Joint Conference of 1996 International Computer Symposium
December 19~21, Kaohsiung, Taiwan, R.O.C.

# An Efficient Access Method for Spatial Databases *

Anthony J.T. Lee

Department of Information Management
National Taiwan University
Taipei, Taiwan R.O.C.
jtlee@im.ntu.edu.tw

## Abstract

*We present a new spatial access method that allows the directory to grow almost linearly with the number of insertions, irrespective of the data distribution. Our method can be seen as a comprise of the Quad tree and the grid file. Because the grid file decreases the performance for highly correlated data, our method is designed to organize such data very efficiently. Such robustness in the design is achieved through the use of a hierarchical directory tree. In our method, the number of disk accesses required to reconstruct the whole directory tree is bounded by $O(N \times log(N))$, where N is the number of points in the database. Reporting on experiments conducted according to the standardized testbed designed by Kriegel et al. to compare multidimensional access methods under arbitrary data distributions and various types of queries, we found that our method outperformed the other method under most conditions.*

## 1 Introduction

In non-standard database applications, such as geographic and CAD applications as well as VLSI design, access methods are required to support efficient manipulation of multidimensional objects (data) on secondary storage. Typical manipulations are the retrieval, insertion and deletion of an object by using the values of its attributes. Also important are so-called range queries, where all object inside of a specified region are selected for further manipulation or display.

In order to handle multidimensional objects efficiently, a database system needs an access method that will help it retrieve, insert and delete objects quickly according to their specified values of attributes. A classical approach is to successively divide the data space into smaller and smaller subspaces. Multidimensional access methods based on this idea include the K-d tree [Ben75], Quad tree [FB74], MD tree [NAOS88], GBD tree [OS90], multidimensional linear hashing [HSW88a], k-d-B tree [Rob81], multidimensional extensible hashing [Tam82], grid file [NH84], BANG file [Fre87], twin grid file [HSW88b], HB-tree [LS89] and buddy-tree [SK90]. All of the methods referenced above are point-based methods, in that objects are in some way represented by single points. For example, in some methods, each object of non-zero size such as lines, regions, and solids, is stored according to its bounding box, i.e., its smallest enclosing multidimensional interval, and then mapped to a point in a higher dimensional space. They partition the data space into small regions so that all points in one region can fit in a disk page. The data space may be divided either into pairwise nonoverlapping subspaces or into overlapping subspaces.

Most of the access methods are rather efficient for uniform and uncorrelated data but not for highly correlated data. According to the results in [KSSS89], the buddy-tree is one of the best access methods; it is robust and efficient for queries on "ugly" data, in which data distributions are strongly correlated and nonuniform. However, the buddy-tree is not a balanced tree and only guarantees that a page in disk contain at least one entry. In the worst case, most of the pages in disk contain only one entry, which substantially increases the space requirement. Performance is also degraded in this case because the height of the tree increases; that is, the search path is longer.

We propose an access method which uses a concept similar to Quad trees for point data, but differs from the Quad trees by partitioning the data space into S ($\geq 4$) subspaces, rather than four, so that each subspace has about equal number of points. Because the buddy tree was superior to the HB-tree, the BANG-file, and the grid file in the performance comparison conducted by Kriegel et al. [KSSS89], we ran performance comparisons to compare our method with the buddy tree. The performance comparisons were conducted according to the standardized testbed designed by Kriegel et al. [KSSS89] to compare multidimensional access methods under arbitrary data distributions and various types of queries. We found that our methods outperformed the buddy tree under most conditions.

In the next section, we introduce the main ideas of the proposed access method. Section 3 contains a detailed description of the algorithms for constructing, updating, and searching for a database system storing points. A performance comparison is given in Section 4. Conclusions are in Section 5.

*This work was supported by National Science Council, Republic of China, under Grant NSC85-2213-E-002-022

## 2 Main Ideas

For simplicity, we will first discuss the two-dimensional (2D) case. Assume that an x-y coordinate system has been imposed on a 2D map. The origin is at the lower left corner of the map. The access method organizes point data using a tree-based directory in which points are stored in leaf nodes and non-leaf nodes store representations of bounding rectangles (bounding rectangles for brevity) that contain all points or bounding rectangles in the child nodes' entries. Because all points (data) are stored in leaf nodes and all bounding rectangles (index entries) are stored in non-leaf nodes, we call leaf nodes data nodes and non-leaf nodes index nodes in the later sections. One data or index node in the tree-based directory corresponds to one page in disk. Assume that a data node can store P points and an index node can store R bounding rectangles. Suppose that there are N points in the map. For simplicity, assume that $N = P \times R^k, k \geq 0$. There are two algorithms to build a directory tree: **Construct** and **Insert**. The **Construct** algorithm can be outlined as follows.

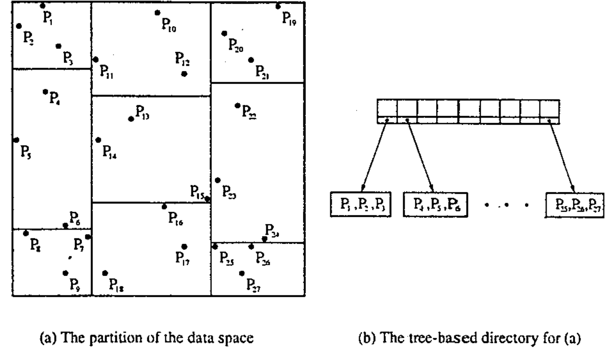
(1) Partition the data space into R rectangular regions, in such a way that each pair of rectangular regions overlap only at the boundaries and each rectangular region has the same number of data points.

(2) Continue to partition the rectangular regions as in step (1) until the number of points in a rectangular region is less than or equal to P, so that the points in one rectangular region fit in a data node.

If $N = P \times R^k, k \geq 0$, k levels of partitions are needed. The first level generates R rectangular regions, each containing $P \times R^{k-1}$ points. For each rectangular region, store its bounding rectangle into one entry of the root node. At the second level, each bounding rectangle stored in the root node is further partitioned into R rectangular regions. Similarly, for each rectangular region, store its bounding rectangle into an entry of a child node of the root. The root has R children; each child contains R rectangular regions. There are then $R^2$ rectangular regions at the second level. At the k-th level, there are $R^k$ rectangular regions, each containing P points. The points in one rectangular region are stored into a data node. For example, Figure 1 shows a set of 27 points and the corresponding tree-based directory for P=3 and R=9.

Now let's consider how much storage is needed. There is one node at the root node level (level 0). At the first level, there are R nodes. At the second level, there are $R^2$ nodes, and so on. At the k-th level, there are $R^k$ nodes. Therefore, there are $1 + R + R^2 + \ldots + R^k = \frac{R^{k+1}-1}{R-1} \approx \frac{R^{k+1}}{R-1} = \frac{N \times R}{P \times (R-1)} \approx \frac{N}{P}$ nodes in the directory tree. That is, we need approximately $\frac{N}{P}$ pages in disk to store the directory tree.

If N is not of the form of $P \times R^k$, say, $P \times R^{k-1} < N < P \times R^k$ ($k \geq 1$), the data space is first partitioned



(a) The partition of the data space (b) The tree-based directory for (a)

Figure 1: A set of points and the corresponding tree-based directory

into M subspaces, where M is $\lceil \frac{N}{P \times R^{k-1}} \rceil$. (Note that $M \geq 2$ because $\frac{N}{P \times R^{k-1}} > 1$.) Then we can apply the above algorithm to partition each subspace. Because M is $\lceil \frac{N}{P \times R^{k-1}} \rceil$, we have

$$\begin{array}{lllll} & M - 1 & < \frac{N}{P \times R^{k-1}} & \leq M \\ \Rightarrow & (M-1) \times P & < \frac{N}{R^{k-1}} & \leq M \times P \\ \Rightarrow & \frac{M-1}{M} \times P & < \frac{N}{M \times R^{k-1}} & \leq P \end{array}$$

$\frac{N}{M \times R^{k-1}}$ is the average number of points in a data node after the partitions. Because $M \geq 2$, $\frac{N}{M \times R^{k-1}} > \frac{P}{2}$. That is, the average number of points in a data node is greater than $\frac{P}{2}$.

Now let's consider how much storage is needed in this case. For each subspace, we have an R-ary full directory tree of k levels. That is, the directory tree contains $1 + R + R^2 + \ldots + R^{k-1} = \frac{R^k - 1}{R-1} \approx \frac{R^k}{R-1}$ nodes. There are M subspaces, so we need $1 + M \times \frac{R^k}{R-1} = 1 + \lceil \frac{N}{P \times R^{k-1}} \rceil \times \frac{R^k}{R-1} \approx \frac{N}{P \times R^{k-1}} \times \frac{R^k}{R-1} = \frac{N \times R}{P \times (R-1)} \approx \frac{N}{P}$ nodes for the directory tree of the whole data space. That is, we need approximately $\frac{N}{P}$ pages in disk to store the directory tree.

The detailed steps of the **Construct** algorithm is given in Section 3.1. We here summarize the properties of the directory tree created by the **Construct** algorithm.

(1) The number of nodes in the directory grows linearly with $\frac{N}{P}$, and hence with N.

(2) The height of the directory tree is $\lceil log_R(\frac{N}{P}) \rceil$.

(3) All data nodes are at the same level.

(4) The average number of points in a data node is greater than $\frac{P}{2}$.

(5) Every index node except the root node contains R entries (bounding rectangles). The root node

contains M entries, where M is $\lceil \frac{N}{P \times R^{k-1}} \rceil$ if $P \times R^{k-1} < N \leq P \times R^k$.

(6) Each index node contains bounding rectangles which contain all points or bounding rectangles in its child nodes' entries.

(7) The bounding rectangles at the same level of the directory tree overlap only at their boundaries.

The **Insert** algorithm can be outlined as follows. Assume that a new point, p, is being inserted into the directory tree.

(1) Search down the directory tree to decide where to insert p and then insert p to the directory tree.

(2) For each node S on the searching path, check whether the number of points stored in the subtree rooted at S is of the form of $P \times R^i, i \geq 1$.

(3) If yes, invoke the **Construct** algorithm to reconstruct the directory tree.

The **Construct** algorithm can be applied with the **Insert** algorithm to rebalance the directory tree. In addition, if the database will not be changed for a period of time after a large amount of data are inserted, the **Construct** algorithm can be used to reconstruct the directory tree.

# 3 Constructing, Updating and Searching the Directory Tree

In the following sections, assume that each index node in the directory tree contains at most R bounding rectangles and each data node has at most P points. Each index node and data node in the directory tree contains two variables: *numberOfPoints* and *parent*. All points in the database are assumed to be distinct.

## 3.1 Construction

Algorithm **Construct**. Given N points in 2D space, construct the corresponding directory tree. Let N/P = $R^k$, R = $r^2$, for some integer r. Let $number_{Y_a}$ be the number of points in $Y_a$ and $number_{X_{i,a}}$ be the number of points in $X_{i,a}$. (These sets will be defined below.)

(1) Sort the N points (x, y) on x (primary sort on x and secondary sort on y). Store the sorted points as an array, X.

(2) Sort the N points (x, y) on y (primary sort on y and secondary sort on x). Store the sorted points as an array, Y.

(3) Recursively execute steps (3.1) to (3.6) until the number of points in each partition is equal to or less than P.

(3.1) Partition X into r arrays so that each array contains the same number of points. Let those arrays be $X_1, X_2, ..., X_r$. The x coordinates of the partitioning lines in the x direction are midway between the x coordinates of the last point of $X_i$ and the first point of $X_{i+1}$. There are r-1 partitioning lines in the x direction.

(3.2) Create r corresponding arrays $Y_1, Y_2, ..., Y_r$ from Y by using the following algorithm. Note that each $Y_i$ is sorted on y.

```
for a=1 to r do number_{Y_a} = 0 enddo;
for b=1 to number of points in Y do
    c=0;
    repeat
        c = c+1;
    until X_c[1] ≤ Y[b] < X_{c+1}[1] /*
        primary comparison on x and sec-
        ondary comparison on y */
    number_{Y_c} = number_{Y_c} + 1;
    Y_c[number_{Y_c}] = Y[b];
enddo
```

(3.3) Partition each $Y_i$ into r arrays so that each array contains the same number of points. Let those arrays be $Y_{i,j}$. Note that the data space is now partitioned into $R = r^2$ regions, each having the same number of points. The y coordinates of the partitioning lines in the y direction are midway between the y coordinates of the last point of $Y_{i,j}$ and the first point of $Y_{i,j+1}$. There are r-1 partitioning lines in the y direction for each partition $Y_i$.

(3.4) For each $X_i$, create r arrays $X_{i,1}, X_{i,2}, ..., X_{i,r}$ from $X_i$ by using the following algorithm. Note that each $X_{i,j}$ is sorted on x.

```
for a=1 to r do number_{X_{i,a}} = 0 enddo;
for b=1 to number of points in X_i do
    c=0;
    repeat
        c = c + 1;
    until Y_{i,c}[1] ≤ X_i[b] < Y_{i,c+1}[1] /*
        primary comparison on y and sec-
        ondary comparison on x */
    number_{X_{i,c}} = number_{X_{i,c}} + 1;
    X_{i,c}[number_{X_{i,c}}] = X_i[b];
enddo
```

(3.5) If the number of points in X is greater than P, create an index node which contains the R bounding rectangles corresponding to the $X_{i,j}$. Set *numberOfPoints* of the index node to be the number of points in X. Let *parent* of the index node point to its parent.

(3.6) For each partition $i, j$, set X = $X_{i,j}$ and Y = $Y_{i,j}$.

(4) Then store the points in each partition into a data node of the directory tree. Set *numberOfPoints* of the data node to be the number of points in that partition. Let *parent* of the data node point to its parent.

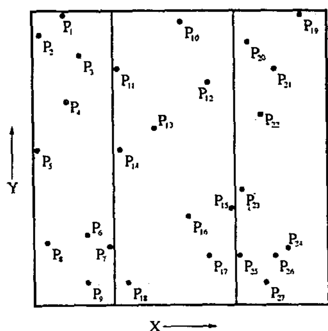


Figure 2: The rectangular regions corresponding to $X_1, X_2, X_3$

For example, Figure 2 shows the rectangular regions after X is partitioned into $X_1, X_2, X_3$ for a set of 27 points with P=3 and R=9. Each $X_i$ contains nine points. Then each rectangular region is further partitioned into three regions in the y direction, so that there are nine rectangular regions as shown in Figure 1 (a).

We now analyze the complexity of the **Construct** algorithm. The cost of steps (1) and (2) (sorts of N points) is $O(N \times log_2(N))$. Then we find a recurrence relation for step (3). Let C(N) be the cost of step (3). The costs of steps (3.1) and (3.2) are $O(N)$. The cost of step (3.3) is $O(N)$, because the cost of partitioning $Y_i$ into $Y_{i,j}$, j = 1, 2, ..., r is $O(N/r)$ and i runs from 1 to r. Similarly, the cost of step (3.4) is $O(N)$. The cost of step (3.5) is $O(R \times (N/R)) = O(N)$, because the cost of finding the smallest bounding rectangle for $X_{i,j}$ is $O(N/R)$ and there are R $X_{i,j}$. Step (3.6) takes a constant time. Therefore, the recurrence relation is

$$C(N) = R \times C(N/R) + O(N).$$

Solving this by the master theorem [CLR90], we find that C(N) is $O(N \times log_R(N))$. The cost of step (4) is O(N). Therefore, the complexity of the **Construct** algorithm is $O(N \times log_2(N))$.

Now let's consider the number of disk accesses of the **Construct** algorithm. If N is sufficient small that that all the sorting and partitioning can be done in main memory, disk accesses are required only when the data are retrieved from the disk and the directory tree is saved in the disk. In this case, the number of disk accesses required to retrieve data is $\frac{N}{P}$. There is one node at the root node level (level 0). At the first level, there are R nodes. At the second level, there are $R^2$ nodes, and so on. At the $k$-th level, there are $R^k$ nodes. Therefore, there are $1+R+R^2+\ldots+R^k = \frac{R^{k+1}-1}{R-1} \approx \frac{R^{k+1}}{R-1} = \frac{N \times R}{P \times (R-1)} \approx \frac{N}{P}$ nodes in the directory tree. That is, we need approximately $\frac{N}{P}$ pages in disk to store the directory tree. So, the number of disk accesses of the **Construct** algorithm is bounded by $O(\frac{N}{P})$.

If N is so large that all the sorting and partitioning can not be done in main memory, disk accesses are required when the data are retrieved, sorted, and partitioned, and the directory tree is saved. That is, we need to use external sort algorithm such as merge sort to sort the data and only a portion of the data is loaded into main memory for partitioning. In this case, the number of disk accesses required to sort the data is bounded by $O(\frac{N}{P} \times log_2(\frac{N}{P}))$. For each level of partitioning, we need to access $O(\frac{N}{P})$ pages in steps (3.1) to (3.4). Also, we need to access $R^i$ pages in step (3.5) for the $i$-th level of partitioning. There is no disk accesses in step (3.6). There are k levels of partitions. Hence, the number of disk accesses required to partition the data is bounded by $O(k \times \frac{N}{P} + \sum_{i=1}^{k} R^i) = O(\frac{N}{P} \times log_R(\frac{N}{P}))$. So, the number of disk accesses of the **Construct** algorithm is bounded by $O(\frac{N}{P} \times log(\frac{N}{P}))$.

If N is not of the form of $P \times R^k$, say, $P \times R^{k-1} < N < P \times R^k$ (k $\geq$ 1), the data space is first partitioned into M subspaces, where M is $\lceil \frac{N}{P \times R^{k-1}} \rceil$. (Note that $M \geq 2$ because $\frac{N}{P \times R^{k-1}} > 1$.) Then we can apply the above algorithm to construct the directory tree for each subspace. If N $\leq$ P, we don't need to execute the **Construct** algorithm but we just store all points into one data node. In this case, the directory tree contains only one data node.

If R is not of the form of $r^2$ and R is not a prime number, we can choose $R=a \times b$ so that a and b are as close to $\sqrt{R}$ as possible. For example, R=24. We can choose R=4 × 6. Then use a and b to partition the data space. If R is a prime number, we can choose an appropriate number R' to approximate R. For example, R=17. We can choose R'=4 × 4.

### 3.2 Insertion

Algorithm **Insert**. Insert a new point p=(x,y) into the directory tree. Let the root of the directory tree be T and S=T.

**(1)** Repeat steps (1.1) and (1.2) until S is a data node.

- **(1.1)** If $log_R((numberOfPoints$ of S + 1)/P) is equal to some positive integer i, invoke the **Construct** algorithm to reconstruct the subtree of S and then stop the **Insert** algorithm. Include p in the construction.
- **(1.2)** Otherwise; if S is an index node, increase *numberOfPoints* by 1 and set S to be the node at the next lower level whose corresponding bounding rectangle contains p. (If p is on a partitioning line, choose the bounding rectangle with the least *numberOfPoints*.)

**(2)** Perform one of the following steps. (At this point, S is a data node.)

- **(2.1)** If S is not full, insert p in S and increase *numberOfPoints* of S by 1.
- **(2.2)** If S is full and S's parent have less than R children, create a new data node $S_1$ and

store one half of points in S into $S_1$ according to the coordinates of the points. Insert p in S or $S_1$ according to the coordinate of p. Let the parent of $S_1$ point to the S's parent. Adjust the *numberOfPoints* of S and $S_1$.

**(2.3)** If S is full and S's parent has R children or S has no parent, split S into two new data nodes, $S_2$ and $S_3$, so that $S_2$ and $S_3$ have about an equal number of points. Insert p in $S_2$ or $S_3$ according to the coordinate of p. Adjust the *numberOfPoints* of $S_2$ and $S_3$. Change S into an index node which contains two data nodes: $S_2$ and $S_3$.

### 3.3 Exact Match Search

Here we only present two types of searching algorithms. The other types of searching can be seen as the variations of these two algorithms.

Algorithm **Search**. Given a point p=(x,y), test whether p is in the database. Let the root of the directory tree be T and S=T.

**(1)** If S is an index node, find all bounding rectangles in S containing p. For each bounding rectangle containing p, recursively invoke **Search** on the corresponding subtree.

**(2)** If S is a data node, for each point q in S, test whether p = q. If yes, report p is in the database.

### 3.4 Range Query

Algorithm **Range Query**. Given a rectangle Q, find all points that are contained in Q. Let the root of the directory tree be T and S=T.

**(1)** If S is an index node, find all bounding rectangles overlapping Q, For each overlapping rectangle, recursively invoke **Range Query** on the corresponding subtree.

**(2)** If S is a data node, check each point q in S to determine whether Q contains q. If yes, q is a qualifying point.

### 3.5 Bounding Rectangles

When the points in a database are clustered, the data space may contain some empty regions. (See (EX1), (EX2), (EX3) and (EX4) in Figure 4.) To speed up the search operations, we can make the bounding rectangles smaller. Instead of storing the rectangular region in step (3.5) of the **Construct** algorithm, we can store the smallest bounding rectangle (SBR for short) with sides parallel to the boundaries of the data space, that contains all points in this region.

The empty data space is not partitioned. Hence, the **Search** algorithm may reject a point p when p is not in any SBR at some level. It is not necessary to search down to the data node and then reject p. Therefore, the search operation becomes faster. Similarly, the range query becomes faster. However, the insertion operation needs more extra work. Consider

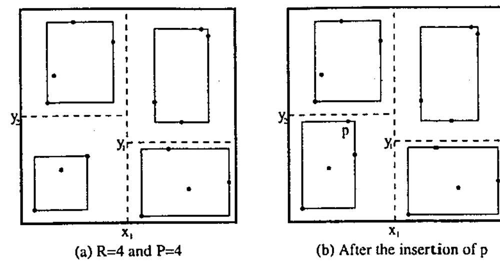



Figure 3: The example of an insertion by using SBRs. Note: The dotted lines are the partition lines.

the case: insert a point p and p is not contained in an SBR at some level. In this case, we need extra information to decide which SBR should be expanded to include p. In particular, each index node in the directory tree must store the x and y coordinates of the partitioning lines of the data space corresponding to the index node. For example, the index node corresponding to the whole rectangle in Figure 3(a) needs to store $x_1$ for the vertical partitioning lines and store $y_1$ and $y_2$ for the horizontal partitioning lines. This requires only extra storage in each index node. After p is inserted in Figure 3, we find that p is in the lower left rectangular region. So, the SBR corresponding to the lower left rectangular region is expanded to include p. The result is shown in Figure 3(b).

## 4 Performance Comparisons

According to the results shown in [KSSS89], the performance of the buddy tree was superior to that of the HB tree[LS89], the BANG file[Fre87] and the Grid file[HN83]. Therefore, we compared our method only with the buddy tree. Our implementation of the buddy tree followed the specifications of the buddy tree in [SK90].

We conducted the performance comparisons on SUN Sparc workstations under UNIX using C++ implementations of the selected methods.

### 4.1 Simulation Model

The procedure used here was similar to the standardized testbed designed by Kriegel et al. [KSSS89]. We performed six experiments on 2-dimensional points with varying distributions. There were $10^5$ points without duplicates in the first five experiments. The data for the last experiment is real satellite data and contains about 34823 points without duplicates.

The data space was normalized to be $[0,1]^2$. In the following, U(a,b) denotes a uniform distribution between a and b and N(m,v) a normal distribution with mean m and variance v. The specifications of the six experiments are shown below. The data distributions of those experiments are shown in Figure 4.

**(EX1) Diagonal:** The points follow a uniform distribution on the main diagonal.

**(EX2) Sine Distribution:** The points follow a sine curve. That is, the x coordinates of the points

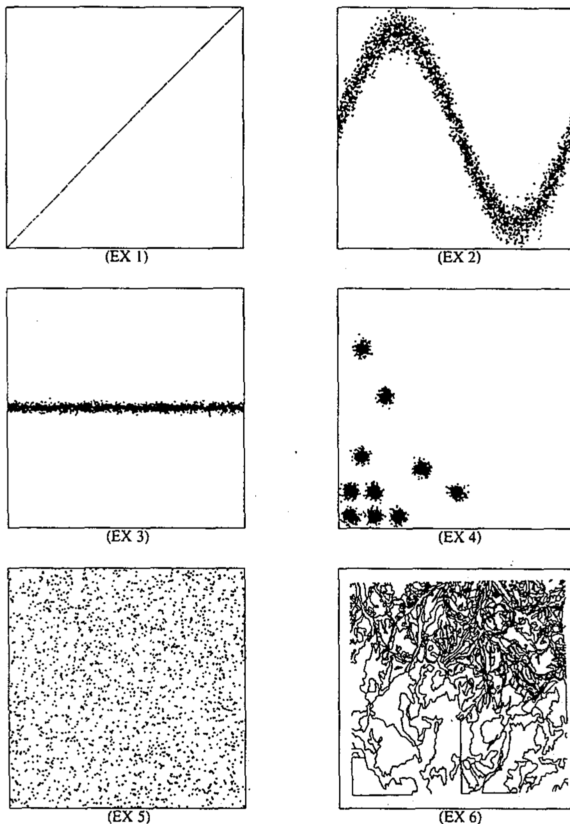



Figure 4: Data distributions

are uniformly distributed and the y coordinates follow N(sin(2 × π× x), 0.1).

**(EX3) x-Parallel:** The x coordinates of the points are uniformly distributed and the y coordinates follow N(0.5, 0.01).

**(EX4) Clustered Points:** The x coordinates of the points follow N($z_x$, 0.05) and the y coordinates follow N($z_y$, 0.05), where $(z_x, z_y)$ are the coordinates of the cluster centers. Here we arbitrarily choose ten cluster centers which are located at the lower left triangle of the data space, with six of the cluster centers located around the lower left corner of the data space. Each group of clustered points contains 10000 points.

**(EX5) Uniform Distribution:** The points are uniformly distributed over $[0, 1]^2$.

**(EX6) Real Data:** The real satellite data represents information on soils in Spearfish, South Dakota. We acknowledge receiving this data from the Construction Engineering Research Laboratory (USACERL), Illinois, USA.

For each experiment we performed five kinds of queries:

**(Q1)** 20 range queries in which the query rectangles were squares with area 0.001 in which the center of the squares followed a uniform distribution over $[0, 1]^2$.

**(Q2)** 20 range queries in which the query rectangles were squares with area 0.01 in which the center of the squares followed a uniform distribution over $[0, 1]^2$.

**(Q3)** 20 range queries in which the query rectangles were squares with area 0.1 in which the center of the squares followed a uniform distribution over $[0, 1]^2$.

**(Q4)** 20 partial match queries in which the specified x coordinate was uniformly distributed over [0,1] and the y coordinate was unspecified.

**(Q5)** 20 partial match queries in which the specified y coordinate was uniformly distributed over [0,1] and the x coordinate was unspecified.

Therefore we had 100 queries in total. For each experiment we performed the 100 queries and computed the average number of disk accesses. We chose the disk page size to be 1024 bytes (1K). We repeated the experiments so as to get small confidence intervals. We performed each experiment two hundred times, which yielded a relative error within 5% with confidence 95%.

## 4.2 Results and Summary

The number of disk accesses for each experiment is shown in Figure 5 to Figure 10, where the range query contains three columns: 0.1%, 1%, and 10% corresponding to (Q1), (Q2) and (Q3) respectively, and the partial match query contains two columns: x-spec and y-spec corresponding to (Q4) and (Q5) respectively. Generally speaking, our method is better than the buddy tree for all three types of range queries over all six types of data distributions. This is because our method creates a more balanced tree than the buddy tree method does.

For the experiment (EX3) (x-parallel), if we modify our method in the way that the number of partitions in the x direction is about ten times as many as the number of partitions in the y direction, because the range of y coordinates of the points is about one tenth of the range of x coordinates of the points, the number of disk accesses for two types of partial match queries (x-spec and y-spec) is 11.3 ± 0.1 and 12.5 ± 1.5 respectively and the average number of disk accesses is $32.6 \pm 1.2$, which is about 80% of the original average number of disk accesses. This implies that how we partition a data space should take into account the data distribution.

The average number of disk accesses over all query types for both methods is shown in Figure 11. Overall, our method is better than the buddy tree in the sense of requiring fewer disk accesses.

| Access Method | Range Query | | | Partial Match Query | |
|---|---|---|---|---|---|
| | 0.1 % | 1 % | 10 % | x-spec | y-spec |
| Buddy Tree | $3.8 \pm 0.3$ | $20 \pm 1.5$ | $168.9 \pm 6.7$ | $4 \pm 0.1$ | $4 \pm 0.1$ |
| Partition Method | $2.6 \pm 0.2$ | $13.1 \pm 1$ | $107.4 \pm 4.3$ | $4.5 \pm 0.1$ | $4.5 \pm 0.1$ |

Figure 5: (EX1): Number of disk accesses for the diagonally distributed points

| Access Method | Range Query | | | Partial Match Query | |
|---|---|---|---|---|---|
| | 0.1 % | 1 % | 10 % | x-spec | y-spec |
| Buddy Tree | $9.6 \pm 0.7$ | $38.3 \pm 2.5$ | $223.3 \pm 9.2$ | $166.7 \pm 7.9$ | $6 \pm 0.2$ |
| Partition Method | $3.8 \pm 0.3$ | $17.7 \pm 1.2$ | $124.4 \pm 5.5$ | $16.9 \pm 0.7$ | $16.4 \pm 0.2$ |

Figure 6: (EX2): Number of disk accesses for the points following a sine distribution

| Access Method | Range Query | | | Partial Match Query | |
|---|---|---|---|---|---|
| | 0.1 % | 1 % | 10 % | x-spec | y-spec |
| Buddy Tree | $6.4 \pm 0.5$ | $29.9 \pm 2.5$ | $201.5 \pm 9.3$ | $4.2 \pm 0.1$ | $79.8 \pm 11.2$ |
| Partition Method | $4.3 \pm 0.4$ | $19.3 \pm 1.9$ | $127.6 \pm 6.4$ | $52.6 \pm 0.2$ | $3.6 \pm 0.3$ |

Figure 7: (EX3): Number of disk accesses for the distributed parallel to the x axis points

| Access Method | Range Query | | | Partial Match Query | |
|---|---|---|---|---|---|
| | 0.1 % | 1 % | 10 % | x-spec | y-spec |
| Buddy Tree | $13.3 \pm 0.9$ | $57.4 \pm 2.9$ | $296.3 \pm 12.1$ | $93.9 \pm 4.4$ | $151.6 \pm 6.7$ |
| Partition Method | $5.6 \pm 0.2$ | $20.8 \pm 1.1$ | $110.9 \pm 5.8$ | $35.7 \pm 0.7$ | $33.5 \pm 0.4$ |

Figure 8: (EX4): Number of disk accesses for the clustered points

| Access Method | Range Query | | | Partial Match Query | |
|---|---|---|---|---|---|
| | 0.1 % | 1 % | 10 % | x-spec | y-spec |
| Buddy Tree | $11.6 \pm 0.1$ | $39.3 \pm 0.4$ | $195.1 \pm 2.9$ | $122.6 \pm 0.5$ | $71.1 \pm 1.1$ |
| Partition Method | $7.1 \pm 0.1$ | $22.8 \pm 0.2$ | $116.6 \pm 1.8$ | $52.7 \pm 0.2$ | $40.5 \pm 0.1$ |

Figure 9: (EX5): Number of disk accesses for the uniformly distributed points

| Access Method | Range Query | | | Partial Match Query | |
|---|---|---|---|---|---|
| | 0.1 % | 1 % | 10 % | x-spec | y-spec |
| Buddy Tree | $11.3 \pm 0.4$ | $35.2 \pm 1.3$ | $120.5 \pm 3.9$ | $263.1 \pm 7.6$ | $4.54 \pm 0.1$ |
| Partition Method | $6.1 \pm 0.2$ | $12.7 \pm 0.4$ | $54.5 \pm 2.1$ | $37.5 \pm 1$ | $22.5 \pm 0.2$ |

Figure 10: (EX6): Number of disk accesses for the real data

## 5 Conclusions

We have proposed a new spatial access method. Like the buddy tree, our method can generate the SBRs in its directory. Hence the data space is not completely covered by those SBRs. In particular empty data space is not reflected in the directory tree. By using Kriegel's standardized testbed, we ran a performance comparison of our method with the buddy tree and the result demonstrated the efficiency and robustness of our method.

In addition, it appears that a directory tree can be reorganized locally or globally in our method. How to reorganize a directory tree and how the reorganization affects the performance needs be further studied. As shown in the result of the performance comparison, the partition of a data space depends on the data distributions. Choosing a good partition for a given data distribution needs be further investigated. Also, we are extending our method from the two dimensional case to higher dimensions.

| Data Distribution | Buddy Tree | Partition Method |
|---|---|---|
| Diagonal | $40.1 \pm 1.3$ | $29.2 \pm 0.9$ |
| Sine | $88.9 \pm 2.6$ | $35.9 \pm 1.2$ |
| X-parallel | $64.4 \pm 2.9$ | $41.5 \pm 1.3$ |
| Cluster | $122.5 \pm 3.1$ | $41.3 \pm 1.2$ |
| Uniform | $87.9 \pm 0.6$ | $47.9 \pm 0.3$ |
| Real Data | $86.9 \pm 1.8$ | $26.7 \pm 0.5$ |

Figure 11: Average number of disk accesses over all five types of queries

## References

[Ben75] J.L. Bently. Multidimensional Binary Search Trees Used for Associative Searching. *Communications of the ACM*, 18(9):509–517, 1975.

[CLR90] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, Massachusetts, 1990.

[FB74] R.A. Finkel and J.L. Bently. Quad Tree: A Data Structure for Retrieval on Composite Key. *Acta Informatica*, 4(1):1–9, 1974.

[Fre87] M. Freeston. The BANG file: a new kind of grid file. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 260–269, 1987.

[HN83] K. Hinrichs and J. Nievergelt. The Grid File: A Data Structure Designed to Support Proximity Queries on Spatial Objects. In *Proc. Workshop on Graph Theoretic Concepts in Computer Science*, pages 100–113, 1983.

[HSW88a] A. Hutflesz, H.W. Six, and P. Widmayer. Globally Order Preserving Multidimensional Linear Hashing. In *Proc. IEEE 4th Intl. Conf. on Data Engineering*, 1988.

[HSW88b] A. Hutflesz, H.W. Six, and P. Widmayer. Twin Grid File: Space Optimizing Access Schemes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 183–190, 1988.

[KSSS89] H.P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. Performance Comparison of Point and Spatial Access Method. In *First Symposium SSD: Design and Implementation of Large Spatial Databases*, pages 89–114, 1989.

[LS89] D.B. Lomet and B. Salzberg. The hB-tree: A Robust Multiattribute Search Structure. In *Proc. IEEE 5th Intl. Conf. on Data Engineering*, 1989.

[NAOS88] Y. Nakamura, S. ABE, Y. Ohsawa, and M. Sakauchi. MD-Tree: A Balanced Hierarchical Data Structure for Multidimesional Data with Highly Efficient Dynamic Characteristics. In *Proc. IEEE Intl. Conf. on Pattern Recognition*, pages 375–378, 1988.

[NH84] J. Nievergelt and H. Hinterberger. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. on Database Systems*, 9(1):38–71, 1984.

[OS90] Y. Ohsawa and M. Sakauchi. A New Tree Type Data Structure with Homogeneous Nodes Suitable for a Very Large Spatial Database. In *Proc. IEEE 6th Intl. Conf. on Data Engineering*, pages 296–303, 1990.

[Rob81] J.T. Robinson. The k-d-B-Tree: A Search Structure for Large Multidimensional Dynamic indexes. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 10–18, 1981.

[SK90] B. Seeger and H.P. Kriegel. The Buddy-Tree: An Efficient and Robust Access Method for Spatial Data Base Systems. In *Proc. 16th Very Large Database Conf*, pages 590–601, 1990.

[Tam82] M. Tamminen. Efficient Spatial Access to a Database. In *Proc. ACM SIGMOD Intl. Conf. on Management of Data*, pages 200–206, 1982.