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

Georgios Evangelidis (giorgos@tsolias.civil.auth.gr)
Ph.D., currently fulfilling military obligation with the greek army
Leoforos Stratou 13, GR-54639 Thessaloniki, Greece, tel. 30-31-814-691

David Lomet (lomet@microsoft.com)
Ph.D., Senior Scientist, Microsoft Corporation
One Microsoft Way, Bldg 9, Redmond, WA 98052-6399, tel. (206) 703-1853

Betty Salzberg (salzberg@ccs.neu.edu)
Ph.D., Professor, College of Computer Science
Northeastern University, Boston, MA 02115, tel. (617) 373-2229

corresponding author: Betty Salzberg
Phone: (617) 373-2229, Fax: (617) 373-5121, E-mail: salzberg@ccs.neu.edu

Categories and Subject Descriptors: E.1 [**Data**]: Data Structures - Trees; E.2 [**Data**]: Data Storage Representations - Composite Structures; H.2.2 [**Database Management**]: Physical Design - Access methods; H.2.4 [**Database Management**]: Systems - Concurrency; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval - Clustering, Retrieval models, Search process.

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

The hB^{Π}-tree: A Multi-attribute Index Supporting Concurrency, Recovery and Node Consolidation

Georgios Evangelidis
Leoforos Stratou 13, GR-54639 Thessaloniki, Greece

David Lomet
Microsoft Corporation, One Microsoft Way, Bldg 9, Redmond, WA 98052-6399

Betty Salzberg
College of Computer Science, Northeastern University, Boston, MA 02115

Abstract

We propose a new multi-attribute index. Our approach combines the hB-tree, a multi-attribute index, and the Π-tree, an abstract index which offers efficient concurrency and recovery methods. We call the resulting method the hB^{Π}-tree. We describe several versions of the hB^{Π}-tree, each using a different node splitting and index term posting algorithm. We also describe a new node deletion algorithm.

We have implemented all the versions of the hB^{Π}-tree. Our performance results show that even the version that offers no performance guarantees, actually performs very well, in terms of storage utilization, index size (fan-out), exact-match and range searching, under various data types and distributions. We have also shown that our index is fairly insensitive to increases in dimension. Thus, it is suitable for indexing high-dimensional applications. This property and the fact that all our versions of the hB^{Π}-tree can use the Π-tree concurrency and recovery algorithms, make the hB^{Π}-tree a promising candidate for inclusion in a general purpose DBMS.

1 Introduction

Traditional Database Management Systems (DBMSs) efficiently organize, access, and manipulate very large quantities of data for business applications in banks, airlines, government agencies, hospitals, and other large organizations. Almost all of them implement some variation of the B^+-tree [BM72, Com79] for ordered single-attribute indexing.

In general, "traditional" data can be viewed as linear data. Each data item is identified by some attribute that distinguishes it from other data items and this is what we call a *primary* key for the data item. The primary key defines a total order for the data items in the database. B^+-tree indexes on a primary key are often (dynamic) *clustering* indexes as well, if the leaf level of the index consists of pages that contain the actual data items. Such a single-attribute clustering index is very efficient for answering range queries on the indexed attribute, since data items with comparable values for their indexed attribute will be in the same or neighboring pages.

However, today new applications that deal with "non-traditional" 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, and design databases for CAD and VLSI.

This data is called **multi-attribute** data. It must be organized in terms of more than one attribute, for example latitude and longitude and height above the earth.

A straightforward way to handle multi-attribute data is to use many single-attribute indexes (e.g., B^+-trees), one for each attribute of interest. Unfortunately, this is a highly inefficient solution. Only one of the indexes can be a clustering index and during insertions or deletions all indexes need updating.

To efficiently handle multi-attribute data one needs explicitly multi-attribute indexes. It is essential to cluster "nearby" k-dimensional data in contiguous areas of a disk. There is no "perfect" way to map multi-attribute data on linear physical disk storage, though. In general, the desirable properties of multi-attribute indexes are [Sal91]:

1. good space utilization in both index and data nodes,
2. high fan-out (the index should be significantly smaller than the data collection),
3. fast exact match search (given the coordinates, the data should be obtained quickly),
4. fair clustering in data pages by all attributes for good range search performance,
5. easy integration with the query, locking, and recovery systems of existing DBMSs,
6. simple design for incremental growth and shrinkage (insertion and deletion algorithms).

1.1 Related Work

There are two basic types of multi-attribute data:

Point data that can be viewed as geometrical points in a k-dimensional space without necessarily being geometrical data. Examples are (x, y, z) triplets in the three-dimensional Euclidean space, or (SSN, city) pairs in a two-dimensional space.

Spatial data that can be viewed as subspaces in a k-dimensional space, for example, polygons in the two-dimensional Euclidean space. This type of multi-attribute data is almost always geographical data.

Proposed point data indexing methods include Z-ordering [OM84], the grid file [NHS84], the K-D-B-tree [Rob81], and the hB-tree [LS90]. Z-ordering maps k attributes to a single one by alternating their bit representations. Then it uses any single-attribute index, e.g., the B^+-tree, to index the resulting attribute. The rest of the mentioned methods explicitly index multi-attribute data.

Proposed methods for spatial (or non-point) data include the R-tree [Gut84] and its variations (R^+-tree [SRF87], R^*-tree [BKSS90]), and the cell tree [Gue89]. An alternative approach maps k-dimensional spatial objects to $2k$-dimensional points. This can be done by taking the minimum bounding box of the spatial object and using a $2k$ dimensional point to represent it (low and high values for each attribute) [Hin85]. Then, any point data method can be used to index the transformed space.

Concurrency control in B^+-trees has been the subject of many papers [BS77, LY81, Sal85, Sag86, SG88, ML89, LS92]. Most of these papers, with the exception of [ML89, LS92], have not addressed the problem of system crashes during structure changes.

1.2 Our Approach

We introduce a new access method for multi-attribute data that we call the **hB^{Π}-tree**. It is based on the hB-tree, a multi-attribute point data access method [LS90] and the Π-tree, an abstract index tree for which a general algorithm for concurrency and recovery is available [LS92].

The Π-tree is essentially a generalization of the B^{link}-tree [LY81]. The B^{link}-tree has **side-pointers** or **links** pointing from each index node to the next index node on the same level of the tree in key order. Links enable index-term posting to be separated from index-node splitting, since when information about a split has not yet been posted to the parent, search can follow the links and still be correct. This implies that the locks needed for splitting can be dropped before the locks for posting are acquired.

A recent study compared the performance of various concurrency control algorithms [SC91]. Its most important conclusion was that algorithms using the link technique provide the most concurrency and the best overall performance.

The Π-tree generalizes the B^{link}-tree because it treats the multi-attribute case, it considers node consolidation, and it considers recovery as well as concurrency. We modify the hB-tree so that it becomes a special case of the Π-tree. This involves minor structural changes to the hB-tree.

In addition, and perhaps more importantly, we invent new node splitting and index term posting, and node consolidation and index term dropping algorithms. The new index node splitting and index term posting algorithms correct an error in the hB-tree. Also, the hB-tree had no node consolidation algorithm.

The result of this modification of the hB-tree is called the hB^{Π}-tree. In a series of experiments, we test the various splitting and posting algorithms and measure the overall performance of the hB^{Π}-tree.

1.3 Overview

In section 2 we review the Π-tree [LS92]. We modify the hB-tree [LS90] to become a subcase of the Π-tree. We call the new method the hB^{Π}-tree and we present its structural characteristics in Section 3.

Section 4 is devoted to the various node splitting and index term posting algorithms for the hB^{Π}-tree. Section 5 presents a node consolidation algorithm for the hB^{Π}-tree.

The splitting/posting and consolidation algorithms have been tested in the implemented hB^{Π}-tree under various data distributions. The data we used was either computer generated or geographical data from the Sequoia 2000 Storage Benchmark [SFGM93]. The results are shown in Section 6.

Finally, in Section 7 we summarize and give some directions for future work.

2 Π-tree Concurrency and Recovery

In this section we briefly review a general algorithm for concurrency and recovery for a wide class of index trees (single-attribute, multi-attribute, or versioned) [LS92]. This algorithm is applicable to an abstract index tree structure, the Π-tree.

The main idea is to make it possible to hold only short term locks on non-leaf nodes. This is achieved by making Π-tree structure changes consist of a sequence of atomic actions [Lom77]. Most of these actions are separate from the transaction whose update triggered the structure change and each one of them leaves the tree in an intermediate well-formed state.

What we do here is review the structure of a Π-tree and the concurrency and recovery algorithms applying to it. In the next section, we will modify the hB-tree only slightly (by adding side pointers) to make it a Π-tree. We will also add some features to aid in node consolidation. This will make the Π-tree concurrency, recovery and node consolidation algorithms applicable to the modified hB-tree.

2.1 Structure

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. For an index node it is the space the node distributes among its children nodes and is described by **index terms**, and for a data node it is the space where existing and potential data points lie.
- 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 called **side pointers**. If a node C contains a side pointer to a node X, C is referred to as the **container** node for X and X is referred to as an **extracted** node of C.

Each **level** of the tree begins as one node, the root. Node-splitting *always* creates a new sibling node *on the same level*. When information about a root-node split is posted, a new root node and a new level to the tree are created. Posting of other split information does not create new levels. Data nodes are all on the bottom level of the tree.

In the Π-tree it is possible for a node to be referred to by more than one parent. This happens whenever the boundary of a parent split cuts across a child boundary. Then both the old parent and its new sibling on the same level of the tree are parents of the child. This child is called a **multi-parent** node. (Nodes which are not multi-parent nodes are **single-parent** nodes.)

2.2 Searching

For exact match searches, a unique path, that may include side pointers, is followed down to the leaf (data) level. Searches start at the root of the Π-tree. At every node visited, if the search point is included in the node's directly contained space a child pointer is followed that leads to a lower level node. Otherwise, a side pointer is followed until a sibling index node that directly contains the search point is found. Eventually a leaf data node is reached. If necessary, side pointers at leaf level lead to a data node whose directly contained space includes the search point. The record for the search point will be present on that node, if it exists at all.

Since we are only interested in the case where node consolidation is possible, we must deal with the fact that nodes can be deallocated. Because of this, searching uses lock-coupling. The lock held on a node is not released until the lock on the next node (its child or sibling) is acquired. In the case an update is to be made at some level of the tree, the nodes are locked with an exclusive lock at the level of the tree where the update is to be made. (In [LS92] a slightly less restrictive lock is used at the update level; we are making a simplification.) On all other levels, or if no updates are to be made, the search locks are share locks.

If an update or read is to be to on a data item, only the database lock on the data item (of whatever granularity the database system supports) is held to end of transaction. The node locks of the searching algorithm of [LS92] need not be held once the search has finished and the data item is read or updated. (Database locks are not part of the Π-tree algorithms, which are mainly concerned with tree structure modifications.)

For range searches, multiple children nodes are visited at each level. These are the nodes whose directly contained space intersects the query window. Eventually, the multiple paths that are followed lead to all data nodes that satisfy the search.

2.3 Node Splitting and Index-Term Posting

When an insertion causes a Π-tree node to overflow, that node is split, with part of the contents going to a new sibling node, and a new index term is posted to the parent. In the Π-tree **node splitting** and the **index-term posting** are performed by separate **recoverable atomic actions**, as follows:

Node splitting: An updating process detects an Π-tree node C that is full and cannot accommodate the update. C is split and part of its contents are moved to a newly created node, X. Node splitting concludes by storing a sibling term and side pointer for X in C. Since C now contains a side pointer to X, C is a container node for X and X is considered extracted from C. (see Figures 1a and 1b). It is not true that C will forever be a container for X. Another node, Y could split from C at a later time and take with it the pointer to X. Then Y would be a container for X and C would be a container for Y. Containment/extraction in the Π-tree is determined by the existence of side-pointers.

Only the node C which is split need be locked (with an exclusive lock) during the node-splitting atomic action. If C is a data node, and the insertion is part of a transaction, the lock on C may need to be held to end of transaction. Splits above the data node level are never part of a database transaction. All splits immediately schedule index-term posting, once their locks are dropped.

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. Since a Π-tree node may have more than one parent, index posting may consist of several separate index-term-posting atomic actions. The node where the posting is to take place is exclusively locked during the posting. In addition, the container child to which the index term refers is locked briefly with a share lock, to check that a side pointer still exists whose information still needs posting. To check that the parent is the correct one, and that it has itself not been changed since the posting was scheduled, state identifiers in the parent are checked against those in the posting request. If they do not match, search for the correct parent begins again in the root. More details can be found in [LS92].

Node splitting is exactly the same (except for possible database locks) at the abstraction level of the Π-tree in [LS92] for both data and index nodes. However, a particular Π-tree (such as a B^+-tree)

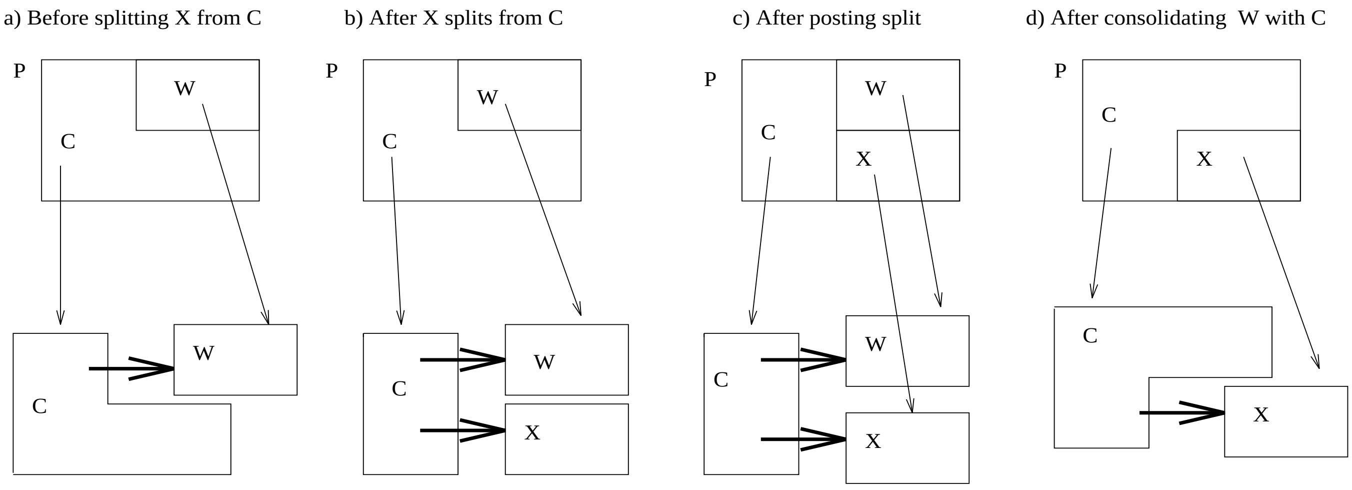


Figure 1: **Splitting, Posting, and Consolidating in the Π-tree.**

may have index terms which look quite different from data records. The details of such splitting will differ from example to example. Much of this paper discusses the details of splitting, index-term posting, and node consolidation for the hB^{Π}-tree. These cannot be deduced from the Π-tree.

When index-term posting involves multiple parents (because a parent split across a child boundary) or when a system failure interrupts index-term posting, 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 side pointer. That is, two instances of the Π-tree can be structurally different, because index-term posting has not been performed or completed, but semantically equivalent. Traversal of a side pointer results in scheduling the index-term-posting atomic action for the missing index term.

Below, we describe in detail the node-splitting and index-term-posting atomic actions:

Node-Splitting Atomic Action

1. A node C (Container) is overfull and cannot accommodate an insertion. C is locked with an exclusive lock.

2. A new node X (eXtracted) is allocated.

3. The directly contained space of C is partitioned into two parts.

4. One part remains in C and the other is moved (delegated) to X.

5. A sibling term for X is included in C. C is unlocked if C is an index node and not a data node. Data nodes involved in splits may have to hold locks until end of transaction.

Index-Term-Posting Atomic Action

0. **Scheduling:** either a tree traversal using a (multi-attribute) KEY traverses a side pointer, or a node split has just been performed, at LEVEL. This causes a posting action to be scheduled.

1. Using KEY find P (Parent), where P is the node at LEVEL+1 that contains KEY. Lock P with an exclusive lock.

2. Find the child C (Container) of P where KEY belongs. Lock C with a short-term share lock.

3. Using KEY find the sibling X (eXtracted) of C (by finding a side pointer).

4. If no such sibling X of C exists, EXIT (posting has already been performed or node has been consolidated) and drop all locks. If X exists, drop the lock on C.

5. *Post Index Term*: include in P an index term that describes the space of X, and update in P the index term that describes the space of C. Then drop the lock on P.

2.4 Node Consolidation

To improve storage utilization, a Π-tree node whose storage utilization drops below a prespecified threshold should be consolidated with another node that can be either its container node or a node that has been extracted from it.

In the Π-tree, regardless of whether the sparse node is a container or an extracted node, the contents of the extracted node are moved to the container node and the extracted node is deallocated. All references to the deallocated node must be removed from its parent(s). That is why in the Π-tree, unlike node splitting (which can be performed using multiple independent atomic actions), node consolidation has to be completed by a single atomic action.

We want atomic actions to involve as few nodes as possible, so we require three conditions for node consolidation:

1. both the container and the extracted nodes must be children of the same parent,

2. the extracted node must be a single-parent node, and

3. the container node has sufficient space to absorb the contents of the extracted node.

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

Node-Consolidation Atomic Action

0. **Scheduling:** a tree traversal using KEY visits a sparse node at LEVEL.

1. Using KEY find P (Parent), where P is the node at LEVEL+1 that contains KEY. Lock P with an exclusive lock.

2. Find the child S (Sparse) of P where KEY belongs. Lock S with an exclusive lock. It is not required that P refer to S. P need only have KEY in its directly responsible space.

3. If S is NOT sparse anymore EXIT and drop locks.

4. If no child of P (container of or extracted from S) is found to merge S with, EXIT and drop locks. Otherwise lock the child to merge with with an exclusive lock.

5. Let C and W be the container and extracted nodes (S can be either one of them).

6. If W is NOT single-parent EXIT and drop all locks.

7. *Drop Index Term*: remove index term for W from P, and adjust C's index term in P to include W's space. If the index term for W had never been posted, this step can be skipped.

8. *Drop Sibling Term*: replace pointer to W in C with W's contents.

9. Deallocate W. Drop all locks.

2.5 Concurrency Efficiency and Recovery Issues

An atomic action holds at most three exclusive locks on existing Π-tree nodes at all times: in the case of node splitting, it is on the node that is about to be split, and in the case of index-term posting, it is on the node where posting will take place. In the case of the node consolidation, it is on the parent and the two children. In the case of index-term posting, a short-term share lock is also held on the container child. Searching holds at most two locks at a time. No other locks are held by atomic actions. Because only a small number of locks are held and because most atomic actions are independent of database transactions, concurrency is efficient.

Atomic actions are logged. If there is a failure before an atomic action completes, it is rolled back. Complete atomic actions (actions all of whose *log records* are on disk) are redone at recovery if their results have not reached the disk. Thus, the transaction manager must know about atomic actions in the sense it knows about database or system transactions. More details can be found in [LS92].

3 The hB$^{\Pi}$-tree: Structure

In this section we first briefly review the hB-tree and then we describe the structural modifications that transform the hB-tree into the hB$^{\Pi}$-tree.

3.1 Review of the hB-tree

3.1.1 Structure

The hB-tree (or holey Brick tree) [LS90] consists of **index** and **data** nodes.

- **Index** nodes are responsible for k-dimensional subspaces. They contain a **kd-tree** [Ben79] which is used to organize information about children on the next lower level of the hB-tree and about regions which have been extracted and transferred to siblings on the same index level.

- **Data** nodes contain the actual data records. Data nodes may contain kd-trees, as well, that enable the data nodes to describe their own **inner** boundaries (collection of records, known as **record lists**, that occupy a k-dimensional region) and the extracted data-level sibling regions.

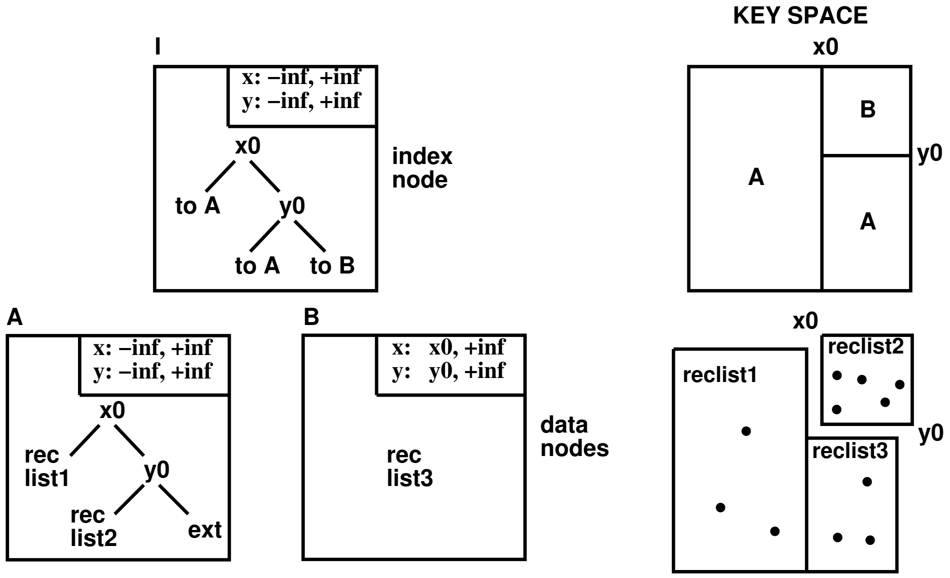
Figure 2 contains an example hB-tree. We have a two-dimensional attribute space. The root of the hB-tree is the index node I, that is responsible for the whole space. Its kd-tree describes the next level of the hB-tree, which consists of two data nodes, A and B. There are three kd-tree paths in I's kd-tree: paths ($x0$-left) and ($x0$-right, $y0$-left) constitute the index term for child node A, and path ($x0$-right, $y0$-right) constitutes the index term for child B. These kd-tree paths indicate that all records in which both $x \in [x0, +\infty)$ and $y \in [y0, +\infty)$ are located in node B (unless otherwise noted, equality is assumed to be on the right path in our kd-tree figures), and the rest of the records are located in node A. Data node A also contains a kd-tree. Its paths describe the spaces occupied by its two record-lists and the space that was extracted from A and was delegated to B. Finally, data node B contains only a record-list.

To summarize, a path from the root of a kd-tree to a leaf kd-tree node can represent either

1. an **index term** for index nodes (that describes a region of a child) or an **inner boundary** for data nodes (that describes the region occupied by a record list), or

2. a **sibling term** (that describes the region of a node that has been extracted from that node due to a prior split).

Figure 2: **A two-dimensional hB-tree with one index node and two data nodes**



In addition, each node stores a description of the space it is responsible for. These are two attribute values (low, high) for each attribute. These values are called the **boundaries** of the hB-tree node. For example, in Figure 2 the boundaries of I and A are the whole space, whereas the boundaries of B are (x: [$x0$, $+\infty$), y: [$y0$, $+\infty$)).

Unlike other multi-attribute indexes that split nodes by hyperplanes (e.g., the K-D-B-tree or the Grid File), the hB-tree can use more than one attribute to describe the extracted region. Therefore, nodes can be bricks with holes that represent extracted regions, hence the name holey Brick tree. For example, in Figure 2 node A is a brick from which a corner has been extracted.

(Note that splitting is not symmetric unless the kd-tree is split at the root. If the kd-tree is *not* split at the root, the boundaries of a node C describe a space which strictly contains the space described by the boundaries of the node X which has been extracted from C. In the case of a split at the root of the kd-tree, neither sibling is designated as the "extracted" or the "container" sibling. Also, for a split at the root, no sibling terms are created.)

3.1.2 Searching

Searching for point data is straightforward. We start the search at the hB-tree root. The root is searched by traversing its kd-tree. Every kd-tree node has information which includes an attribute and its value. By comparing this value to the value of the corresponding attribute of the search point, one can decide on which of the two children of the kd-tree node one should visit next.

This process leads us to lower levels of the hB-tree and eventually to a data node. That data node contains the region where the search point belongs, if it exists at all. Finally, the points of the node are searched with the help of the kd-tree of the node.

To illustrate, assume that $y0 = 0$ and $x0 = 0$ and assume that we are searching for point (10, -5) using the example hB-tree of Figure 2. We start from the root I of the hB-tree and we follow the kd-tree path the search point directs us to, that is, ($x0$-right, $y0$-left). This path leads us to data node A, where the same procedure using A's kd-tree leads us to record-list reclist2. This record-list is the place where our search point is located if it exists at all.

3.1.3 Node Splitting

When an insertion causes a data or index hB-tree node to overflow, its kd-tree is used to find a kd-subtree whose size is between one and two thirds of the contents of the node. For data nodes, if necessary, a record list is split using one or more attributes and new kd-tree nodes are introduced to describe the resulting subspaces. A new hB-tree node is allocated and part of the contents of the overflowing node are moved to it. When the split is not at the root of the kd-tree, a special marker, called an **external** marker, is included in the original node to indicate that part of its contents have been extracted. (When the split is at the root of the kd-tree, there is no external marker.)

For example, in Figure 2, A's kd-tree shows that A, which initially was responsible for the whole attribute space, was split by a corner (both x and y attributes were used). All records in which both $x \in [x0, +\infty)$ and $y \in [y0, +\infty)$ were moved to a newly allocated node B. Two kd-tree nodes, $x0$, and $y0$, were introduced to describe the new space decomposition. The resulting kd-tree in A describes A's inner boundaries and the extracted region (indicated by the external marker).

The kd-tree nodes in the path from the kd-tree root of the node to the extracted subtree which describe the extracted region and have not been posted during another splitting are posted to the parent hB-tree node. Posting of index terms may trigger additional node splits at higher levels. In Figure 2, kd-tree nodes $x0$ and $y0$ were posted in the index node I to describe the splitting of node A.

When the splitting boundary of an index node intersects the space of a child, that child becomes a **multi-parent** node. Its address is stored both in the original and the new sibling node. All of the parents of a multi-parent node are on the same level of the tree since all splits create a new node on the same level of the tree as the original node.

3.2 Making the hB-tree a Π-tree

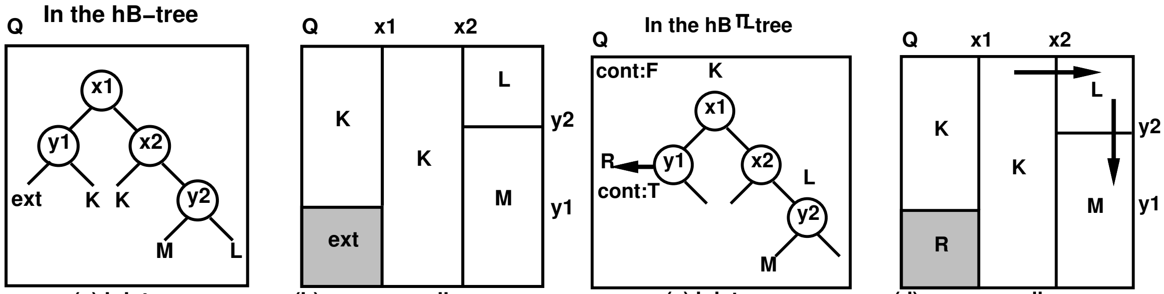
In the following discussion we describe the transformations that make the hB-tree a Π-tree. In order to transform the hB-tree into a case of the Π-tree when a split at node N occurs, we need to place the actual address of the node that is extracted from N in N. When the split is not at the root of the k-d-tree, we will simply replace external markers with this address. We must also treat the case of a split at the root. We shall use Figure 3 to illustrate these changes and the changes we make to support node consolidation.

Figure 3a shows the intra-node organization of an index hB-tree node Q, and Figure 3b shows the corresponding space decomposition. Each path in the kd-tree of node Q corresponds to either an index term or a sibling term. Here, the path ($x1$-left, $y1$-left) corresponds to a sibling term and describes space that has been extracted from Q and delegated to a sibling on the same level as Q. This space is represented by the shaded region of Figure 3b. The remaining four paths in the kd-tree of Q correspond to index terms. They describe the decomposition of the space node Q is directly responsible for among its children, namely nodes K, L, and M. These spaces are represented by the white regions of Figure 3b.

3.2.1 Side-pointers

The first and most important modification is the replacement of external markers by pointers to the extracted nodes, called **side-pointers**. In Figure 3c the thick arrow with the address of node R represents the side-pointer that is now used in the place of the external marker. Right after the split of R from Q, R will contain a k-d tree which had been a subtree of the k-d tree which was in Q. R is on the same level as Q. R is not on the same level as L, M and K, which are children of Q. In Figure 3d the shaded region represents the space the extracted node R is responsible for.

Figure 3: **Intra-node organization of hB-tree and hB$^{\Pi}$-tree nodes using kd-trees**



(a) kd−tree (b) corresponding space (c) kd−tree (d) corresponding space

Now, we can be lazy about posting index terms that describe a node split. A search that should visit the extracted node can still visit it through its container node.

For example, in the hB-tree the index term that describes the extraction of node R from node Q (see Figure 3a) must be posted in the parent of Q at the time of the split, otherwise one cannot visit node R. In the hB$^{\Pi}$-tree, the inclusion of the side-pointer to R in Q allows us to perform the posting any time after a split without affecting search correctness (see Figure 3c).

3.2.2 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 the root. In the original hB-tree, one of the subtrees becomes the kd-tree of 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 no external marker is created for this split. In the hB-tree, splitting at the root produced symmetric children where neither can be identified as the "container" or "extracted" child.

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. Now the container/extracted relationship is also indicated by the side pointer from the container to the extracted sibling node. In this case, we are transforming a symmetric relationship to an asymmetric one, where one sibling is arbitrarily declared to be the container and the other the extracted sibling. Π-trees require this asymmetry.

3.3 Supporting Node Consolidation

To support node consolidation, we make further structural changes. Given an hB$^{\Pi}$-tree node N, the idea is to be able to determine the following information without having to visit extra hB$^{\Pi}$-tree nodes:

1. the containment order of the children of N and

2. whether a child node of N is multi-parent or not.

For example, in the hB-tree of Figure 3a, we are not in position to know the above described information: (1) we do not know the address of the node that was extracted from Q, (2) we do not know whether the child L of Q was extracted from the child M of Q, or vice versa, and (3) we do not know whether the child K of Q is multi-parent or not.

3.3.1 Decorations

To determine the containment order of the children of a node, we begin by associating each k-d-tree node with the the address of the hB$^{\Pi}$-tree node it was posted from. We use the term **decoration** for the children addresses associated with k-d-tree nodes.

We can achieve space savings and reduced bookkeeping by following two conventions. First, a kd-tree node that has the same decoration as its parent kd-tree node does not need to store the decoration again (for example, in Figure 3c kd-tree nodes $y1$ and $x2$ did not store their decoration since they were sharing the same decoration with their parent). Second, if a kd-tree node's left or right pointer is a child-pointer and is the same as the kd-tree node's decoration then that pointer does not need to be stored (for example, in Figure 3c, kd-tree nodes $y1$ and $y2$ did not store their right pointers and kd-tree node $x2$ did not store its left pointer since they were the same as their decorations). We say that decorations and child-pointers that are not stored are NULL. Note that decorations are relevant to index nodes only. Data nodes do not have children.

The collection of kd-tree nodes with the same decoration forms a **decorated fragment** that describes the partitioning of the space of the child node appearing as decoration among its siblings. We observe that now one can determine the **containment order** of the children of a node by just looking at the kd-tree of that node. For example, the kd-tree of node Q in Figure 3c 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 arrows in the space decomposition of Figure 3d indicate the containment order of the children of node Q.

3.3.2 Continuation flags

Whenever a decorated fragment is partitioned by an index hB$^{\Pi}$-tree node split, the extracted kd-subtree is decorated with the same decoration as the split decorated fragment. After the completion of the split, the child hB$^{\Pi}$-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.

We distinguish the multiple parents of an hB$^{\Pi}$-tree node N, by referring to the parent where the original root of the N-fragment is located as the **prime** parent of N. All the other parents of N, where the N-fragment is continued, are called **secondary** parents of N. We also refer to the concatenation of the decorated fragments for a node in its parent nodes as the **full decorated fragment** for that node. The root of the full decorated fragment is always located in the prime parent.

Since node consolidation in the Π-tree requires that the node that is being deleted is referenced only by a single parent, we have to be able to detect whether a node is multi-parent or not by examining its current parent. To accomplish this, 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 decorated 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. For example, the TRUE continues-to flag of the side-pointer to R in Figure 3c indicates that the child node K of node Q is multi-parent, and that node R is its other parent.

In addition, some of the index term posting algorithms that we describe in the next section, require that we are able to determine whether a parent node P of a node N is the prime parent or a secondary parent of N. Clearly, if N is the decoration of a kd-subtree of P's kd-tree, P is the prime parent of N. But if N happens to be the decoration of the root of P's kd-tree, it may be the case that the N-fragment in P is a piece of the full N-fragment. For this reason all hB$^{\Pi}$-tree nodes also store a special **continues-from** flag. When a decorated fragment is partitioned by an index node split we make the continues-from flag of the extracted node TRUE, to indicate that it is a secondary parent of the child node that appears as the decoration of the root. For example, the FALSE continues-from flag of hB$^{\Pi}$-tree node Q in Figure 3c indicates that Q is the prime parent of K. Note that regardless of the value of Q's continues-from flag, Q is the prime parent of both L and M.

3.4 Terminology and Notation

Table 1 shows the terminology that we will be using in the following sections.

Table 1: **hB$^{\Pi}$-tree terminology**

Term	Description
hB$^{\Pi}$-tree node boundaries	$2k$-dimensional array stored in an hB$^{\Pi}$-tree node that describes the space the node is responsible for
index term	kd-nodes in the path(s) from the root of a kd-tree to a child node pointer
sibling term	kd-nodes in the path from the root of a kd-tree to a sibling node pointer
decorated fragment	set of kd-tree nodes with common decoration
full decorated fragment (e.g. full A-fragment)	the concatenation of all the decorated fragments for a node in its multiple parents
decorated subtree	kd-subtree rooted at a decorated kd-tree node
P, C, X	(P)arent, (C)ontainer and e(X)tracted nodes
PtoC-path	a path from the root of the C-fragment in P to C
CtoX-path	the path from C's kd-tree root to X

In Figure 3c, the index term for node K consists of all the kd-tree nodes in the two paths from the root of Q's kd-tree to K, i.e., $x1$, $y1$, and $x2$. 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 PtoC-path's: ($x1$-left, $y1$-right) and ($x1$-right, $x2$-left). In the space decomposition figure (3d), index terms for the children of Q correspond to one or more white rectangles, and their containment order is represented by the thick arrows.

Also, in Figure 3c, the sibling term for node R consists of all the kd-tree nodes in the path from the root of Q's kd-tree to R, i.e., $x1$, and $y1$. 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 space decomposition figure (3d), the sibling term for the sibling R of Q corresponds to the shaded rectangle.

4 The hB$^{\Pi}$-tree: Splitting/Posting

In the previous section we showed how kd-trees are used to represent index and sibling terms. In this section, we describe in detail how kd-trees should be split, how sibling terms are created during hB$^{\Pi}$-tree node splitting, and finally, how index terms should be posted, and more importantly, what they should look like.

4.1 Description of A Problem

We begin by showing here how one could adapt the original hB-tree node splitting and index term posting algorithms (described in [LS90]) for use in the hB^{Π}-tree. This is a straightforward adaptation since the main innovation of the hB^{Π}-tree are the side-pointers and the fact that node splitting and index term posting are performed by separate atomic actions.

We then show that the original algorithms are flawed and that an hB^{Π}-tree (or an hB-tree) that uses them is not well-formed. In particular, search is not correct. Our new algorithms, described in section 4.2, will correct this flaw.

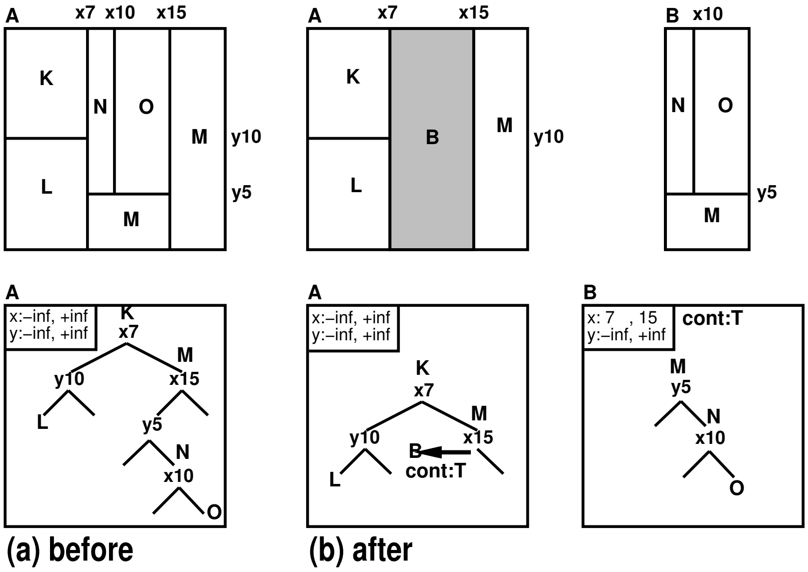
4.1.1 Original Index Node Splitting

We distinguish index node splitting from data node splitting. Index nodes contain a kd-tree only, whereas, data nodes contain either a collection of records known as a record-list, or a kd-tree and one or more record-lists.

Index node splitting is straightforward. In [LS90] it is shown that one can always split off a kd-subtree with between one third and two thirds of the kd-tree nodes. This is the extracted kd-subtree that is moved to the extracted node (a newly allocated hB^{Π}-tree node). The root of the extracted kd-subtree takes the decoration of the decorated fragment it belonged to. The parent of the root of the extracted kd-subtree is made to point to the extracted node. This is done by replacing the child pointer of the parent corresponding to the extracted kd-subtree with a pointer to the extracted hB^{Π}-tree node.

For example, in Figure 4, node B was split off node A. (In Figure 4 and subsequent figures, we use the notation xn as shorthand for $x = n$.) The kd-subtree rooted at $y5$ was extracted and a side-pointer to B, indicated by the thick arrow, was included in A's kd-tree. Note that the child M of A is now referenced to by both A and B and hence is a multi-parent node. Also, note that the space directly contained by A has now become holey.

Figure 4: **Original index node splitting**



The procedure for locating the root of the extracted kd-subtree is very simple:

1. Start from the kd-tree root.
2. Visit the child that represents the root of the larger kd-subtree.
3. If the size of that kd-subtree is between one third and two thirds of the page size, then this is the root of the extracted kd-subtree, else repeat from step 2.

Note that the extracted space is described by the kd-tree path from the root of the split node to the extracted kd-subtree. This is always a k-**dimensional region** (or brick).

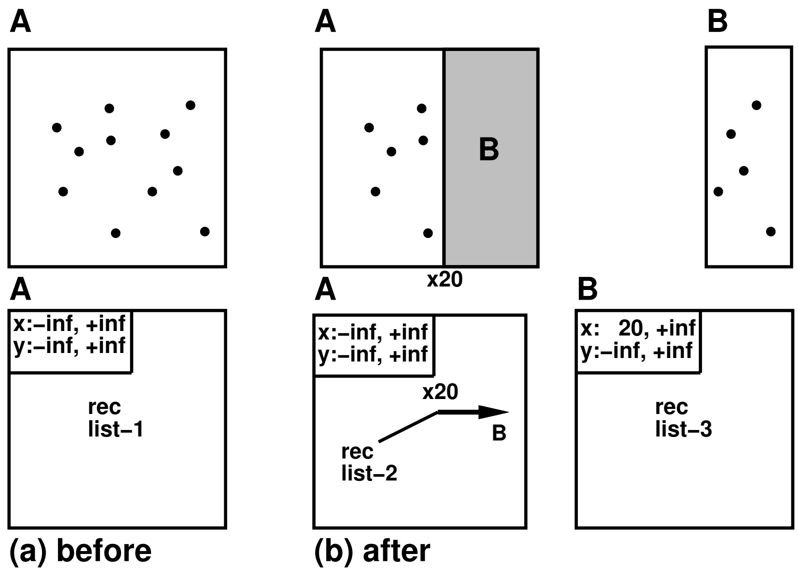
4.1.2 Original Data Node Splitting

To describe data node spitting, let's assume that we start with an hB$^{\Pi}$-tree that has a single data node (and, of course, no index nodes at all). That node will contain a single record-list. Once the node becomes full and cannot accommodate another insertion, it has to be split.

In the hB$^{\Pi}$-tree (and the hB-tree) one can use simultaneously more than one attribute to split a record-list. We talk about **hyperplane** or k-**dimensional corner** splits when one or k attributes are used, respectively. Every record-list keeps track of the attribute that was last used to split it. The splitting algorithm uses this information to split the record-list by a different attribute next time in a round-robin fashion.

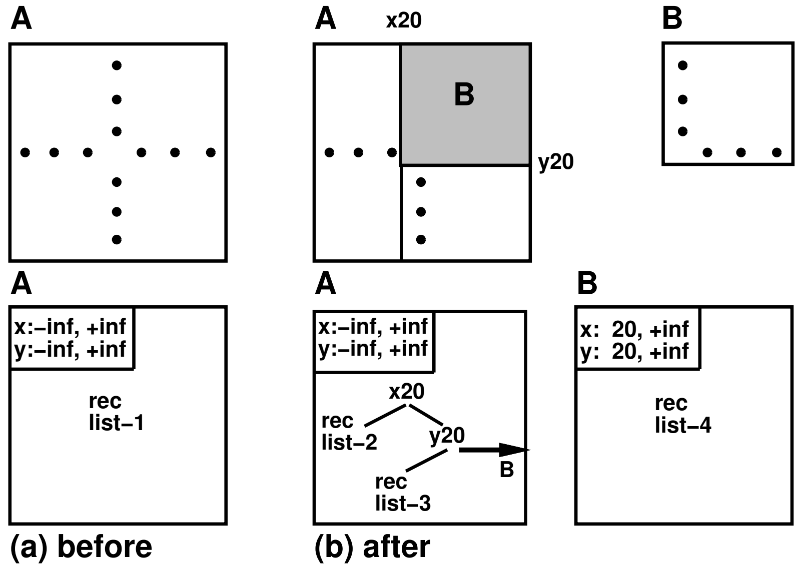
We first try to split our record-list using the current splitting attribute and so that at worst a one third/two thirds split is made. If this is not possible, then we try with the next attribute and so on. If a split with one attribute is possible (this was almost always the case in our experiments), we create a new kd-tree node whose attribute is the same as the splitting attribute. One of its children is the remaining record-list and the other a side-pointer to the extracted node, where the extracted record-list is moved. For example, in Figure 5 data node A was split and all records with $x \geq 20$ were extracted to a newly created node B. The appropriate kd-tree node was included in A to describe the node's "inner" boundaries and the extracted region.

Figure 5: **Data node hyperplane split**



In the case that no hyperplane split is possible, we can use more than one attribute, and then, we have a corner split. For example, in Figure 6 there is no way to extract between one third and

Figure 6: **Data node corner split**



two thirds of the node's contents by performing a hyperplane split. On the other hand, we can use both indexing attributes and extract all records that have both $x \geq 20$ and $y \geq 20$. This kind of split introduces two kd-tree nodes in node A that describe the resulting space decomposition. In [LS90], it is shown that it is always possible to achieve a one third to two thirds corner split.

If there is a kd-tree in the data node, we may be able to find and extract a kd-subtree that refers to a collection of record-lists, instead of splitting a record-list. In that case, data node splittings do not introduce new kd-tree nodes.

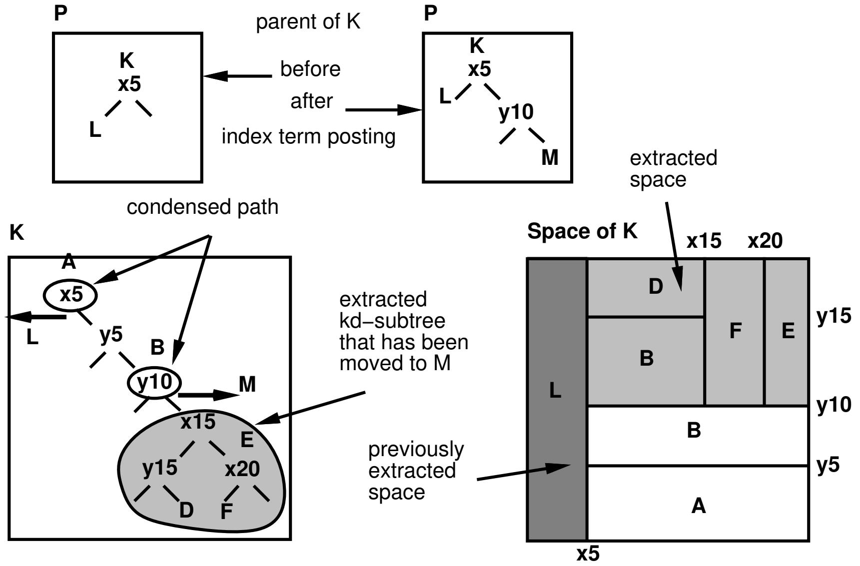
4.1.3 Original Index Term Posting

Suppose a node C (for Container) is split and a new node X (for eXtracted) is created. Once the node splitting atomic action is over, or when the side-pointer from C to X is traversed, we want to post the description of the split to the current parent node P (for Parent) of C. The objective of index term posting is to include (post) a description (index term) of X's space in P, and also, to modify the already existing index term in P for C so that it reflects the new (shrunk) space of C.

For example, in Figure 7 the shaded kd-subtree was split off node K and now resides in node M. The parent P of K initially only contains index terms for its children K and L. Index term posting has to include in P the minimum number of kd-tree nodes in the path from the root of the kd-tree of the container node K to the extracted node M that describe the extracted space. This is the so-called **condensed path**. In our example, it consists of kd-tree nodes $x5$ (that has already been posted) and $y10$. Note that $y5$ does not belong to the condensed path, since $y10$ is a tighter boundary for the extracted space. Once posting has been performed, node P contains a new index term for M, and has an updated index term for K: before the boundaries for K were ($x \geq 5$ and $y =$ anything) and now they are ($x \geq 5$ and $y < 10$).

We have just sketched the basic ideas behind index term posting. We will not describe this procedure in more detail. The reason is that we have found that the posting algorithm for the hB-tree [LS90] is not correct. In the following sections, first we demonstrate the problem, and then we present in detail various alternative splitting and posting policies that remedy the problem.

Figure 7: **Original index term posting**



4.1.4 hB-tree Splitting/Posting Algorithm Flaw

The directly contained space of a data hB^{Π}-tree node, i.e., the one that does not include space that has been delegated to a sibling node due to a split, 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 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/or the remaining directly contained space of the nodes do not correspond to a DSB there will be search correctness problems. We say that such splits do not **preserve** DSBs. Also, we say that a kd-tree **defines** DSBs when the space decomposition it describes preserves DSBs.

With the help of the scenario demonstrated in Figures 8 through 11, we show that the splitting and posting algorithm of the hB-tree [LS90] is erroneous. These figures use an hB^{Π}-tree, but the argument we will use also holds for hB-trees. In the first three phases of our scenario we can see the state of an hB^{Π}-tree after:

- C2 was extracted from C1, and the condensed path ($y5$, $x10$) was posted to B1 (see Figure 8),
- B2 was extracted from B1, and the condensed path ($y5$) was posted to A1 (see Figure 9),
- C3 has just been extracted from C1, and we are ready to post the index term for that split (see Figure 10).

The algorithm described in [LS90] does not cope with space decompositions that do not preserve DSBs. As shown in Figure 11, it would post kd-tree node $x5$ above $y5$ in B1, and change the C1 decoration to a C3 decoration in B2. Note that in the hB^{Π}-tree we need two separate index term posting atomic actions to do this. This is because node C1 happens to be a multi-parent node.

But, then we would have a major problem. A process searching for point (2, 9), which is in C1, would be directed from node A1, to node B2, and eventually to node C3. That is, search is not correct. The problem occurred because node B1 was split along a boundary that did not preserve DSBs.

What is needed to correct the hB-tree flaw is to preserve DSBs at the index levels. We have found various policies that accomplish this. Basically, they involve restrictions on the places where nodes

Figure 8: **Phase 1: C2 was split off C1 and the condensed path ($y5$, $x10$) was posted to B1**

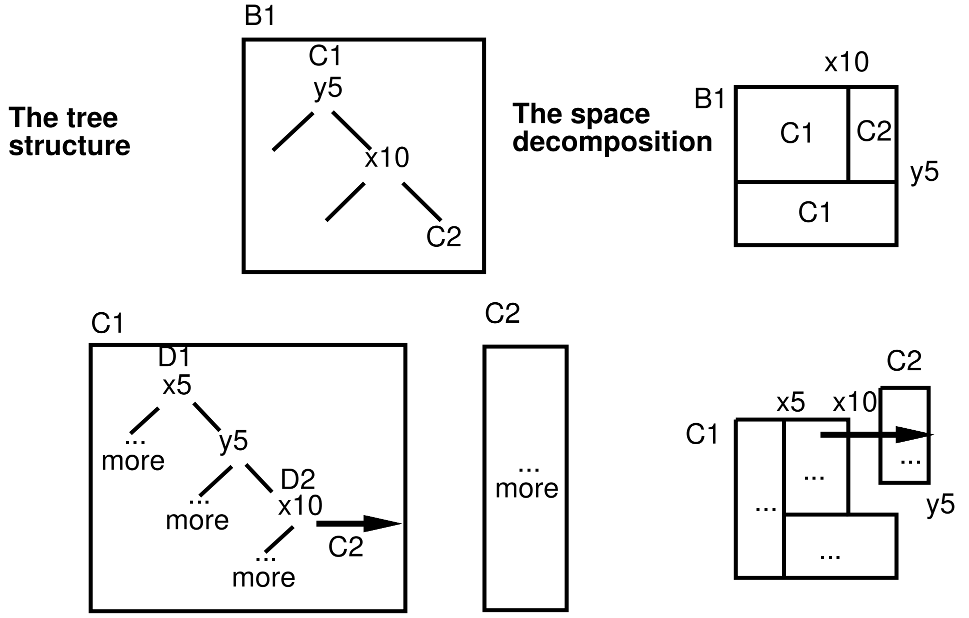


Figure 9: **Phase 2: B2 was split off B1 and the condensed path ($y5$) was posted to A1**

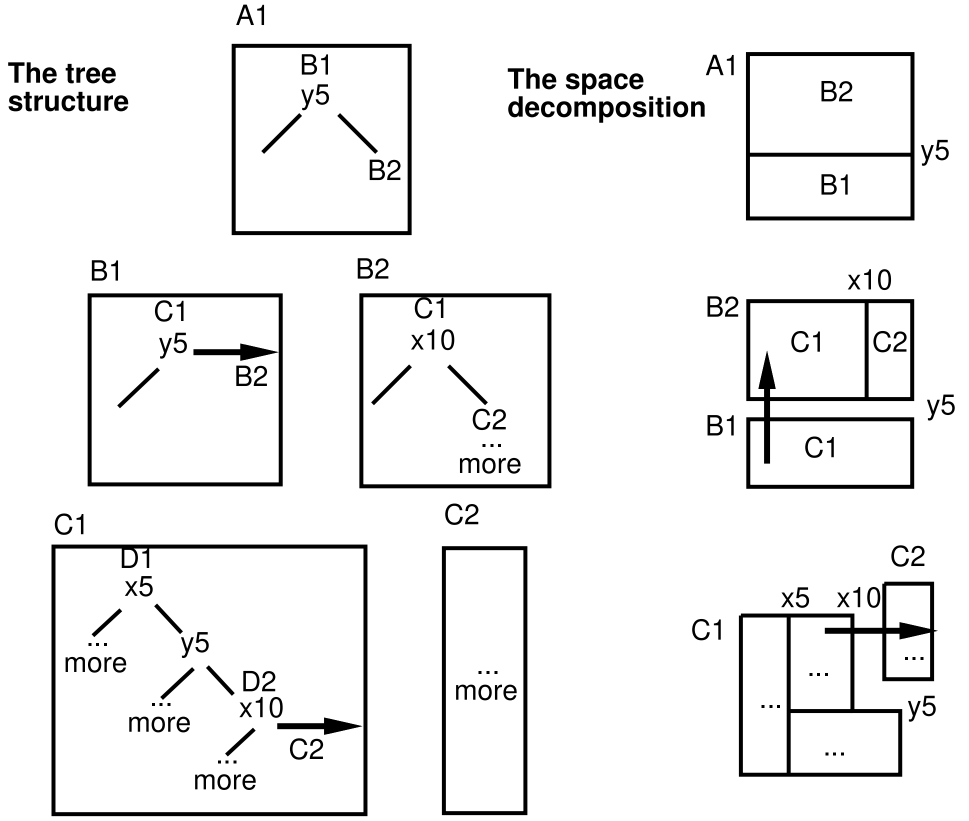


Figure 10: **Phase 3: C3 was split off C1 and we are ready to post the condensed path**

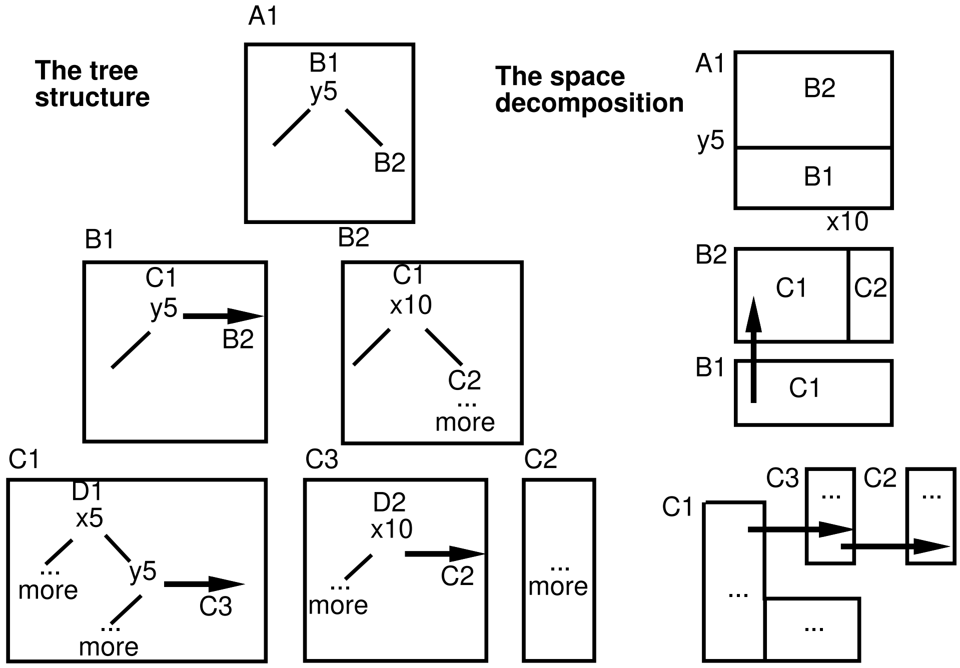
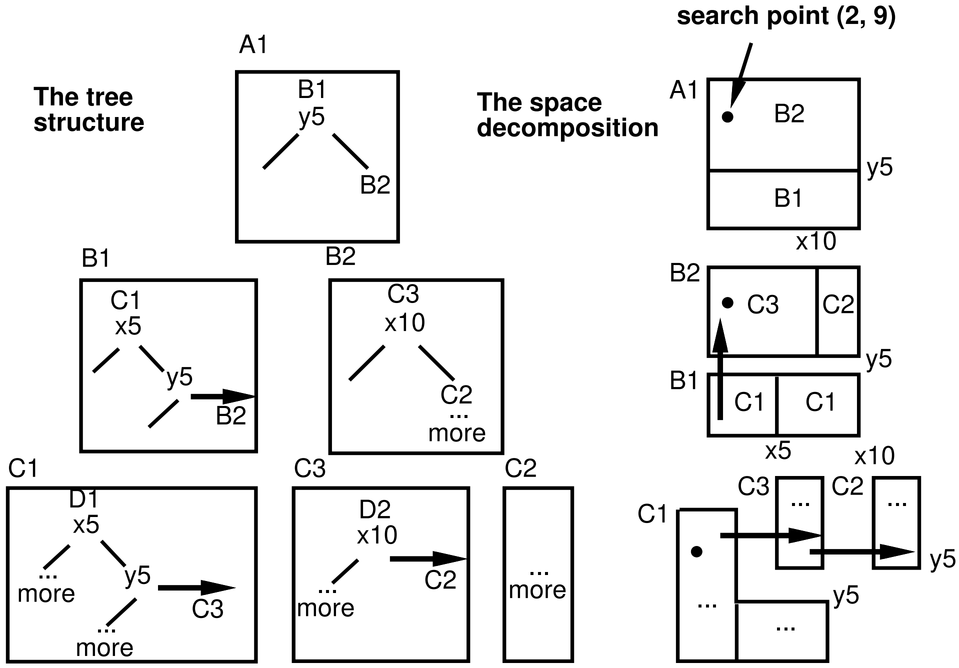


Figure 11: **Phase 4: The condensed path** ($x5$, $y5$) **was posted to B1, and the C1 decoration was changed to a C3 decoration in B2, but now searches for point (2, 9) are not correct**



can be split, or the nature of the index term that is being posted. In the next section we present a hierarchy of the various resulting splitting/posting algorithms that preserve DSBs.

4.2 New Approaches to Splitting and Posting

We introduce three different approaches to splitting and posting which correct the flaw in the hB-tree. All of these approaches preserve DSBs. One of them, the CB (Complete Boundaries), has the best performance characteristics. However, implementors may prefer to use the other approaches for reasons of algorithmic simplicity as the performance differences are not great. In any case, reading the descriptions of the simpler approaches aids in understanding the Complete Boundaries approach.

4.2.1 Solution 1: Post Full Paths (fp)

One way to preserve DSBs is to always post the **full** path during index term posting. Here the interesting thing to observe is that this reduces posting to appending to or relabeling (with new decorations) an already posted k-d tree. (This is already the case when posting index terms from the data level splits in the hB-tree.)

Claim: Posting reduces to appending or relabeling when full paths are always posted.

Proof of Claim: First we treat the case where scheduled posting occurs in the same order as the corresponding node-splitting. We also assume that there is only one parent node for the node(s) to be split, and it has room for all the new postings.

Suppose first we extract X from C. We then post the full path from the root of the kd-tree in C to the root of the kd-tree of X (the CtoX-path) to the parent P. The next split could be at C or at X. If it is at X, suppose Y is extracted from X. Then the path from the root of the kd-tree in X to the root of Y is appended to the path already in P.

Suppose, instead that the second split is in C. Say a node Z is extracted from C. Then the path from the root of the kd-tree in C to the root of the kd-tree in Z intersects the CtoX-path (possibly only at the root of the kd-tree in C). The new path then is connected to the already posted CtoX-path where they intersect by appending the part that is not shared. It is also possible that the new path is a subpath of the already posted CtoX-path. Then the already posted CtoX-path need only be decorated with the address of Z in the appropriate place. This new decoration is a kd-tree node decoration if Z now contains the side-pointer to X, or it is a kd-tree leaf decoration if not.

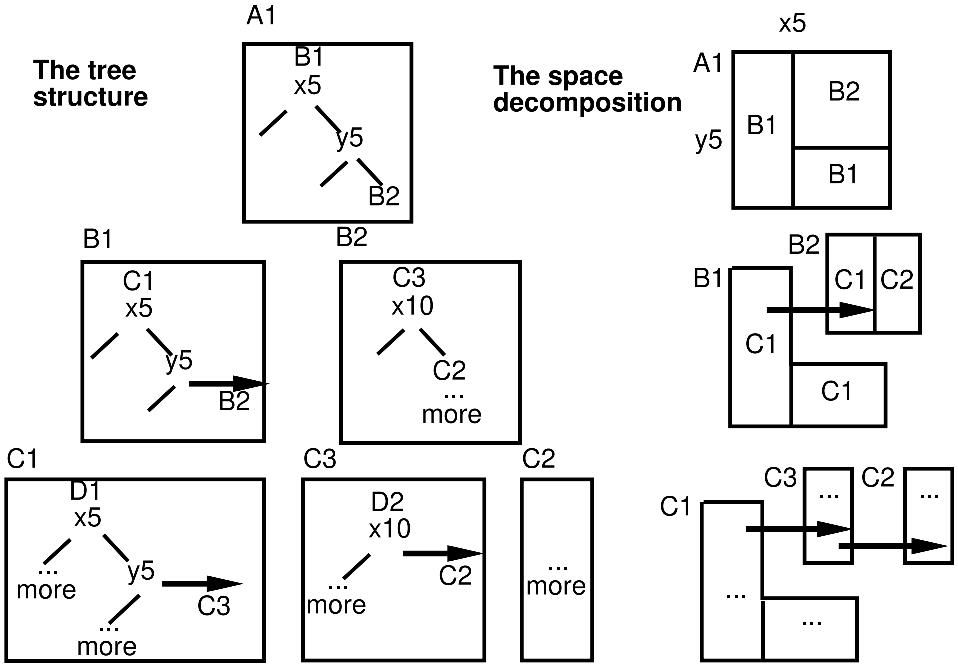
Now suppose we have asynchrony and we are posting full paths. Can it be that a later posting is missing the earlier split information to append to? This might happen if A split from B and the information was not posted and then D split from A. However, we adapt the policy of posting only split information from the first side pointer we see in a chain starting from a child whose address we already have. Using this policy, posting full path always appends to an existing path.

Now suppose that the parent P has been split and part of the posted information is in one node and part is in a sibling. We claim that this does not change the fact that new posting information is always appended to an existing path. This is because the kd-trees on one level of a hB^{Π}-tree can logically be considered as forming one large kd-tree by concatenating, following sibling pointers. **This proves the claim.**

Thus, the **kd-tree** (concatenation of all kd-trees) **of a level** of the hB^{Π}-tree is a prefix of the kd-tree of the level below. Since the level 0 (data level) kd-tree defines DSBs, the same will be true for the level 1 kd-tree, and so on. Hence, a path from the root of a kd-tree to any kd-tree node in the kd-tree defines DSBs.

Figure 12 shows the hB^{Π}-tree of Figure 10 if we post the full paths. Now, kd-tree node $x5$ was posted to B1 as part of the index term for C2. Therefore, the index term for the subsequent splitting

Figure 12: **Posting the full paths: kd-trees at index levels define DSBs**



of B1 defines a DSB.

There is a drawback in posting full paths, though, that has to do with the size of the index terms. The hB-tree uses condensed paths in order to have a worst case guarantee of $2k + 1$ kd-tree nodes for the size of the index term. Posting full paths cannot offer any guarantee on the size of the index term.

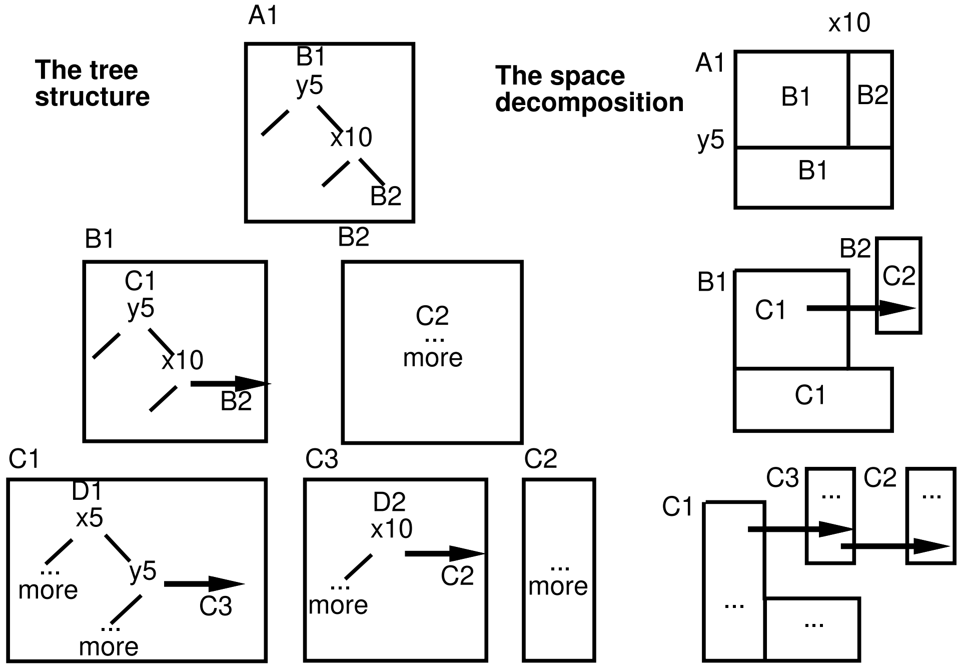
4.2.2 Solution 2: Split at Decorations Only (D)

Another way to preserve DSBs is to split nodes only at boundaries defined by their children. This is also a recursive procedure that preserves DSBs across the tree levels. By definition, data nodes define DSBs because they are collections of record lists. At higher hB^{Π}-tree levels, when index hB^{Π}-tree nodes are split only at boundaries defined by their children (so that the root of the new hB^{Π}-tree node was a decorated kd-tree node before the split), no multiparent nodes are created. (We call this policy **splitting at Decorations**) This means each new hB^{Π}-tree index node has the boundaries of the union of its children. Recursively, these are DSBs.

Figure 13 shows the hB^{Π}-tree of Figure 10 when nodes are split at decorations only. Node B1 was not allowed to be split by extracting the kd-subtree rooted at $x10$. Instead, the extracted kd-subtree was the right child of $x10$ whose root was C2 decorated (not shown in the figure). Therefore, the index term for B2 in A1 does include $x10$. Note that in order to split off C3 from C1 we used a decorated kd-subtree, too.

One advantage of splitting at decorations is that since no multi-parent nodes are ever created, index term posting can always be completed with a single atomic action. On the other hand, index nodes cannot be split anywhere. This can degrade index node storage utilization. Whenever it is not possible to find a decorated kd-subtree with between one third and two thirds of the size of a page, we have a "bad" quality split.

Figure 13: **Splitting at decorations: kd-trees at index levels define DSBs**



4.2.3 Solution 3: Split at Complete Boundaries Only (CB)

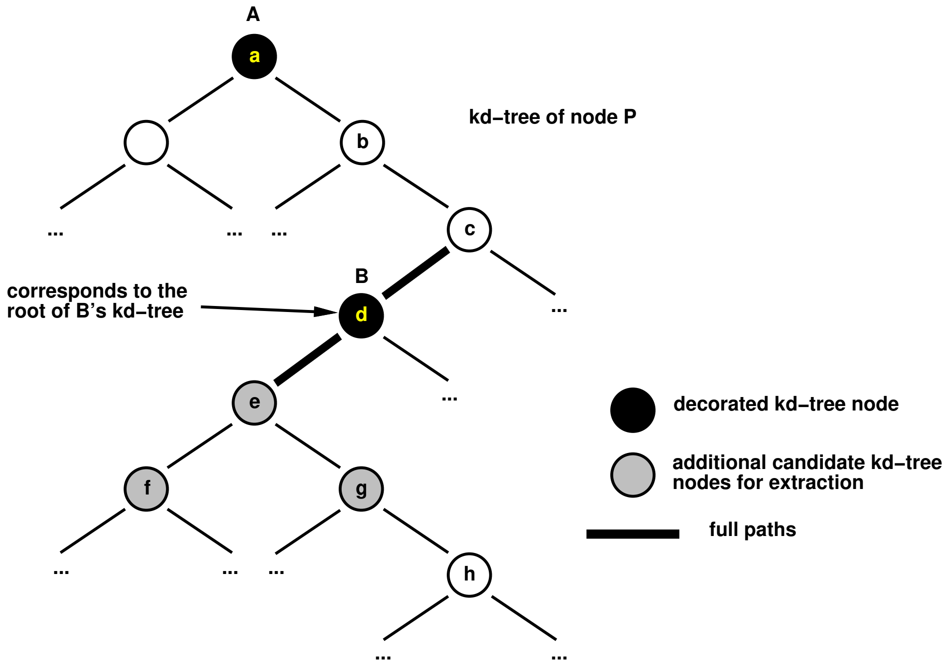
The two previous solutions tell us that we have DSBs anywhere in a kd-tree when we always post the full paths, or at decorated kd-tree nodes regardless of the posting policy (full or condensed paths). In this section we describe a new splitting policy which is a combination of the two previous solutions. It assumes that the condensed paths are being posted but offers more candidate kd-subtrees for extraction (in addition to the decorated ones).

Let us assume that we have a decorated kd-tree node, i.e, a root of a decorated fragment, in an index hB$^{\Pi}$-tree node at level L. Let n_1 be that kd-tree node and n_0 be its parent kd-tree node (which, of course, belongs to a different decorated fragment). Imagine that the kd-tree path n_0, n_1, n_2, ..., n_p, $p \geq 1$, is full, i.e., there are no missing kd-tree nodes between any two kd-tree nodes in the path. In other words, we already know that the path from the the previous decoration that ends in n_0 defines DSBs (it is the index term for the child node appearing as the decoration of n_1), and we now extend it with a full path that, by definition, also defines DSBs. We can extract the kd-subtree rooted at n_1 because n_1 is decorated. But we can also extract kd-subtrees rooted at any child of a node on the path n_1, n_2, ..., n_p, and still preserve DSBs.

For example, in Figure 14 we can see the kd-tree of node P. Two of the children of P are nodes A and B. We can see that kd-tree node d corresponds to the root of the kd-tree of node B. This is because it is decorated and there are no missing kd-tree nodes above it. Also, kd-tree node c corresponds to the kd-tree node that carries the side-pointer to B in the kd-tree of A that is the container of B. We can extract any of the following kd-trees: (a) the one rooted at d because d is decorated (solution 2), (b) the one rooted at e because there are no missing kd-tree nodes above d, (c) the one rooted at f because there are no missing nodes above d and e, and (d) the one rooted at g for the same reason we can extract the one rooted at f.

We say that these additional splitting points in the kd-trees of index nodes define **Complete Boundaries**. In order to make this splitting approach work, we need to have a way to determine

Figure 14: **Splitting at complete boundaries: kd-trees at index levels define DSBs**



whether a kd-tree path is full or not. The bookkeeping needed to implement this splitting approach will be described in a later section.

The complete boundaries are not the only places where DSBs are preserved when posting the condensed paths. Imagine that there exists a kd-tree path n_1, n_2, ..., n_p ($p \geq 2k$), with no missing kd-tree nodes, that contains both a low and a high boundary for each one of the k indexing attributes. Then, if the kd-tree path n_p, n_{p+1}, ..., n_q, $q \geq p$, is also full, clearly, a kd-subtree rooted at any child of a kd-tree node in that path is a candidate for extraction.

It is not very likely that there will be full paths which contain low and high boundaries for every attribute and which are also reasonably short. Even for a two-dimensional hB$^{\Pi}$-tree such a path will have to be at least four kd-tree nodes long and may be much longer. Therefore, we choose not to consider this scenario in our "complete boundaries" solution.

4.3 Algorithm Hierarchy

In the previous section we defined one new policy for posting index terms and two new policies for splitting nodes. We summarize all the existing node splitting and index term posting policies, and we present a hierarchy of splitting/posting algorithm that preserve DSBs at the index levels.

The term **splitting policy** refers to the the way index nodes are split. When splitting an index node we have three options regarding the way we split its kd-tree:

A Anywhere: This policy may split a decorated fragment Consequently, multi-parent nodes may be introduced. At most one multi-parent node is introduced per split.

D At decorated kd-subtrees: That is, we always extract a decorated subtree. All hB$^{\Pi}$-tree nodes are single-parent nodes.

CB At complete boundaries: In addition to splitting at decorated fragments, we can split at any kd-tree node after a decorated kd-tree node, as long as the path from the parent of the decorated kd-tree node to that kd-tree node is full, i.e., there are no missing kd-tree nodes. This policy introduces multi-parent nodes, as well.

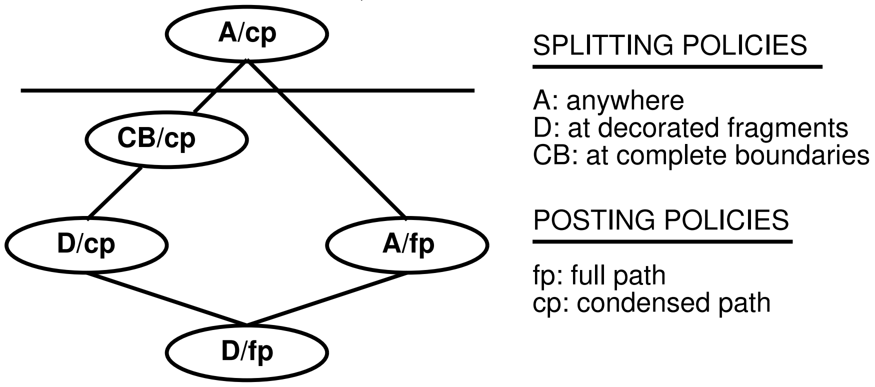
The term **posting policy** refers to the nature of the index terms that are posted. An index term is comprised of copies of some of the kd-tree nodes in the path from the root of the kd-tree of the container node to the extracted node. An index term must describe the space that was delegated to the extracted node. We have two options regarding the nature of the index term:

fp We can use as an index term the **full path** to the extracted node: That is, all kd-tree nodes in the path are posted by a posting action.

cp We can use as an index term the **condensed path** to the extracted node: These are the kd-tree nodes of the full path that define the "tighter" low and high boundaries for each dimension. Hence, $2k+1$ is the maximum size of an index term (the extra kd-tree node may be needed in order to correctly merge the condensed path with the existing kd-tree of the parent where posting takes place).

In figure 15, we present all possible algorithm configurations. Algorithm CB/fp is irrelevant, since there are no missing kd-tree nodes when posting the full path. CB/fp is the same as A/fp.

Figure 15: **Splitting/Posting algorithm hierarchy**



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 DSBs, and therefore, we can achieve the best possible node storage utilization. Thus, the restriction we impose on splitting applies only to index nodes.

We observe that algorithm A/cp corresponds to the original splitting/posting algorithm for the hB-tree, the one we have shown to be erroneous because it does not preserve DSBs. In the following sections we will examine the other four algorithms (below the horizontal line of Figure 15) that preserve DSBs and discuss the advantages and disadvantages of each one. We expect to obtain "better" hB^{Π}-trees when using algorithms that are higher in the graph of figure 15.

4.4 Algorithm D/fp

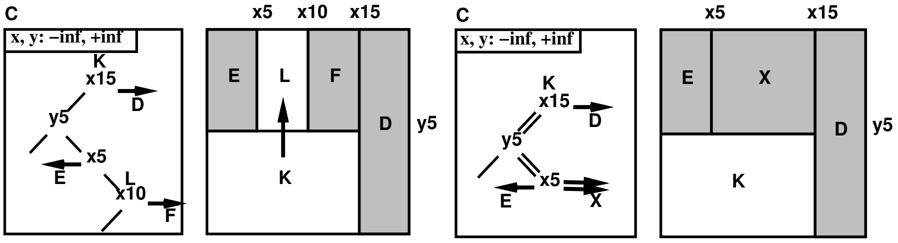
In this section we describe the simplest splitting/posting algorithm: it splits index nodes only by extracting decorated subtrees (D) and posts the full CtoX-path (fp), hence the name D/fp. This algorithm uses both Solutions 1 and 2 in order to preserve DSBs.

4.4.1 Splitting at Decorations (D)

When we split at decorations in the hB^{Π}-tree, data nodes are split anywhere (like the hB-tree). But, for index hB^{Π}-tree nodes, 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 most suitable decorated subtree for extraction. In some cases it may not be possible to locate a decorated subtree whose size is between one and two thirds of a node's size. We always extract the best possible decorated subtree, i.e., the one whose size is closest to half the hB^{Π}-tree node size.

Finally, if no decorated subtree exists we can drop the splitting action and reschedule it in the future (that would be the case in Figure 16a if kd-tree node $x10$ were missing). (This case never occurred in our experiments since most kd-tree nodes were decorated.) 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. This is acceptable, since search is still correct.

Figure 16: **D/fp: X has been split off C: the full CtoX-path must be posted to the parent of C**



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

(b) C and its space after X was split off C. NOTE: the double path is the CtoX–path.

Here is the algorithm for splitting at decorations, in a node C:

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).

Next, we show how to post the full CtoX-path of Figure 16b, that is indicated by the double edges, to the parent P of C. After the posting has been performed, node P will be able to direct searchers directly to X.

4.4.2 Posting the full path (fp)

In this variation of the hB^{Π}-tree we post 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. Thus we only need to 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 PtoC-path up to date so that it reflects the space decomposition described by the CtoX-path.

The resulting algorithm is quite straightforward. All we have to do is compare the PtoC-path 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.

Figures 17 through 19 demonstrate the three cases discussed above. Node P corresponds to the parent of node C of Figure 16. In each case, P and the space it is responsible for are shown before and after the posting takes place. The double edges in the kd-trees indicate the PtoC-path that in each case is compared to the CtoX-path of Figure 16b. The algorithm is the following:

CASE 1 The PtoC-path is longer than the 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 PtoC-path which no longer refers to space in C (see Figure 17).

CASE 2 The PtoC-path and the CtoX-path are the same length: again, all kd-tree nodes of the CtoX-path had already been posted. We make the last kd-tree node of the PtoC-path point to X (see Figure 18).

CASE 3 The PtoC-path is shorter than the CtoX-path: that is, the PtoC-path is a prefix of the CtoX-path. We append a copy of the extra CtoX-path to the PtoC-path, with the last posted node pointing to X (see Figure 19).

Figure 17: **D/fp: PtoC-path > CtoX-path: X-decorate remaining PtoC-path**

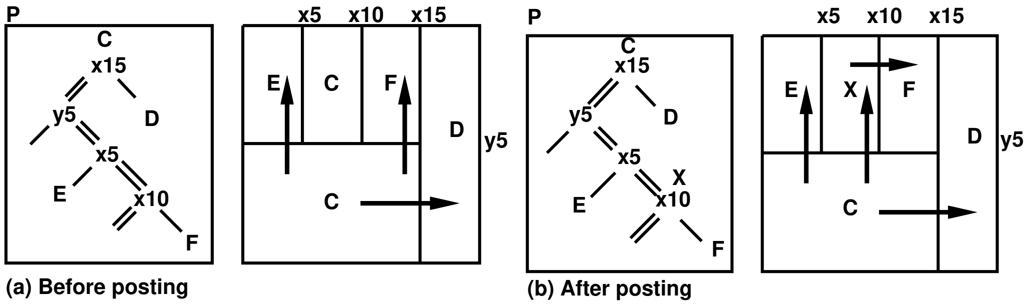
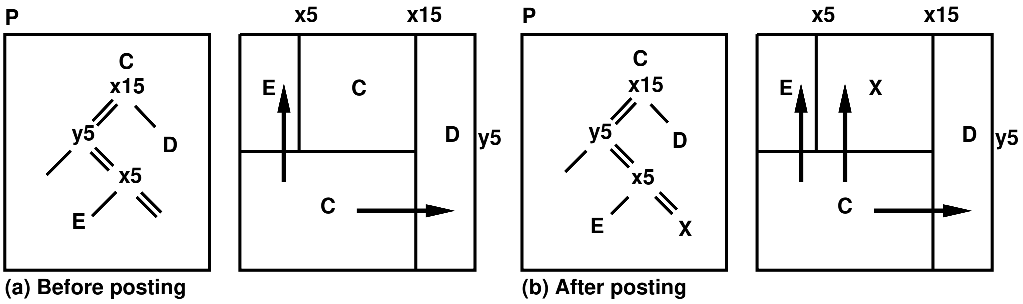


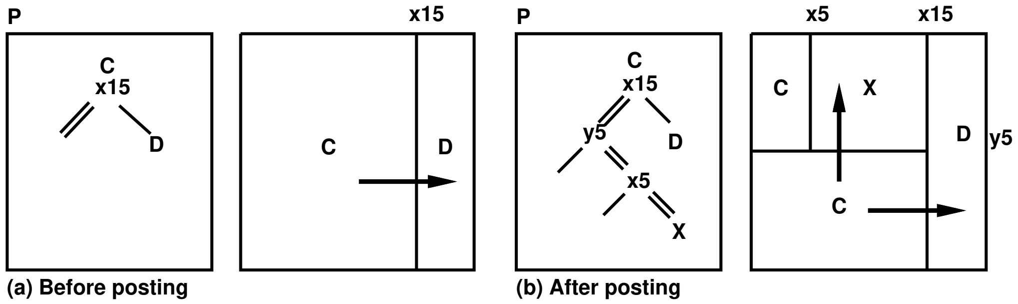
Figure 18: **D/fp: PtoC-path == CtoX-path: make last node of PtoC-path point to X**



4.4.3 Discussion

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 decorations (*D*) requires an exhaustive search of the whole kd-tree of the index hB^{Π}-tree node to find the largest decorated subtree whose

Figure 19: **D/fp: PtoC-path < CtoX-path: append the extra nodes of CtoX-path to PtoC-path**



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.

4.5 Algorithm A/fp

In this section we relax the splitting policy by allowing hB^{Π}-tree nodes to be split anywhere (A), but we still require that the full CtoX-paths are posted (fp), hence the name A/fp. This algorithm uses Solution 1 to preserve DSBs.

4.5.1 Splitting Anywhere (A)

For this algorithm, when an index node needs to be split, there is no limitation on where its kd-tree can be split. The procedure for finding the kd-subtree that has to be extracted was described earlier in the description of the original index node splitting algorithm.

4.5.2 Posting the full path (fp)

Although the posting policy (fp) is the same as in algorithm D/fp, the new splitting policy (A), that may introduce multi-parent nodes, affects the way posting is done. The posting scenarios we describe are based on Figure 16 of Section 4.4. We repeat this figure below, as Figure 20, in order to make the description of the examples easier to follow. Below, we examine all the possible cases.

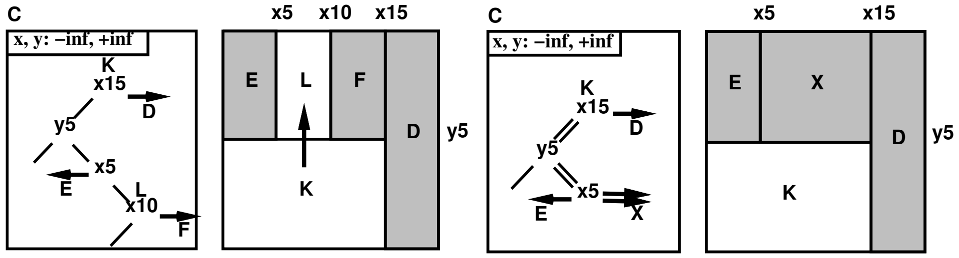
CASE 1 P is a prime parent of C: identical to D/fp.

CASE 2 P is a secondary parent of C: then, the C-fragment in P is part of the full C-fragment. The root of the full C-fragment, which corresponds to the root of C's kd-tree (and the first kd-tree node of the CtoX-path), is located in a container of P. Therefore, we have to find the point in the CtoX-path where posting should start. In order to do that, we start with the boundaries of C and we refine them as we traverse the CtoX-path. Our purpose is to match them with the boundaries of P. There are two scenarios:

CASE 2.1 The CtoX-path runs out of kd-tree nodes, i.e., we have to follow the side-pointer to X, in order to match the boundaries of P. This indicates that the space described by the C-fragment in P, now belongs to X. We simply X-decorate the root of P's kd-tree (which also is the first kd-tree node of the PtoC-path). This is the case in the example of Figure 21.

CASE 2.2 We match the boundaries of P and there are more kd-tree nodes in the CtoX-path. The kd-tree node that comes next in the CtoX-path is the point where posting should start. We post the remaining CtoX-path in P as described in CASE 1. Figure 22 demonstrates this scenario. Once we match the boundaries of P at kd-tree node $y5$ of C (see Figure 20), we follow CASE 1, regarding kd-tree node $y5$ as the root of the CtoX-path.

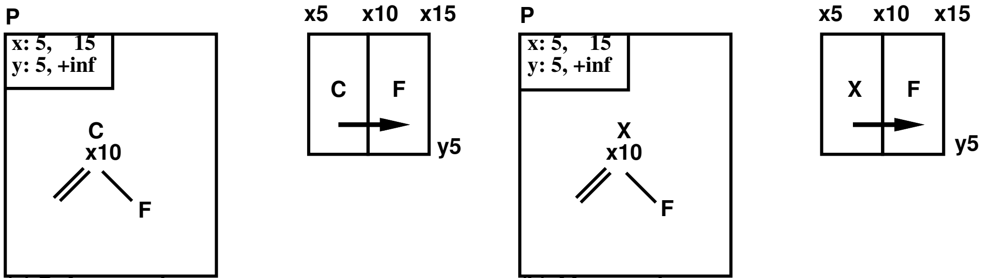
Figure 20: **A/fp: X has been split off C: the condensed CtoX-path must be posted to the parent(s) of C**



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

(b) C and its space after X was split off C. NOTE: the double path is the CtoX–path.

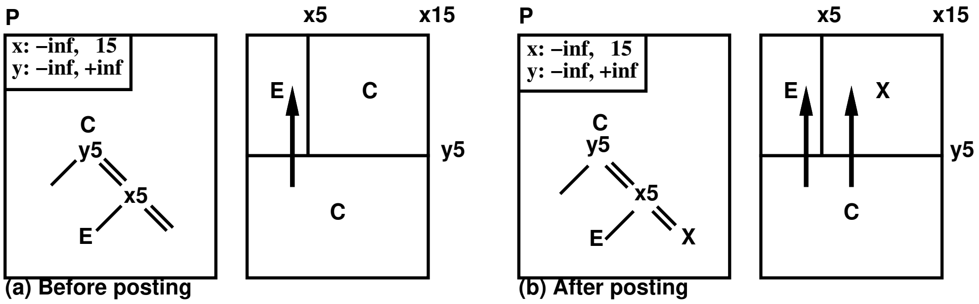
Figure 21: **A/fp: P is a secondary parent of C: P's space is equal to X's space**



(a) Before posting

(b) After posting

Figure 22: **A/fp: P is a secondary parent of C: P's space contains X's space**



(a) Before posting

(b) After posting

4.5.3 Discussion

A/fp still does not guarantee the worst case size for the index terms, since it posts the full paths like D/fp. What is new here is the splitting policy which is a great improvement over the D/fp algorithm, as now we can always split a kd-tree by removing between one third and two thirds of its contents. Index node space utilization is expected to be better than the D/fp algorithm.

4.6 Algorithm D/cp

In this section we describe an algorithm that relaxes the posting policy of algorithm D/fp, instead of relaxing its splitting policy (as A/fp does). It uses Solution 2 to preserve DSBs.

4.6.1 Splitting at Decorations (D)

We split hB^{Π}-tree nodes at decorations only. As a result all nodes are single-parent nodes. This splitting policy is described in Section 4.4.

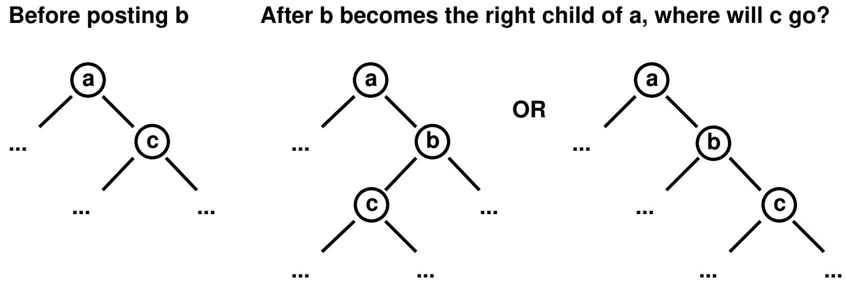
4.6.2 Posting the condensed path (cp)

In Section 4.1.3 we introduced the notion of the **condensed path**. It is the minimum collection of the kd-tree nodes of the CtoX-path that are necessary to describe the extracted region. In the worst case the condensed path consists of $2k+1$ kd-tree nodes. These are two tightest low and high boundaries for each one of the k attributes. We may also have to post an extra kd-tree node, called the **divergence** node, in order to glue pieces of the kd-tree where posting is performed together.

When we post the condensed paths, we may have "missing" kd-tree nodes in the middle of kd-tree paths. Posting them later, during an index term posting atomic action, is similar to filling in gaps in existing kd-trees.

We are now faced with a new problem, that was first brought to our attention by [Bar92]. This is illustrated in Figure 23. After inserting kd-tree node c in between kd-tree nodes a and b, we need to determine whether the kd-subtree rooted at c, which is the previous right child of a, will become a left or a right child of b. We call the kd-subtree rooted at c, the **hanging tree**.

Figure 23: **D/cp: hanging tree problem**



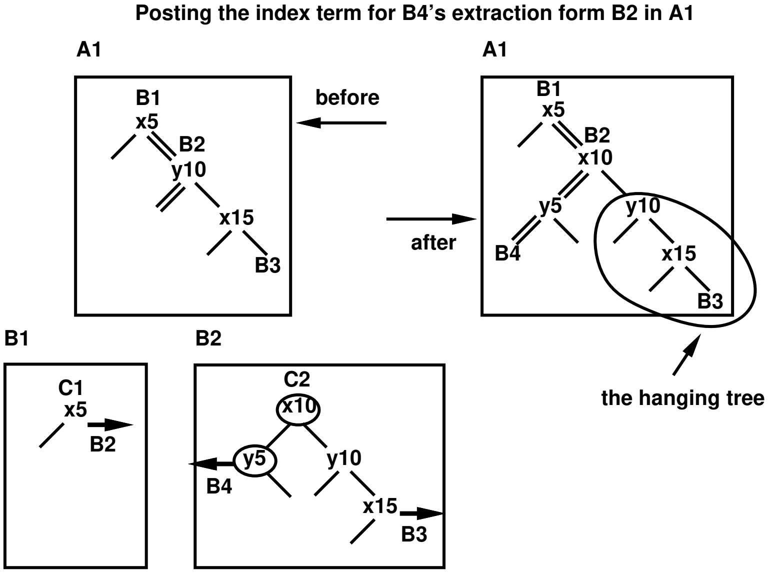
Here is the answer to this problem. Assume that node X is extracted from node C and while posting the index term for this split in the parent P of C, we have to insert a missing kd-tree node, say b, in the C-fragment of P. The fact that b was missing indicates that during another index term posting (say, for the extraction of Y from C), b was not in the condensed path. That is, there was a tighter boundary (some other kd-tree node d with the same attribute) further down the path from C to Y that was posted instead.

We claim that d will still be present in the hanging tree. To see that, remember that we split at decorations only. Thus, the C-fragment resides in its whole in a single parent.

We look for d in the hanging tree. If d's value is smaller than b's value then the hanging tree becomes a left child of b, otherwise a right child of b.

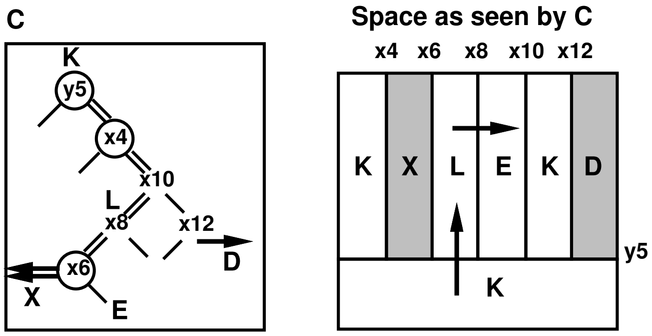
Figure 24 demonstrates the above problem. The parent A1 of B2 is shown before the index term that describes B4's extraction from B2 has been posted. Note that the condensed path for the extraction of B3 from B2 did not include kd-tree node $x10$, since $x15$ further down the path from B2 to B3 was a tighter boundary for the extracted space. That kd-tree node, i.e., $x15$, is used to determine where to put the hanging tree rooted at $y10$, after we post $x10$ in P.

Figure 24: **D/cp: solving the hanging tree problem**



In order to describe all the cases that can be encountered during index term posting, we use the example of Figure 25. Node X has been extracted from C, and kd-tree nodes $y5$, $x4$, and $x6$ comprise the condensed path. $x4$ is a lower and $x6$ is a higher boundary for attribute x, whereas, $y5$ is a lower boundary for attribute y. Figure 26 shows the parent P of C.

Figure 25: **D/cp: X has been split off C and we have to post the condensed path in the parent P of C**



The algorithm for posting the condensed path in the parent P of C, will have to traverse the CtoX-path and the PtoC-path in parallel. For every kd-tree node in the CtoX-path, if the node:

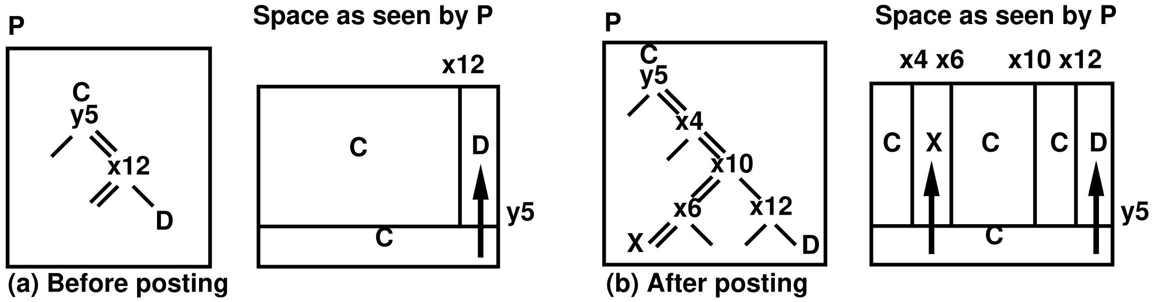
1. **Belongs to the condensed path, but has already been posted**: we do nothing. For example, in figure 26a, node $y5$ was posted when the extraction of D from C was posted.

2. **Belongs to the condensed path, and has to be posted above existing kd-tree nodes of the C-fragment**: we insert it between the current and the next kd-tree nodes of the PtoC-path. We examine the hanging tree (the one rooted at the next node in the PtoC-path) to find a node with the same attribute as the newly posted node, and based on the attribute-value of that node we determine if the hanging tree becomes a left or a right child of the newly posted node. In figure 26b, node $x4$ is posted above $x12$. In this example $x12$ happened to be the root of the hanging tree, and since 12 is greater than 4, $x12$ became the right child of $x4$. Also, kd-tree node $x10$ is not the tightest high x-boundary for X, but has to be posted to P because is a divergence node, that is, it is needed in order to make possible the merging of the condensed path into the kd-tree of P. (In the case that the newly posted node is now the root of the C-fragment, as with $x10$ in Figure 24, it gets the C-decoration.)

3. **Belongs to the condensed path, and has to be posted below existing kd-tree nodes of the C-fragment**: we appended it in the C-fragment in P. This is the case of node $x6$ in figure 26b.

4. **Is a redundant node**: we skip that node. This is the case of node $x8$ in figure 25.

Figure 26: **D/cp: posting the condensed path to the parent P of C**



4.6.3 Discussion

Algorithm D/cp has the drawbacks inherent in the splitting policy D, that is, a "good" split may sometimes not be achievable. Its best and very desirable property is that there is an upper bound on the size of the index term that is posted by a posting action. The maximum size of the condensed path is $2k + 1$ nodes.

4.7 Algorithm CB/cp

The CB/cp algorithm also posts condensed paths. But, in addition to allowing kd-trees to be split at decorations, we also allow them to be split at full paths after a decoration. These are places in the kd-trees where we have complete boundaries, as described in the discussion of Solution 3 in Section 4.2.3. Therefore, the algorithm preserves DSBs.

Since we must be able to detect these complete boundaries, we introduce a new field in every kd-tree node, called the **counter** field. First we present a procedure that sets and updates the counters of kd-tree nodes during posting, and then we describe the node splitting policy CB that uses the counter fields. Finally, we describe how index term posting is affected by the new splitting policy.

4.7.1 The counter field

The idea is to store in every kd-tree node a **counter** that indicates how many kd-tree nodes are missing between this kd-tree node and its current parent. Kd-tree nodes at the data level of the hB^{Π}-tree will always have a zero counter. Similarly, since full paths are always posted to the level just above the leaves, even when *cp* posting is used, that level will have only zero counters.

Below, we describe the procedure for setting the counter of a newly posted kd-tree node at the levels above the parent-of-leaf level of the hB^{Π}-tree. As usual, we assume that hB^{Π}-tree node X has been extracted from C and we post the condensed CtoX-path in the parent P of C. To simplify the explanation, we assume that (i) first, *all* not previously posted kd-tree nodes are being posted, and not only the ones in the condensed path, and (ii) then, the kd-tree nodes that are not in the condensed path, and therefore should not have been posted, are eliminated. When a node is posted or when a node is eliminated, the counter in the root of its hanging tree is also adjusted.

PROCEDURE FOR SETTING OR UPDATING the **counter** field of kd-tree nodes in P
    LET a be a kd-tree node in the full CtoX-path
    LET a' be the copy of a posted to P
    LET b' be the root of the hanging tree (if any) that becomes a child of a'
PHASE 1: POST ALL IN PATH
    FOR every kd-tree node a in the full CtoX path that has not already been posted to P {
        post a' as a copy of a
        counter(a') = counter(a)
        IF there exists a hanging tree with b' being its root
        THEN counter(b') = counter(b') - [counter(a') + 1] }
PHASE 2: DROP NODES NOT IN CONDENSED PATH
    FOR each kd-tree node c' that should have not been posted to P {
        LET d' be the only child of c'
        counter(d') = counter(d') + [counter(c') + 1]
        eliminate c' }

We demonstrate the use of the above procedure with the example of Figure 27. In Figure 27a, node B3 was extracted from B1, and the condensed path ($x30$) was posted to the parent A1 of B1. Notice that in B1, counter($x30$) = 0, but in A1, counter($x30$) = 2, since $x10$ (with counter = 1) was not in the condensed path and was not posted to A1. Now, assume that B4 is extracted from B1. In Figure 27b, the first phase of our counter setting and updating procedure posts all non-posted kd-tree nodes of the full path from B1 to B4. Notice how the counter of $x30$ changes back to zero. Finally, in Figure 27c, the second phase of the procedure eliminates kd-tree node $x20$ that should not have been posted, and appropriately updates the counter of $x15$.

4.7.2 Splitting at Complete Boundaries (CB)

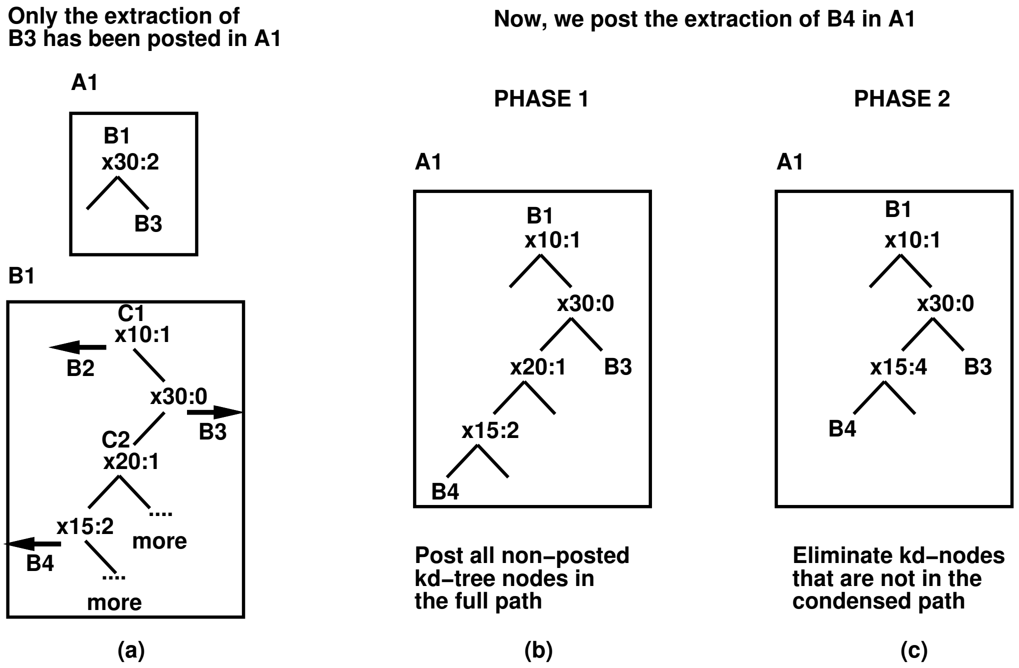
Here we describe the new splitting policy CB. Whenever we want to split an index hB^{Π}-tree node, we use the following algorithm to determine the root of the extracted kd-subtree (see also Solution 3 in Section 4.2.3):

STEP A: Start from the root of the kd-tree.

STEP B: Search the tree recursively testing all kd-subtrees whose root:

1. is decorated, or

Figure 27: **CB/cp: setting and updating the counter fields during posting**



2. is not decorated, but has its parent kd-tree node belonging to a path that has the following properties:
   (a) the path starts at a decorated kd-tree node, and
   (b) all of the path's kd-tree nodes have their counter equal to zero.

STEP C: Among all those candidate kd-subtrees, choose the one with contents closer to half the size of an hB^{Π}-tree node.

4.7.3 Posting the condensed path (cp)

Since decorated fragments can be split, we may have multi-parent nodes. Therefore, it may be necessary to post the index term for a split to more than one parent. Here is the algorithm for posting the condensed path:

CASE 1 P is a prime parent of C: same as algorithm D/cp. Also, we need to set or update the kd-tree node counter fields.

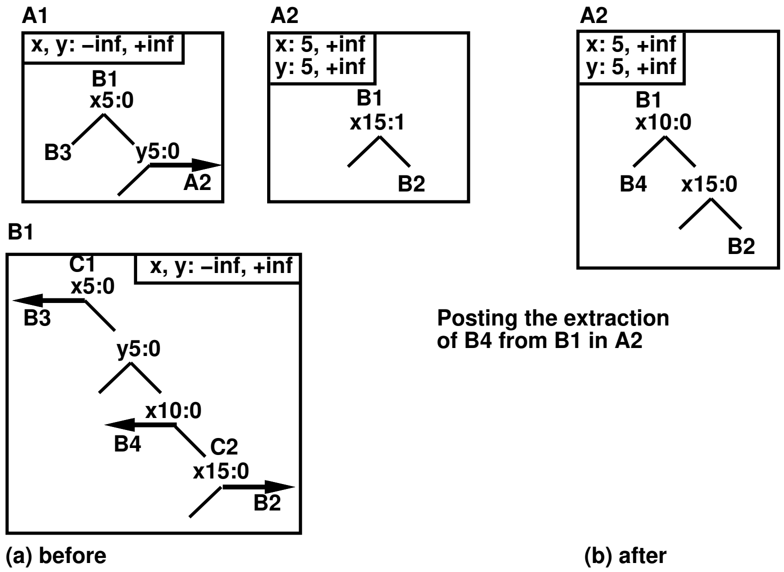
CASE 2 P is a secondary parent of C: same as algorithm A/fp, but we post the condensed path as described in algorithm D/cp. Also, we need to set or update the kd-tree node counter fields.

During posting we can always solve the problem of the hanging tree in the same way we did in algorithm D/cp. Our argument is still valid despite the fact that now we may split decorated fragments. A missing kd-tree node indicates that during an index term posting atomic action a tighter boundary than this missing node was posted instead. The tighter boundary cannot belong to another (extracted) part of the decorated fragment because decorated fragments are partitioned by node splits only when there is a full path from their root to the extracted kd-subtree. This is not the case here since we have a missing kd-tree node. Therefore, the part of the decorated fragment where the missing node is posted always contains the tighter boundary for that missing node. As in algorithm D/cp we use this boundary in order to determine where to put the hanging tree.

Figure 28 demonstrates an example of posting to a secondary parent. In Figure 28a, the index terms for the extractions of B2 and B3 from B1 had already been posted to the parent A1 of B1, when A2 was extracted from A1. Using the CB splitting policy, $x15$ was a candidate kd-subtree for extraction, since its parent $y5$ belonged in a path that started from a decoration (B1) and all of its kd-tree nodes had a zero counter. Since the B1-fragment was split, B1 became a multi-parent node.

According to CASE 2 of the posting algorithm, we start with the boundaries of B1 and we refine them, following the path from B1 to B4, until we match the boundaries of A2. Then, we start posting the condensed version of the rest of the path from B1 to B4, that is, kd-tree node $x10$. In Figure 28b, we see A2 after the posting has been performed. Notice the new updated counter value for $x15$ in A2.

Figure 28: **CB/cp: posting to a secondary parent**



4.7.4 Discussion

The CB splitting policy is a significant improvement over the D splitting policy. There are more potential splitting points for kd-trees, and we expect to have a good index node utilization. Also, it is even more unlikely that we will end up not being able to split a kd-tree and having to drop a posting action, as described in the discussion of algorithm D/fp.

One drawback of this algorithm is the fact that all kd-tree nodes need to store an extra field. Since all kd-tree nodes at the data level and the parent-of-leaf level have their counters equal to zero, and most of the time, kd-tree nodes at the higher index levels do not have missing kd-tree nodes above them, we may choose not to store zero counter fields. Also, even non-zero counters will be small numbers. Therefore, their storage representation will be a small number of bits.

4.8 Summary of Algorithm Properties

Table 2 summarizes the advantages and disadvantages of the various splitting/posting algorithms we described in the previous sections. We do not include an entry for algorithm A/cp.

Table 2: **Splitting/posting algorithm comparison**

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

5 The hB$^{\Pi}$-tree: Consolidating

Node consolidation in the hB$^{\Pi}$-tree is performed along the lines of Π-tree node consolidation: the 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. In the following, the term extracted node will refer to the node we want to deallocate. So, the three conditions for hB$^{\Pi}$-tree node consolidation are:

1. the extracted (sparse) node shares the same parent with its container,
2. the extracted node is a single-parent node, and
3. the container node has sufficient space to absorb the contents of the extracted node.

Conditions 1 and 2 are not as restrictive as they appear to be. Note that since at most one decorated fragment is split per hB$^{\Pi}$-tree node split there is a limit on the number of multi-parent nodes created. 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. With a fan-out of 140 (typical for an hB$^{\Pi}$-tree with node size 4K), at most 0.7% of the nodes at a given level will be multi-parent. The same argument holds for hB$^{\Pi}$-tree nodes that do not share the same parent with their container node. These are exactly the nodes whose decoration (child-pointer) appears at the root of the kd-tree of their parent. In the worst case there will be as many such nodes at a given level as parent nodes at the level above.

On the other hand, Condition 3 may be quite restrictive. A sparse node cannot be consolidated unless its container node is empty enough to absorb it. This condition may delay node consolidation.

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.

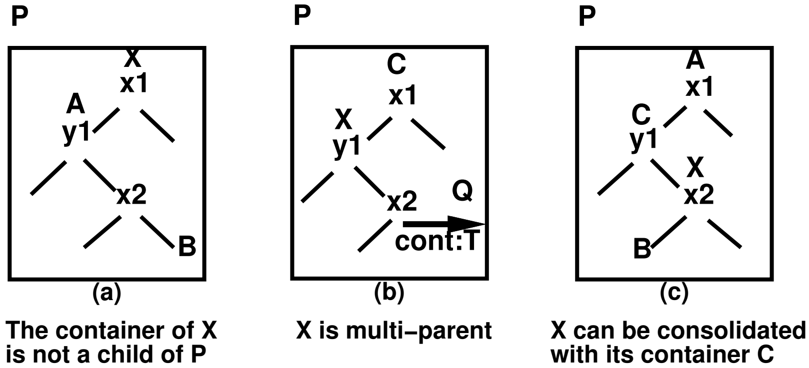
In this section, we first show how one can determine whether a node can be deleted by examining the kd-tree of its parent. This is independent of the posting policy in use (*fp* or *cp*). Then we show how to use pruning to reorganize kd-trees. Pruning is affected by the posting policy.

5.1 Determining When Consolidation Is Possible

The bookkeeping information we keep in the hB$^{\Pi}$-tree enables us to determine whether the two conditions for node consolidation are met by only examining the kd-tree of the parent of the sparse node.

Let us assume that C is the container of the sparse node X (extracted) and P is the parent of X. Node X is single-parent when all (if any) continues-to flags in the X-fragment in P are FALSE. Also, C is a child of P if X is not the decoration of the root of P's kd-tree. In that case, C corresponds to the last decoration seen before the X decoration in the path from the root of P's kd-tree to the X decoration. Figure 29 demonstrates cases when nodes can and cannot be consolidated.

Figure 29: **Determining when consolidation is possible**



In Figure 29a, X is the decoration of the root of the kd-tree of P. This indicates that either X has no container, or the container of X is not a child of P. Since condition (1) does not hold, consolidation cannot proceed. In Figure 29b, C is the container of X and condition (1) holds. But the X-fragment extends to the sibling Q of P, that is, X is multi-parent. Condition (2) does not hold and consolidation cannot proceed. Later, if Q is consolidated with P, perhaps X can be deallocated. In Figure 29c, both conditions hold and X can be consolidated with C if there is space in C.

According to condition (2) for node consolidation, a node cannot be consolidated unless there is a container node for it. There is one exception when the root of the hB$^{\Pi}$-tree is left with a single child. In this case the root of the hB$^{\Pi}$-tree is consolidated with this node and the height of the hB$^{\Pi}$-tree is decreased by one.

5.2 kd-tree Pruning Algorithm

We distinguish two different cases for kd-tree pruning depending on the posting policy in use (*fp* or *cp*). In both cases, kd-tree nodes from the kd-trees of P and C (when C is a data node) may be candidates for elimination. In general, kd-tree nodes are eliminated when their absence does not change the space decomposition and preserves the properties of the kd-trees according to the posting policy in use.

When the CB/cp algorithm is used and we eliminate kd-tree nodes, we have to accordingly update the counter field of its only, if any, child kd-tree node. In particular, if kd-tree node n is eliminated and n' is its child kd-tree node, then $counter(n') = counter(n') + (counter(n) + 1)$.

5.2.1 Full-path Pruning

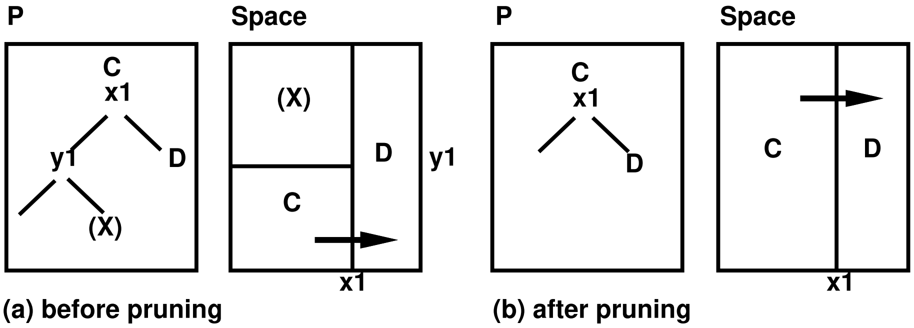
First, we treat the case of pruning when fp posting is used. After a sparse node X is consolidated with its container C, there may be a chance to eliminate certain kd-tree nodes in P's kd-tree, as shown in Figure 30.

In Figure 30a, after the removal of the reference to X from P's kd-tree, kd-tree node $y1$ can be eliminated since both its children are NULL. In other words, now that X seizes to exist its index term has to be dropped. It happens that $y1$, that belongs to the index term for X, does not belong in any index term for some other child of P. Hence, $y1$ is redundant. The pruned kd-tree is shown in Figure 30b.

All our pruning algorithms traverse the kd-tree in P in an upwards direction from the dropped reference to X. Here is the pruning algorithm for index node P:

```
FULL-PATH PRUNING IN INDEX NODE P
  Requirement: X-fragment was empty in P (X was a child pointer)
FOR each kd-tree node in (ascending) order in the path from the reference to X
             (that now is NULL) to the C decoration in P's kd-tree
   IF both the children of the kd-tree node are NULL
      make the child of the kd-tree node's parent NULL
      eliminate the kd-tree node
   ELSE
      DONE
```

Figure 30: **Full-path pruning in P**

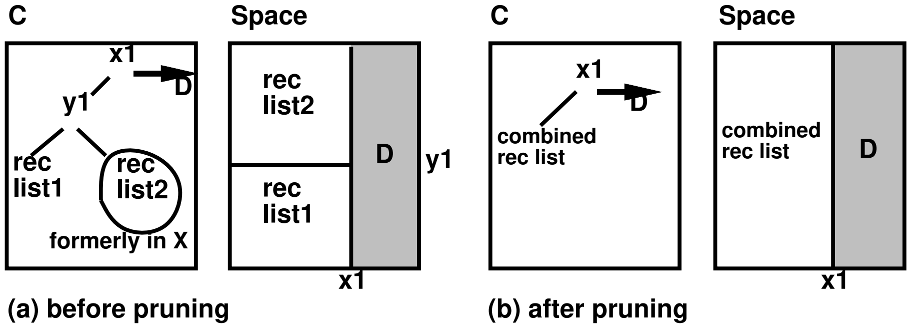


Note that this kind of pruning can take place only when the X-fragment is empty in P. Also, it can eliminate the whole kd-tree of P. For example, in Figure 30b, kd-tree node $x1$ will be eliminated when D is consolidated with C. In this case we cannot consolidate P with its container as part of the atomic action that consolidated D with C. Instead, P will remain empty, containing only a child pointer to C (an empty C-fragment), until it is consolidated by another atomic action. The scenario we have described above is usually avoided, since P would have already been consolidated before it became empty.

A similar pruning algorithm may be applicable to the kd-tree of the container C of X. Figure 31a demonstrates such a scenario with C and X being data nodes. Node X that contained a single record-list is consolidated with node C. The side-pointer to X is replaced by the contents of X making kd-tree node $y1$ redundant. In Figure 31b $y1$ has been eliminated and its two record-lists have been combined and become a child of its parent kd-tree node $x1$. The pruning algorithm for data node C follows:

```
FULL-PATH PRUNING IN DATA NODE C
  Requirement: X's contents was a single record list
FOR each kd-tree node in (ascending) order in the path from the side-pointer
         to X (that now points to X's contents) to the root of C's kd-tree
   IF both the children of the kd-tree node are record lists
      create a new record list by merging the two lists
      make the new list a child of the kd-tree node's parent
      eliminate the kd-tree node
   ELSE
      DONE
```

Figure 31: **Full-path pruning in C**



When C is an index node and X's contents is an empty kd-tree (see discussion in the previous section) the full-path pruning algorithm for the kd-tree of C is identical to the one used to prune the parent P.

When the root node of the hB$^{\Pi}$-tree is left with a single child node, the above algorithm will eliminate all kd-tree nodes in the kd-tree of the root. No kd-tree is needed to describe the space decomposition among the children of the root, since there is only one child. In this case, the root is deallocated and its child node becomes the new root of the hB$^{\Pi}$-tree.

5.2.2 Condensed-path Pruning

The kd-tree pruning algorithm described above is also applicable when the posting policy is to post condensed paths. But, now, one may be able to eliminate additional kd-tree nodes in the kd-trees of P and C. Figure 32a demonstrates such a scenario for the parent P of X. The reference to X is dropped in the kd-tree of its parent P making kd-tree node $x1$ redundant. In Figure 32b $x1$ has been eliminated and since it happened to be the root of the C-fragment its child $x2$ is decorated with C. Notice that node D that was previously a sibling of X now becomes a sibling of C.

A kd-tree node is **redundant** when the space decomposition described by the kd-tree the node belongs to is not affected by the elimination of that kd-tree node. We assume for algorithmic simplicity that the ancestor-descendent relationship of pairs of kd-tree nodes is not changed by pruning. Only nodes in the path from the reference to X ascending to the C decoration in P may be eliminated. A consequence of this decision is that sometimes, a kd-tree node both of whose children are kd-subtrees cannot be eliminated by our algorithm even if it is redundant. Such a kd-tree node is called a **divergence** kd-tree node. In Figure 33, kd-tree node $x2$ is a divergence node.

Here is an algorithm that determines whether a kd-tree node n of the kd-tree of an hB$^{\Pi}$-tree node H is non-divergence redundant (**ND-redundant**) or not. Note that this test works both for index and data hB$^{\Pi}$-tree nodes.

Figure 32: **Condensed-path pruning in P**

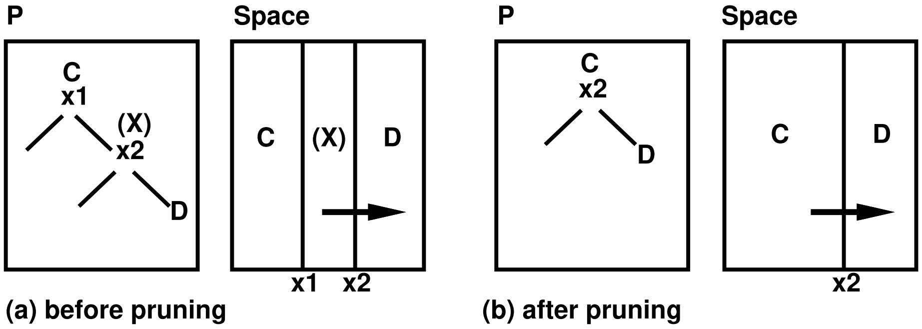
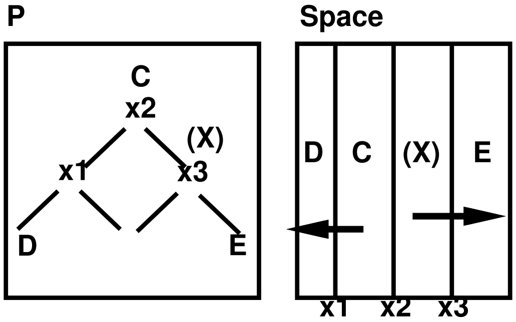


Figure 33: **Redundant divergence kd-tree nodes cannot be eliminated**




```
KD-TREE NODE ND-REDUNDANCY TEST
IF one of the children of n is NULL (H is an index node)
                    or a RECORD-LIST (H is a data node)
   find the boundaries of all children and siblings of H
                    that are reachable from H through n
   IF n appears in at least one of the node boundaries
      RETURN FALSE
   ELSE
      RETURN TRUE
ELSE
   RETURN FALSE
```

In the ND-redundancy test algorithm, the *boundaries* we are talking about are boundaries calculated from the kd-tree node paths in H. Using the above kd-tree node redundancy test we can now proceed to the description of the additional pruning procedures for P and C.

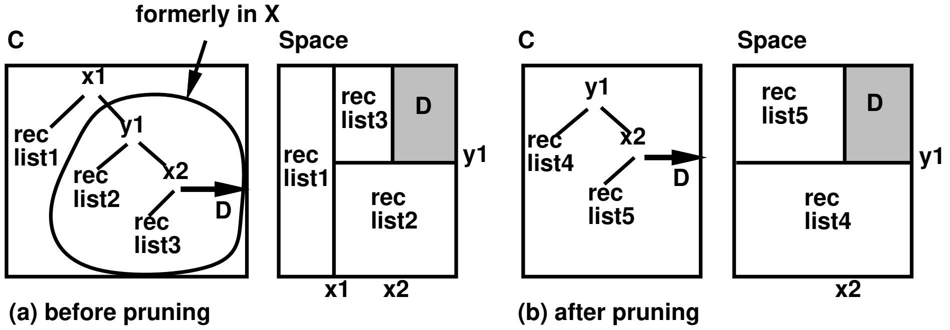
```
CONDENSED PATH PRUNING IN INDEX NODE P
FOR each kd-tree node in order in the path from the reference to X
             (that now is NULL) to the C decoration in P's kd-tree
   IF the kd-tree node is ND-redundant
      make its parent point to the kd-tree node's only child
      eliminate the kd-tree node
```

An example of condensed path pruning in a parent index node is shown in Figure 32. Node X is consolidated with node C and kd-tree node $x1$ in P's kd-tree becomes ND-redundant and is eliminated.

We may be able to eliminate additional kd-tree nodes in C. These are the kd-tree nodes that lie in the path from C's kd-tree root to the side-pointer to X (that now has been replaced by X's contents) and that are ND-redundant.

The condensed path pruning algorithm when C is an index node is exactly the same as the one used to prune P. The same is true when C is a data node, with the only difference being the fact that after the elimination of a ND-redundant node, the records of its record list will have to be re-inserted in the resulting pruned kd-tree. This is a local re-insertion and does not involve any hB$^{\Pi}$-tree traversals. This scenario is demonstrated in Figure 34, where kd-tree node $x1$ is eliminated, since it is not needed to describe D's space. Records from record list 1, that was $x1$'s child, are re-inserted in the pruned kd-tree and modify record lists 2 and 3.

Figure 34: **Condensed-path pruning in C**



6 Performance Results

In this section we first describe the nature of multi-attribute data we used to test the various versions of the hB$^{\Pi}$-tree, and then we present the performance results we obtained.

6.1 Workload

We used both computer-generated data and data from the Sequoia 2000 Storage Benchmark [SFGM93].

6.1.1 Computer-Generated Data

Computer-generated data was created by skewing values obtained from a random number generator. To do this we performed the following transformation on each value (v is the old value and v' is the new one):

$$v' = v^n, \text{ for } 0 \leq v \leq 1$$

where n determines the degree of skewness. For example, for $n = 21.85$ we get a 90:10 skewed distribution [Knu68, Lom83, GSE$^+$94].

In our experiments we used one quarter million records, each one consisting of twelve 4-byte attributes. To simulate hB$^{\Pi}$-trees of various dimensionalities we used between one and twelve of them as indexing attributes. In order to stress the various algorithms and assess the performance of the hB$^{\Pi}$-tree under extreme situations, we used only skewed values for all the indexing attributes (obtained using $n = 3000$ for a 90:10 distribution). In the rest of this section we will be referring to this kind of data as "computer-generated data".

6.1.2 Sequoia Data

The Regional version of the Sequoia 2000 Storage Benchmark [SFGM93] consists of geographic data from the state of California. This is point, polygon, graph, and raster data. In our experiments we used the point data. The point data file consists of 62,584 California place names along with their coordinates, in the following format:

$$easting : northing : name$$

The first two fields represent distance in kilometers from the center of the coordinate system. The third field is a variable length string representing the name of the place. Here is a portion of the file:

```
-1645982:-659926:Abbot, Mount
-1844542:-422024:Abbott
-1779525:-893112:Abbott Canyon
-1680181:-722305:Abbott Creek
-1902477:-126821:Abbott Lake
...
```

In the rest of this section we will be referring to this kind of data as "Sequoia data".

6.2 hB$^{\Pi}$-tree Performance: Point Data

We present performance results both for the computer-generated and the Sequoia data. In particular, we measure node space utilization, index size (that is related to fan-out), and range query performance.

First we report a very interesting result. Regardless of (a) the data distribution we used, (b) the total number or size of records we inserted, or (c) the splitting/posting algorithm we used, the *average number of kd-tree nodes posted per index term posting atomic action was only slightly larger than one.*

This is because record-lists are almost always split using exactly one attribute (i.e., by a hyperplane). Hence, all posting actions that involve data nodes (i.e., that post to the first index level above the data level) have to post exactly one kd-tree node. Also, most of the time index nodes have balanced kd-trees that are usually split by extracting either child of their kd-tree root (i.e., by a hyperplane). This kind of index node split also requires the posting of exactly one kd-tree node.

6.2.1 Node Space Utilization

Figures 35 and 36 show the node space utilization of the hB$^{\Pi}$-tree when using computer-generated and Sequoia point data respectively. Node space utilization is the percent of the node which has information (rather than empty space). For comparison, the B^{+}-tree is known to have a 69% node space utilization.

We observe that in both figures, algorithms D/fp and D/cp perform identically. The same is true for algorithms A/fp and CB/cp. This is as expected because splitting policies A and CB achieve much better splitting quality, i.e., the extracted contents are closer to half the size of a page, than splitting policy D. The comparably low index node space utilization when the page size is 4Kbytes is attributed to the fact that the number of index nodes is very small (only 20) and the sparse root page is included in the calculations.

Table 3 shows the quality of the various splitting policies when we use skewed computer-generated data. A "good" quality split is a split that extracts between one and two thirds of a node's contents.

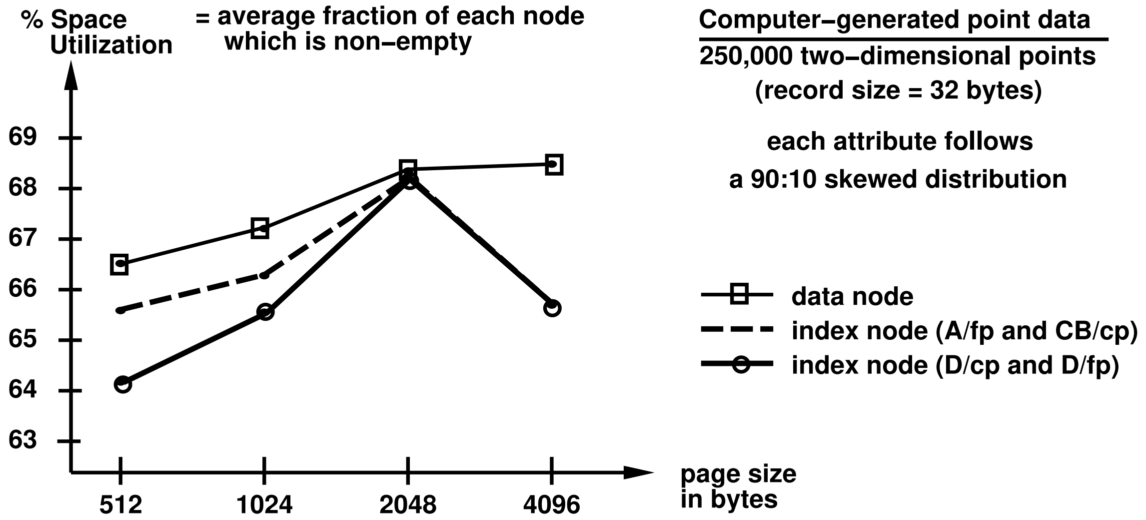
Table 3: **Good versus bad split cases under various algorithms and node sizes**

Node Size $\longrightarrow$	512	1024	2048	4096	512	1024	2048	4096
Algorithm	**Good Split**				**Bad Split**			
D/cp or D/fp	1498	388	89	23	253	16	3	0
CB/cp	1696	399	91	23	2	0	0	0
A/fp	Always				Never			

As we can see, splitting policy CB is a great improvement over splitting policy D. In our experiments it essentially performs as well as policy A.

In general, node space utilization is very high even for small page sizes and is comparable to B^+-tree node space utilization.

Figure 35: **Space utilization when using skewed computer-generated data**



6.2.2 Index Size

Figure 37 shows the proportion of the hB^{Π}-tree which is above the leaves (when using computer generated data). We call this the *size* of the index. By "index" we refer to the collection of index hB^{Π}-tree nodes. We count the percentage of the index nodes over the total number of hB^{Π}-tree nodes. The inverse of that number is an approximation of the fan-out of the hB^{Π}-tree. For example, if 1% of the nodes are index nodes, then the fan-out is close to 100 (i.e., each node has approximately 100 children).

Again, all algorithms perform almost identically. This is explained by the fact that we post one new kd-tree node per posting on average. Also, even when we post condensed paths, the missing kd-tree nodes are often later posted by other posting actions.

The smaller the percentage of index nodes is, the larger the fan-out of the tree will be. In Figure 37 we observe that for a node size of 4K, the index hB^{Π}-tree nodes are less that 1% of the total number of hB^{Π}-tree nodes, yielding a fan-out of 130. In the current implementation of the hB^{Π}-tree we use a fixed kd-tree node size (20 bytes).

Figure 36: Space utilization when using point data from the Sequoia Benchmark

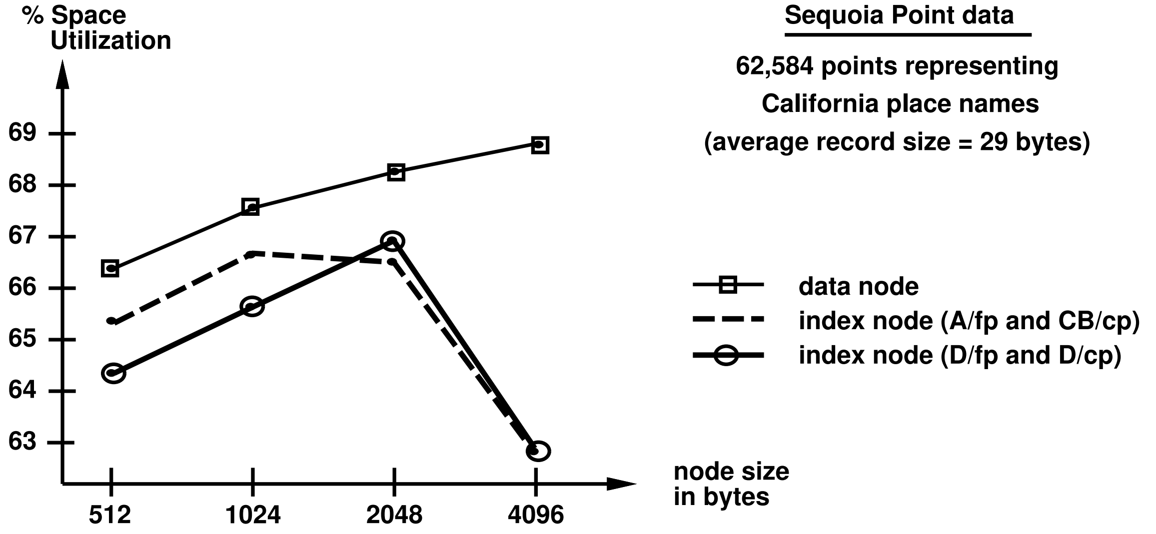
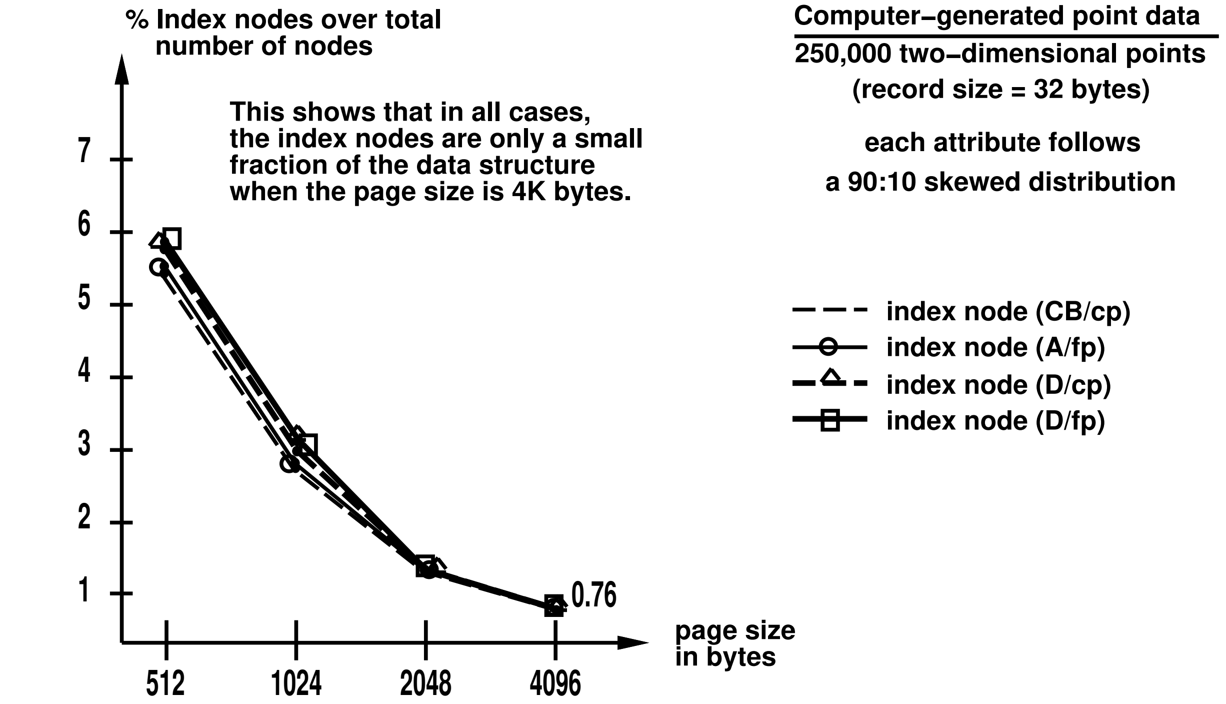


Figure 37: Index size when using skewed computer-generated data. This is the proportion of the tree which is above the leaves.



Since in our experiments with two-dimensional skewed data, the condensed paths were always the same as the full paths, we ran the same experiments with the same data records, but using only one indexing attribute (essentially, we used the hB$^{\Pi}$-tree as a B^{+}-tree). Table 4 shows data concerning the kd-tree nodes that were posted at the index levels using the various algorithms, and under various node sizes. Each table entry consists of three numbers in the format **a:b:c** where **a** is the number of kd-tree nodes posted, **b** is the number of kd-tree nodes that were not posted, i.e., they did not belong to the condensed paths, and **c** is the number of the missing kd-tree nodes that were eventually posted since they belonged to some condensed path.

Table 4: **kd-tree nodes posted and not posted under various algorithms and node sizes**

Node Size ⟶	512	1024	2048	4096
Algorithm	**a:b:c**	**a:b:c**	**a:b:c**	**a:b:c**
D/fp	14875:0:0	6671:0:0	3196:0:0	1565:0:0
A/fp	14890:0:0	6674:0:0	3196:0:0	1565:0:0
D/cp	14823:120:68	6665:12:6	3195:1:0	1564:1:0
CB/cp	14804:121:66	6665:14:7	3194:2:1	1564:1:0

Table 4 shows that even for one-dimensional skewed point data the condensed paths are almost the same as the full paths, especially for large node sizes. Less than 1/400 of the total number of kd-tree nodes is not posted because it does belong to the condensed path when the node size is 0.5K and we use either algorithm D/cp or CB/cp. Of course, there may be data distributions that considerably increase this number.

6.2.3 Range Queries

Finally, we have tested the range search performance of the hB$^{\Pi}$-tree using the Sequoia data. We performed the same series of 104 range searches with varying query selectivity and different 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 we chose the extent of the window for each attribute be a random ratio of the domain range for that attribute.

The results, shown in Figure 38, indicate very good range search performance for query selectivities greater than 0.5%, and sufficiently good for even 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.

6.3 hB$^{\Pi}$-tree Performance with High-Dimensional Data

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 39, from [ES93], illustrates this fact.

Figure 38: **Range search performance under various node sizes and query selectivities**

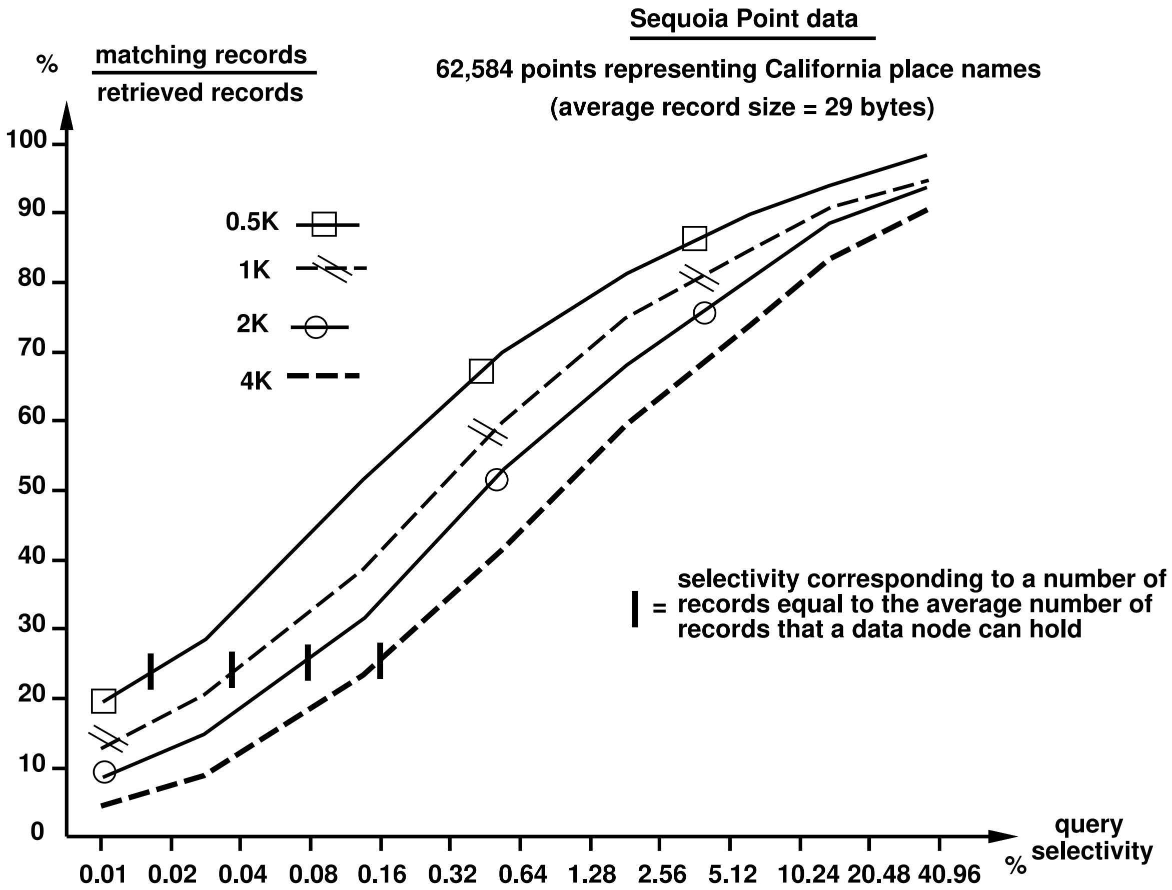
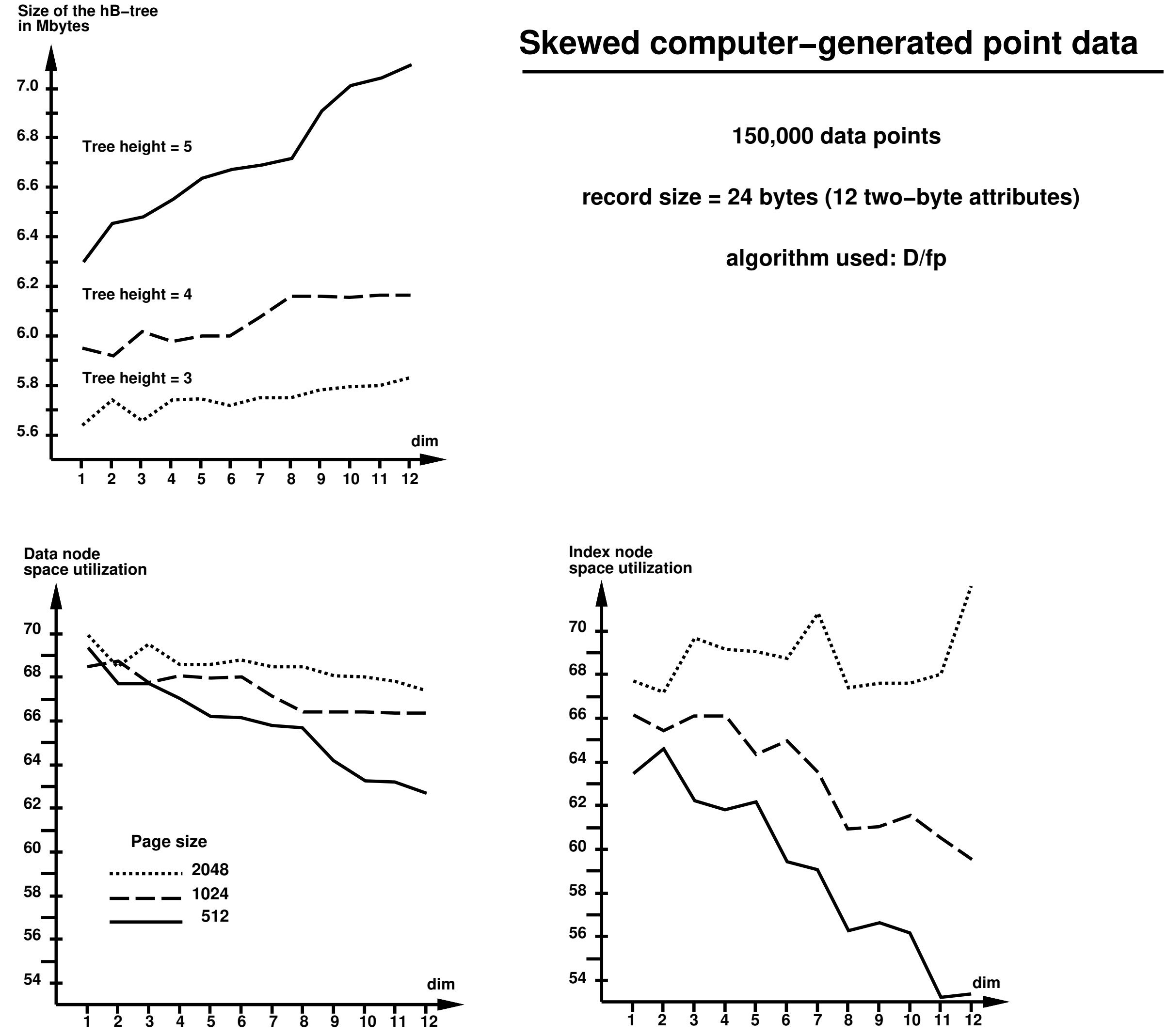


Figure 39: **The hB$^{\Pi}$-tree is fairly insensitive to dimensions when node sizes are greater than 1K bytes**



In this experiment the version of the hB^{Π}-tree that uses the D/fp algorithm was used. 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 (hB^{Π}-tree node boundaries) and not index term size. For example, when we use 12 indexing attributes and the size of the value of an attribute is 4 bytes, we need 12*2*4=96 bytes to store the boundaries of a node. This is a considerable amount of space for small node sizes (almost 20% of the space of a 0.5 Kbyte node). On the other hand, it is an almost negligible percentage of the space of a large node (around 2% of the space of a 4 Kbyte node). With a page size of 1K bytes and larger, there is almost no effect on the size of the hB^{Π}-tree and the node space utilization as the dimensions increase. (Page sizes larger than 2K bytes are not shown.)

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.

7 Conclusion

7.1 Summary

Indexing of multi-attribute data in general purpose DBMSs is a very desirable feature. This is because the number of applications that deal with multi-attribute data is continually increasing. Recently, for example, there is a great deal of activity in data warehousing and OLAP (on-line analytical processing), where many attributes of business data are used to analyze historical data for trends. Also, there are a growing number of applications for Geographic Information Systems or GIS.

These applications expect that the DBMS offers the same functionality for this kind of data as it offers for traditional data. The DBMS should use efficient and reliable ways to store, index, and access the data. For many of these applications it is also important to maximize concurrent accessing of the data by as many users as possible at the same time, and be able to recover from application errors or system crashes that result in data inconsistency.

Approaches that use multiple single-attribute indexes are quite inefficient. That is why there has been extensive research on explicitly multi-attribute indexing. Most proposed multi-attribute indexes do not offer performance guarantees and well understood methods for concurrency and recovery. But these are the requirements for the inclusion of an index in a general purpose DBMS.

We wanted to propose a multi-attribute index that would be appropriate for inclusion in a general purpose DBMS. Our approach was to combine the hB-tree, a multi-attribute index with promising performance guarantees, and the Π-tree, which offers well understood and efficient concurrency and recovery methods. We called the resulting method the hB^{Π}-tree.

We presented the necessary modifications that transform the hB-tree into a case of the Π-tree, and yield the hB^{Π}-tree. Unfortunately, in the process we discovered a flaw in the node splitting and index term posting processes of the hB-tree that affects its well-formedness making searches incorrect. We proposed various solutions to the problem. They work by restricting the places nodes can be split, and/or by increasing the amount of information that needs to be posted to describe a split. Finally, we demonstrated how sparse nodes can be consolidated.

Depending on the way we choose to solve the splitting/posting problem, we get versions of the hB^{Π}-tree with different characteristics. In order to access the performance of these various versions and compare them, we have implemented all of them. We ran extensive experiments with various type and distributions of data and we concluded that even the most restrictive versions of the hB^{Π}-tree, that do not offer worst case storage utilization and index term size guarantees, actually perform very well.

7.2 Future Work

We have shown that the hB^{Π}-tree performs very well on point data. We would like to do further work on spatial (non-point) data. By mapping k-dimensional bounding rectangles of spatial objects to $2k$-dimensional points, we no longer cluster the objects in terms of Euclidean distance, but in terms of size, i.e., large objects are clustered together and small objects are clustered together. It is interesting to see how separation by size can benefit certain kinds of range queries.

This kind of mapping can also result in an altered data distribution. For example, imagine that our entire data collection consists of small sized one-dimensional spatial data, like short line-segments, that follow a normal distribution. When we map them to two-dimensional points of the form (line-start-point, line-end-point), we get a highly correlated distribution. We need to further assess the performance of our method when on such unusual data distributions. In particular, we want to know how they affect range search performance. The polygon and graph data from the Sequoia 2000 Benchmark [SFGM93] would be appropriate for this purpose.

Another very interesting and important problem are spatial joins. An example of a spatial join is "give me all cities that are 10 miles away from the Mississippi river". Assuming that we have two spatial indexes, one for the cities and one for the rivers, we need efficient ways of answering the above query. Spatial joins are a hard research problem.

Often indexes are built on existing sets of data. It is not acceptable to build such a new index by inserting all data items one by one. Methods for "bulk-loading" should be available. We would like to investigate this problem on the hB^{Π}-tree.

Finally, it would be very interesting to use the hB^{Π}-tree in a real general purpose DBMS, or a GIS database, so that we obtain an even clearer picture of its capabilities and possible limitations.

References

[Bar92] M. Barrena. Private communication. 1992.

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

[BKSS90] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 322–331, 1990.

[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.

[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.

[GSE+94] J. Gray, P. Sundaresan, S. Englert, K. Baclawski, and P. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 243–252, Minneapolis, MN, May 1994.

[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.

[Hin85] K. H. Hinrichs. *The grid file: implementation and case studies of applications*. PhD thesis, Swiss Federal Institute of Technology, Zurich, Switzerland, 1985.

[Knu68] D. E. Knuth. *The art of computer programming*, volume 3. Addison-Wesley, Reading, MA, 1968.

[Lom77] D. B. Lomet. Process structuring, synchronization, and recovery using atomic actions. *SIGPLAN Notices*, 12(3):128–137, March 1977.

[Lom83] D. B. Lomet. Bounded index exponential hashing. *ACM Transactions on Database Systems*, 8(1):136–165, March 1983.

[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.

[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.

[Rob81] J. T. Robinson. The K-D-B-Tree: A search structure for large multidimensional dynamic indexes. In *Proceedings of ACM/SIGMOD Annual Conference on Management of Data*, pages 10–18, New York, NY, April 1981.

[Sag86] Y. Sagiv. Concurrent operations on b*-trees with overtaking. *Journal of Computer and System Sciences*, 33(2):275–296, 1986.

[Sal85] B. Salzberg. Restructuring the Lehman-Yao tree. Technical Report NU-CCS-85-21, College of Computer Science, Northeastern University, Boston, MA, 1985.

[Sal91] B. Salzberg. Practical spatial database access methods. In *Proceedings of the Symposium on Applied Computing*, pages 82–90, Kansas City, MI, 1991.

[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.