

Bkd-tree: A Dynamic Scalable kd-tree

Octavian Procopiuc^{1*}, Pankaj K. Agarwal^{1**},
Lars Arge^{1***}, and Jeffrey Scott Vitter^{2†}

¹ Department of Computer Science, Duke University
Durham, NC 27708, USA

² Department of Computer Science, Purdue University
West Lafayette, IN 47907, USA

Abstract. In this paper we propose a new data structure, called the Bkd-tree, for indexing large multi-dimensional point data sets. The Bkd-tree is an I/O-efficient dynamic data structure based on the kd-tree. We present the results of an extensive experimental study showing that unlike previous attempts on making external versions of the kd-tree dynamic, the Bkd-tree maintains its high space utilization and excellent query and update performance regardless of the number of updates performed on it.

1 Introduction

The problem of indexing multi-dimensional point data sets arises in many applications and has been extensively studied. Numerous structures have been developed, highlighting the difficulty of optimizing multiple interrelated requirements that such multi-dimensional indexes must satisfy. More precisely, an efficient index must have high space utilization and be able to process queries fast, and these two properties should be maintained under a significant load of updates. At the same time, updates must also be processed quickly, which means that the structure should change as little as possible during insertions and deletions. This makes it hard to maintain good space utilization and query performance over time. Consequently, the quality of most indexing structures deteriorates as a large number of updates are performed on them, and the problem of handling massive update loads while maintaining high space utilization and low query response time has been recognized as an important research problem [9].

* Supported by the National Science Foundation through research grant EIA-9870734 and by the Army Research Office through MURI grant DAAH04-96-1-0013. Part of this work was done while visiting BRICS, University of Aarhus, Denmark. Email: tavi@cs.duke.edu

** Supported by Army Research Office MURI grant DAAH04-96-1-0013, by a Sloan fellowship, by NSF grants ITR-333-1050, EIA-9870724 and CCR-9732787 and by a grant from the U.S.-Israeli Binational Science Foundation. Email: pankaj@cs.duke.edu

*** Supported in part by the National Science Foundation through ESS grant EIA-9870734, RI grant EIA-9972879 and CAREER grant CCR-9984099. Part of this work was done while visiting BRICS, University of Aarhus, Denmark. Email: large@cs.duke.edu

† Supported in part by the National Science Foundation through research grants CCR-9877133 and EIA-9870734 and by the Army Research Office through MURI grant DAAH04-96-1-0013. Part of this work was done while visiting BRICS, University of Aarhus, Denmark. Part of this work was done while at Duke University. Email: jsv@purdue.edu

In this paper we propose a new data structure, called the Bkd-tree, that maintains its high space utilization and excellent query and update performance regardless of the number of updates performed on it. The Bkd-tree is based on a well-known extensions of the kd-tree, called the K-D-B-tree [22], and on the so-called logarithmic method for making a static structure dynamic. As we show through extensive experimental studies, the Bkd-tree is able to achieve the almost 100% space utilization and the fast query processing of a static K-D-B-tree. However, unlike the K-D-B-tree, these properties are maintained over massive updates.

Previous Results. One of the most fundamental queries in spatial databases is the orthogonal *range query* or *window query*. In two dimensions a window query is an axis-aligned rectangle and the objective is to find all points in the database inside the rectangle. Numerous practically efficient multi-dimensional point indexing structures supporting window queries have been proposed, most of which can also answer a host of other query types. They include K-D-B-trees [22], hB-trees [18, 10], and R-trees [13, 6]. If N is the total number of points and B is the number of points that fit in a disk block, $\Omega(\sqrt{N/B} + K/B)$ is the theoretical lower bound on the number of I/Os needed by a linear space index to answer a window query [15]. Here K is the number of points in the query rectangle. In practice, the above indexing structures often answer queries in much fewer I/Os. However, their query performance can seriously deteriorate after a large number of updates. Recently, a number of linear space structures with guaranteed worst-case efficient query and update performance have been developed (see e.g. [5, 15, 12]). The so-called cross-trees [12] and O-trees [15] answer window queries in the optimal number of I/Os and can be updated, theoretically, in $O(\log_B N)$ I/Os, but they are of limited practical interest because a theoretical analysis shows that their average query performance is close to the worst-case performance. See e.g. [11, 3] for more complete surveys of multi-dimensional indexing structures. While some of the above indexing structures are specifically designed for external memory, many of them are adaptations of structures designed for main memory. In this paper we only focus on external memory adaptations of the original main memory kd-tree proposed by Bentley [7] (see also [23]).

External-Memory Dynamic kd-trees. While static versions of the kd-tree have been shown to have excellent query performance in many practical situations, an efficient dynamic version has proven hard to develop. In the following, we give a brief overview of the internal memory kd-tree structure and then discuss the two most important previous approaches for obtaining external memory dynamic kd-trees. In two dimensions, the kd-tree consists of a height $\lceil \log_2 N \rceil$ binary tree representing a recursive decomposition of the plane by means of axis-orthogonal lines partitioning the point set into two equal subsets.³ On even levels the line is orthogonal to the x -axis, while on odd levels it is orthogonal to the y -axis. The data points themselves are stored in the leaves, which form a partition of the plane into disjoint rectangular regions containing one point each. In the worst case a window query on a kd-tree requires $O(\sqrt{N} + K)$ time [16], but average case analysis [24] and experiments have shown that in practice it often performs

³ For simplicity we only consider two-dimensional kd-trees in this paper. However, all our results work in d dimensions.

much better. One way of performing an insertion on a kd-tree is to first search down the tree for the leaf corresponding to the rectangle containing the point, and then split this leaf into two in order to accommodate the new point. While this insertion procedure runs efficiently in $O(\log_2 N)$ time, the kd-tree can grow increasingly unbalanced when many insertions are performed, resulting in deteriorating query performance. In fact, the resulting tree is no longer a kd-tree, since the lines in the internal nodes no longer partition the points into equal sized sets. Unfortunately, while many other tree structures can be rebalanced efficiently in time proportional to the root-to-leaf path, it can be shown that in order to rebalance a kd-tree after an insertion, we may need to reorganize large parts of the tree [23]. Thus it seems hard to efficiently support insertions while at the same time maintaining good query performance. These considerations show that the kd-tree is mainly a static data structure with very good window query performance.

One main issue in adapting the kd-tree to external memory is how to assign nodes to disk blocks in order to obtain good space utilization (use close to N/B disk blocks) and good I/O query performance. In the first external memory adaptation of the kd-tree, called the K-D-B-tree [22], the kd-tree is organized as a B^+ -tree. More precisely, a K-D-B-tree is a multi-way tree with all leaves on the same level. Each internal node v corresponds to a rectangular region and the children of v define a disjoint partition of that region obtained using a kd-tree partitioning scheme. The points are stored in the leaves of the tree, and each leaf and internal node is stored in one disk block. Like a kd-tree, a K-D-B tree can be bulk loaded such that it exhibits excellent space utilization (uses close to N/B blocks) and answers queries I/O-efficiently (worst case optimally in $O(\sqrt{N/B} + K/B)$ but often much better in practice). Unfortunately, it also exhibits the kd-tree insertion characteristics. To insert a point into a K-D-B-tree, a root-to-leaf path is followed in $\lceil \log_B(N/B) \rceil$ I/Os and after inserting the point in a leaf, the leaf and possibly other nodes on the path are split just like in a B^+ -tree. However, unlike the B^+ -tree but similar to the kd-tree, the split of an internal node v may result in the need for splits of several of the subtrees rooted at v 's children—refer to Figure 1. As a result, updates can be very inefficient and, maybe more importantly, the space utilization can decrease dramatically since the split process may generate many near empty leaves [22].

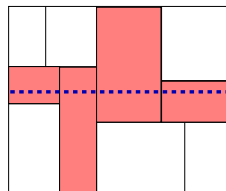


Fig. 1. Splitting a K-D-B-tree node. The outer rectangle corresponds to a node v being split. The darker regions correspond to children that need to be split recursively when v splits.

Following the K-D-B-tree, several other adaptations of the kd-tree to external memory have been proposed. An important breakthrough came with the result of Lomet and Salzberg [18]. Their structure, called the hB-tree (holey brick tree), significantly

improved the update performance over the K-D-B-tree. The better performance was obtained by only splitting nodes on one root-to-leaf path after an insertion. However, to be able to do so, the definition of internal nodes had to be changed so that they no longer corresponded to simple rectangles, but instead to rectangles from which smaller rectangles have been removed (holey bricks). The hB-tree update algorithm is theoretically efficient, although quite complicated. As we show in our experimental results, the hB-tree can still suffer from degenerating space utilization, although to a smaller extent than the K-D-B-tree (see also [10]). All other attempts at externalizing the kd-tree suffer from similar inefficiencies.

Our Results. In this paper, we present the first theoretically and practically efficient *dynamic* adaptation of the kd-tree to external memory. Our structure, which we call the Bkd-tree, maintains the high storage utilization and query efficiency of a static K-D-B-tree, while also supporting updates I/O-efficiently. We have conducted extensive experiments that show that the Bkd-tree outperforms previous approaches in terms of storage utilization and update time, while maintaining similar query performance.

The main ingredients used in the design of the Bkd-tree are an I/O-efficient K-D-B-tree bulk loading algorithm and the so-called logarithmic method for making a static data structure dynamic [8, 21]. Instead of maintaining one tree and dynamically rebalance it after an insertion, we maintain a set of $\log_2(N/M)$ static K-D-B-trees and perform updates by rebuilding a carefully chosen set of the structures at regular intervals (M is the capacity of the memory buffer, in number of points). This way we maintain the close to 100% space utilization of the static K-D-B-tree. The idea of maintaining multiple trees in order to speed up insertion time has also been used by O’Neill et al. [20] and Jagadish et al. [14]. Their structures are used for indexing points on a single attribute and their techniques cannot be extended to efficiently handle multi-dimensional points.

To answer a window query using the Bkd-tree, we have to query all the $\log_2(N/M)$ structures instead of just one, but theoretically we actually maintain the worst case optimal $O(\sqrt{N/B} + K/B)$ query bound. Using an optimal $O(\frac{N}{B} \log_{M/B} \frac{N}{B})$ I/O bulk loading algorithm, an insertion is performed in $O(\frac{1}{B} (\log_{M/B} \frac{N}{B}) (\log_2 \frac{N}{M}))$ I/Os amortized. This bound is much smaller than the familiar $O(\log_B N)$ B⁺-tree update bound for all practical purposes. One disadvantage of the periodical rebuilding is of course that the update bound varies from update to update (thus the amortized result). However, queries can still be answered while an update (rebuilding) is being performed, and (at least theoretically) the update bound can be made worst case using additional storage [21].

While our Bkd-tree has nice theoretical properties, the main contribution of this paper is a proof of its practical viability. We present the result of an extensive experimental study of the performance of the Bkd-tree compared to the K-D-B-tree using both real-life (TIGER) and artificially generated (uniform) data. In addition, we used a carefully chosen family of data sets to show that both the K-D-B-tree and the hB^{II}-tree (an improved version of the hB-tree, see [10]) can have poor space utilization (as low as 28% for the K-D-B-tree and 36% for the hB^{II}-tree), while the space utilization of the Bkd-tree is always above 99%. At the same time, an insertion in a Bkd-tree can be up to 100 times faster than an insertion on the K-D-B-tree, in the amortized sense.

The main practical question is of course how the use of $\log_2(N/M)$ structures affects the query performance. Even though the theoretical worst case efficiency is maintained, the querying of several structures instead of just one results in an increased number of random I/Os compared to the more localized I/Os in a single structure. Our experiments show that this makes no or relatively little difference, and thus that the dynamic Bkd-tree maintains the excellent query performance of the static K-D-B-tree.

Finally, we regard the demonstration of the practical efficiency of the logarithmic method as an important general contribution of this paper; while the main focus of the paper is on making the kd-tree dynamic, the logarithmic method is applicable to any index structure for which an efficient bulk loading algorithm is known. Thus our results suggest that in general we might be able to make practically efficient static index structures dynamically efficient using the method.

The rest of this paper is organized in three sections. The details of the Bkd-tree are given in Section 2. Then, in Section 3, we describe the hardware, software, and data sets used in our experimental study. The results of the experiments are reported and analyzed in Section 4.

2 Description of the Bkd-tree

As mentioned, the Bkd-tree consists of a set of balanced kd-trees. Each kd-tree is laid out (or *blocked*) on disk similarly to the way the K-D-B-tree is laid out. To store a given kd-tree on disk, we first modify the leaves to hold B points, instead of just one. In this way, points are packed in N/B blocks. To pack the internal nodes of the kd-tree, we execute the following algorithm. Let B_i be the number of nodes that fit in one block. Suppose first that N/B is an exact power of B_i , i.e., $N/B = B_i^p$, for some p , and that B_i is an exact power of 2. In this case the internal nodes can easily be stored in $O(N/(BB_i))$ blocks in a natural way. Starting from the kd-tree root v , we store together the nodes obtained by performing a breadth-first search traversal starting from v , until B_i nodes have been traversed. The rest of the tree is then blocked recursively. Using this procedure the number of blocks needed for all the internal nodes is $O(N/(BB_i))$, and the number of blocks touched by a root-leaf path—the path traversed during a point query—is $\log_{B_i}(N/B) + 1 = \Theta(\log_B(N/B))$. If N/B is not a power of B_i , we fill the block containing the kd-tree root with less than B_i nodes in order to be able to block the rest of the tree as above. If N/B is not a power of 2 the kd-tree is unbalanced and the above blocking algorithm can end up under-utilizing disk blocks. To alleviate this problem we modify the kd-tree splitting method and split at rank power of 2 elements, instead of at the median elements. More precisely, when constructing the two children of a node v from a set of p points, we assign $2^{\lceil \log_2 p \rceil}$ points to the left child, and the rest to the right child. This way, only the blocks containing the rightmost path—at most $\lceil \log_{B_i}(N/B) \rceil$ —can be under-full.

From now on, when referring to a kd-tree, we will mean a tree stored on disk as described above.

2.1 Bulk Loading kd-trees

Classically, a kd-tree is built top-down, as outlined in Figure 2 (left column). The first step is to sort the input on both coordinates. Then (in Step 2) we construct the nodes in a recursive manner, starting with the root. For a node v , we determine the splitting position by reading the median from one of the two sorted sets associated with v (when splitting orthogonal to the x -axis we use the file sorted on x , and when splitting orthogonal to the y -axis we use the file sorted on y). Finally we scan these sorted sets and distribute each of them into two sets and recursively build the children of v . Since the kd-tree on N points has height $\log_2(N/B)$ and each input point is read twice and written twice on every level, the algorithm performs $O((N/B) \log_2(N/B))$ I/Os, plus the cost of sorting, which is $O((N/B) \log_{M/B}(N/B))$ I/Os [2], for a total of $O((N/B) \log_2(N/B))$ I/Os.

Algorithm Bulk Load (binary)	Algorithm Bulk Load (grid)
(1) Create two sorted lists;	(1) Create two sorted lists;
(2) Build kd-tree top-down: Starting with the root node, do the following steps for each node, in a depth-first-search manner:	(2) Build $\log_2 t$ levels of the kd-tree:
(a) Find the partitioning line;	(a) Compute t grid lines orthogonal to the x axis and t grid lines orthogonal to the y axis;
(b) Distribute input into two sets, based on partitioning line;	(b) Create the grid matrix A containing the grid cell counts;
	(c) Create a subtree of height $\log_2 t$, using the counts in the grid matrix;
	(d) Distribute input into t sets, corresponding to the t leaves;
	(3) Build the bottom levels either in main memory or by recursing step (2).

Fig. 2. Two algorithms for bulk loading a kd-tree

An improved bulk loading method was proposed in [1]. Instead of constructing one level at a time, this algorithm constructs an entire $\Theta(\log_2(M/B))$ -height subtree of the kd-tree at a time. The major steps of the algorithm are outlined in Figure 2 (right column). As before, the first step is to sort the input on both coordinates. Then (in Step 2) we build the upper $\log_2 t$ levels of the kd-tree using just three passes over the input file, where $t = \Theta(\min\{M/B, \sqrt{M}\})$. We achieve this by first determining a $t \times t$ grid on the input points: t horizontal (vertical) grid lines are chosen (in Step 2a) so that each horizontal (vertical) strip contains N/t points—refer to Figure 3(a). Then (in Step 2b) the number of points in each grid cell is computed by simply scanning the input file. These counts are stored in a $t \times t$ grid matrix A , kept in internal memory (the size of the matrix, t^2 , is at most M). The upper subtree of height $\log_2 t$ is now computed (in step 2c) using a top-down approach. Assume the root node partitions the points using a vertical line. This split line can be determined by first computing (using the cell counts in matrix A) the vertical strip X_k containing the line. After that we can easily compute which block to read from the list sorted by x -coordinate in order to determine the point

defining the split. Next the grid matrix A is split into two new matrices, $A^<$ and $A^>$, storing the grid cell counts from the left and from the right of the split line, respectively. This can be done by scanning the contents of the vertical strip X_k . Figure 3(b) shows how a cell $C_{j,k}$ from the original grid is split into two cells, $C_{j,k}^<$ and $C_{j,k}^>$. The number of points in $C_{j,k}^<$ is stored in $A_{j,k}^<$, and the number of points in $C_{j,k}^>$ is stored in $A_{j,k}^>$, for each j , $1 \leq j \leq t$. Using matrices $A^<$ and $A^>$, the split corresponding to two children of v can be computed recursively. For each node we produce, the size of the matrix A in internal memory grows by t cells. Since $t \leq O(\sqrt{M})$, it still fits in memory after producing $\log_2 t$ levels, that is $2^{\log_2 t} = t$ nodes, of the tree. After producing this number of levels, the resulting subtree determines a partition of the space into t rectangles. At this point we distribute the input points into these rectangles by scanning the input and, for each point p , using the constructed subtree to find the rectangle containing p (Step 2d). If the main memory can hold $t + 1$ blocks—one for each rectangle in the partition, plus one for the input—the distribution can be done in $2N/B$ I/Os. This explains the choice of $t = \Theta(\min(M/B, \sqrt{M}))$. Finally, the bottom levels of the tree are constructed (in Step 3) by recursing Step 2 or, if the point set fits in internal memory, by loading it in memory and applying the binary bulk load algorithm to it.

Since Step 2 scans the input points two times, it follows that $\Theta(\log_2(M/B))$ levels of the kd-tree can be built using $O(N/B)$ I/Os. Thus the entire kd-tree is built in $O((N/B) \log_{M/B}(N/B))$ I/Os. This is a factor of $\Theta(\log_2(M/B))$ better than the binary bulk load algorithm. For most practical purposes, the logarithmic factor is at most 3, so the bulk loading complexity is effectively linear.

The algorithm presented in this section uses only the characteristics of the internal memory kd-tree, and not the specific disk layout. Consequently, other I/O-efficient data structures based on the kd-tree can be bulk loaded using this algorithm. In particular, the algorithm can be readily used to bulk load an hB^H -tree, which was mentioned as an open problem in [10].

2.2 Dynamic Updates

A Bkd-tree on N points in the plane consists of $\log_2(N/M)$ kd-trees. The i th kd-tree, T_i , is either empty or contains exactly $2^i M$ points. Thus, T_0 stores at most M points.

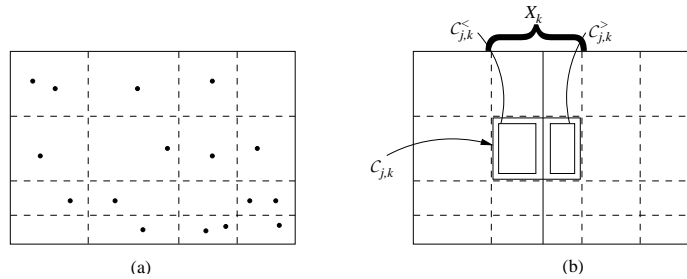


Fig. 3. Finding the median using grid cells. (a) Each strip contains N/t points. (b) Cells $C_{j,k}^<$ and $C_{j,k}^>$ are computed by splitting cell $C_{j,k}$.

In addition, a structure T_0^M containing at most M points is kept in internal memory. Figure 4 depicts the organization of the Bkd-tree. This organization is similar to the one used by the logarithmic method [8, 21].

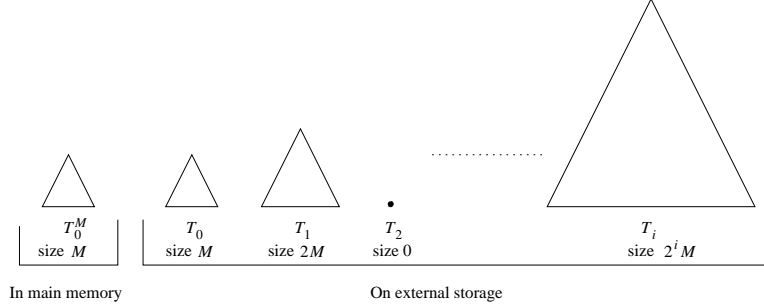


Fig. 4. The forest of trees that makes up the data structure. In this instance, T_2 is empty.

The algorithms for inserting and deleting a point are outlined in Figure 5. The simplest of the two is the deletion algorithm. We simply query each of the trees to find the tree T_i containing the point and delete it from T_i . Since there are at most $\log_2(N/M)$ trees, the number of I/Os performed by a deletion is $O(\log_B(N/B) \log_2(N/M))$.

The insertion algorithm is fundamentally different. Most insertions ($M - 1$ out of M consecutive ones) are performed directly on the in-memory structure T_0^M . Whenever T_0^M becomes full, we find the smallest k such that T_k is an empty kd-tree. Then we extract all points from T_0^M and T_i , $0 \leq i < k$, and bulk load the tree T_k from these points. Note that the number of points now stored in T_k is indeed $2^k M$ since T_0^M stores M points and each T_i , $1 \leq i < k$, stores exactly $2^i M$ points (T_k was the first empty kd-tree). Finally, we empty T_0^M and T_i , $0 \leq i < k$. In other words, points are inserted in the in-memory structure and gradually “pushed” towards larger kd-trees by periodic reorganizations of small kd-trees into one large kd-tree. The larger the kd-tree, the less frequently it needs to be reorganized.

To compute the amortized number of I/Os performed by one insertion, consider N consecutive insertions in an initially empty Bkd-tree. Whenever a new kd-tree T_k is constructed, it replaces all kd-trees T_j , $1 \leq j < k$, and the in-memory structure T_0^M . This operation takes $O((2^k M/B) \log_{M/B}(2^k M/B))$ I/Os (bulk loading T_k) and moves exactly $2^k M$ points into the larger kd-tree T_k . If we divide the construction of T_k between these points, each of them has to pay $O((1/B) \log_{M/B}(2^k M/B)) = O((1/B) \log_{M/B}(N/B))$ I/Os. Since points are only moving into larger kd-trees, and there are at most $\log_2(N/M)$ kd-trees, a point can be charged at most $\log_2(N/M)$ times. Thus the final amortized cost of an insertion is $O\left(\frac{\log_{M/B}(N/B) \log_2(N/M)}{B}\right)$ I/Os.

Algorithm Insert(p)	Algorithm Delete(p)
(1) Insert p into in-memory buffer T_0^M ;	(1) Query T_0^M with p ; if found, delete it and return;
(2) If T_0^M is not full, return; otherwise, find the first empty tree T_k and extract all points from T_0^M and T_i , $0 \leq i < k$ into a file F ;	(2) Query each non-empty tree in the forest (starting with T_0) with p ; if found, delete it and return;
(3) Bulk load T_k from the items in F ;	
(4) Empty T_0^M and T_i , $0 \leq i < k$.	

Fig. 5. The **Insert** and **Delete** algorithms for the Bkd-tree

2.3 Queries

To answer a window query on the Bkd-tree we simply have to query all $\log_2(N/M)$ kd-trees. The worst-case performance of a window query on one kd-tree storing N points is an optimal $O(\sqrt{N/B} + K/B)$ I/Os, where K is the number of points in the query window. Since the kd-trees that form the Bkd-tree are geometrically increasing in size, the worst-case performance of the Bkd-tree is also $O(\sqrt{N/B} + K/B)$ I/Os. However, since the average window query performance of a kd-tree is often much better than this worst-case performance [24], it is important to investigate how the use of several kd-trees influences the practical performance of the Bkd-tree compared to the kd-tree.

3 Experimental Platform

In this section we describe the setup for our experimental studies, providing detailed information on the software, hardware, and data sets that were used.

Software Platform. We implemented the Bkd-tree in C++ using TPIE. TPIE [4] is a templated library that provides support for implementing I/O-efficient algorithms and data structures. In our implementation we used a block size of 16KB for internal nodes (following the suggestions of Lomet [17] for the B-tree), resulting in a maximum fanout of 512. The leaves of a kd-tree, stored in 16KB blocks as well, contain a maximum of 1364 (key, pointer) elements. We implemented the Bkd-tree using the grid bulk loading algorithm during insertions and a linear array as the internal memory structure T_0^M (more sophisticated data structures can be implemented for better CPU performance). For comparison purposes, we also implemented the K-D-B-tree, following closely the details provided in the original paper [22] regarding the insertion algorithm. As mentioned in the Introduction, the K-D-B-tree is the point of departure for the hB-tree [18] and the hB^H-tree [10]. The latter is the state-of-the-art in indexing data structures for multi-dimensional points. We used the authors' implementation of the hB^H-tree for the space utilization experiments. The provided implementation is in-memory, but it simulates I/Os by counting accesses to data blocks. For the rest of the experiments, we chose not to use this implementation of the hB^H-tree, since we wanted to emphasize the running times of the Bkd-tree for data sets much larger than main memory.

Data Sets. We chose three different types of point sets for our experiments: real points from the TIGER/Line data [25], uniformly distributed points, and points along a diagonal of a square. The real data consists of six sets of points generated from the road features in the TIGER/Line files. *TIGER Set* i , $1 \leq i \leq 6$, consists of all points on CD-ROMs 1 through i . Note that the largest set, TIGER set 6, contains all the points in the road features of the entire United States and its size is 885MB. Table 1 contains the sizes of all 6 data sets. Figure 6(a) depicts TIGER set 1, representing 15 eastern US states. It can be seen that points are somewhat clustered, with clusters corresponding to urban areas. The uniform data consists of six sets, each containing uniformly distributed

Set	1	2	3	4	5	6
Number of points	15483533	29703113	39523372	54337289	66562237	77383213
Size (MB)	177.25	340.00	452.38	621.94	761.82	885.69

Table 1. The sizes of the TIGER sets.

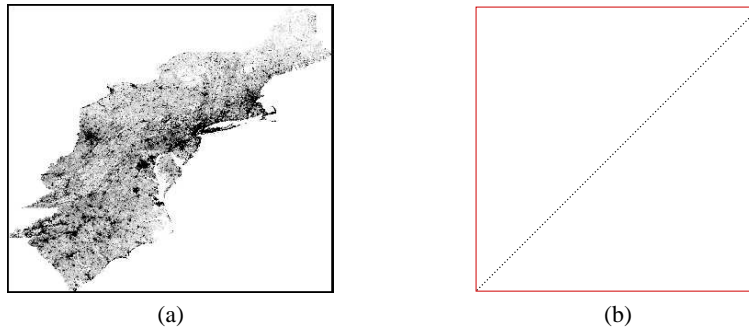


Fig. 6. (a) An image of TIGER set 1 (all the points in the road features from 15 eastern US states). The white area contains no points. The darkest regions have the highest density of points. (b) A diagonal data set.

points in a square region. The smallest set contains 20 million points, while the largest contains 120 million points. Table 2 contains the sizes of all 6 sets. The final group

Set	1	2	3	4	5	6
Number of points (millions)	20	40	60	80	100	120
Size (MB)	228.88	457.76	686.65	915.53	1144.41	1373.29

Table 2. The sizes of the uniform data sets.

of sets contains points arranged on a diagonal of a square, as shown in Figure 6(b). We used these sets only for space utilization experiments. In all sets, a point consists of three integer values: the x -coordinate, the y -coordinate, and an ID, for a total of 12 bytes per point. Thus, the largest data set we tested on, containing 120 million points, uses 1.34GB of storage.

Hardware Platform. We used a dedicated Dell PowerEdge 2400 workstation with one Pentium III/500MHz processor, running FreeBSD 4.3. A 36GB SCSI disk (IBM Ultra-star 36LZX) was used to store all necessary files: the input points, the data structure, and the temporary files. The machine had 128MB of memory, but we restricted the amount of memory that TPIE could use to 64MB. The rest was used by operating system daemons. We deliberately used a small memory, to obtain a large data size to memory size ratio.

4 Experimental Results

4.1 Space Utilization

As mentioned previously, the Bkd-tree has close to 100% space utilization. To contrast this to the space utilization of the K-D-B-tree and the hB^{II} -tree, we inserted the points from each of the diagonal data sets, sorted by x -coordinate, in all three data structures, and measured the final space utilization. The results are depicted in Figure 7(a). As expected, the Bkd-tree space utilization is almost 100% (between 99.3% and 99.4%). For the K-D-B-tree, the space utilization is as low as 28%, while for the hB^{II} -tree it is as low as 38%. In the case of the K-D-B-tree, the diagonal pattern causes most leaves of the tree to be inside long and skinny rectangles, with points concentrated on one end of the rectangle. When an internal node is split, some of these leaves are cut, resulting in empty leaves. As data sets get larger, the effect is compounded (empty leaves are split as well), resulting in increasingly lower space utilization. In the case of the hB^{II} -tree, node splits are not propagated down to leaves. Indeed, the space utilization of the leaves remains at 50% or better, as reported in [10]. However, node splits cause redundancy: Some kd-tree nodes are stored in multiple hB -tree nodes. Consequently, the size of the index grows dramatically, resulting in low fanout, large tree height, and poor overall space utilization. In our experiments, the K-D-B-tree had lower tree height than the corresponding hB^{II} -tree.

These results underscore the sensitivity of the K-D-B-tree and the hB^{II} -tree to data distribution and insertion order. Indeed, much better space utilization is obtained when the points in a diagonal data set are inserted in random order, rather than sorted on the x coordinate.

To investigate the space utilization for more practically realistic data sets, we repeated the experiment using the TIGER data. The structures were built by repeated insertions, and the order of insertion is given by the order in which the points were stored in the original TIGER/Line data. Unfortunately, we were not able to run the hB^{II} -tree experiments in a reasonable amount of time. Experiments on smaller TIGER data sets show the space utilization of the hB^{II} -tree to be around 62% (consistent with the results reported in [10] for similar geographic data). Although not as extreme as the diagonal sets, the real life data sets result in relatively poor space utilization—refer to Figure 7(b). For these sets, the space utilization of the K-D-B-tree is around 56%, still far from the 99.4% utilization of the Bkd-tree.

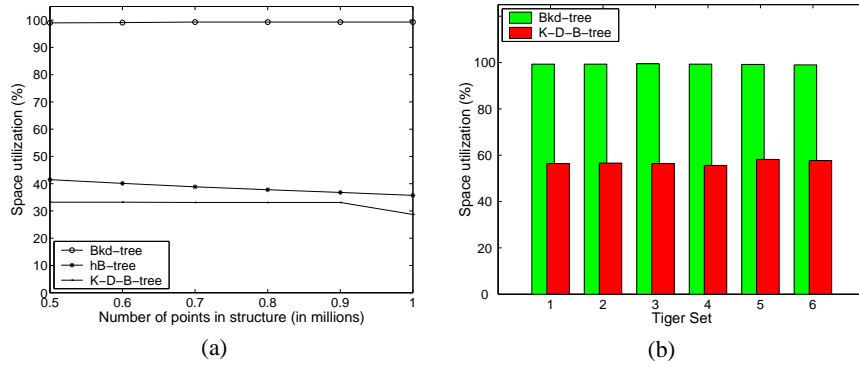


Fig. 7. (a) Space utilization for the diagonal sets. (b) Space utilization for the TIGER sets.

4.2 Bulk Loading Performance

To compare the two kd-tree bulk loading algorithms presented in Section 2.1, we tested them on both the uniform and the real data sets. Figure 8 shows the performance for the uniform data sets and Figure 9 shows the performance for the TIGER data sets. The figures reflect only the building time, leaving out the time needed to sort the data set on each coordinate, which is common for the two methods.

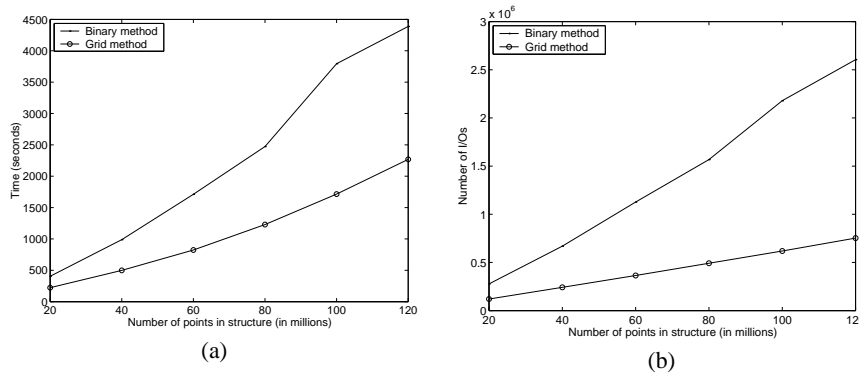


Fig. 8. Bulk loading performance on uniform data: (a) Time (in seconds), (b) Number of I/Os.

The experiments on uniformly distributed data (Figure 8(a)) show that, in terms of running time, the grid method is at least twice as fast as the binary method and, as predicted by the theoretical analysis, the speedup increases with increased set size. When comparing the number of I/Os (Figure 8(b)), the difference is even larger. To better understand the difference in the number of I/Os performed by the two methods, we can do a “back-of-the-envelope” computation: for the largest size tested, the binary method reads the input file 14 times and writes it 13 times (two reads and two writes

for each of the upper levels, and two reads and one write for the lower levels, which are computed in memory), while the grid method reads the input file 5 times and writes it 3 times (one read to compute the grid matrix, two reads and two writes for all the upper levels, and two reads and one write for the lower levels, which are computed in memory). This means that the grid method saves 9 reads of the entire file and, more importantly, 10 writes of the entire input file. To put it differently, the grid method performs less than a third fewer I/Os than the binary method. This corresponds perfectly with the results from Figure 8(b). The difference between the running time speedup (approximately 2) and the I/O speedup (approximately 3) reflects the fact that the grid method is more CPU-intensive.

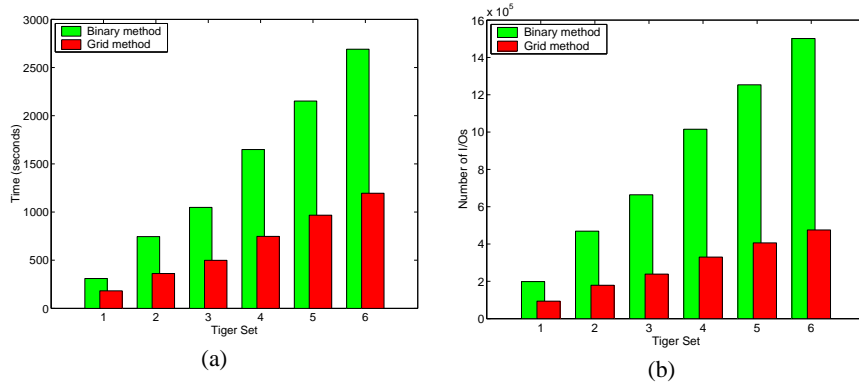


Fig. 9. Bulk loading performance on TIGER data: (a) Time (in seconds), (b) Number of I/Os.

The experiments on the TIGER data (Figure 9) show a similar pattern. Note that the kd-tree bulk loading performance is independent of the data distribution, which means that the bulk loading performance can be predicted very accurately only from the number of points to be indexed. To illustrate this, consider the uniformly distributed set containing 40 million points, and TIGER set 3, containing 39.5 million points. Comparing the bulk loading times for the two sets, we find virtually identical values.

4.3 Insertion Performance

To compare the average insertion performance of the Bkd-tree with that of the K-D-B-tree, we inserted all the points of each TIGER set into an initially empty structure, and we divided the overall results by the number of points inserted. Figure 10 shows the average time and the average number of I/Os for one insertion. In terms of elapsed time, a Bkd-tree insertion is only twice as fast as a K-D-B-tree insertion. When I/Os are counted, however, the Bkd-tree values are not even visible on the graph, since they are well below 1. This dissimilarity in the two performance metrics can be easily explained by the layout of the TIGER data and caching effects. Since points are inserted in the K-D-B-tree in the order in which they appear in the original data sets (points in the same

county are stored together), the K-D-B-tree takes advantage of the locality existent in this particular order and the fact that we cache root-leaf paths during insertions. If the next point to be inserted is next to the previous one, the same path could be used, and the insertion may not perform any I/Os.

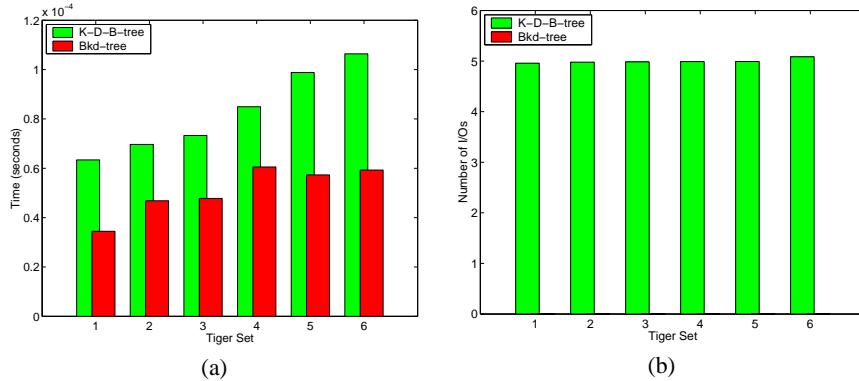


Fig. 10. Insertion performance on K-D-B-trees and Bkd-trees (TIGER data): (a) Time (in seconds), (b) Number of I/Os.

We also compared the average insertion performance of the Bkd-tree and the K-D-B-tree using the artificially generated data. The insertions in these experiments exhibit less (or no) locality since points were inserted in random order. Figure 11 shows the average time and number of I/Os for one insertion, using the uniform data sets. For the Bkd-tree, the values were obtained by inserting all points one by one in an initially empty structure and averaging. For the K-D-B-tree, however, we have not been able to perform the same experiment. Even for the smallest set, containing 20 million points, inserting them one by one takes more than 2 days! This is due to the lack of locality in the insertion pattern; even if all internal nodes are cached, each insertion still makes at least two I/Os (to read and to write the corresponding leaf) because chances are that the relevant leaf is not in the cache. This results in 40 million random I/Os for the 20 million point set.

Since we could not build the K-D-B-tree by repeated insertions, we designed a different experiment to measure the K-D-B-tree insertion performance. We bulk loaded a K-D-B-tree using the input points (filling each leaf and node up to 70% of capacity) and then we inserted 1000 random points into that structure. As predicted by the theoretical analysis, a Bkd-tree insertion is several orders of magnitude faster than a K-D-B-tree insertion, both in terms of elapsed time and number of I/Os; in terms of elapsed time, the Bkd-tree insertion is more than 100 times faster than the K-D-B-tree insertion, for all data sizes. In terms of number of I/Os, the Bkd-tree is up to 230 times faster. The discrepancy between the two numbers comes, again, from the fact that we cache nodes and leaves. Since the Bkd-tree implicitly uses the entire main memory as cache, we allowed the K-D-B-tree to do the same. However, due to the randomness of the data, very few leaves were found in the cache.

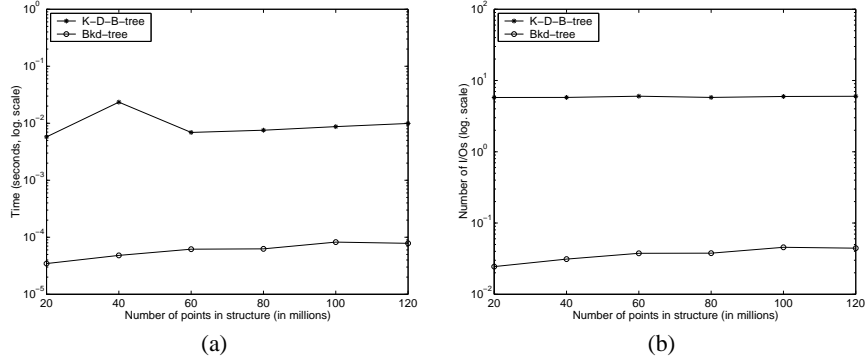


Fig. 11. Insertion performance on K-D-B-trees and Bkd-trees (uniformly distributed data): (a) Time (in seconds), (b) Number of I/Os.

4.4 Query Performance

Although the worst case asymptotic bounds for a window query on a Bkd-tree and a K-D-B-tree are identical, we expect the Bkd-tree to perform more I/Os, due to the multiple trees that need to be searched. To investigate this, we queried a Bkd-tree and a K-D-B-tree with the same window. Figure 12 shows the running times and number of I/Os of a square-shaped window query that covers 1% of the points in each of the uniform data sets. These values are obtained by averaging over 10 queries of the same size, whose position is randomly chosen in the area covered by the points. It can be seen that the Bkd-tree performs roughly the same number of I/Os as the K-D-B-tree. This somewhat unexpected result is the consequence of a number of factors. First, the average number of kd-trees forming the Bkd-tree is less than $\lceil \log_2(N/M) \rceil$ (the maximum possible). Table 3 shows the number of non-empty kd-trees and the number of maximum kd-trees for each of the 6 uniform data sets. It can easily be shown that in the course of $2^p M$ insertions into an initially empty Bkd-tree, the average number of non-empty kd-trees is $p/2$. As a result, the number of kd-trees that need to be searched during a window query is smaller than the maximum. Second, the individual kd-trees in the Bkd-tree have smaller heights than the K-D-B-tree built on the same data set. This is due to the geometrically decreasing sizes of the kd-trees and to the fact that, as noted in Section 3, the fanout of the Bkd-tree is larger than the fanout of the K-D-B-tree. As a result, the number of internal nodes read during a window query is small. Third, the kd-tree query performance is very efficient for these data sets. Table 4 shows, for the uniform data sets, the number of points returned by the query as a percentage of the total number of points retrieved. As a result, both data structures read roughly the same number of leaf-level blocks, which is close to optimal.

In terms of running time, the K-D-B-tree is faster than the Bkd-tree. This can be explained by the fact that the queries are performed on a bulk loaded K-D-B-tree. The trees constructed by the bulk loading algorithms described in Section 2.1 exhibit a high level of locality, in the sense that points that are nearby on disk are likely to be spatially close. Queries performed on the K-D-B-tree are able to take advantage of this locality, resulting in a more sequential access pattern. On the other hand, the Bkd-tree has less

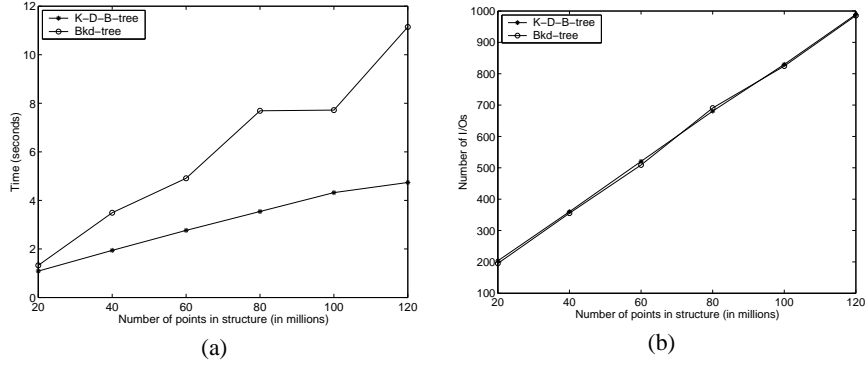


Fig. 12. Range query performance on the uniform data (the range area is 1% of entire area): (a) Time (in seconds), (b) Number of I/Os.

locality, since multiple trees have to be queried to obtain the final result. In a real-world spatial database the K-D-B-tree is often obtained by repeated insertions. This typically results in a structure with low space utilization and poor locality. This behavior can be

Number of points (in millions)	20	40	60	80	100	120
Non-empty kd-trees	3	3	3	4	4	4
Max kd-trees ($\lceil \log_2(N/M) \rceil$)	4	5	6	6	7	7

Table 3. The number of non-empty kd-trees and the maximum number of kd-trees, for each Bkd-tree built on the uniform data sets.

Number of points (in millions)	20	40	60	80	100	120
Bkd-tree	78.4	84.7	88.1	86.5	90.4	90.6
K-D-B-tree	74.8	83.6	86.2	87.9	90.2	90.6

Table 4. The number of points returned by a window query as a percentage of the total number of points retrieved. For each set, the window covers 1% of the total number of points

observed in the experiments performed on the TIGER sets. As explained in Section 4.3, the K-D-B-tree for the TIGER sets was obtained by repeated insertions. As a result, it exhibits much less locality. Figure 13 shows that the two structures perform similarly in terms of time, attesting to the fact that both structures have to perform some random I/O (the Bkd-tree because it queries multiple kd-trees, and the K-D-B-tree because it exhibits less locality). In terms of I/O, the Bkd-tree is performing half as many I/Os as the K-D-B-tree. This is due to the poor space utilization of the K-D-B-tree, which was shown to be around 56% for the TIGER data sets (see Section 4.1).

In order to measure the effect of the window size on the query performance, we ran a set of experiments with various window sizes. Figure 14 shows the results of these experiments. Both the K-D-B-tree and the Bkd-tree are built on the largest data set, containing 120 million uniformly distributed points. On the graph showing elapsed time, we see again the effects of a freshly bulk loaded K-D-B-tree, resulting in a more sequential I/O pattern than the Bkd-tree. But the I/O performance of the two structures is virtually identical for the entire range of query sizes, confirming the results obtained

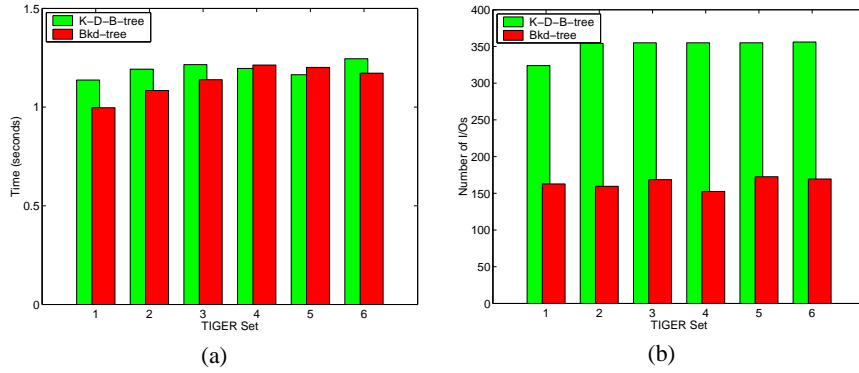


Fig. 13. Range query performance on the TIGER data: (a) Time (in seconds), (b) Number of I/Os.

on the 1% query, namely that the Bkd-tree’s window query performance is on par with that of existing data structures. Thus, without sacrificing window query performance, the Bkd-tree makes significant improvements in insertion performance and space utilization: insertions are up to 100 times faster than K-D-B-tree insertions, and space utilization is close to a perfect 100%, even under massive insertions.

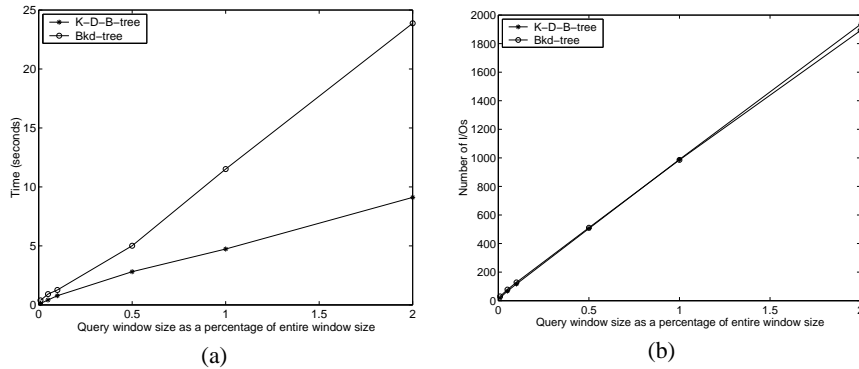


Fig. 14. Performance of range queries of increasing size (the data set consists of 120 million points uniformly distributed in a square): (a) Time (in seconds), (b) Number of I/Os.

Acknowledgments

We would like to thank Georgios Evangelidis for providing us the hB^H -tree code.

References

1. P. K. Agarwal, L. Arge, O. Procopiuc, and J. S. Vitter. A framework for index bulk loading and dynamization. In *Proc. Intl. Colloq. Automata, Languages and Programming*, pages 115–127, 2001.
2. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Commun. ACM*, 31:1116–1127, 1988.

3. L. Arge. External memory data structures. In J. Abello, P. M. Pardalos, and M. G. C. Resende, editors, *Handbook of Massive Data Sets*, pages 313–358. Kluwer, 2002.
4. L. Arge, O. Procopiuc, and J. S. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proc. European Symp. on Algorithms*, pages 88–100, 2002.
5. L. Arge, V. Samoladas, and J. S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proc. ACM Symp. Principles of Database Systems*, pages 346–357, 1999.
6. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 322–331, 1990.
7. J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, Sept. 1975.
8. J. L. Bentley. Decomposable searching problems. *Inform. Process. Lett.*, 8:244–251, 1979.
9. S. Berchtold, C. Böhm, and H.-P. Kriegel. Improving the query performance of high-dimensional index structures by bulk load operations. In *Proc. Intl. Conf. on Extending Database Technology*, volume 1377 of *Lecture Notes Comput. Sci.*, pages 216–230, 1998.
10. G. Evangelidis, D. Lomet, and B. Salzberg. The hB^{II}-tree: A multi-attribute index supporting concurrency, recovery and node consolidation. *The VLDB Journal*, 6:1–25, 1997.
11. V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.
12. R. Grossi and G. F. Italiano. Efficient cross-tree for external memory. In J. Abello and J. S. Vitter, editors, *External Memory Algorithms and Visualization*, pages 87–106. American Mathematical Society, 1999. Revised version available at <ftp://ftp.di.unipi.it/pub/techreports/TR-00-16.ps.z>.
13. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 47–57, 1984.
14. H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and R. Kanneganti. Incremental organization for data recording and warehousing. In *Proc. Intl. Conf. on Very Large Data Bases*, pages 16–25, 1997.
15. K. V. R. Kanth and A. K. Singh. Optimal dynamic range searching in non-replicating index structures. In *Proc. Intl. Conf. on Database Theory*, volume 1540 of *Lecture Notes Comput. Sci.*, pages 257–276, 1999.
16. D. T. Lee and C. K. Wong. Worst-case analysis for region and partial region searches in multidimensional binary search trees and balanced quad trees. *Acta Informatica*, 9:23–29, 1977.
17. D. Lomet. B-tree page size when caching is considered. *SIGMOD Record*, 27(3):28–32, 1998.
18. D. B. Lomet and B. Salzberg. The hB-Tree: A multiattribute indexing method with good guaranteed performance. *ACM Trans. on Database Systems*, 15(4):625–658, Dec. 1990.
19. J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Trans. on Database Systems*, 9(1):38–71, Mar. 1984.
20. P. E. O’Neil, E. Cheng, D. Gawlick, and E. J. O’Neil. The log-structured merge-tree (LSM-tree). *Acta Informatica*, 33(4):351–385, 1996.
21. M. H. Overmars. *The Design of Dynamic Data Structures*, volume 156 of *Lecture Notes Comput. Sci.* Springer-Verlag, 1983.
22. J. T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. In *Proc. SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1981.
23. H. Samet. *The design and analysis of spatial data structures*. Addison-Wesley, 1990.
24. Y. V. Silva Filho. Average case analysis of region search in balanced *k*-d trees. *Inform. Process. Lett.*, 8:219–223, 1979.
25. *TIGER/Line Files, 1997 Technical Documentation*. U.S. Census Bureau, 1998. <http://www.census.gov/geo/tiger/TIGER97D.pdf>.