5 Varieties of Multi-attribute Data

In this section we demonstrate the performance of the hB$^{\Pi}$-tree on multi-attribute point data. We use both computer-generated data and geographic data from the Sequoia project [SFGM93] and we measure node space utilization and range search performance.

We also examine the problem of using a multi-attribute point data method, like the hB^{Π}-tree, for spatial data. First, we show that when mapping is used to transform a k-dimensional objects to $2k$-dimensional points, the resulting mapped space has some interesting properties that are likely to enhance the performance of spatial queries. Second, we claim that the hB^{Π}-tree is very well suited for indexing the mapped space because it is fairly insensitive to dimension.

5.1 Point Data Performance

5.1.1 Node Space Utilization

In the first part of our experiment, whose results are shown in Figure 8, we inserted half a million 32-byte records (eight 4 byte attributes in each record) into the hB^{Π}-tree.

We attempted to test the limits of algorithm *D/fp* by making the values for all indexed attributes follow a 90:10 skewed distribution, instead of the uniform distribution. With reasonably large node sizes (greater than 1K bytes) space utilization is very good regardless of the number of indexed attributes. Note that the decline in utilization is due to increased control information and not index term size (see Section 5.3). Also, the size of the index is very small: for node sizes 0.5K, 1K, 2K, and 4K, 5%, 2.5%, 1.4%, and 0.75% of the total number of hB^{Π}-tree nodes are index nodes respectively.

Finally, our performance results show that when the hB^{Π}-tree is used as a single-attribute index it performs comparably to the B^{+}-tree.

In the second part of our experiment, data from the Sequoia 2000 Storage Benchmark [SFGM93] was inserted in the hB^{Π}-tree. These are 62,584 points representing California place names. We tested two of the splitting/posting algorithms we have described in this paper: *D/fp*, and *A/fp*. As expected, the more relaxed index node splitting strategy *A* yielded better index node space utilization compared to splitting strategy *D* (see Figure 9).

Note that the space utilization results are comparable to the ones we obtained when using computer generated data. Also, the comparably low index node space utilization when the node size is 4K is attributed to the low number of index nodes (only 6 index nodes).

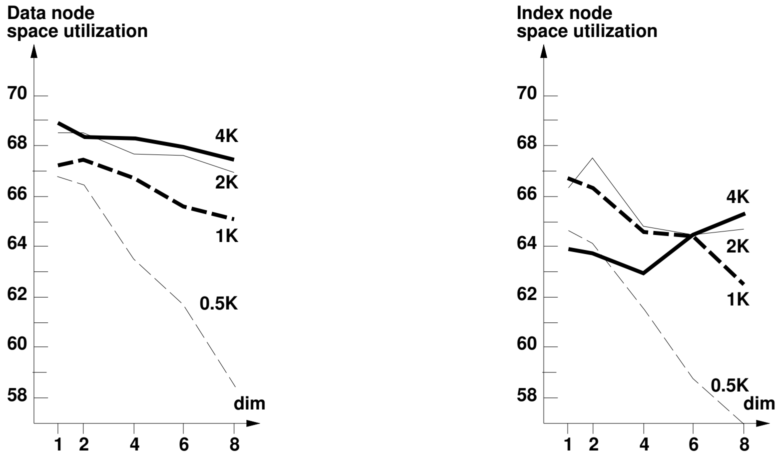


Figure 8: Node space utilization for computer generated data under varying node sizes and dimensions when algorithm *D/fp* is used.

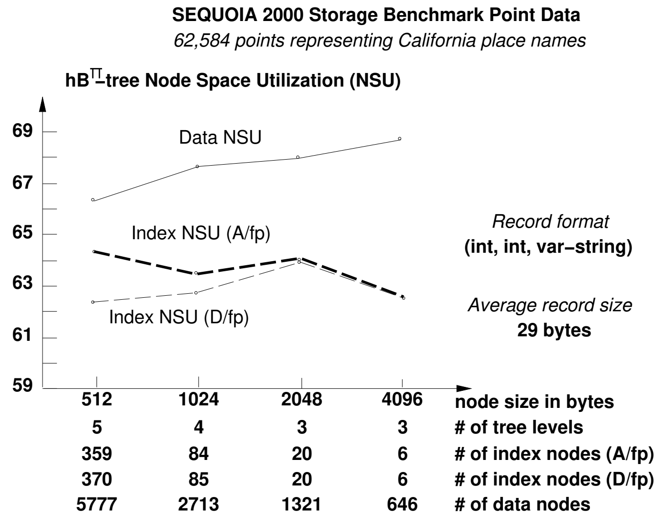


Figure 9: Node space utilization for the Sequoia 2000 Storage Benchmark point data under varying node sizes and index node splitting strategies.

5.1.2 Range Searches

Finally, we have tested the range seach performance of the hB$^{\Pi}$-tree. We performed the same series of 104 range searches with varying query selectivity and node size. The query window was rectangular and was formed by taking a randomly chosen existing point as its center. To achieve various query selectivities, the extent of the window for each attribute was a random fraction of the domain range for that attribute.

The results, shown in Figure 10, indicate very good range search performance for query selectivities greater than 0.5%, and sufficiently good even at smaller query selectivities. Note that when the query selectivity is approximately equal to the average number of records in a data node, 25% of the records retrived satisfy the query. This is as expected because it is likely that in this case the query window will overlap on average four data nodes.

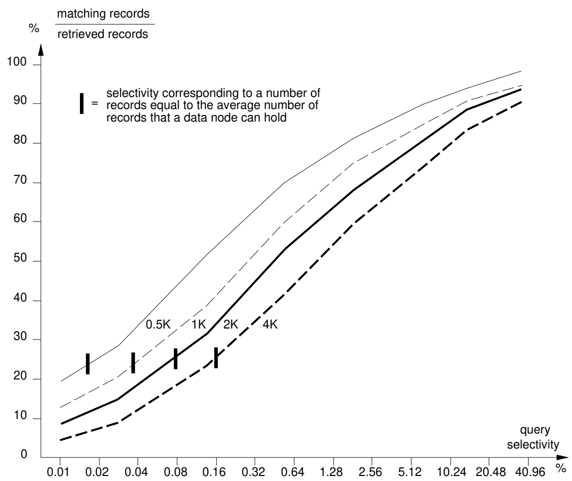


Figure 10: hB$^{\Pi}$-tree range search performance in terms of the ratio of retrieved points that satisfy the query over total number of retrieved points per range search, under various node sizes (0.5, 1, 2, and 4 Kbytes) and query selectivity.

5.2 Mapping Spatial Data to Point Data

A spatial object in k-dimensional space can be indexed by its bounding box. The boundaries of the box are mapped to a point in $2k$-dimensional space [NH83]. This mapping has some very interesting properties: if two points in the $2k$-dimensional space have similar values in all coordinates then the k-dimensional objects will (a) be similar in size, (b) be close in space, if they are small, and (c) overlap, if they are large. Thus, any multi-attribute point data indexing method which clusters nearby points will group records of large objects together in pages with other overlapping large objects. It will also group small objects together with nearby small objects in pages.

This clustering is efficient for typical spatial queries. Large objects are likely to be the answer to many queries. Having them clustered in disk pages will increase the locality of reference. Having the small objects clustered together will decrease the number of pages which must be accessed for a particular query [Lom91].

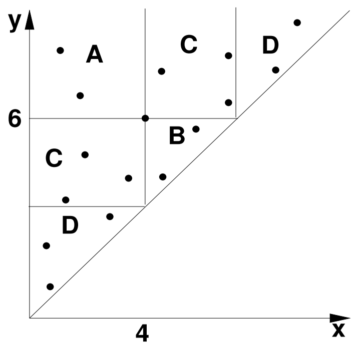


Figure 11: Mapping of line segments to points. Points close to the line $x = y$ will represent small line segments.

To illustrate, we look at one dimensional spatial objects, which are line segments with a begin value and an end value. We map these objects to points in two-dimensional space using the x-coordinate for the begin value of the line segment and the y-coordinate for the end value. Since the begin value is always less than or equal to the end value, all points will lie on or above the line $x = y$. Points representing small line segments will be near the diagonal line $x = y$ and points representing large line segments will be far from the line $x = y$ (figure 11).

Common spatial queries such as inclusion, intersection, or disjointness are represented by rectangular bricks in the transformed ($2k$-dimensional) space. For example, as illustrated in figure 11, all line segments containing the line segment (begin = 4, end = 6), which is represented by point (4, 6), will be in area A, the ones included in it in area B, the ones intersecting it in area C, and the ones that have no common points with it in area D.

5.3 The hB$^{\Pi}$-tree in High Dimensions

One main objection to using multiattribute point-based methods for spatial objects is that the number of dimensions needed to represent the objects doubles, making the index too large. But the hB$^{\Pi}$-tree is essentially insensitive to increases in dimension.

A kd-tree node always stores the value of exactly one attribute. Thus, the size of a kd-tree node (and, consequently, the size of the kd-trees that reside in the hB$^{\Pi}$-tree nodes) does not depend on the number of indexing attributes.

But, in addition to a kd-tree, every hB$^{\Pi}$-tree node stores its own boundaries (i.e., low and high values for all attributes that describe the space the node is responsible for). These are $2k$ attribute values for a k-dimensional hB$^{\Pi}$-tree. An increase on the number of dimensions does increase the space required to store a node's boundaries. This additional space is not significant for large page sizes.

Figure 12 illustrates this fact. Node space utilization is defined as the ratio of the size of a node's kd-tree and the size of a page. The decline in utilization is due to increased control information and not index term size. With a page size of 1K bytes and larger, there is almost no effect on the size of the hB$^{\Pi}$-tree and the node space utilization as the dimensions increase. (Page sizes larger than 2K bytes are not shown.) It is interesting to notice that these performance results were obtained using algorithm *D/cp* and are comparable to the results of figure 8 where algorithm *D/fp* was used.

This is in contrast, for example, with the R-tree [Gut84], where index entries are bounding coordinates of objects plus a pointer. Thus, in the R-Tree (and its variants) the size of the index is proportional to the dimension of the space. If data is uniformly distributed with respect to all index attributes, the grid file [NHS84] can be efficient for large dimension. However, in the case of correlated data, for example, it can be an $O(n^k)$ size index, where n is the number of points and k is the dimension of the space. Z-ordering [OM84] usually requires that the field expressing the interleaved attributes is appended to each record. This obviously results in a substantial increase in data space consumed and hence index size, which becomes worse as the dimension of the data increases.

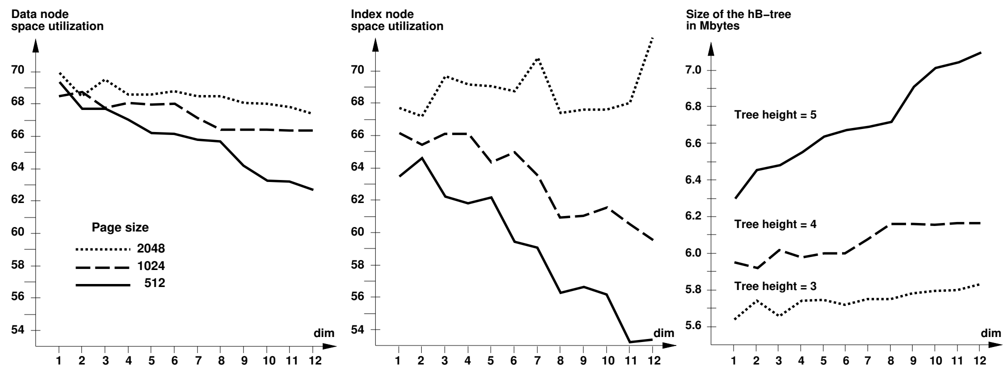


Figure 12: Index and data hB$^{\Pi}$-tree node space utilization and size of the hB$^{\Pi}$-tree in terms of height and Mbytes under different page sizes and dimensions (indexed attributes). For page sizes greater than 1K bytes the hB$^{\Pi}$-tree is fairly insensitive to dimension. (In all cases the same 150,000 24-byte records were inserted with their attribute values following a 90:10 skewed distribution.)

6 Summary

The hB$^{\Pi}$-tree is a combination of the hB-tree [LS90] and the Π-tree [LS92]. It inherits the good performance of the hB-tree and the high concurrency of the Π-tree.

We have implemented and tested various splitting/posting algorithms for the hB$^{\Pi}$-tree. We have found that if we post more information than is actually needed (the full path), or if we restrict index node splits to certain places on their kd-tree (the decorated fragments), our algorithms become simpler. Our experiments show that even with these simpler algorithms the performance is very good. We have also developed a deletion algorithm for the hB$^{\Pi}$-tree [ELS94]. In addition, the hB$^{\Pi}$-tree is very well suited for indexing spatial data that has been mapped to points in higher dimensions. This is because it is fairly insensitive to increases in the dimensions of the stored data [ES93].

We intend to further assess the performance of additional splitting/posting algorithms, based on even more flexible splitting strategies and using polygon and graph data from the Sequoia 2000 Storage Benchmark [SFGM93].

References

[Ben79] J. L. Bentley. Multidimensional binary search trees in database applications. *IEEE Transactions on Software Engineering*, SE-5(4):333–340, July 1979.

[BM72] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[BS77] R. Bayer and M. Schkolnick. Concurrency of operations on B-Trees. *Acta Informatica*, 9(1):1–21, 1977.

[Com79] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(4):121–137, 1979.

[ELS94] G. Evangelidis, D. Lomet, and B. Salzberg. Node Deletion in the hB^{Π}-Tree. Technical Report NU-CCS-94-04, College of Computer Science, Northeastern University, Boston, MA, 1994 (submitted for publication).

[ES93] G. Evangelidis and B. Salzberg. Using the Holey Brick Tree for Spatial Data in General Purpose DBMSs. *IEEE Database Engineering Bulletin*, 16(3):34–39, September 1993.

[Gue89] O. Guenther. The design of the cell tree: an object oriented index structure for geometric databases. In *Proceedings of IEEE Data Engineering Conference*, pages 598–605, Los Angeles, CA, 1989.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 47–57, Boston, MA, June 1984.

[Lom91] D. Lomet. Grow and Post Index Trees: role, techniques and future potential. 2nd Symposium on Large Spatial Databases (SSD91) (August, 1991) Zurich. In *Advances in Spatial Databases, Lecture Notes in Computer Science 525*, pages 183–206, Berlin, 1991. Springer-Verlag.

[LS90] D. Lomet and B. Salzberg. The hB-Tree: A multiattribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, December 1990.

[LS92] D. Lomet and B. Salzberg. Access method concurrency with recovery. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 351–360, San Diego, CA, June 1992.

[LY81] P. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Transactions on Database Systems*, 6(4):650–670, December 1981.

[ML89] C. Mohan and F. Levine. ARIES/IM: an efficient and high concurrency index management method using write-ahead logging. IBM Research Report RJ 6846, IBM Almaden Research Center, San Jose, CA, August 1989.

[NH83] J. Nievergelt and Hinrichs. The Grid File: A Data Structure to Support Proximity Queries on Spatial Objects. In *Proceedings of the International Workshop on Graph Theoretic Concepts in Computer Science*, pages 100–113, Linz, Austria, 1983.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An adaptable, symmetric, multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[OM84] J. A. Orenstein and T. Merrett. A class of data structures for associative searching. In *Proceedings of SIGART-SIGMOD 3rd Symposium on Principles of Database Systems*, pages 181–190, Waterloo, Canada, 1984.

[SC91] V. Srinivasan and M. Carey. Performance of B-tree concurrency control algorithms. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 416–425, Denver, CO, May 1991.

[SFGM93] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Storage Benchmark. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 2–11, Washington, DC, May 1993.

[SG88] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Transactions on Database Systems*, 13(1):53–90, March 1988.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-tree: a dynamic index for multi-dimensional objects. In *International Conference on Very Large Data Bases*, pages 1–24, Brighton, England, 1987.

Contents