The hB-Tree: A Multiattribute Indexing Method with Good Guaranteed Performance

DAVID B. LOMET
Wang Institute of Graduate Studies
and
BETTY SALZBERG
Northeastern University

A new multiattribute index structure called the hB-tree is introduced. It is derived from the K-D-B-tree of Robinson [15] but has additional desirable properties. The hB-tree internode search and growth processes are precisely analogous to the corresponding processes in B-trees [1]. The intranode processes are unique. A k-d tree is used as the structure within nodes for very efficient searching. Node splitting requires that this k-d tree be split. This produces nodes which no longer represent brick-like regions in k-space, but that can be characterized as holey bricks, bricks in which subregions have been extracted. We present results that guarantee hB-tree users decent storage utilization, reasonable size index terms, and good search and insert performance. These results guarantee that the hB-tree copes well with arbitrary distributions of keys.

Categories and Subject Descriptors: D.4.3 [**Operating Systems**]: File Systems Management—*access methods, file organization*; H.2.2 [**Database Management**]: Physical Design—*access methods*; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*file organization*

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: Access methods, B-trees, dynamic files, multiattribute indexing

1. INTRODUCTION

There are a number of application areas where it is very common to perform searches using the values of several attributes. Examples of such areas include geographic or geometric data, VLSI design, and certain kinds of document retrieval.

Where multiple-attribute searches are the rule and single-attributes searches the exception, there are advantages to using one multiattribute index compared with several single-attribute indexes. First, the clustering of index terms and data on disk can dramatically reduce the number of I/O accesses needed for the search. Second, when new records are inserted, a multiattribute organization

Authors' current addresses: David Lomet, Digital Equipment Corp., Cambridge Research Laboratory, One Kendall Square, Bldg. 700, Cambridge, MA 02139; Betty Salzberg, College of Computer Science, Northeastern University, Boston, MA 02115.

needs only a single update of its index. Multiple single-attribute indexes require multiple updates.

Despite these two important advantages, multiattribute index organizations have not commonly been used. One reason for this is that multiattribute organizations yield poor clustering for single attributes. A "cross section", or "partial match" query, where the value of only one of the attributes is specified, requires searching all pages in the "hyperplane" of the multiattribute index. An example would be finding all the baseball games played on a given date, in a multiattribute file organized both by team and by date. Figure 1 illustrates this.

Of course, single-attribute secondary indexes also lose data clustering. Thus, the choice between a multiattribute index or multiple single-attribute indexes is application-dependent. However, multiattribute organizations are rarely used in general-purpose database systems, even for applications which would benefit from their use.

A substantial part of the reason for the lack of use of multiattribute index methods is that they fail, in one way or another, to be good search structures. Good search structures must have at least the following properties:

(1) good average storage utilization both in the index and in the data pages;
(2) large index fan-out, resulting in a small index and a small number of disk accesses;
(3) easy *incremental* reorganization as the file grows;
(4) simple algorithms with an absence of special cases; and
(5) an ability to handle range searches (and partial match searches) as well as exact match searches.

Many existing multiattribute search methods exhibit these characteristics some of the time. However, a good search organization must guarantee these characteristics all of the time in the face of arbitrary data.

In this paper we introduce a new multiattribute search structure which guarantees good performance in the face of any pattern of data distribution and any pattern of insertions and deletions. This method promises good space utilization both in the data pages and in the index.

Our point of departure is the K-D-B-tree of Robinson [15]. K-D-B-tree nodes always represent rectangular regions, or bricks. In our structure, the nodes represent bricks from which smaller bricks have been removed, or "holey bricks." We call our structure the *hB-tree*, or holey brick B-tree. A similar idea was used in the one-dimensional case for DL*-trees in [9]. Holey bricks have also been used in [5] and in [13].

We distinguish between *data nodes* that are pages which contain the records of the database and *index nodes* that contain k-d trees. The data nodes are the leaves of the hB-tree. The index nodes are the internal nodes of the hB-tree. Thus, the hB-tree is similar to the B^+-tree [4].

The hB-tree grows from the leaves and has all leaves at the same level, just as a B-tree does. Since it is a B-tree-like structure, enhancements such as partial expansions [10] and Bounded Disorder files [7, 11] can be used to improve its efficiency.

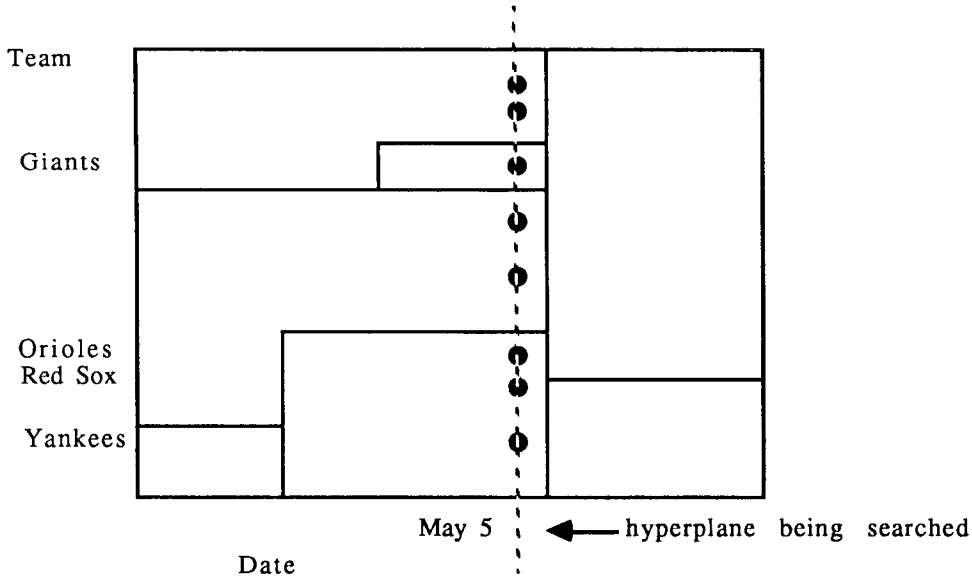


Fig. 1. Clustering for a single attribute is compromised in any multiattribute index.

Robinson does not specify the internal structure of his K-D-B-tree nodes. In order to present information about node utilization and average fan-out, we must choose some specific *node structure*. We argue that the k-d tree is a good choice for the internal structure of the index node of the hB-tree. That is, we use the k-d trees *within hB-tree index nodes* to organize information about lower levels of the hB-tree. The k-d tree also assists in organizing data nodes of hB-trees.

The original version of the k-d tree [2] is a binary tree which requires a fixed alternation from one attribute to another as one descends the tree. We use a generalization of this structure [3], which allows the dynamic insertion of data to determine the choice of attribute. This means that some attribute identifier (age, salary, x-coordinate) must be stored in the k-d tree node as well as the comparison value.

In addition, in order to represent holey bricks, several leaves of the k-d tree in an hB-tree index node may refer to the same hB-tree node at a lower level of the hB-tree. This construction is illustrated in Figure 2.

As with the single-attribute B^+-tree, when new records are added, a data node must occasionally split, and new information is posted to its parent. When too much information is posted to an index node, the index node must split. When an index node splits, information about that split is posted to the parent of the index node. Eventually, the root will split and the hB-tree will grow in height. We shall give precise algorithms for these operations on the hB-tree and show how these algorithms lead to node utilization that is excellent on average, and still quite good in the worst case. We also establish bounds on the amount of information posted to higher levels, so that we can establish a lower bound on hB-tree node fan-out.

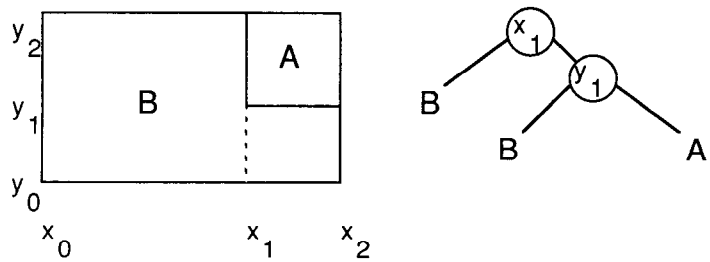


Fig. 2. A holey brick is represented via a k-d tree. A holey brick is a brick from which a smaller brick has been removed. Two leaves of the k-d tree are required to reference the holey brick region denoted by B.

An overview of some of the other proposed multiattribute search structures is given in Section 2. We then introduce the hB-tree and show how to search for points (exact match queries) and for regions (range queries) in Section 3.

In subsequent sections we describe insertion and updating. In Section 4 we show how the insertion process compares with that of single-attribute B^+-trees; this leads to our node utilization analysis and our guarantees on fan-out. In Section 5 we give the details of node splitting, and in Section 6 show how to post index information. In the last section we summarize and discuss these results. In the appendix, we include proofs of the correctness of some of our assertions.

In sections where it is relevant, we establish worst-case results for storage utilization and for size of index entries. These worst-case results *guarantee* that the hB-tree will have good space utilization in the index and data pages. Exact match queries are *guaranteed* to have good performance and the index is *guaranteed* to be significantly smaller than the data collection. These guarantees hold for all data distributions, any order of insertion, and any query patterns.

2. OTHER MULTIATTRIBUTE SEARCH STRUCTURES

There have been many structures proposed for multiattribute searching. The simplest idea is to concatenate the key attributes in some order, and use a single-attribute search structure such as a B-tree. The disadvantage of this method is that some attributes are favored over others. A partial match query specifying the value of the first attribute will be very efficient, since the records matching on that attribute will be clustered. A partial match query on the last attribute will require that most, if not all, of the search structure be accessed.

2.1 Bit Interleaving

Another approach is to interleave the keys of several attributes, using first a binary digit from one, then a binary digit from another. This method is used in [14]. In bit interleaving, the ordering of digits from the various attributes is fixed, and typically is the same for the entire key space. Bit interleaving guarantees good node utilization since a single-attribute search structure with good node utilization may be chosen once the interleaving pattern is fixed.

If the assumptions made in choosing this ordering do not apply to the entire key space, or change over time, the only recourse is to choose a new ordering and

reorganize the file. The danger here is that skewed distributions where many records have the same leading bits in some attribute will cause the data to be organized on the disk by the other attributes, as if concatenation had been used.

2.2 The Grid File

The grid file [12, 19] is a good search structure for uniformly distributed multiattribute keys. However, if the distribution of the attributes of the keys are skewed, the index can use large amounts of space. On a single data page overflow, it is possible that a $k - 1$ dimensional slice must be added to the index. This increases the number of divisions in one of the dimensions by one. If the numbers of value divisions in each dimension (the length of the linear scale for a given dimension) are near some constant c, then adding this new $k - 1$ dimensional slice adds c^{k-1} new index entries.

In the worst case, the number of index entries may be $O(-n^k)$, where n is the number of records in the file and k is the number of attributes in the key. This occurs when the data is correlated, for example, lying entirely on the diagonal.

We are concerned with databases for which the grid file index will not fit in memory. In this case, the grid file requires two disk accesses for exact match queries: one access for the index and one for the data.

For range queries a large *index* is especially harmful. It will take many disk accesses just to read the index entries. In fact, the number of data pages for a given range query might be considerably smaller than the number of index pages. This problem is especially sensitive to the number of attributes in the key.

2.3 The K-D-B-Tree

The K-D-B-tree of Robinson [15, 18] works analogously to a B-tree but, instead of nodes containing search values in disjoint intervals of a one-dimensional space, each node "covers" a brick-like region of k-dimensional space. Thus, K-D-B-trees inherit from B-trees the balanced tree property, that is, all paths to leaves of the tree are equal in length. Further, like B^+-trees, all data is stored in leaf nodes, internal nodes containing only index entries which direct the search.

A K-D-B-tree has many of the desirable properties of a good index structure. In particular, because its growth method is similar to that of the B-tree, it adapts to the distribution of attribute values presented to it.

2.3.1 *Splitting Nodes in the K-D-B-Tree.* However, there is an implicit assumption in [15] that data nodes can be split evenly using one attribute value. Figure 3 shows an example where this assumption fails. In Section 4, we give a mathematical analysis of this phenomenon. The even split assumption is made not only in the K-D-B-tree algorithm, but also in many other papers on multiattribute structures. None provide any bounds on how bad performance can be when the assumption is violated.

Index node splitting is more complex. Figure 4 (from [15]) illustrates how Robinson suggests splitting an index node. Such splitting requires a series of cascading splits of nodes in the subtree whose root is the node being split. The problem here is that any single plane through the space represented by an index node may split the space of one or more descendant nodes. Such cascading splitting of descendant nodes, in addition to making the splitting process costly, can also adversely impact storage utilization.

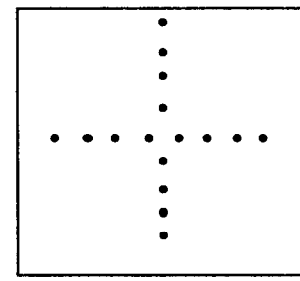


Fig. 3. In this example, one fourth of the records lie on each half axis. No vertical or horizontal line, denoting a single value for one of the attributes, can evenly split this data page. Thus no one attribute value can be used to split the page.

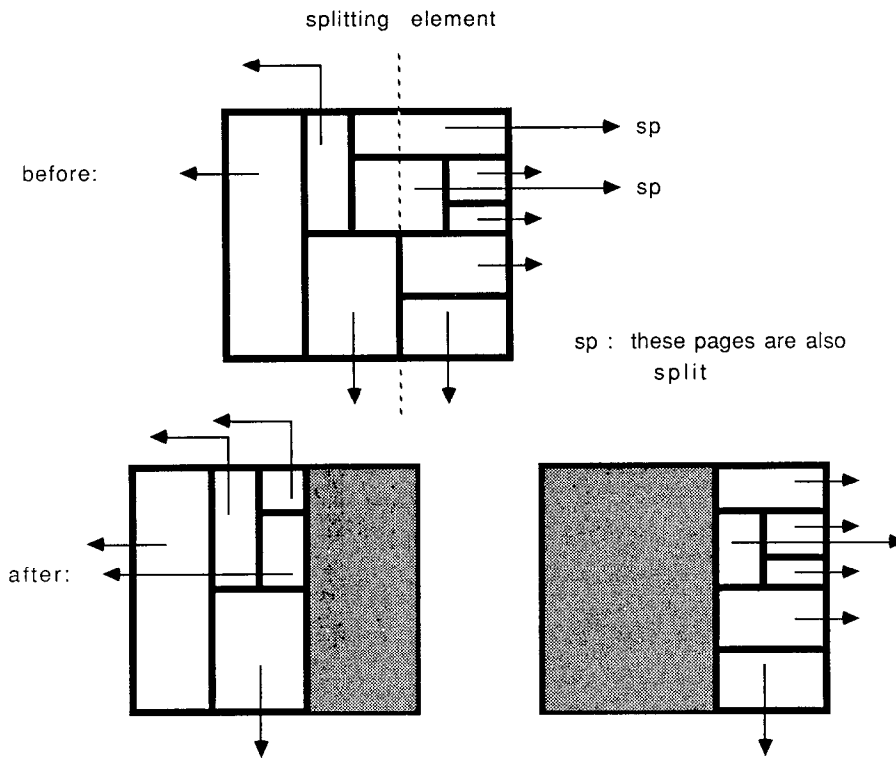


Fig. 4. Index node splitting in K-D-B-trees (from [15]). Here the split of one index node can cause descendant nodes to be split as well. This may cause sparse index nodes to be created on a lower level.

2.3.2 *Internal Index Node Organization in the K-D-B-Tree.* The K-D-B-tree organization, like many file organizations, does not specify how to organize its data within nodes. For single-attribute methods, such as the B-tree, several successful methods exist, most involving sorting the entries and placing them in an array to permit binary search. Such an array is also easy to deal with during node splitting.

For k-attribute searching, however, it is not so clear how entries should be organized within a node. Multiattribute search structures within a node may also

make it difficult to reorganize nodes after they are split. What is desired, obviously, is a search structure that is space efficient, and supports fast search while being compatible with an effective splitting process.

3. THE hB-TREE

The hB-tree is a variant to Robinson's K-D-B-tree [15]. There are two distinguishing features of hB-trees.

(1) Index nodes are organized as k-d trees.

(2) Splitting of a node may require the participation of more than one attribute.

The result is that nodes no longer correspond to bricks in k-space, but rather to "holey" bricks, or bricks from which smaller bricks have been removed. Because of this, hB-trees can avoid cascading splits. Like B-trees, however, splits can propagate up the tree.

In B^+-trees, values acting as index terms in index nodes are redundant in the sense that these values are derived from keys stored in the leaves of the B^+-tree. This is true of hB-trees too. However, with hB-trees, the index terms must partition a k-dimensional space, not merely a one-dimensional space.

3.1 The Advantages of k-d Trees

With K-D-B-trees, each index node represents a region or brick in k dimensional space as in Figure 5. Each such brick is split into sub-bricks whose union is the region represented by the node. These sub-bricks represent regions on a lower level of the index. At the bottom level of the index the sub-bricks represent data nodes. No indication is given in [15] as to how these sub-bricks are represented within the index node.

In Figure 5a we show a two-dimensional space partitioned into separate regions as in [15]. In Figure 5b we show a k-d tree which realizes this partitioning, while Figure 5c shows another way to describe the partition, by explicitly listing the boundaries of each brick. While there may be other ways of representing multiattribute index terms, these are two clear choices, and k-d trees have substantial advantages in both search speed and space.

3.1.1 *Intranode Search Speed.* In Figure 5b, we need only make two comparisons to find the correct node, using the k-d tree. Figure 5c shows the same information kept as a list of the boundaries of the bricks. Even if the point we are searching for is in the first brick on the list, we must make four comparisons, for the four boundary values, just to verify that this is the correct brick. If the point we are searching for is in the last brick on the list, we would make at least one comparison (and possibly more) in order to *rule out* the first three boundary descriptions, and then make four comparisons to verify that the last brick was correct. (This last can be avoided if we know that the bricks cover the entire space.)

The reason for the k-d tree advantage is that bricks share boundaries. In the boundaries representation, these are checked redundantly, while for a k-d tree, a boundary is typically checked only once. When we deal with holey bricks, the k-d tree retains its advantage over the boundaries representation on average, but

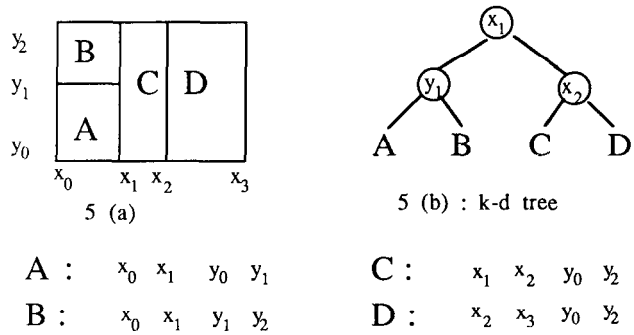


Fig. 5. The internal representation of K-D-B-tree nodes.

loses it in the worst cases. We return to this after introducing holey bricks in Section 3.2.

3.1.1.1 *k-d Tree Representation for Bricks.* Usually, searching for a correct lower level node using a k-d tree requires order log n comparisons (i.e., if the tree is well-balanced, where n is the number of bricks). There are $n - 1$ nodes in the k-d tree with n references to the bricks. In the worst case, the nodes will all be in a line (i.e., the k-d tree will be linear). For this tree, one needs to make an average of $(n - 1)/2$ comparisons. The worst-case search for this worst-case tree requires $n - 1$ comparisons.

If we assume that $k = 2$ and that $n = 100$, then the costs are as follows. For the worst-case k-d tree, when the tree is linear, the worst-case search is $n - 1 = 99$ comparisons and the average search is $(n - 1)/2 = 49.5$ comparisons, since the number of k-d tree nodes is one less than the number of bricks referenced. With an "average" tree (i.e., one that is approximately balanced) about $\log_2(n) = 7$ comparisons are needed for all searches.

3.1.1.2 *Boundaries Representation for Bricks.* For the list of boundaries representation, the search cost is higher. On average, one must reject $(n - 1)/2$ bricks before finding the correct brick. At worst, one rejects $n - 1$ bricks. To confirm that a brick is correct, one must make $2k$ comparisons, one for each of the $2k$ boundaries.

For each brick that does not contain the point that is the search argument, the cost, in comparisons, is as follows. For each dimension of a rejected brick, one boundary value must always be compared to the point's coordinate in that dimension. The point will satisfy the first boundary comparison half the time before being rejected by the other boundary. Thus, assuming no optimizations based on the point's coordinates, to check a dimension requires an average of 1.5 comparisons. If the point survives the first dimension checked, an average of 2 + 1.5 comparisons will be needed, and so on.

What is the probability that a brick is rejected in a check on a given dimension? To answer this, we assume that each brick is about the same size and that each of the k dimensions is subdivided into b parts (where $b = n^{1/k}$) by the bricks.

Then the probability that a point survives the first dimension checked is $1/b$. For example, if $n = 100$, and $k = 2$, b is 10, and the probability that a given point is within the boundaries of a given dimension for one of the bricks is 1/10.

On average, the number of comparisons needed to reject a brick is

$$1.5 + (1/b)(2 + 1.5) + (1/b^2)(4 + 1.5) + \ldots =$$

$$1.5 + \sum_{i=1}^{k-1} \frac{2i + 1.5}{b^i}.$$

In our example, this is 1.85 comparisons per rejected brick.

Since $(n - 1)/2$ bricks must, on average, be rejected before finding the correct brick, and confirming the correct brick requires $2k$ comparisons, the average number of comparisons needed to find the correct brick (again for $k = 2$ and $n = 100$) is $1.85(99/2) + 4 = 95.6$. The worst case for this is $(n - 1) \times 2k = 396$ comparisons, when a brick is not rejected until all boundaries have been compared and the correct brick is the last brick examined.

3.1.2 *Space Used with Intranode Representations.* It should be obvious that the k-d tree representation also takes up less space than the boundaries list representation. We assume that pointers, both intranode pointers and external pointers, are equal in size to attribute values. This almost surely overstates their size and works against the k-d tree representation, which has more pointers and fewer attribute values. Even with this handicap, the k-d tree representation is superior.

3.1.2.1 *k-d Tree Representation for Bricks.* Each internal node of the k-d tree contains only one comparison value, pointers to a left and right child, and a flag to indicate which attribute is being compared. We assume the flag, which serves other purposes as well, is equal to half an attribute value, again a generous estimate. The entire collection of necessary information here is not likely to exceed two bytes, while attribute values will frequently be longer than four bytes. Hence, each node of the k-d tree is 3.5 attribute value equivalents. (Note that the boundaries representation will almost surely require control information as well, but it has been ignored here.)

For n bricks, there are $n - 1$ k-d tree nodes. This means $(n - 1) \times 3.5$ attribute value equivalents. Note that the number of dimensions of the space k does not affect the space used, when dealing with bricks, except in the encoding of the attribute to which each node corresponds.

3.1.2.2 *Boundaries Representation for Bricks.* The boundary representation contains many duplicate attribute values. Regardless of the structure or distribution of bricks or holey bricks, the amount of space consumed by the boundaries representation is the same. There must be $n \times 2k$ attribute values in the boundary representation as well as n pointers. Thus, the space consumed is $(2k + 1) \times n$ attribute value equivalents. Since $k >= 2$, when dealing with bricks, the boundaries representation always requires at minimum $5 \times n$ attribute value equivalents of space, and this increases with k. The k-d tree is always better than this.

In our example in Figure 5, we needed 16 boundary values and four pointers for the brick boundary representation in Figure 5c, but only three boundary

values and six pointers for Figure 5b. Given our assumptions, the k-d tree has space equal to 10.5 attribute value equivalents, while the boundaries representation takes 20 attribute value equivalents.

3.2 Holey Bricks

3.2.1 *Solving the Node Splitting Problem.* As illustrated in Figure 3, splitting data nodes using a single attribute value may present difficulties because the resulting nodes have substantially different utilizations. In Figure 8, we show an example where an index node cannot be split at the root of the k-d tree for the same reason. In K-D-B-trees, an index node was split by drawing a plane through its region, splitting subregions in the process. This technique required lower level nodes to be split, descending down the tree, and possibly causing sparse nodes at lower levels.

By removing the restriction that nodes must be split using only a single attribute, these problems are avoided. The regions of k-space resulting from this kind of splitting are no longer bricks, but rather bricks with, perhaps, smaller bricks removed from them. We call these regions *holey bricks*, since the subregions removed may be anywhere in the original region, even completely interior to the region. We call the region removed the *extracted region*, and the holey region the *enclosing region.*

In the Appendix, we prove that good space utilization can be guaranteed when splitting a data node if the extracted region is a corner of the enclosing region. Importantly, corner regions only require at most one boundary per dimension. Our analysis below assumes this form of data node splitting. Other forms of splitting do not necessarily have guarantees.

3.2.2 *Representing Holey Bricks.* Like bricks, holey bricks can be represented either via k-d trees or using a boundaries representation. The boundaries representation is unchanged, although we may wish to order the list of regions with more care. The situation is somewhat different with k-d trees.

In order to represent holey bricks using k-d trees, several leaves of the k-d tree may refer to the same node on a lower level in the hB-tree. This is illustrated in Figure 2. One leaf refers to the extracted region (A). Two leaves refer to the enclosing region (B).

Using holey bricks as the regions described by our hB-tree index does not affect the search algorithm. It does, however, occasionally cause hB-tree nodes to have more than one parent. This means that the hB-tree is not truly a tree; it is a directed acyclic graph, or DAG. We shall continue to use tree terminology, as levels, height, ancestors, and children are still concepts that make sense in this context.

We use fragments of k-d trees as our index terms and compose them with other such k-d tree fragment index terms into the k-d tree that exists within an hB-tree index node. In addition, we can bound the size of these index terms, which enables us to derive lower bounds on hB-tree index node fan-out.

3.2.3 *Performance Impact of Holey Bricks.* With the boundaries representation, the space consumed is the same as required for bricks. The search must now find the smallest brick that contains the argument point, since these bricks can be

nested. Aside from this, however, search is the same as before, and the worst-case bounds are the same. The average case search can be made the same as before by ordering the boundaries list, putting enclosing brick after extracted brick in the ordering.

With holey bricks, the k-d trees will sometimes be larger. When an index term is posted, it can consist of as many as k k-d tree nodes when reflecting a data node split. The k is the maximum number of boundaries for an extracted corner region. Hence, the worst-case linear tree has $(n - 1) \times k$ k-d tree nodes. This contrasts with $(n - 1)$ nodes when we deal with bricks without holes.

The result of the above difference is to multiply the worst-case search cost and the worst-case space consumption by a factor of k. (This should be regarded as only an approximate characterization.) For the average case, we would expect that only somewhat more than one node is actually needed in order to successfully split a data node. Below, we assume 1.5 nodes (e.g., that half the time a data node can be split with a single attribute value while two attribute values are required the rest of the time).

Returning to our previous example of $n = 100$ nodes with $k = 2$, we have the following.

The average search for the worst-case (linear) k-d tree requires $(n - 1) \times k/2 = 99$ comparisons. The worst-case search on the worst-case tree requires $(n - 1) \times k = 198$ comparisons. This is comparable to the average and worst-case results for the boundaries representation. This tree consumes $3.5 \times k \times (n - 1)$ attribute value equivalents of space, which is worse than the $(2k + 1) \times n$ attribute equivalents needed by the boundaries representation.

However, about $\log_2 (1.5 \times n)$ comparisons are all that are needed for an "average," approximately balanced k-d tree with about 1.5 k-d tree nodes posted for a split. For a balanced tree, again only about seven comparisons are all that are needed, even in the worst case. With respect to space, this average tree requires $5.25 \times (n - 1)$ attribute equivalents of space, which is comparable to the space required by the boundaries representation.

Thus, in the holey brick case, the k-d tree representation still has a decided performance advantage compared to the boundaries representation, though it does not perform as well as when only bricks are permitted. (Of course, with bricks, one does not get space utilization and fan-out guarantees.) The space required for an average tree is comparable to the boundaries representation when $k = 2$, and is smaller with $k > 2$. For the worst-case tree, space consumption is not good for the k-d tree representation. The internal pointers of the k-d tree form constitute too much overhead. Fortunately, the worst-case linear tree with k attributes per holey brick will almost never occur.

3.3 Searching Using the hB-tree

The k-d tree leaves in an hB-tree index node refer to lower level hB-tree nodes. In each internal node of the k-d tree, there is an indicator of which attribute is to be compared, what the comparison value is, and whether equality is on the left branch (less than or equal) or the right branch (greater than or equal). We cannot always have equality on the left (or always on the right) and still guarantee even splitting (Section 4).

3.3.1 *Exact Match Queries.* To do an exact match search, one follows a unique path from the root of the hB-tree down the hB-tree to a data page. The number of hB-tree nodes accessed is the height of the hB-tree, which depends only on the hB-tree fan-out. The k-d trees within hB-tree nodes should only be regarded as directing the search to the next lower level of the hB-tree, just as index values in B^+-trees or boundary values in other hierarchical multiattribute search structures direct such a search.

3.3.2 *Range Queries.* A *range query* specifies a range of values for one or more of the attributes. If a comparison value in a k-d-tree node is greater than all the same attribute's values in the search range, one goes to the left. If it is smaller, one goes to the right. If a comparison value is in the middle of a search range, one follows both the left and right branches. Several hB-tree nodes at each level may be accessed. Range searching results in a similar search pattern for the one-dimensional B^+-trees as well, but it is easy to overlook this.

3.3.3 *Region Data.* The hB-tree can be used for region data. Brick-like regions can be considered data points if they are described by the upper and lower coordinates in each attribute. For example, a rectangle in two-dimensional space can be specified by its lower y and upper y and lower x and upper x coordinates. A rectangle is thus a point in a four-dimensional space. For regions of different shapes, their enclosing rectangles can be used to identify candidate regions for the search. More detailed examination of the enclosing rectangles is needed to make a final determination. This examination is beyond the scope of this paper.

Exact match search for a region is the same as exact match for any other data point. Values are given for all the upper and lower coordinates.

Finding all the regions D *contained in* a given region R is a range query. The high x coordinate of D must be less than or equal to the high x coordinate of R. The low x coordinate of D must be greater than or equal to the low x coordinate of R, and so forth. Similar inequalities are needed to find all the regions *intersecting* a given region.

Thus, searching for region data is similar to searching for point data. Only the number of attributes being compared is different, that is, it is doubled.

3.3.4 *Summary of Searching in hB-trees.* Searching for data within an hB-tree index node is a very straightforward process, essentially the same as searching using a binary search tree. For exact match searches, the number of hB-tree nodes accessed is exactly the height of the tree. As with single-attribute B-tree indexes, the upper levels of this tree will be in memory, so that with decent node fan-out, this should result in no more than *two disk accesses* for most files. We make estimates of node fan-out in Section 4.

Range searches may require accessing several hB-tree nodes on each level. This is also true for single-attribute B-tree indexes. Region data can be treated like point data, but with double the number of dimensions.

4. INSERTING AND UPDATING IN hB-TREES

4.1 Commonality with B^+-trees

In this section, an algorithm for updating an hB-tree is described in general terms. The algorithm is, at this level of description, exactly the same as the

corresponding algorithm for B^+-trees. The steps of the algorithm are as follows:

General Updating Algorithm

1. Search the hB-tree for the node which is to contain the record to be inserted. This search uses k-d trees in hB-tree index nodes, as other constructs are used in B-tree index nodes, but the end result is to have retrieved the node into which the record is to be inserted.
2. Within this node, find the location where the new record is to be placed and insert it there. There are two possibilities:
   a. The record fits into the space available for it. In this case, the insertion is complete.
   b. The record is too large for the available space. In this case, the data node must be split.
      i. Allocate a new node and split the contents of the original node between the original node and the new node as evenly as possible.
      ii. Post an index term in the next higher level of the tree which identifies the new key space partitioning between original node and new node. If the higher level index node overflows, split it analogously.
      iii. Reinsert the record into the resulting restructured tree.

4.2 Divergence from B^+-trees

The hB-tree differs from the B^+-tree algorithm in some important details, including:

(1) How a data node is organized and how splitting is accomplished.
(2) How an index node is organized and how splitting is accomplished.
(3) What the index terms are and how they are posted to the next higher level of the tree.
(4) What the guaranteed storage utilization is.

These details are explained in the next sections.

4.3 Data Node Splitting

We model a file with a two-attribute hB-tree using an x-y coordinate system. A point (x_0, y_0) in such a space represents a record whose value in the first attribute is x_0 and whose value in the second attribute is y_0.

Suppose now that half the records lie on the x-axis and half on the y-axis, as in Figure 3. Of the records on the y-axis, half lie above the x-axis and half lie below. A similar configuration of points is on the x-axis. In other words, each half-axis contains one-fourth of the points. Then there is no horizontal line or vertical line which will split the space more evenly than in a 3:1 ratio, with $\frac{3}{4}$ of the points on one side of the line and $\frac{1}{4}$ of the points on the other side. A horizontal line would split the data node using one value of the second (y) attribute. A vertical line would split the data node using one value of the first (x) attribute. This demonstrates that using one attribute value alone will not always split a data node evenly.

This argument generalizes to k dimensions. If we have a k-dimensional space, we can place $1/(2k)$ points on each of the $2k$ half-axes, and no one attribute value will split the space in better than a $(2k - 1)$:1 ratio. That is, for this point

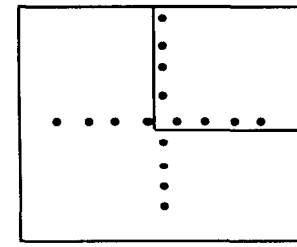


Fig. 6. A corner split data node. Using the values for several of the attributes we can split any data node in at least a 2:1 ratio.

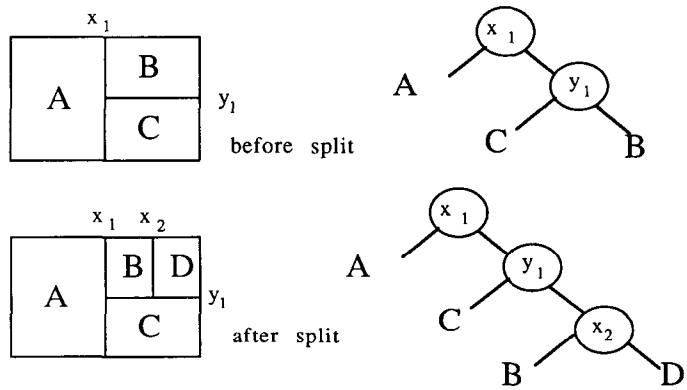


Fig. 7. The usual case for a data node split. The data node B is a brick, and the split is along the x attribute at x_2 forming two bricks. One brick remains at disk address B while a new brick is allocated at D.

distribution, at best, $1/(2k)$ of the points will be on one side of the hyperplane determined by our single attribute value and $(2k - 1)/(2k)$ points lie on the other side of the hyperplane.

With at most k attribute values (in a k-dimensional space) determining a "closed corner" as in Figure 6, we can guarantee at least a 2:1 split in the data points for any point distribution. That is, the data points can be divided so that no more than $\frac{2}{3}$ of them are in one node and no less than $\frac{1}{3}$ of them are in the other. We prove this in Appendix 1. We can guarantee minimum data node utilization of at least $\frac{1}{3}$. Note that we usually do much better than 2:1. In Figure 6, we split the data described in a 1:1 ratio using two attributes.

If the full data node represents a brick in k-dimensional space before the split, then there is only one reference to it in the index. In this case, we replace that reference with the k-d tree representing the split. This will usually be a split with only one attribute, and we will be adding only one k-d tree node to the index. One of its pointers will refer to the original hB-tree data node while the other will refer to the new hB-tree data node. This usual case is illustrated in Figure 7.

In the general case, the full data node is a holey brick, and there are several references to it in the index. Despite this, we can guarantee no more than k new

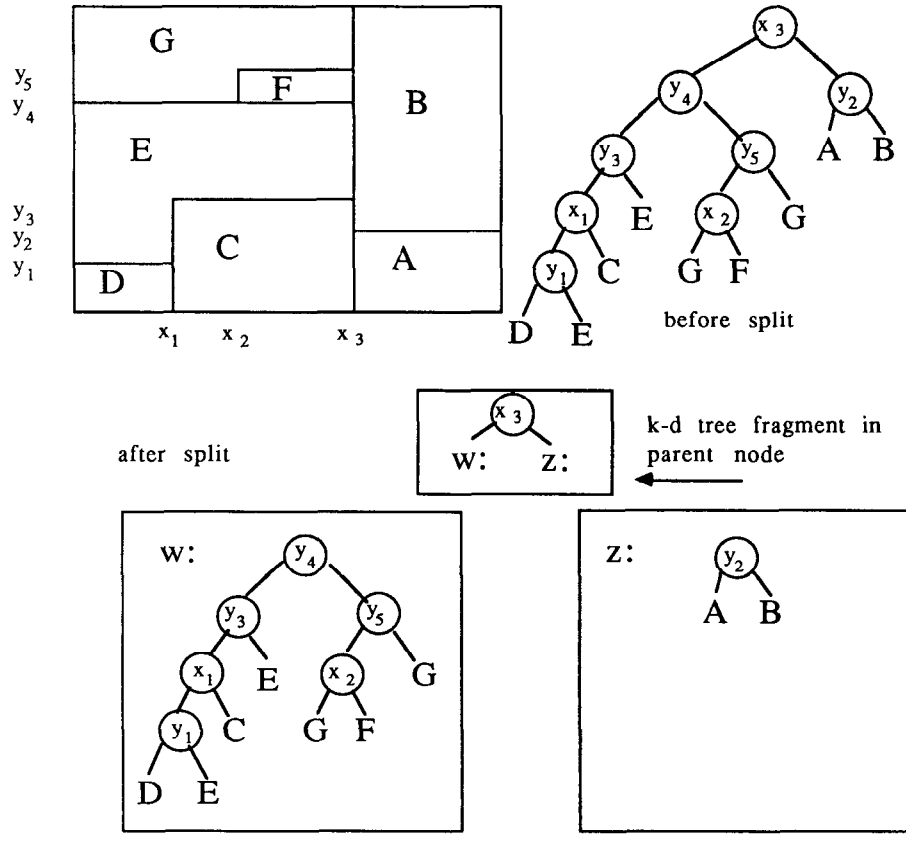


Fig. 8. An example where splitting an index node at the root of the k-d tree results in poor fan-out and low node utilization.

k-d tree nodes posted to the index in the worst case, and sometimes we post no new nodes; we only change references. Section 5 describes the general algorithm.

4.4 Index Node Splitting

Once we have decided to use k-d trees in our hB-tree index nodes, we must decide how to split *them*. In [15], a split of a K-D-B-tree index nodes was made by slicing it with a hyperplane. This necessitated changes on lower level K-D-B-tree nodes, sometimes creating new sparse lower level nodes and sometimes precipitating changes down the tree several levels. We use the k-d tree intranode organization to avoid these problems.

If we always split at the root of the k-d tree, as in Figure 8, we run the risk of creating sparse index nodes. Instead, we exact a subtree as in Figure 9.

Suppose there are N nodes in the k-d tree. We can always find some subtree with between floor($N/3$) and ceiling($2N/3$) nodes. We prove this in Appendix 2. We thus split it into an extracted subtree and an enclosing subtree. This guarantees a minimum node utilization of $\frac{1}{3}$ in the index level nodes. The worst

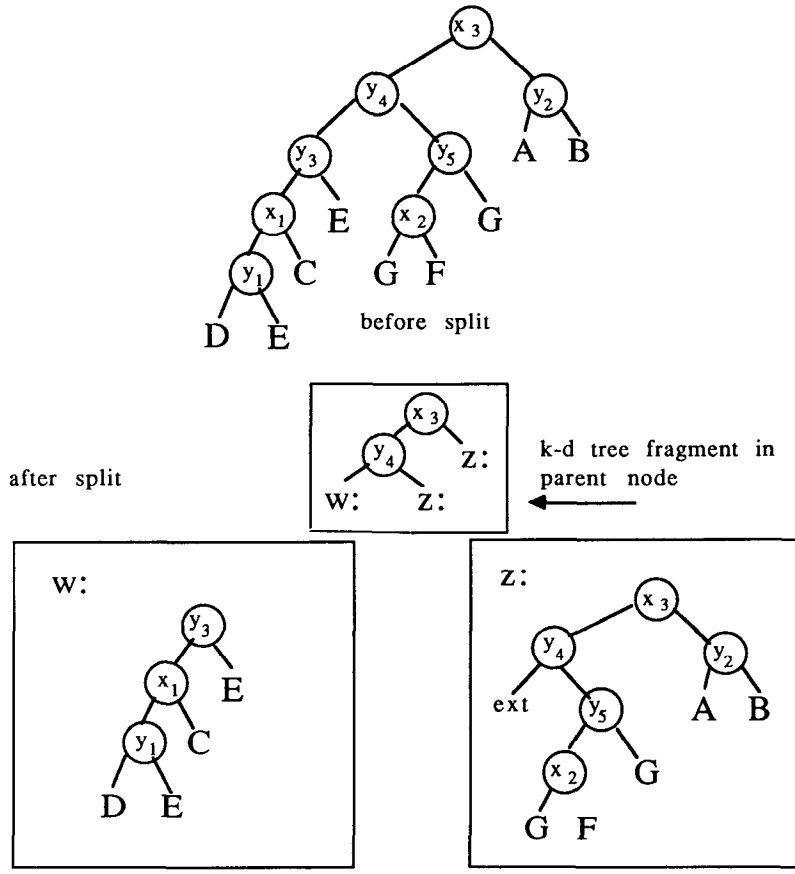


Fig. 9. The same example as in Figure 8, where the k-d tree of an index node is too skewed to split at the root. We find a subtree which has between $\frac{1}{3}$ and $\frac{2}{3}$ of the nodes and extract that subtree.

case, then, is that in one resulting node utilization begins at .66 and in the other at .33.

When an index node is split, we must post information in the parent to distinguish the extracted tree from the enclosing tree. The extracted tree represents a k-dimensional brick. Thus, at worst, we need to post the $2k$ boundaries to the parent. However, by using a subset of the nodes on the path from the root of the extracted tree to the root of the enclosing tree, we can do better most of the time. Also, in this way, the algorithm for posting information is simpler. However, to gain these advantages, we must occasionally post one additional node, so that in the worst case we post $2k + 1$ k-d tree nodes as our hB-tree index term. We give the details of this algorithm in Section 6.

4.5 Anticipated hB-tree Utilization

Determining storage utilization in index organizations subject to uneven splitting was a problem faced in Digital B-trees [8]. The analysis done there used the uniform growth assumption and involved computing recurrence relations. The

result, specialized to single bucket nodes of the sort that we have, and assuming growth by local doubling via node splitting, is

$$U = S \log\left(\frac{1}{S}\right) + (1 - S)\log\left(\frac{1}{1 - S}\right)$$

where S is the fraction of the data going to one new node and $(1 - S)$ is the fraction going to the other. For our worst case 2:1 split, $S = 0.667$ and $U = 0.637$.

While this utilization is not as good as B^+-trees, recall that this is a worst-case result. That is, under the uniform growth assumption, utilization must be at least .637. More typically, some intermediate split factor would occur, e.g., 1.5:1 yielding $S = 0.6$ and $U = 0.673$, which is very close to B^+-tree utilization. Partial expansions [10] can be employed, as with B^+-trees, should higher utilization be desirable.

We emphasize that it is this sort of analysis that is missing from most other papers on multiattribute structures. Many structures will work well with uniformly distributed data, or with data whose distribution can be predicted in advance. However, real data collections have neither uniform nor predictable distributions.

The reason that B^+-trees are the most popular *single-attribute* indexing method in use today is their worst-case guarantees, which show that space utilization and query time cannot be arbitrarily bad. B^+-trees can handle highly skewed and highly unpredictable dynamic data collections. Our hB-tree results provide analogous worst-case guarantees for the multiattribute case.

4.6 Fan-Out

We are interested in two results. First, we would like to estimate the fan-out for hB-trees so as to determine the storage burden imposed by the index levels of the tree. Second, we want to compare fan-out with B^+-tree fan-out, as B^+-trees are the standard benchmark for index organizations.

Fan-out is a function of three quantities, node size ns, node utilization U, and size of index terms t. Thus,

$$\text{fanout} = \frac{U \times ns}{t}.$$

We assume here a node size of 4K bytes and a utilization, from Section 4.5, of 0.673. We compare this to B^+-tree fan-out, where we assume the same node size and a utilization of 0.693. Both utilizations are the result of analyses. The 4K-node size is typical of large database systems.

The hard part of determining fan-out is to derive an average size for index terms. Comparison values (keys) that appear in an index can vary widely in size. However, eight bytes should be a fairly typical size, especially with judicious use of key compression. Pointers to disk pages can be three bytes. An additional two control bytes are probably required as well. In these two bytes, we keep track of (i) which attribute is used; (ii) the "weight" of the subtree (used in splitting); and (iii) whether equality is on the right or the left. This results in B^+-tree index terms being 11 bytes, and k-d tree nodes in hB-tree index nodes of 16 bytes (comparison value, two control bytes, and two pointers).

We need to know the size of an hB-tree index term, not merely the size of a k-d tree node. In the worst case, we need k k-d tree nodes for index terms that reference hB-tree data nodes and $2k + 1$ k-d tree nodes for index terms that reference hB-tree index nodes. In both cases, however, we usually need only a single k-d tree node. We assume, somewhat arbitrarily, that the average is about 1.5 k-d tree nodes per index term. Such index terms thus average about 24 bytes.

Factoring in the small difference in storage utilization results in B^+-trees with about twice the fan-out of hB-trees. That is, B^+-trees have a fan-out of 258 while hB-tree fan-out is 115. Note that the size of all index nodes in an hB-tree is less than one percent of the size of the data nodes, while, of course, B^+-tree index size is even smaller. However, and importantly, this demonstrates, for reasonable size nodes, that the size of the index in hB-trees is *not* a significant factor.

For small values of k, e.g., $k = 2$, it is almost surely the case that the average index term for hB-trees is close to one k-d tree node. In this case, fan-out is even better, being 172, or 0.67 of the fan-out of a B^+-tree. The worst case with $k = 2$ is a fan-out of 86 for the bottom index level. This should be exceedingly rare, and is not a disaster in any event.

5. NODE SPLITTING

In this section we show how to use k-d trees to split holey brick nodes. The previous split planes are used whenever possible, and each k-d tree node represents such a split plane.

Each hB-tree node has stored with it its outer boundaries. This information is required for posting index terms, as we shall see. We have ignored the space used for this in our fan-out analysis. This is $2k$ eight-byte values in each node—in our fan-out analysis. (For $k = 2$, it is 32 bytes in each 4K node.) (B^+-trees also typically contain some overhead information in each node.)

Since each node represents a holey brick, the "internal" boundaries of this brick, from which smaller bricks may have been extracted, are stored in it. These internal boundaries are stored as a k-d tree, which represents the partitioning of the node that was necessary to extract subregions during previous splits. It is called the *local k-d tree.*

A local k-d tree contains four kinds of nodes:

(1) Internal nodes.
(2) Three kinds of leaf nodes:
   (a) a leaf that references a collection of data records (in data nodes only);
   (b) a leaf that references an hB-tree node on the next lower level in the hB-tree (in internal hB-tree nodes only); and
   (c) a marker that denotes that a subtree is missing (called an *external marker*). Such a marker indicates that a subtree has previously been extracted. This subtree is referenced through a different search path in higher level hB-tree nodes.

Data nodes contain both data records and a local k-d tree, which is typically quite small. All the information in a local k-d tree of a data node is repeated in higher levels of the hB-tree. Index nodes contain *only* the local k-d tree, typically

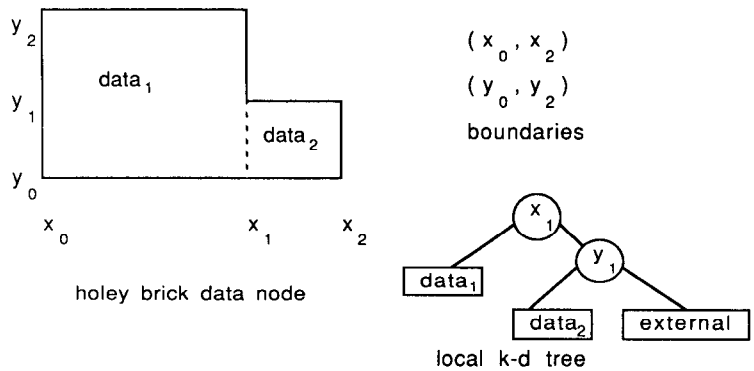


Fig. 10. Boundary and tree information stored in data nodes. The local k-d tree references data in the node and contains "external markers" identifying regions that have been extracted.

quite large. A small portion of the local k-d tree of an index node is repeated in higher levels of the hB-tree. The data node configuration is shown in Figure 10.

5.1 Terminology

The subsequent discussion is simplified by the introduction of some technical terms, which allow us to systematically distinguish between nodes of the hB-tree and nodes of k-d trees. To emphasize the distinction between these trees, terms referring to the hB-tree use capital letters while terms referring to the k-d tree are in lowercase. We use the letter *f* to stand for the *Full* tree or node that is to be split. We use *n* for the *eNclosing* tree or node and *x* for the *eXtracted* tree or node. Finally, we use *i* for *Index*, when referring to levels of the hB-tree above the node being split.

Thus, when an hB-tree node requires splitting, we need to discuss the following items:

(1) **F-NODE.** The hB-tree node that requires splitting. It stores the data or index terms of the original region.

(2) **N-NODE.** The hB-tree node that contains data or index terms relevant to the enclosing region, which results from the F-NODE after the extracted region has been removed. Typically, the N-NODE occupies the same disk space as did the F-NODE.

(3) **X-NODE.** The hB-tree node that contains data or index terms relevant to the extracted region, which is removed from the F-NODE. The X-NODE is typically newly allocated during the splitting process.

(4) **I-NODE.** The hB-tree node (or nodes) that contains the fragment of the k-d tree that indexes the regions extracted from the F-NODE and refers to the N-NODE and the X-NODE. The I-NODE is at a higher level in the hB-tree than the F-NODE.

(5) **f-tree.** The local k-d tree of the F-NODE, describing the holey brick of the full region. It contains, as subtrees, the original forms of the n-tree and the x-tree. Nodes of the f-tree are called f-nodes while its root is the f-root.

(6) **n-tree.** The local k-d tree of the N-NODE, describing the holey brick of the enclosing region. Nodes to the n-tree are called n-nodes while its root is the n-root.

(7) **x-tree.** The local k-d tree of the X-NODE, describing the holey brick of the extracted region. Nodes of the x-tree are called x-nodes while its root is the x-root.

(8) **i-tree.** The k-d tree fragment in the I-NODE (or nodes), which describes the regions extracted from an F-NODE. Nodes of the i-tree are called i-nodes while its root is the i-root. The i-tree is the index term(s) referring to the N-NODE and the X-NODE.

5.2 Splitting at the f-Root

If the f-root can be used to split the F-NODE into two nodes, each of which contains between $\frac{1}{3}$ and $\frac{2}{3}$ of the contents (the k-d tree nodes or the data records), we do so. In this case, we arbitrarily denote the right subtree as the basis for the x-tree. The f-tree, with the right subtree extracted is the basis for the n-tree. We move the right subtree (and when the F-NODE is a data node, its data records) to the X-NODE.

We change the boundary description in the F-NODE to a boundary description relevant to the N-NODE. The N-NODE will not in this case, nor in general, have the same boundary as the F-NODE. The boundary for the N-NODE, and its resulting n-tree, are formed by removing nodes from the f-tree, starting from the f-root, and finding the smallest subtree that contains all the N-NODE leaves referring to data or to lower levels on the hB-tree. The leaves that are markers referring to previously extracted subtrees are not necessary, since they refer to data or k-d trees which were not in the F-NODE. When splitting at the f-root, we are assured that the f-root itself is always removed from the n-tree.

The right subtree becomes the basis for the x-tree. As with the n-tree, we reduce this right subtree by removing nodes until we have found the smallest subtree containing all the X-NODE leaves referring to data or to lower levels of the hB-tree. The extracted region has boundaries, as always, that reflect the x-tree that we have determined. We illustrate splitting an F-NODE using the f-root in Figure 11.

5.3 Splitting at a Lower Subtree of the f-Tree

If the value in the f-root does not split the contents in a 1:2 ratio or better, one subtree refers to more than $\frac{2}{3}$ of the contents. Follow the arc leading to this subtree. Applying this strategy recursively to successive f-nodes will lead either to a subtree that corresponds to between $\frac{1}{3}$ and $\frac{2}{3}$ of the contents or else it will lead to a leaf f-node (in the case of a data node only) that refers to more than $\frac{2}{3}$ of the contents. We treat the subtree case first.

The contents, to which the subtree found above refers, are moved to the new X-NODE. This subtree forms the basis for the x-tree, the local subtree for the X-NODE. This subtree is reduced, as mentioned above, before being established as the x-tree, and the boundary for the X-NODE is shrunk accordingly.

The boundary description in the N-NODE is inherited from the F-NODE, as before. It too must be reduced, as mentioned above, but in this case the boundary

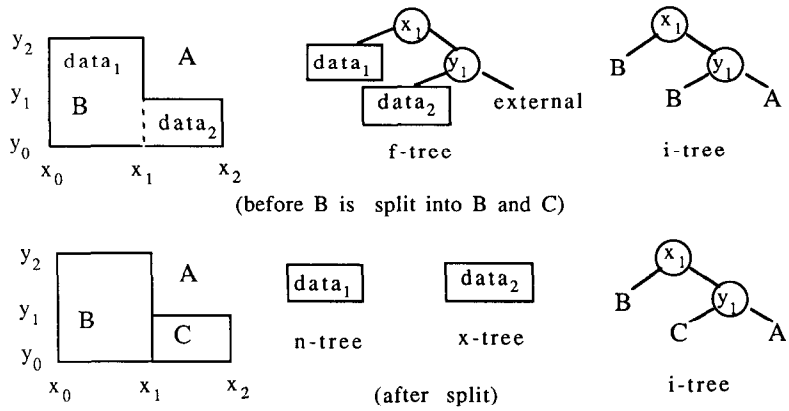


Fig. 11. The F-NODE is split by the value in the f-root. No new i-nodes are added to the i-tree. The left subtree becomes the n-tree for the N-NODE. The right subtree, the basis for the x-tree, is reduced.

does not necessarily change. Further, in the n-tree, the extracted subtree is replaced with a marker to note that it has been extracted. This splitting situation is illustrated in Figure 12.

So far this process enables us to occasionally make the hB-tree nodes more like full bricks than holey ones. That is, it reduces the number of k-d tree nodes in the n-tree and the x-tree. If either the n-tree or the x-tree no longer contains external markers, it represents a brick. In this case the local k-d tree of the data node degenerates to the empty tree.

5.4 Splitting a Leaf of the f-Tree

In data nodes, the splitting process sometimes leads to a f-tree leaf corresponding to more than $\frac{2}{3}$ of the contents. (Recall that the contents of the node include both the data records and the k-d tree nodes of the local tree.) This happens, for instance, if the data node is a brick, not a holey brick. Consider the records corresponding to the leaf of the f-tree containing more than $\frac{2}{3}$ of the contents. Call this collection of records S. Call the collection of all the records in the F-NODE T. If the F-NODE is a brick, $S = T$.

We are guaranteed by the theorem in Appendix 1 to find a corner (which can be described by at most one boundary value for each of the k attributes) with between $\frac{1}{3}$ and $\frac{2}{3}$ of the contents in the entire F-NODE. This corner will become the X-NODE X. And, it will contain data only from S. The N-NODE will contain all the records not in S plus the records which are in S, but not in X. The f-tree is extended by replacing the leaf in the f-tree corresponding to the region representing S with the k-d tree representing this split.

An algorithm can be constructed from the proof in Appendix 1 once an ordering for the attributes has been chosen. We attempt first to use the median of the first attribute to split the node. If this does not produce the desired 2:1 ratio (or better), we attempt the second attribute, then the third, and so on. If none of the medians work, we try the upper corner produced by the first and second attributes

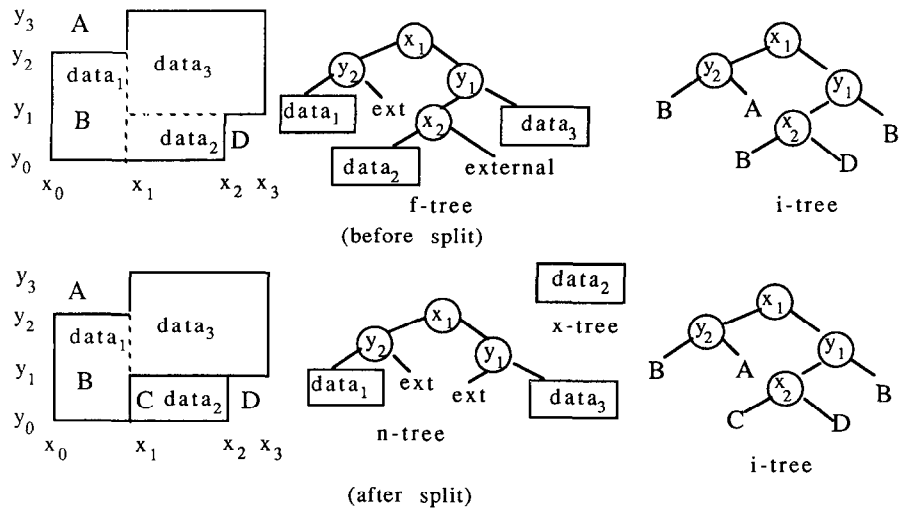


Fig. 12. We assume here that the f-root of B does not suffice to split B evenly. However, a lower subtree works. No new i-nodes are added to the i-tree. The subtree becomes the basis for the x-tree. The x-tree is replaced by a marker in the n-tree.

and then the lower corner produced by the first and second attributes. If this does not work, we try the first, second, and third attributes, and so forth. The theorem guarantees that the ratio will be obtained by this process.

We have not specified how to order the attributes. Data node splitting does not require that the trials described above be performed in the order specified either. There are many potential strategies for both attribute order and composition of attributes to perform a data node split. The above sequence, together with round-robin ordering of attributes, is but one possibility. Optimizing for good data clustering by attribute, for even division of data and for minimizing index term size, is a complex optimization problem that depends greatly on how important each of the mentioned characteristics is. This is surely a topic for further research.

The holes in the brick may cause the area of a holey brick to become noncontiguous. This does not cause any problems with the node utilization estimates or the search algorithms. Figure 13 illustrates both the contiguous and the noncontiguous possibilities.

Node-Splitting Algorithm

1. Start at the f-root. While the subtree of the f-tree contains more than $\frac{2}{3}$ of the contents, proceed to its descendent node containing the larger amount of contents. Either a leaf is reached or a subtree (the precursor for the x-tree) with between $\frac{1}{3}$ and $\frac{2}{3}$ of the contents found.
2. If an f-leaf is reached, which references more than $\frac{2}{3}$ of the contents (only possible in data nodes), extend the f-tree so as to partition these records as described in Section 4.3 above. The extended f-tree can be used in step 3. One leaf of the extended f-tree refers to between $\frac{1}{3}$ and $\frac{2}{3}$ of the contents. This leaf is the "subtree" referred to in the following steps.

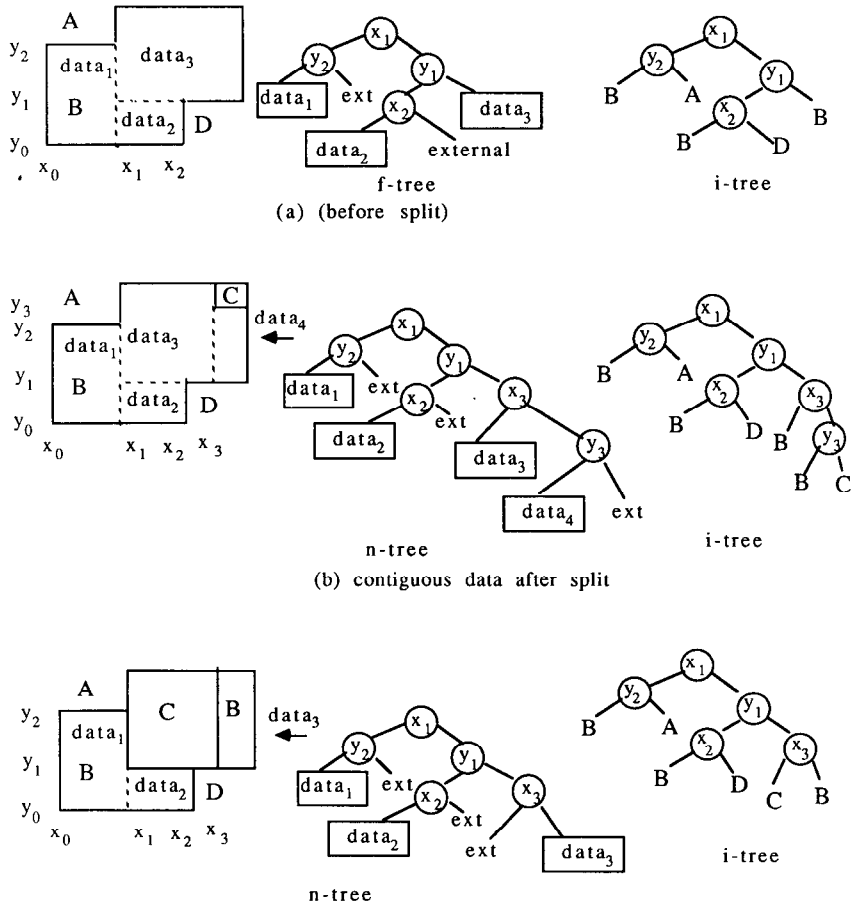


Fig. 13. One of the leaves of the f-tree refers to more than $\frac{2}{3}$ of the contents of the F-NODE. We replace the reference to this leaf, in both the i-tree and the n-tree, with a k-d tree describing the corner split. The local tree of the X-NODE is empty.

3. Replace the subtree with an "external" node marker. Now remove f-nodes, starting at the f-root, until the smallest subtree is found that contains all the leaves referring to data or to lower level hB-tree nodes. This is the n-tree. Refine the boundary for the F-NODE to be the N-NODE boundary by restricting the boundary to exactly that required by the new n-tree.
4. The subtree extracted is the basis for the x-tree. Move the subtree found in step 3 to the X-NODE. Reduce this subtree to the smallest subtree containing all the leaves referring to data or to lower level hB-tree nodes. This is now the x-tree. The boundary for the X-NODE is found by the restriction process described in step 3.

6. POSTING INDEX TERMS

When an F-NODE is split, we must post information in the parent to distinguish the X-NODE from the N-NODE. The X-NODE represents a k-dimensional brick. Thus, at worst, we need to post its $2k$ boundaries to the parent. However,

because some of the boundaries of the X-NODE are already in the i-tree for the F-NODE being split, it is frequently possible to post fewer i-nodes (boundaries). All potential boundary values are on the path from the f-root to the x-root. Thus, our task is to choose exactly the nodes on this path required to identify the X-NODE.

Sometimes, two f-nodes in the path will give bounds in the same direction for the same attribute. For example, one f-node may express "salary greater than $30,000," while an f-node lower down the path may express "salary greater than $40,000." In this case, we only use the f-node with the more restrictive bound. The more restrictive bound is always lower in the path. We thus limit ourselves to $2k$ nodes for bounding, even if the path is longer.

If another tree has previously been extracted from the same f-tree, some of these f-nodes may have already been posted in the i-tree. Also, the path may not contain boundaries in all directions. For instance, a "corner" (rather than a "hole") requires only k boundaries. For these two reasons the number of f-nodes to be posted will usually be considerably less than $2k$, frequently zero or one.

Because this information is to augment an existing i-tree, the resulting i-tree must be a well-formed k-d tree. This sometimes requires an extra f-node to be posted so as to permit the new i-nodes to be connected to the existing i-tree without reordering the i-nodes. We call this extra f-node the *divergence node.* It is the lowest node on the path from the f-root to the x-root, which is itself an ancestor of a previously extracted subtree. It may or may not have already been posted to the I-NODE(s).

We call the part of the path we post the *condensed path.* It contains the most restrictive bounds in each attribute that have not previously been posted, plus possibly one divergence node. In the worst case we post $2k + 1$ f-nodes to the I-NODE(s). We give a formal definition of condensed path.

Definition. The *condensed path* consists of those f-nodes on the path from the f-root to the x-root that:

(1) have not previously been posted and
(2) satisfy one of the following:
    (a) is a least upper bound or a greatest lower bound in some attribute for the x-tree or
    (b) is the divergence node.

CLAIM. *The condensed path correctly determines whether or not a record is contained in the region of an x-tree.*

PROOF OF CLAIM. The full path from the f-root to the parent of the extracted subtree correctly determines whether or not a point is in the region described by the extracted subtree. We have only to show that, in removing an f-node from this path, which is not a least upper bound or greatest lower bound of the remaining attribute values on the path, this search property remains invariant.

Suppose then, without loss of generality, that there is a f-node A that contains as comparison value "a," which is a least upper bound for one of the attributes. Suppose that further up the path in an f-node B there is an upper bound "b" on the same attribute that is larger than "a."

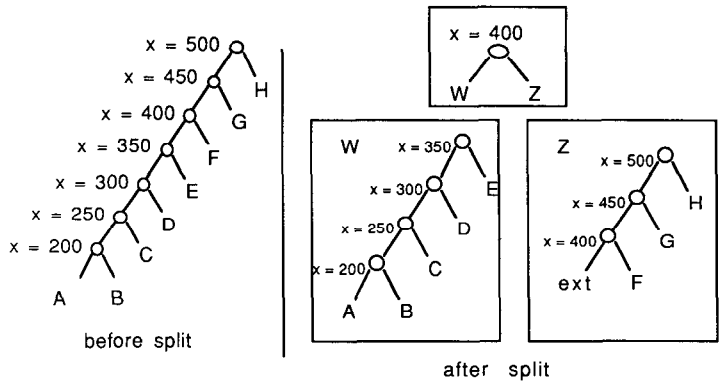


Fig. 14. A skewed k-d tree is flattened by the splitting process.

In a search for records that do not lie in the extracted subtree, those whose values are greater than "b" in the given attribute will be eliminated at B. If we remove this f-node B, those same records will be eliminated lower down the path at f-node A. So searches for records whose values in this attribute are not in the extracted region will still be correct. If a record *is* in the region, it will not be eliminated at B or at A. Removing B will make no difference.

Thus the condensed path, if posted to the parent by including it in the i-tree, will provide a correct determination of whether or not a record is in a region described by an extracted subtree. The divergence node is not needed for this determination. The divergence node is only needed to integrate a new condensed path with already posted nodes. □

We emphasize that no matter how "skewed" the f-tree is, only at most $2k + 1$ *f-nodes are posted.* Skewed f-trees are "flattened" by the splitting process. This implies that searches in an underlying large skewed k-d tree involve only a small subset of the k-d tree nodes that would be followed if the underlying tree were not organized as an hB-tree. A split of a highly skewed f-tree is illustrated in Figure 14.

Thus, using the hB-tree can be regarded as a form of "rotation" for the k-d tree, somewhat like the way that AVL trees rotate binary search trees. There is a strictly bounded amount of repetition, which gains an overall balancing of the tree. In many cases the hB-tree can be more efficient than a k-d tree index, even if both the hB-tree and the k-d tree index are memory-resident. The height of the tree and the resulting CPU search time will be shortened by the hB-tree organization.

6.1 Algorithm for Posting Index Terms

Our algorithm makes use of two flags in each internal f-node. One flag indicates whether or not it has already been posted to the I-NODE(s), and the other indicates whether or not it is an ancestor of a previously extracted tree. Posted f-nodes are always ancestors of previously extracted trees, but the reverse is not

true: ancestors of previously extracted trees are not always posted. These flags are always *set* for internal nodes of f-trees in *data* nodes, since these f-nodes always have these properties.

In addition, in each hB-tree node, we keep boundaries of the region it identifies. We call this boundary information the *region identifier*. We need this to find the i-root. We present this algorithm in detail, so that it can be easily coded. The main idea is simple: integrate the condensed path into the i-tree.

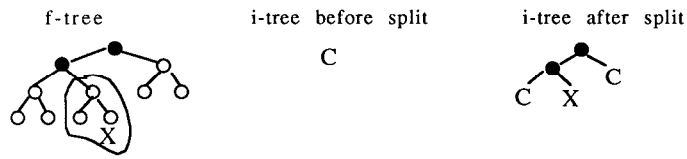
Index Term Posting Algorithm

1. Using the region identifier of the F-NODE, we do a range search to find the i-root. We begin at the root of the hB-tree, which is in memory. Usually, most of the other ancestors of the F-NODE will also be in memory. When we reach a k-d tree leaf pointing to the F-NODE, or when we reach a k-d tree node where we must follow both branches, we have arrived at the i-root.
2. We are now ready to post the condensed path. We descend both the i-tree and the f-tree, beginning at the i-root and f-root, respectively. We follow the path in both trees that will lead to the x-root. Nodes in each of these trees are processed one node at a time. A cursor is associated with each tree, which maintains a record of what nodes in these trees have been traversed thus far. The i-tree cursor is set to the parent of the i-root. The f-tree cursor is similarly set "above" the f-root.

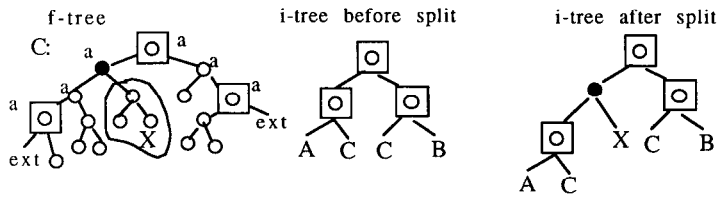
    a. When a previously posted f-node is encountered in the full path in the F-NODE, it corresponds to an i-node. The f-node and the i-node share the same attribute and attribute value. Advance the cursors in both the i-tree and the f-tree past these nodes.
    b. When an f-node on the full path is encountered that represents a redundant bound of the extracted region, advance the F-NODE cursor past it. The cursor for the i-tree path is not advanced.
    c. When an f-node on the condensed path is encountered in traversing the full path, post it to the i-tree. This is done by creating an i-node with the same attribute and attribute value as the f-node. This i-node is then inserted so that it becomes a descendent of the i-node that is currently referenced by the i-tree cursor. The previous i-tree descendent becomes the descendent of the newly posted i-node. The i-tree cursor is advanced to the newly posted i-node.

3. Let P be the parent of the x-root. P is in the condensed path if it has not already been posted, and hence is always represented in the I-NODE(s) after step 2 has been executed. Except possibly for P in the i-tree, references to hB-tree nodes from the posted condensed path are references to the F-NODE, and will become references to the N-NODE (because the disk address of the F-NODE will be inherited by the N-NODE).
4. The x-tree nodes present in the i-tree must be updated.

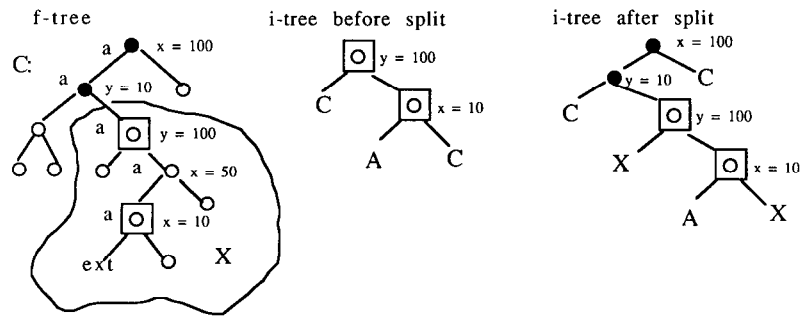
    a. If the x-root is not an ancestor of a previously extracted tree, there are no x-tree nodes in the i-tree. One of the leaf references of P is then to the X-NODE. This is the only case where a leaf reference in the condensed path is not a reference to the N-NODE. If P has previously been posted, its reference to the N-NODE must be changed to refer to the X-NODE.
    b. If the x-root is an ancestor of previously extracted trees, leaf references in the x-tree nodes posted to the i-tree that currently reference the N-NODE are replaced by references to the X-NODE.



(a) The f-root is not an ancestor of a previously extracted tree.



(b) The f-tree had previously extracted trees, but not in the x-tree.



(c) Previously extracted trees in the x-tree. Here we also show the bounds of x and y variables in the nodes.

Fig. 15. Some examples of posting index split information. Blackened f-nodes refer to the condensed path. Previously posted f-nodes are enclosed in a box. Ancestors of previously extracted trees are marked with a lowercase "a".

5. In the f-tree, all posted nodes are marked as posted. All f-nodes on the full path from the f-root to the x-root are marked as ancestor nodes (they may be already so marked). The f-tree is then transformed into the n-tree, as in Section 5. This prepares the n-tree for the next application of this algorithm.

To give an idea of how this works, we illustrate several cases in Figure 15.

6.2 Posting Index Terms for Data Nodes

Posting information for splitting data nodes is an important special case of the general case described above. We have already seen how this works in Section 5.

When we find a subtree of the f-tree representing between $\frac{1}{3}$ and $\frac{2}{3}$ of the contents in a data node, the condensed path is trivial, since all the nodes in the f-tree have duplicates in the i-tree. In this case, we need only concern ourselves with step 4b, where we change the references in the x-tree part of the i-tree that referred to the N-NODE to references to the X-NODE.

When, on the other hand, we find ourselves at an f-leaf representing more than $\frac{2}{3}$ of the contents, the condensed path is nontrivial and consists of the new k-d tree nodes which are added to the f-tree. Note that we never need more than k additional k-d tree nodes to identify a (data) X-NODE, and hence post at most k i-nodes. Also note that it is this case which guarantees that the f-tree is duplicated in the i-tree.

Since this is the only case where a data node split posts new i-nodes, at most one split can be induced at the parent-of-leaf level in the hB-tree (in the one hB-tree node containing the copy of the f-leaf representing $\frac{2}{3}$ of the contents.) That is, data node splits can cause *at most one split* at the next level.

Although it is possible that posting index terms above the parent-of-leaf level causes two or more splits on the same hB-tree level, it is not likely. For this to happen, several nodes from the condensed path must be posted to different hB-tree index nodes, each of which is full and must be split. In addition, as in the B^+-tree, splits at higher levels are increasingly improbable.

7. SUMMARY AND DISCUSSION

The hB-tree is space and time efficient for any data distribution. We have shown that, even in the worst cases, the fan-out and space utilization remain competitive with the best single-attribute methods. An hB-tree works well for partial match, exact match, and range searches. We have presented detailed, straightforward algorithms.

The hB-tree has been discussed in the context of multiattribute searching. It is worth pointing out that its performance as a single-attribute search method is competitive with B^+-tree performance. While hB-trees have more capability than is strictly necessary for the single-attribute case, nonetheless, its use imposes minimal extra overhead. Data nodes, in the one-dimensional case, can always be split evenly, that is, in a 1:1 ratio, and the resulting regions are always one-dimensional "bricks." Index nodes are organized as degenerate k-d trees (i.e., as binary search trees). These trees can be searched very efficiently, typically as efficiently as using a binary search. The biggest penalty is that the k-d tree consumes more space than the usual vector-oriented binary search organization.

We have not described how to organize data records in data nodes of the hB-tree. Recall that data nodes have local trees, organized as k-d trees, which lend some structure to data nodes. But the data records are not part of the local tree. In fact, our data node corner-splitting requires that we split the records without regard to tree structure for the data. There are many possible organizations, for instance, an indexed heap. We do not expect performance to depend heavily on the specific organization. The number of records is in the tens, not the hundreds. Searches and updates involving this number of entries should not be a large part of system cost.

Splitting index nodes may produce holey bricks, but the hB-tree remains balanced, just as the B^+-tree does, and for the same reason (i.e., the hB-tree increases in height only when the root splits). It is worth repeating one last time that hB-trees differ from B^+-trees only in the organization of index terms into k-d trees and in the splitting of data (or index information) between nodes during

the file growth process. These are what might be called local differences or differences in the small. Globally, hB-trees behave as B^+-trees.

The full details of a deletion algorithm remain for future work. Record deletions that do not yield underutilized nodes do not change the hB-tree structure, but are done locally within a node. If a data node is underutilized, we reverse the process of node splitting, combining siblings which might have previously been paired as N-NODE and X-NODE. We then simplify the i-tree, eliminating i-nodes which only serve to distinguish the two siblings. This is complicated, and results in a number of different cases that need to be treated. However, it is clear in concept what is required. Further, because hB-trees deal with holey bricks, it is always possible to find another node in which to store the data from a node being deleted. This is in contrast to the grid file and the K-D-B tree, which require that nodes represent full (rectangular) bricks. At times it is impossible to find a node to pair with the node being deleted such that the combined volumes form a brick.

Since node splitting occurs, from the global hB-tree view, in exactly the same manner as it occurs in B^+-trees, it is possible to link together all nodes at the same level of the tree, into what has been called a sequence set. This sequence set is simply a linked list of the nodes. With B^+-trees, the single-attribute keys of the nodes are ordered. Thus, the keys in one node all precede or all follow all the keys of another node. For hB-trees, we do not have this ordering relation. With these links, however, and with a few minor adjustments to the search algorithm, the Lehman–Yao concurrency method [6, 16, 17] could be applied to the hB-tree.

Most important, the hB-tree is one of few multiattribute search structures that provides an analysis of space utilization. It shares with the B^+-tree a simple node-splitting process, which occasionally proceeds up the tree to the ancestors of the node being split and never cascades downwards. The hB-tree does not require knowing the distribution of the data ahead of time; it, like the B^+-tree, adjusts gracefully to any pattern of incoming data.

APPENDIX 1. Proof for Data Node Split

In this appendix we show that, with at most k attribute values, we can guarantee at least a 1:2 split in the data points. Thus, we can guarantee both data node utilization of at least $\frac{1}{3}$ and index terms for data nodes that do not involve more than k attributes.

Definition. A *d-dimensional closed upper corner* in a finite subset of k-dimensional space can be defined formally as the set of points $(x_1, x_2, \ldots, x_k)$ such that

$$x_1 \geq m_1,\ x_2 \geq m_2,\ \ldots,\ x_d \geq m_d$$

where $d \leq k$ and where m_i is the median value for the ith coordinate for the points in the space. A *d-dimensional closed lower corner* is defined similarly.

A *d-dimensional closed corner* is either a d-dimensional closed *upper* corner, or else it is a d-dimensional closed *lower* corner.

LEMMA. *If no median hyperplane splits the space in a ratio less than or equal to* 2:1, *and if some d-dimensional closed upper corner, D contains more than* $\frac{2}{3}$ *of the points, then any* $(d + 1)$*-dimensional closed upper corner contained in D has more than* $\frac{1}{3}$ *of the points.*

PROOF. Suppose we have such a d-dimensional closed upper corner $(x_1, x_2, \ldots, x_d)$ of points D such that

$$x_1 \geq m_1, x_2 \geq m_2, \ldots, x_d \geq m_d,$$

which includes more than $\frac{2}{3}$ of the total points in the space.

Then the complement of D is the set of points C such that $x_1 < m_1$ OR $x_2 < m_2 \ldots$ OR $x_d < m_d$. Since D contains more than $\frac{2}{3}$ of the points, C contains less than $\frac{1}{3}$ of the points.

The set of points X such that $x_{d+1} < m_{d+1}$ contains less than $\frac{1}{3}$ of the points, since no median hyperplane splits the space in a 2:1 ratio. (The labeling "d + 1" of the next coordinate is arbitrary; any coordinate not already used to define D could be used here.)

Therefore, the union of C and X contains less than $\frac{2}{3}$ of the points, and its complement, the set of points $(x_1, x_2, \ldots, x_d, x_{d+1})$ such that

$$x_1 \geq m_1, x_2 \geq m_2, \ldots, x_{d+1} \geq m_{d+1}$$

must therefore contain more than $\frac{1}{3}$ of the total points. □

THEOREM. *Some d-dimensional closed corner contains between* $\frac{1}{3}$ *and* $\frac{2}{3}$ *of the points.*

PROOF. When d = 1, we are talking about splitting the space with hyperplanes. If some median hyperplane works, we are done. If not, no median hyperplane splits the space in a 2:1 ratio, and all k 1-dimensional closed corners must contain more than $\frac{2}{3}$ of the points. (They contain more than half, since the split point is the median value; if they also contain less than $\frac{2}{3}$, this hyperplane splits in a 2:1 ratio).

We use mathematical induction on d to show that if the theorem is not true, then there is a d-dimensional closed upper corner with more than $\frac{2}{3}$ of the points and a d-dimensional closed lower corner (which may be defined on the same coordinates) with more than $\frac{2}{3}$ of the points. When d = k, we arrive at a contradiction.

We have shown that if the theorem is not true, then the induction hypothesis holds for d = 1.

Now suppose the induction hypothesis is true for d. That is, there is a closed upper corner D, consisting of points $(x_1, x_2, \ldots, x_k)$ such that

$$x_1 \geq m_1, x_2 \geq m_2, \ldots, x_d \geq m_d,$$

which contains more than $\frac{2}{3}$ of the total points. The closed lower corner, consisting of points $(x_1, x_2, \ldots, x_k)$ such that

$$x_1 \leq m_1, x_2 \leq m_2, \ldots, x_d \leq m_d,$$

also contains more than $\frac{2}{3}$ of the points.

Then, by the lemma, the set of points $(x_1, x_2, \ldots, x_k)$ such that

$$x_1 \geq m_1,\ x_2 \geq m_2,\ \ldots,\ x_{d+1} \geq m_{d+1}$$

contains more than $\frac{1}{3}$ of the total points. If it also contains less than $\frac{2}{3}$ of the points, then the theorem is true. Thus, it must contain more than $\frac{2}{3}$ of the points if the theorem is false. By symmetry, this is also true for closed lower corners. This finishes the induction step.

So, by induction on d, we have shown that the closed upper k-dimensional corner U and the closed lower k-dimensional corner L both contain more than $\frac{2}{3}$ of the points. But then the complement K of U is the set of points where $x_1 < m_1$ OR $x_2 < m_2 \ldots$ OR $x_k < m_k$.

We thus know that K has less than $\frac{1}{3}$ of the points. Thus, the points which are in L but not in K are more than $\frac{1}{3}$ of the points. But this is the single point $(m_1, m_2, \ldots, m_k)$. □

The theorem establishes that any collection of points can have a subregion extracted, which constitutes between $\frac{1}{3}$ and $\frac{2}{3}$ of the points. To establish that it is possible to "split" a holey brick by extracting a *full* brick from it, we need a more specialized result. The following corollary accomplishes this.

COROLLARY. *Suppose a finite set S of points in k-dimensional space contains over $\frac{2}{3}$ of the points of a larger finite set T. Then we can augment S with a new set of points Q such that $|S \cup Q| = |T|$ (The cardinality of S union Q is the same as the cardinality of T), and we can find a closed corner K of $S \cup Q$ such that between $\frac{1}{3}$ and $\frac{2}{3}$ of the points of T lie in K. (These points are also in S).*

PROOF. Suppose that for the set of all points $(x_1, \ldots, x_k)$ in S we have upper and lower bounds

$$l_i \leq x_i \leq u_i$$

for each $1 \leq i \leq k$. These upper and lower bounds define a rectangular convex hull for S. That is, they define the boundaries of the smallest rectangular region containing all the points of S.

Let the points of Q be defined as follows. Let $|Q| = |T - S|$. Half of the points of Q are to lie in the quadrant where

$$\begin{aligned} x_1 &< l_1 \\ x_2 &< u_2 \\ x_3 &> u_3 \\ &\cdots \\ x_k &> u_k \end{aligned}$$

That is, except for the first coordinate, all the coordinates are larger than the coordinates of any point in S. The first coordinate is smaller than the first coordinate of any point in S. Symmetrically, the other half of the points of Q lie in the quadrant

$$\begin{aligned} x_1 &> u_1 \\ x_2 &< l_2 \\ x_3 &< l_3 \\ &\cdots \\ x_k &< l_k \end{aligned}$$

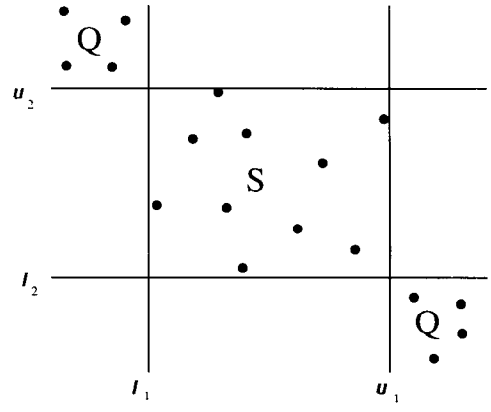


Fig. 16. A projection of data from Q onto two dimensions is shown as it is placed in the mixed corners of imaginary region I which includes S. This permits the splitting of the set T that is comprised of both S and Q in a ratio of 2:1 or better while extracting only a subregion of S. The l_i are lower bounds for coordinates of S while the u_i are upper bounds.

A projection of S and Q onto the two-dimensional space (x_1, x_2) is given in Figure 16.

By the theorem, some d-dimensional closed corner K of S ∪ Q contains less than $\frac{2}{3}$ and more than $\frac{1}{3}$ of the points of S ∪ Q. If $d \geq 2$, then K does not contain any points of Q. Thus, all of the points in K are in S, and hence also in T.

If d = 1, then K contains all the points of S ∪ Q whose ith coordinate is greater than or equal to the median. Such a "closed corner" has more than half of the points of S ∪ Q, and contains half of the points of Q. The other half of the points of Q are on the other side of the hyperplane defined by the median.

But the number of points in half of the points of Q is less than $\frac{1}{6}$ of the number of points of T. The rest of the points are in S (and hence in T). So at least $\frac{1}{3}$ of the points of T are in K. □

We have shown above that data nodes can always be split so that the division of data between the two resulting data nodes is better(less) than 2:1. The method we have employed makes use of closed corners of k-space (i.e., corners that contain their boundaries). In the one-dimensional case, we can clearly divide the data in close to a 1:1 ratio. We conjecture that it should be possible to split data more evenly in higher dimensional spaces by making use of semiclosed corners of the space being divided. These are corners which contain some, but not necessarily all, of the boundaries (i.e., they are closed in some dimensions and open in others). The rationale for this conjecture is that we should be able to pick and choose boundaries so as to balance the split ratio.

APPENDIX 2. Index Node Splitting

In this appendix, we show how an index node can be split into two index nodes with at worst one third of the information in one node and two thirds in the other. We can always find a subtree to extract where the number of (internal) nodes in the extracted subtree is between one third and two thirds of the total number of (internal) nodes. (The *internal* nodes are the nodes of the k-d tree whose information is actually stored in the index. *Leaf* nodes are addresses of hB-tree data nodes or index nodes on a lower level or are external markers.) We thus guarantee worst-case node utilization of one third. Let us prove this formally.

Definition. Let n be the total number of internal nodes in the k-d tree to be split. Suppose n is greater than 1. Let lb be floor($n/3$). Let ub be ceiling($2n/3$). Say the node N *satisfies the bound condition* if

$$lb \leq x \leq ub$$

where x is the number of internal nodes in the subtree whose root is N.

CLAIM. *We can always find a node satisfying the bound condition.*

PROOF. Associate with each internal node N in the tree the number count(N), the number of internal nodes in the subtree whose root is N. We begin at the root of the k-d tree. If either child satisfies the bound condition, we are done. Suppose neither child satisfies the bound condition. First we show that this implies that count(C) will be greater than ub for one of the children, C.

Let c_1 = count(left child) and let c_2 = count(right child). Then

$$c_1 + c_2 + 1 = n.$$

If both $c_1 \leq ub$ and $c_2 \leq ub$, then since neither child satisfies the bound condition, we have $c_1 < lb$ and $c_2 < lb$.

Thus

$$n = c_1 + c_2 + 1 < lb + lb + 1 \leq n/3 + n/3 + 1.$$

This implies $n < 3$.

But a tree with only two nodes has a child of the root satisfying the bound condition. We are not considering trees with only one node in our claim.

This shows that if there is no child satisfying the bound condition, one of the children will be the root of a tree with more than ub nodes.

Continue descending the tree, choosing a child C of the current node such that count(C) $> ub$. After a finite number of steps, less than the height of the tree, neither of the two children will be the roots of trees with more than ub nodes. Because the number of nodes of a tree is an integer, and this number strictly decreases as we descend the tree.

Let p be count(parent) when we stop our descent. So $p > ub$. By our assumption, c_1 = count(child 1) and c_2 = count(child 2) at this point are both less than or equal to ub. If either child satisfies the bound condition at this point, we are done. Assume not. Then

$$c_1 < lb \quad \text{and} \quad c_2 < lb,$$

as before. Also,

$$c_1 + c_2 + 1 = p > ub.$$

Thus

$$ub < c_1 + c_2 + 1 < lb + lb + 1.$$

This implies

$$\text{ceiling}(2n/3) < \text{floor}(n/3) + \text{floor}(n/3) + 1.$$

If n is not divisible by 3, this becomes

$$2k + 1 < 2k + 1,$$

where $n = 3k + 1$, or

$$2k + 2 < 2k + 1,$$

where $n = 3k + 2$. If n is divisible by 3, then $c_i < lb$ implies $c_i \leq n/3 - 1$. Thus

$$ub < c_1 + c_2 + 1 \leq n/3 - 1 + n/3 - 1 + 1.$$

This implies $2n/3 < 2n/3 - 1$. We thus have a contradiction to our assumption that neither child satisfied the bound condition. □

REFERENCES

1. BAYER, R., AND MCCREIGHT, E. M. Organization and maintenance of large ordered indices. *Acta Inf. 1*, 3 (1972), 173–189.
2. BENTLEY, J. L. Multidimensional binary search trees used for associative searching. *Commun. ACM 18*, 9 (Sept. 1975), 509–517.
3. BENTLEY, J. L. Multidimensional binary search trees in database applications. *IEEE Trans. Softw. Eng. SE-5*, 4 (July, 1979), 333–340.
4. COMER, D. The ubiquitous B-tree. *ACM Comput. Surv. 11*, 2 (June 1979), 121–138.
5. FREESTON, M. The BANG file: A new kind of grid file. In *Proceedings of the ACM SIGMOD Conference on Management of Data* (San Francisco, May 1987). ACM, New York, 1987, 260–269.
6. LEHMAN, P., AND YAO, S. B. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst. 5*, 3 (Sept. 1980), 339–353.
7. LITWIN, W., AND LOMET, D. A new method for fast data searches with keys. *IEEE Softw. 4*, 2 (Mar. 1987), 16–24.
8. LOMET, D. Digital B-trees. In *Proceedings of the 7th Conference on Very Large Data Bases* (Cannes, Sept. 1981), 333–343.
9. LOMET, D. DL*-trees. IBM Res. Rep. RC10860, IBM Thomas J. Watson Research Center, Yorktown Heights, N.Y., Nov. 1984.
10. LOMET, D. Partial expansions for file organizations with an index. *ACM Trans. Database Syst. 12*, 1 (Mar. 1987), 65–84.
11. LOMET, D. A simple bounded disorder file organization with good performance. *ACM Trans. Database Syst. 13*, 3 (Dec. 1988), 525–551.
12. NIEVERGELT, J., HINTERBERGER, H., AND SEVCIK, K. C. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. Database Syst. 9*, 1 (Mar. 1984), 38–71.
13. OHSAWA, Y., AND SAKAUCHI, M. The BD-tree—A new n-dimensional data structure with highly efficient dynamic characteristics. In *Proceedings IFIP Congress* (Paris, 1983). North Holland, 1983, 539–544.
14. ORENSTEIN, J. A., AND MERRETT, T. A class of data structures for associative searching. In *Proceedings of ACM SIGACT-SIGMOD Principles of Database Systems Conference* (Waterloo, 1984). ACM, New York, 1984, 181–190.
15. ROBINSON, J. T. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings ACM SIGMOD Conference on Management of Data* (Boston, June 1984). ACM, New York, 1984, 10–18.
16. SAGIV, Y. Concurrent operations on B-trees with overtaking. In *Proceedings ACM SIGACT-SIGMOD Symposium on Principles of Database Systems* (Portland, Ore., 1985). ACM, New York, 1985, 28–37.
17. SALZBERG, B. Restructuring the Lehman–Yao tree. Tech. Rep. BS-85-21, Northeastern Univ., Boston, 1985.
18. SALZBERG, B. Using the buddy system on the K-D-B-tree. Tech. Rep. BS-85-21, Northeastern Univ., Boston, 1985.
19. SALZBERG, B. Grid file concurrency. *Inf. Syst. 11*, 3 (1986), 235–244.

Received July 1987; revised July 1988; revised and accepted October 1989