The hB$^{\Pi}$-tree: A Modified hB-tree Supporting Concurrency, Recovery and Node Consolidation *

Georgios Evangelidis
Leoforos Stratou 13
GR-54639 Thessaloniki, Greece

David Lomet
Microsoft Corporation
Redmond, WA 98052

Betty Salzberg
Northeastern University
Boston, MA 02115

Abstract

We describe a new access method, the hB$^{\Pi}$-tree, an adaptation of the hB-tree index to the constraints of the Π-tree . The Π-trees, a generalization of the B^{link}-trees, provide high concurrency with recovery, because they break down structure modification into a series of short atomic actions. In addition, the Π-trees include a node consolidation algorithm. The hB-tree is a multi-attribute index which is highly insensitive to dimensionality, but which has no node consolidation algorithm and has a flaw in its split/post algorithm in certain special cases. The hB$^{\Pi}$-tree corrects the splitting/posting algorithm and adapts the concurrency, recovery and node consolidation of the Π-tree to the hB-tree. The combination makes the hB$^{\Pi}$-tree suitable for inclusion in a general purpose database management system supporting multi-attribute and spatial queries.

keywords: indexing, B-trees, multi-attribute access methods, spatial access methods, concurrency, recovery

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 [2, 4].

* This work was partially supported by NSF grants IRI-91-02821 and IRI-93-03403.

Proceedings of the 21st VLDB Conference, Zurich, Switzerland 1995

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 by latitude and longitude and height above the earth, for example, rather than linearly by one attribute.

Spatial organization of large databases is a largely unsolved problem. There have been a number of proposals for multi-attribute and spatial indexing in the past 15 years (for example, [8, 15, 7, 18, 10]), but none of them has been integrated into a commercial general purpose DBMS. One reason for this is that there are very complicated or no concurrency and recovery methods for them.

Concurrency in B^{+}-trees has been the subject of many papers [3, 12, 19, 13, 11]. Most of these papers, with the exception of [13, 11], have not addressed the problem of system crashes during structure changes.

The *hB$^{\Pi}$-tree* is a new access method for multi-attribute point or spatial data that is based on the hB-tree [10] and can be used in a general purpose DBMS since a robust concurrency and recovery algorithm is available for it [11] (the "Π-tree paper"). This algorithm is applicable to an abstract index tree structure, the Π-tree, which is a generalization of the B^{link}-tree [12]. A recent study [20] 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. The Π-tree paper improves the B^{link}-tree by breaking down structure modification operations into a series of independent atomic actions, each of short duration and holding a small number of locks. The Π-tree paper includes node consolidation as one of the atomic actions.

The hB$^{\Pi}$-tree is a modification of the hB-tree so that it becomes a special case of the Π-tree. This involves structural changes to the hB-tree. In addi-

tion, new splitting and posting algorithms were invented for the hB$^{\Pi}$-tree which correct an error in the hB-tree, and the node deletion atomic action of the Π-tree paper was adapted to the hB$^{\Pi}$-tree.

We have implemented the hB$^{\Pi}$-tree and tested it in extensive experiments with computer-generated skewed point data and with point data from the Sequoia 2000 Storage Benchmark [21]. We show that the hB$^{\Pi}$-tree is relatively insensitive to dimension.

This paper is organized as follows. Section 2 briefly reviews the Π-tree. Section 3 introduces the hB$^{\Pi}$-tree, which is a combination of the hB-tree and the Π-tree. Section 4 introduces the new splitting and posting algorithms which correct the error in [10]. Finally, Section 5 reports performance results.

2 Concurrency & recovery: the Π-tree

In this section we review the Π-tree and its algorithms. The Π-tree concurrency method is effective because it breaks down structure modification into a series of short atomic actions, each of which holds only a small number of locks. During recovery after system failure, incomplete atomic actions are undone and pending atomic actions are scheduled lazily.

2.1 Π-tree structure

Briefly, the Π-tree is based on a partial ordering of nodes at each level of the tree. A binary relationship is formed when a node splits. Part of the contents of the node (as in the B^{+}-tree) goes to a new sibling. The sibling is said to be **extracted**. The old node is said to be the **container**. The closure of the container/extracted relationship is only a partial ordering since, for example, two new siblings can be extracted in two separate splits from the same old container node in such a way that neither of the two new siblings is a container of the other.

More formally, as a generalization of the B^{link}-tree [12], 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 analogous to those in the B^{link}-tree.

2.3 Node splitting and index 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** 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). Only the container node need be locked during the node-splitting atomic action.

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 index term posting atomic actions (one for each parent). Only the parent node need be write-locked. (The child is read-locked for a short time to see if posting is still needed.)

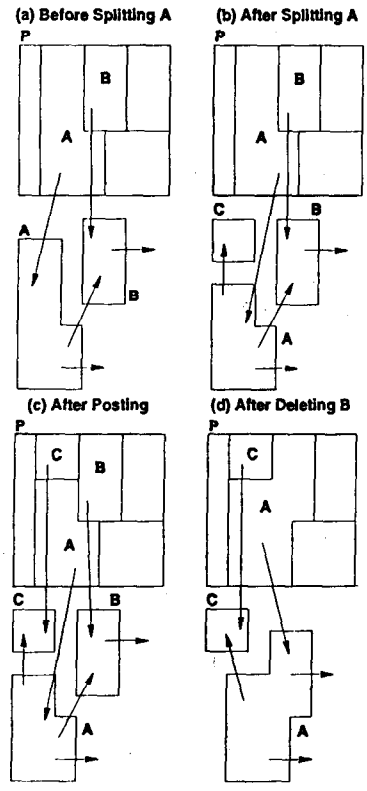


Figure 1: Splitting, Posting, and Consolidating in the Π-tree.

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 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. In addition to the obvious requirement that the container node have sufficient free space, we require two additional conditions 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. Only the parent and the two children are locked.

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 System failures

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 following a sibling pointer from its container node. The same is true when, for 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 a missing index term is scheduled when a sibling pointer to it is traversed. Node consolidations are not necessary for correct search. They can be rescheduled when a sparse node is visited.

3 The hB-tree as a Π-tree

We review the hB-tree in this section and show how we modify it to become a special case of the Π-tree and how we modify it to facilitate node consolidation. We call the resulting tree the hB$^{\Pi}$-tree.

3.1 Multi-attribute indexing

The hB-tree [10] is a multi-attribute point data indexing method. 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 [1].

In Figures 2a and 2b we can see the 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). Let S_q denote the space for which Q is directly responsible. The remaining four paths in the kd-tree of Q describe the decomposition of S_q among Q's children, namely nodes K, L, and M (white regions of Figure 2b).

In order to transform the hB-tree into a case of the Π-tree we need to have pointers to extracted nodes. In addition, so that we can find candidate siblings for node consolidation, we need a convenient way to be able to determine the containment order of the children of an index node and 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$^{\Pi}$-tree. The first and second are needed to make it a Π-tree. The third simplifies

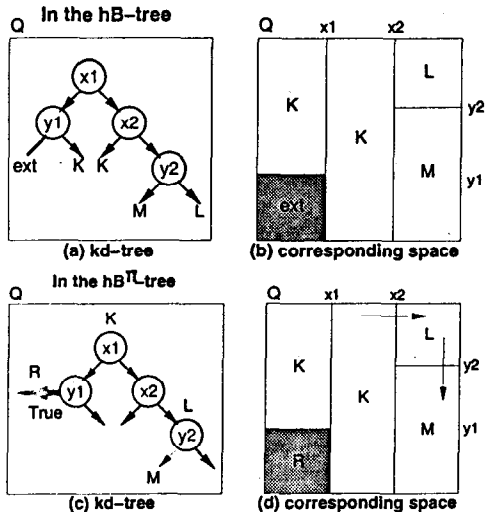


Figure 2: Intra-node organization of hB-tree and hB$^{\Pi}$-tree nodes using kd-trees.

kd-tree management and both third and fourth aid in node consolidation.

3.2 Side pointers

We replace external markers by pointers to extracted nodes, called **side-pointers**. In Figure 2c the thick arrow with the address of node R denotes the side-pointer 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

In addition to replacing external markers, to get all the side pointers we need, we must modify 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 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. This modification will help us to determine whether or not there is a suitable child for consolidation with a sparse child. Consolidation can only take place when one sibling is the container of the other. (This modification will also help us define some simple splitting algorithms.)

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 describing the space for which a child node is responsible. 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**. We refer to pecific decorated subtrees by means of their decorations. For example, in Figure 2c, the decorated subtree rooted at $x1$ is referred to as the K-subtree, the decorated subtree at $y2$ as 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.
2. a sibling node address: the index node has 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.

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 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 decoration L at $y2$ is directly responsible for. Decorations are relevant to index nodes only. Data nodes do not have children.

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. (Note that R is **not** a child of Q; R is a sibling of Q.)

3.5 Continuation flags

To deallocate an extracted node when it is consolidated with its container, we must know that it is a single-parent node. Continuation flags in the parent are used to make this determination.

Multi-parent nodes are created when an index hB$^{\Pi}$-tree node is split and an undecorated kd-subtree (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. After the completion of the split, the child hB$^{\Pi}$-tree node that appears as decoration will be a **multi-parent** node 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. We 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
decor'd fragmnt	kd-tree nodes with common decoration
decor'd subtree	kd-subtree rooted at decorated node
P, C, X	Parent, Container and eXtracted nodes
Cto-Ppath	from P's C-subtree root to C-fragment null leaf
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$^{\Pi}$-tree restructuring

In this section, we present new splitting and posting policies which correct the flawed splitting/posting algorithm of [10]. This flaw does not appear until there are three levels in the index above the leaves and then only in special cases. A construction of such a four-level tree can be found in [17] and [5]. Here we first explain the reason for this error and then show the new posting and splitting algorithms which correct it.

4.1 Data space boundaries

The directly contained space of a data hB$^{\Pi}$-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**. 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. This occurred in the hB-tree. 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.

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 restrictions on splitting and/or posting. In this section we consider

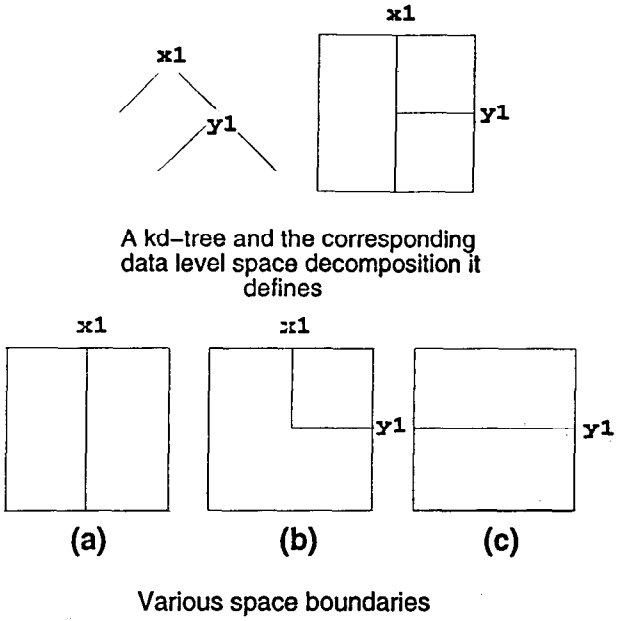


Figure 3: Data level space decomposition and various data and non-data space boundaries.

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.

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. Hence the space utilization guarantees claimed for the hB-tree apply to the hB^{Π}-tree at the leaf level (where over 95% of the storage resides).

4.2 Splitting at decorations (D)

Splitting at decorations has the very attractive property that it makes all nodes single-parent nodes. The hB^{Π}-tree in this case is a true tree, not a DAG. This will make the index-term posting algorithm simpler.

For index hB^{Π}-tree nodes, when splitting at decorations, the extracted subtree must be a decorated subtree. 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^{Π}-tree node size. This implies that the one-third/two-thirds guarantee of index node utilization promised by the hB-tree does not hold.

In fact, if no decorated subtree exists at all in an index node, because none of the k-d-tree nodes carry decorations, splitting at decorations is not possible, and one must defer the splitting action (that would be the case in Figure 4a if kd-tree node $x10$ were missing). Remember that index hB^{Π}-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. Thus deferring splitting does not affect correctness of search. However, it can reduce search performance as each side pointer traversal accesses an extra hB^{Π}-tree node. In any case, in our experiments, most (over 90%) of the nodes were decorated, so this situation did not arise. (Most splits post only one k-d-tree node and this node then becomes decorated.)

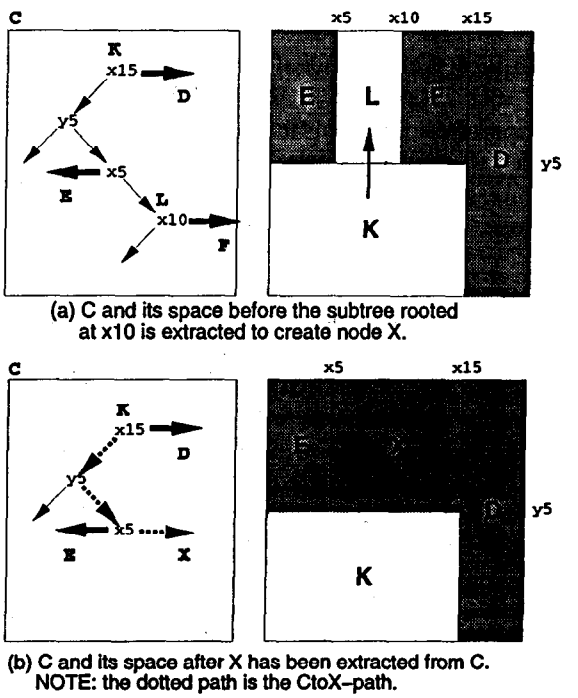


Figure 4: Node X is extracted from node C. kd-tree nodes from the CtoX-path will have to be posted to the parent of C.

Here is the algorithm for splitting an index hB^{Π}-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 kd-tree nodes describing an hB$^{\Pi}$-tree node split into an existing kd-tree in the parent hB$^{\Pi}$-tree index node. Intuitively, a kd-tree in an hB$^{\Pi}$-tree index node is well-formed if its kd-tree nodes appear in the same order as the corresponding kd-tree nodes of its children. Two kd-tree nodes correspond if (a) they are located in consecutive hB$^{\Pi}$-tree levels, and (b) the one at the hB$^{\Pi}$-tree parent has been created as a copy of the one at the hB$^{\Pi}$-tree child 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). In the *fp* variation of the hB$^{\Pi}$-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.

By posting the full CtoX-paths we preserve DSBs across the levels of the hB$^{\Pi}$-tree. Any subtree (decorated or not) of an hB$^{\Pi}$-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. Since we split at decorations, all hB$^{\Pi}$-tree nodes have exactly one parent, so one need only post the index term for a split to one hB$^{\Pi}$-tree node.

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).

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).

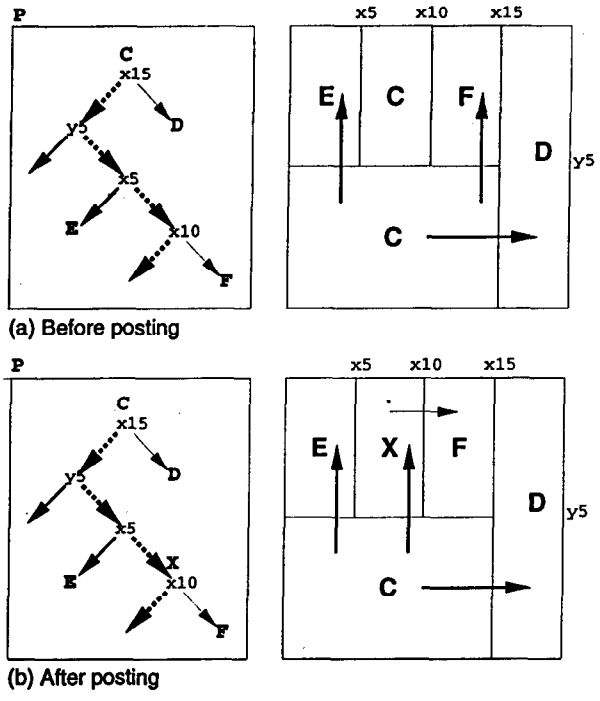


Figure 5: len(Cto-Ppath for X) > len(CtoX-path): X-decorate the first kd-tree node of the Cto-Ppath which no longer refers to space in C.

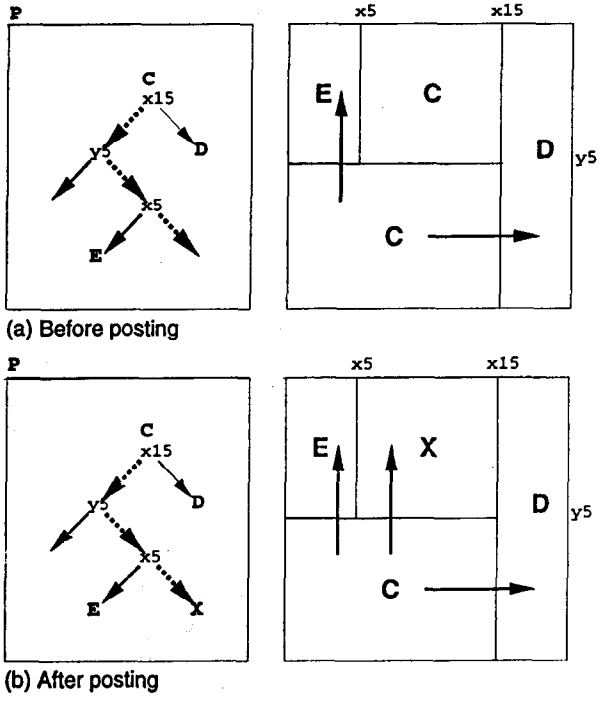


Figure 6: len(Cto-Ppath for X) = len(CtoX-path): make the last kd-tree node of the Cto-Ppath point to X.

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).

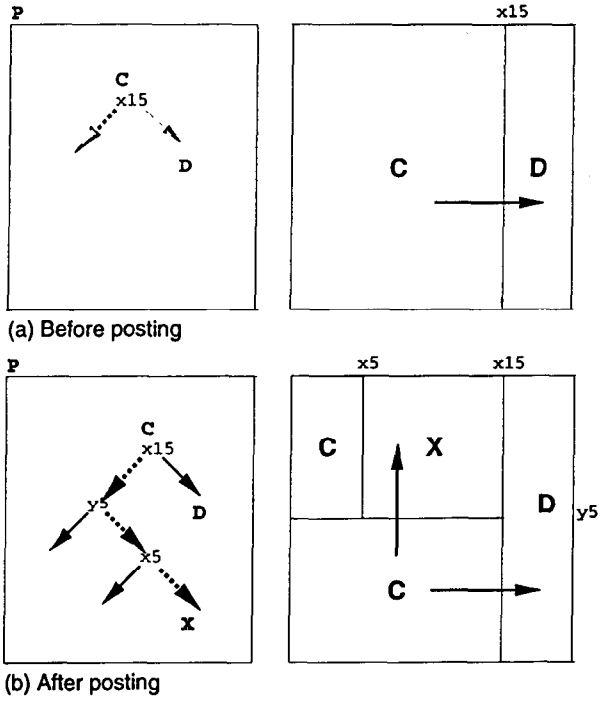


Figure 7: len(Cto-Ppath for X) < len(CtoX-path): append the extra kd-tree nodes of the CtoX-path to the Cto-Ppath.

4.4 Other splitting/posting algorithms

Posting the full path (*fp*) may increase the size of the index terms posted. When the data is skewed, we may end up posting long CtoX-paths. The guarantees for size of index terms posted in the hB-tree no longer hold.

Splitting at decorations (*D*) requires an exhaustive search of the whole kd-tree of the index hB$^{\Pi}$-tree node to find the subtree whose size is closest to half the size of an hB$^{\Pi}$-tree node. There is no guarantee we can find a good 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 node space and range search 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 split anywhere if we use strategy A. 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 [10]. The strengths and weaknesses of the other algorithms are summarized in Table 2.

Property/Algorithm	D/fp	A/fp	D/cp
Splitting	Restrictive	Flexible	Restrictive
Posting	Append	Append	Merge
Worst split	Skewed	Balanced	Skewed
Worst index size	Large	Large	Small
Multiple parents	No	Yes	No
Concurrency	High	High	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. If these conditions are not satisfied, or if the contents of the two nodes are too large for a single consolidated node, then node consolidation does not take place. (Remember that node consolidation is optional and that the goal is to preserve overall utilization. And our two conditions are commonly satisfied.)

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^{Π}-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^{Π}-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^{Π}-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 [5], 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 Performance

In this section we demonstrate the performance of the hB^{Π}-tree on multi-attribute point data. We use both computer-generated data and geographic data from the Sequoia project [21] and we measure node space utilization and range search performance.

We show that the hB^{Π}-tree is insensitive to dimension. Thus it can be used efficiently for any high-dimensional applications.

5.1 Point data

5.1.1 Node 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 explored the limits of algorithm *D/fp* by skewing the values for all indexed attributes by 90:10, instead of using a uniform distribution. With reasonable node sizes (greater than 1K bytes) space utilization is very good regardless of the number of indexed attributes. The decline in utilization is due to increased control information and not index term size (see Section 5.2). 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.

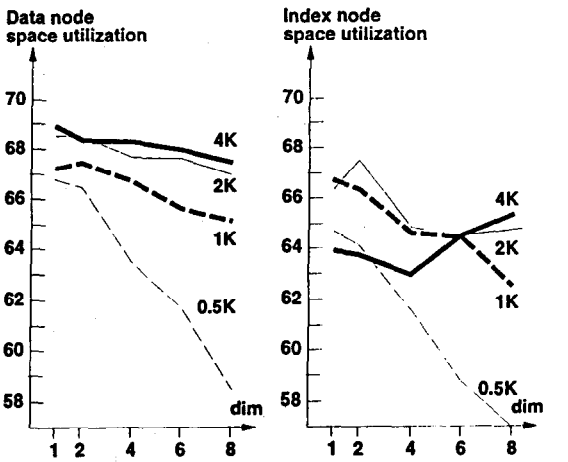


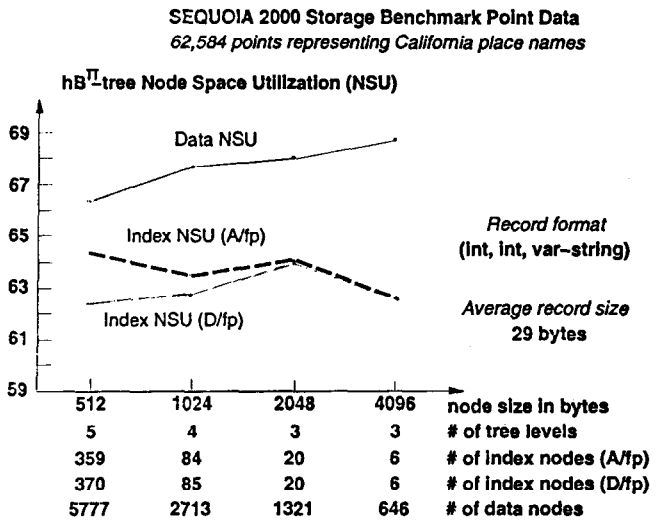
Figure 8: Node space utilization for computer generated data under varying node sizes and dimensions when algorithm *D/fp* is used.

In the second part of our experiment, data from the Sequoia 2000 Storage Benchmark [21] 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).

5.1.2 Range searches

Finally, we have tested range search performance of the hB^{Π}-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 vary query selectivities, the extent



node size in bytes	512	1024	2048	4096
# of tree levels	5	4	3	3
# of index nodes (A/fp)	359	84	20	6
# of index nodes (D/fp)	370	85	20	6
# of data nodes	5777	2713	1321	646

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

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.

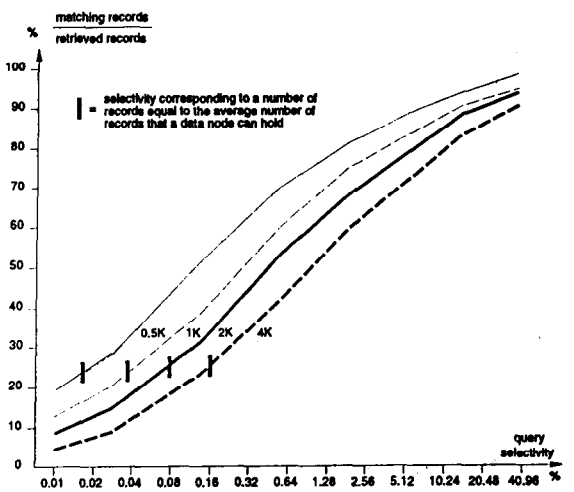


Figure 10: hB$^{\Pi}$-tree range search performance in terms of the ratio of retrieved points that satisfy the query over total number of retrieved points per range search, under various node sizes (0.5, 1, 2, and 4 Kbytes) and query selectivity.

5.2 The hB$^{\Pi}$-tree in high dimensions

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 11 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.

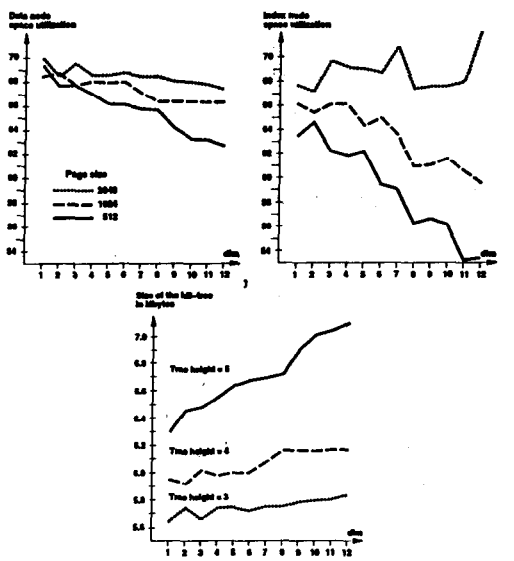


Figure 11: Index and data hB$^{\Pi}$-tree node space utilization and size of the hB$^{\Pi}$-tree in terms of height and Mbytes under different page sizes and dimensions (indexed attributes). For page sizes greater than 1K bytes the hB$^{\Pi}$-tree is fairly insensitive to dimension. (In all cases the same 150,000 24-byte records were inserted with their attribute values following a 90:10 skewed distribution.)

This is in contrast, for example, with the R-tree [8], 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 [15] 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 [16] 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.

6 Summary

The hB$^{\Pi}$-tree is a combination of the hB-tree [10] and the Π-tree [11]. It corrects the flaw in the hB-tree and, with suitable modifications, inherits the concurrency, recovery and node-deletion algorithms of the Π-tree. It is insensitive to dimension and so is suitable for high-dimensional applications [6].

We have implemented and tested various splitting/posting algorithms for the hB$^{\Pi}$-tree. We 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 [5].

We intend to assess the performance of additional splitting/posting algorithms, based on even more flexible splitting strategies, using polygon and graph data from the Sequoia 2000 Storage Benchmark [21].

References

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

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

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

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

[5] G. Evangelidis, D. Lomet, and B. Salzberg. Node Deletion in the hB$^{\Pi}$-Tree. Report NU-CCS-94-04, Northeastern Univ, Boston, MA, 1994

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

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

[8] A. Guttman. R-trees: A dynamic index structure for spatial searching. *ACM/SIGMOD Conf.*, 47–57, Boston, MA, 1984.

[9] D. Lomet. Grow and Post Index Trees: role, techniques and future potential. *Symp. on Large Spatial Databases (1991) Zurich.* In *Advances in Spatial Databases, LN in Computer Science 525*, 183–206, Berlin, 1991. Springer-Verlag.

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

[11] D. Lomet and B. Salzberg. Access method concurrency with recovery. *ACM/SIGMOD Conf.*, 351–360, San Diego, CA, 1992.

[12] P. Lehman and S. B. Yao. Efficient locking for concurrent operations on B-trees. *ACM Trans. on Database Sys.*, 6(4):650–670, Dec 1981.

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

[14] J. Nievergelt and Hinrichs. The Grid File: A Data Structure to Support Proximity Queries on Spatial Objects. *Int'l Workshop on Graph Theoretic Concepts in Computer Science*, 100–113, Linz, Austria, 1983.

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

[16] J. A. Orenstein and T. Merrett. A class of data structures for associative searching. *SIGART-SIGMOD Symp. on Prin. of Database Sys.*, 181–190, Waterloo, Canada, 1984.

[17] B. Salzberg. On indexing spatial and temporal data. *Information Sys.*, 19(6):447–465, 1994.

[18] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^{+}-tree: a dynamic index for multi-dimensional objects. *VLDB Conf.*, 1–24, Brighton, England, 1987.

[19] D. Shasha and N. Goodman. Concurrent search structure algorithms. *ACM Trans. on Database Sys.*, 13(1):53–90, Mar 1988.

[20] V. Srinivasan and M. Carey. Performance of B-tree concurrency control algorithms. *ACM/SIGMOD Conf.*, 416–425, Denver, CO, 1991.

[21] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The Sequoia 2000 Storage Benchmark. *ACM/SIGMOD Conf.*, 2–11, Washington, DC, 1993.