Using the Holey Brick Tree for Spatial Data in General Purpose DBMSs*

Georgios Evangelidis Betty Salzberg

College of Computer Science
Northeastern University
Boston, MA 02115-5096

1 Introduction

There is leverage to be gained by bringing spatial data within the purview of general purpose database systems. A spatial access method embedded in a general purpose DBMS would have several advantages [Lom91]:

1. Spatial data could be integrated with other data. Spatial objects are likely to have attributes other than the location of their bounding boxes. These other attributes can be queried using traditional methods.
2. Multiple users (some making updates) would be supported by concurrency algorithms already in place.
3. In case of a system failure, the restart process would be able to recover the database to a consistent state.
4. Robust system utilities for loading data, analyzing performance, producing reports and so forth could be applied.

To this end, we have been modifying the hB-tree (or holey Brick tree) [LS90], an efficient multi-attribute search structure, for use with the concurrency and recovery systems available in general purpose DBMSs. This will make it more likely that vendors of DBMSs will be able to provide support for spatial data.

In section 2, we review the hB-tree. In section 3, we show why structures which cluster points in space as the hB-tree does will also be efficient for clustering k-dimensional spatial objects when their bounding box coordinates are entered as points in $2k$-dimensional space. We give here performance results which demonstrate that the hB-tree is especially well suited for such spatial data, since the space used for the index is not sensitive to the number of dimensions. In section 4, we indicate how the concurrency and recovery algorithm of [LS92] can be applied to a modified hB-tree.

2 The hB-tree

The hB-tree is a multi-attribute index structure. Its inter node search and growth processes are precisely analogous to the corresponding processes in B-trees. Its nodes represent bricks (i.e., multi-dimensional rectangles), or "holey" bricks, that is, bricks from which smaller bricks have been removed.

It consists of **index** and **data** nodes. Index nodes are responsible for multi-dimensional subspaces. They contain a **kd-tree** [Ben79] which is used to organize information about lower levels of the hB-tree and extracted

*This work was partially supported by NSF grant IRI-91-02821.

index-level regions. Data nodes contain the actual data, which are points in multi-dimensional space. Since the space a data node describes may be holey, data nodes may contain kd-trees, as well, that enable the data nodes to describe their own "inner" boundaries and the extracted data-level regions.

A kd-tree is a binary tree that is capable of describing a region (possibly holey) of the multi-dimensional space of attributes. The leaf nodes of a kd-tree may: (a) refer to other hB-tree nodes of the next lower level, (b) be markers, called **external markers**, indicating that a subtree is missing (this happens when a node is split and part of the attribute space it was describing is now described by another sibling node), or (c) refer to a collection of data records that are kept in a **record list** (in data nodes, only).

Figure 1 contains an example hB-tree. We have a two-dimensional attribute space. The root of the hB-tree is the index node I, that describes the whole space. Its kd-tree describes the next level of the hB-tree, which consists of two data nodes. All records with attributes $x \in (x_0, +\infty)$ and $y \in (y_0, +\infty)$ are located in data node B, and the rest of the records are located in data node A.

A's kd-tree shows that A, which initially was responsible for the whole attribute space, was split and part of its records moved to node B (that is what the external marker denotes).

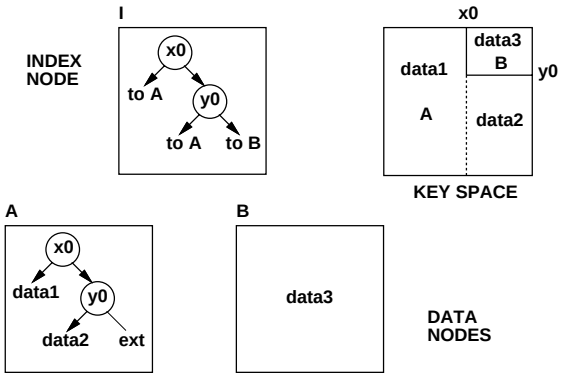

Figure 1: An example hB-tree.

2.1 Searching

Searching for point data using the hB-tree 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, we can decide on which of the two children of the kd-tree node we should visit next.

This process will lead us to lower levels of the hB-tree, and eventually to a data node. That data node contains the part of the attribute space where the search point belongs. Finally, the points of the node are searched with the help of the kd-tree of the node.

2.2 Node Splitting and Index Term Posting

The idea behind node splitting is the same as in B-trees. When a node becomes full it has to be split. Its kd-tree is used to find a subtree whose size is between one and two thirds of the contents of the node. A new hB-tree node is allocated and the extracted contents are moved to it. An external marker is included in the original node indicating that part of its contents have been extracted.

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.

3 Using the hB-tree for Spatial Objects

The hB-tree can be used for spatial objects as can any other multiattribute point index. The spatial objects are represented in the index by the coordinates of their bounding boxes.

We show first that any multiattribute index (such as the hB-tree) which clusters data points which are nearby in space, will, using boundary coordinates, cluster together records of nearby spatial objects. Since the number

of coordinates used to describe a bounding box is twice the number of dimensions of the space, indexes whose size does not depend on the number of coordinates of the data points will be superior. We show that the hB-tree disk space utilization is insensitive to the dimension of the space.

3.1 Mapping Spatial Objects to Points

Say we are using $2k$ coordinates to describe a brick in k-dimensional space. If two points in the $2k$-dimensional space have similar values in all coordinates, (a) the objects are similar in size, (b) if the objects are small, they are close in space, and (c) if the objects are large, they will overlap. Thus, any method which clusters nearby points will group records of large objects together in pages with other overlapping large objects. It will also group small objects together with nearby small objects in pages.

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

To illustrate these points, we look at one-dimensional spatial objects, which are line segments with a begin value and an end value. We map these objects to points in two-dimensional space using the x-coordinate for the begin value of the line segment and the y-coordinate for the end value. Since the begin value is always less than or equal to the end value, all points will lie on or above the line $x = y$. Points representing small line segments will be near the diagonal line $x = y$ and points representing large line segments will be far from the line $x = y$ (figure 2).

Common spatial queries such as inclusion or intersection are represented by rectangular bricks in the transformed ($2k$-dimensional) space. For example, as illustrated in figure 2, all line segments containing the point "6" would be in the area where the begin value x is less than or equal to 6 and where the end value y is greater than or equal to 6.

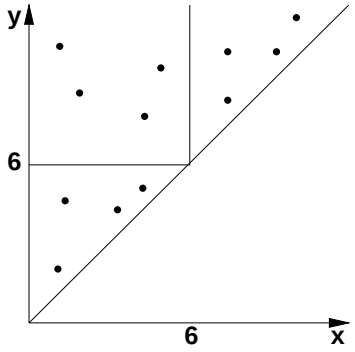

Figure 2: Mapping of line segments to points. Points close to the line $x = y$ will represent small line segments.

3.2 Performance of the hB-tree in High Dimensions

The main objection to using multiattribute point-based methods for spatial objects is that the number of dimensions needed to represent the objects doubles, making the index too large. But the hB-tree is especially 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-tree nodes) does not depend on the number of indexing attributes.

But, in addition to a kd-tree, every hB-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-tree. An increase on the number of dimensions does increase the space required to store a node's boundaries. The additional amount of space required as the dimensions increase becomes a non-factor for large enough page sizes. Figure 3 illustrates this fact. With a page size of 2K bytes and larger, there is almost no effect on the size of the hB-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. The grid file [NHS84] is even worse since it is a multidimensional array. Thus doubling the number

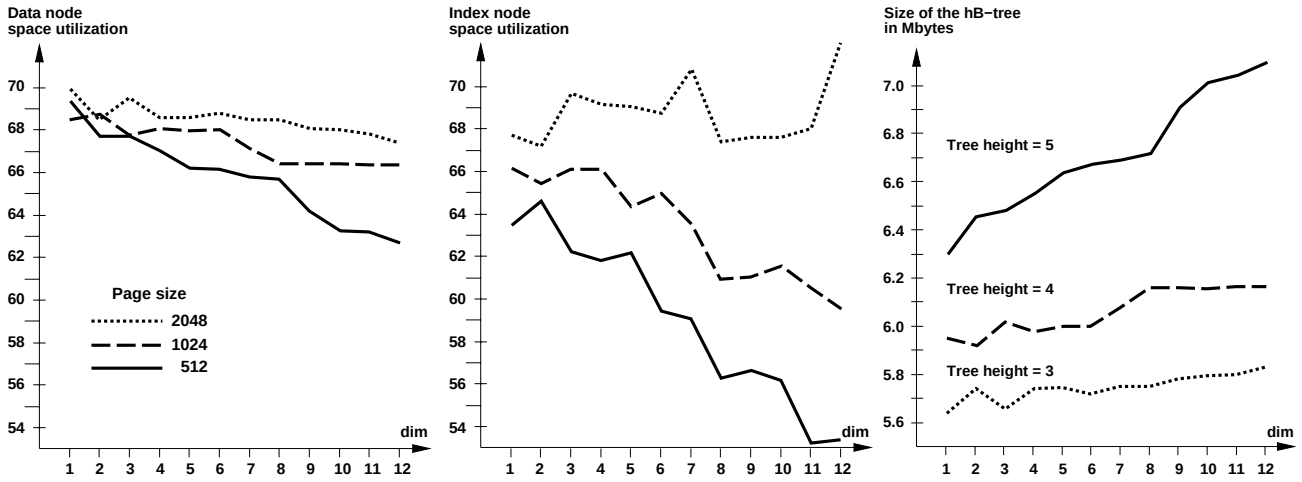

Figure 3: Index and data hB-tree node space utilization and size of the hB-tree in terms of height and Mbytes under different page sizes and dimensions. For page sizes greater than 2K bytes the hB-tree is insensitive to the number of dimensions. In all cases the same 150,000 24-byte records were inserted.

of dimensions squares the size of the index. In the worst case it is an $O(n^k)$ size index, where n is the number of points and k is the dimension of the space. Z-ordering, [OM84] on the other hand, like the hB-tree is insensitive to dimension.

4 Dealing with Concurrency and Recovery

New access methods which are advantageous for spatial data cannot be integrated into general purpose DBMSs unless they use the existing concurrency and recovery methods provided by the systems. The hB-tree has been properly modified so that the efficient algorithm for concurrency and recovery of [LS92] is applicable on it.

The most important modification is the replacement of external markers by actual pointers to the extracted nodes. The hB-tree becomes a subcase of the Π-tree [LS92], an abstract tree-structure which is a generalization of the B^{link}-tree [LY81]. We call the modified hB-tree an hB^{Π}-tree. Now, we can be lazy about posting index terms that describe a node split. In other words, two instances of the hB^{Π}-tree can be structurally different, because some index term postings have not been performed, but semantically equivalent.

4.1 Splitting and Posting

As with simple B-trees, when an insertion causes an hB^{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 hB^{Π}-tree the node splitting phase and the index posting phase are performed by separate atomic actions, as following:

Node splitting phase: An updater detects an hB^{Π}-tree node that is full and cannot accommodate the update. This node, called the **container node**, is split and part of its contents are moved to a newly created node, called the **extracted node**. The node splitting phase concludes by storing a side pointer in the container node that points to the extracted node (figure 4b).

Index posting phase: An index term that describes the space that was extracted from the container is posted to a parent of the container. Since an hB^{Π}-tree node may have many parents, the index posting phase may consist of several separate **index posting atomic actions**. An index posting atomic action always posts to a single parent (figure 4c).

An index posting atomic action can be scheduled after: (a) **a node splitting phase is over**, in which case it involves the parent of the container node that lies in the current search path, or (b) **a side pointer traversal**, which is an indication that either the posting action that was scheduled after the splitting was not performed for some reason, or the container node is a multiparent node and the current search path is through a parent other than the one involved in the node splitting phase.

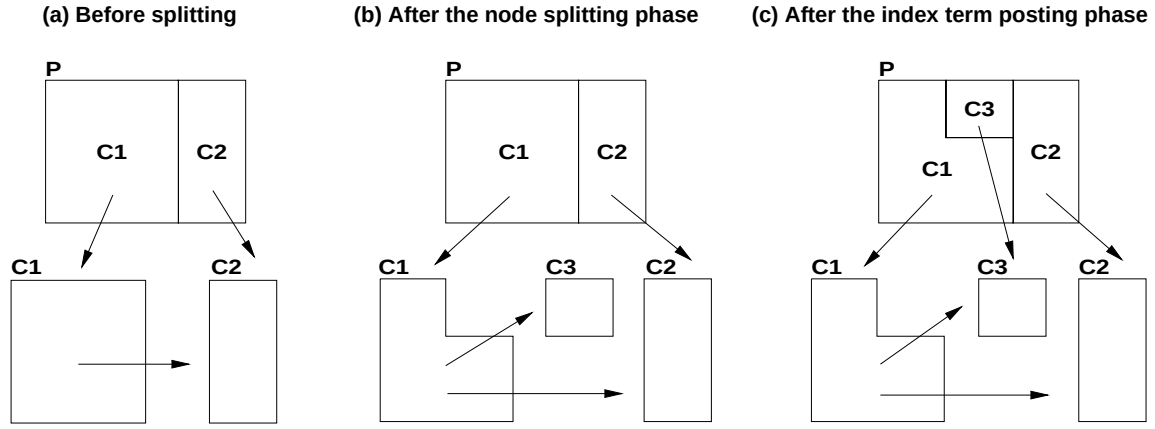

Figure 4: In the node splitting phase, node C1 is split, part of its contents move to the new node C3 and a side pointer to C3 is included in C1. In the index term posting phase an index term that describes the space of C3 is posted to its parent P.

In case an index posting atomic action can not take place for any reason, the hB$^{\Pi}$-tree is left in a consistent state. Searchers can always traverse or visit the extracted node by going through its container node and following the side pointer. System failure is not the only thing that prevents posting actions from completing. Depending on the algorithm used for performing the posting of index terms, a posting action may be dropped because the algorithm decides that it is not a good idea to perform the posting at that particular moment for performance reasons. Being too lazy in posting index terms may degrade the performance of the hB$^{\Pi}$-tree. An algorithm has to balance the tradeoff between an efficient and simple scheme for posting index terms and more expensive traversals of the hB$^{\Pi}$-tree due to a large number of side pointer traversals.

In the hB$^{\Pi}$-tree we have found that if we post more information (extra kd-tree nodes) than is actually needed, or if we restrict index node splits to certain places on their kd-tree, our algorithms become simpler. Such simplification could have negative consequences on performance since worst case guarantees of index term size and index node space utilization no longer hold as in the hB-tree. However, our preliminary experiments show that even with simple algorithms the performance is very good [ELS93].

5 Conclusions

The hB-tree is especially well-suited for spatial data. It clusters points using median attributes. It is insensitive to increases in dimension. Like any other point-clustering method, it can be expanded for use with bounding boxes of spatial objects. The resulting clustering will be effective for typical spatial queries such as nearest-neighbor and intersection searching. In addition, we are applying an efficient concurrency and recovery method to a modified hB-tree (called the hB$^{\Pi}$-tree) making possible its integration into a general purpose DBMS. This will enable users of spatial data to take advantage of the extensive and robust capabilities of these systems.

References

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

[ELS93] G. Evangelidis, D. Lomet, and B. Salzberg. Towards a concurrent and recoverable multiattribute indexing method. Technical Report in preparation, College of Computer Science, Northeastern University, 1993.

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

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

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

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

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

[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.