Performance Comparison of Segment Access Methods Implemented on Top of the Buddy-Tree

Bernhard Seeger*
Dept. of Comp. Science, University of Waterloo
Waterloo, Ontario, Canada N2L 3G1

Abstract

Multidimensional segment access methods efficiently organize one- and multidimensional intervals on secondary storage. In this paper, we present a detailed discussion and a comprehensive experimental performance comparison of these methods. Each of the segment access methods is implemented on top of an ordinary point access method using the techniques clipping, overlapping regions and transformation. As the basic point access method we have chosen the buddy-tree. The performance of the buddy-tree is less influenced by the underlying data distribution, and therefore, the resulting segment access methods can be judged with respect to their individual techniques. Besides the different versions of buddy-trees, the R*-tree (an improved approach of the R-tree) and the grid file participate in our experimental performance comparison.

1 Introduction

An efficient organization of spatial data objects requires the ability to cluster them according to their spatial locations in the underlying (Euclidean) data space. These data objects may represent points, rectangles, polygons or semantically richer composite objects. Data files storing these objects are large and therefore, retrieval operations almost always require access to secondary storage. The time efficiency of operations is usually measured by the number of required disk accesses. During the last decade, a large number of access methods has been proposed to maintain an efficient organization of spatial objects. A survey of such access methods is given in [Sam 89]. We distinguish between point ([Rob 81, NHS 84, Fre 87, LS 90, SK 90]) and spatial access methods as being able to organize multidimensional point data and spatial objects with an extension, respectively. Most work on spatial access methods

*Bernhard Seeger is on leave of absence from the University of Bremen, Germany.

([Gut 84, Hin 85, SRF 87, SK 88, SW 88, HSW 89, HSW 90]) was done under the assumption that objects with an extension can be approximated by their (rectilinear) minimum bounding rectangles. Therefore, the organization of d-dimensional spatial objects is transformed into the simpler problem of organizing d-dimensional segments. In the following we refer to these access methods as segment access methods, or SAMs for short.

As shown in [SK 88], SAMs can be implemented on top of point access methods (PAMs) using one of the following techniques: clipping, overlapping regions, and transformation. Therefore, the performance of a SAM depends on the choice of both, the technique and the underlying PAM. Until now, less work has been done to compare SAMs with respect to these techniques. An experimental performance comparison with regard to the different techniques seems to be not fair as long as different PAMs are used for the implementation of the SAMs. In this paper, we present how these different techniques are implemented on top of the buddy-tree. Moreover, we present the results of a comprehensive performance comparison in which all these SAMs have been participated.

The choice for the buddy-tree is not by accident at all. The buddy-tree [SK 90] offers efficient performance almost independent of the underlying data distribution. In a previous experimental performance comparison of different PAMs [KSSS 89], the buddy-tree has been shown to be superior to some competitors, like the BANG-file [Fre 87], hB-tree [LS 90] and the grid file [NHS 84]. Thus we expect that SAMs, implemented on top of the buddy-tree, are in general very efficient. To demonstrate this, the performance comparison include other schemes like the grid file (with the transformation technique) and an improved version of R-trees, called R*-tree [BKSS 90].

The paper is organized as follows. In section 2 we review the principles and the properties of the buddy-tree. In section 3, we present how these three techniques clipping, overlapping regions and transformation are implemented on top of the buddy-tree. The performance comparison is presented in section 4. Section 5 concludes the paper.

2 Review of the Buddy-tree

During the last decade several multidimensional point access methods (PAMs) have been proposed. All PAMs partition the multidimensional data space into page regions. A page region belongs to a data page in the way that all multidimensional records stored in the page must be in the page region. A classification of these PAMs can be made with regard

to certain properties of their page regions. The design goals of the buddy-tree [SK 90] are quite different to the ones of previously proposed PAMs. The most important property is that the union of the page regions does not cover the complete data space. Instead the buddy-tree avoids partitioning empty data space. We will give a brief overview of the data structure by an example, see Figure 1. We assume a 2-dimensional data space with page regions S1,..,S3,R1,..,R4 as illustrated on the left side of Figure 1. All the data records are assumed to be covered by these page regions.

The buddy-tree organizes records with a tree-based data structure whose internal and leaf nodes are called directory and data pages, respectively. Data pages contain only data records whereas the directory pages contain so-called directory entries. A directory entry comprise a pointer to a subtree and a directory rectangle that covers all the records which are stored in the corresponding subtree. It might be the minimum bounding rectangle, but this is not necessary. However, all directory rectangles in a page have to be disjoint. In the middle of Figure 1, the depicted directory refers to the page regions illustrated on the left side. Here, the rectangles R and S, which are stored in the root page, are the minimum bounding rectangles of R1,..,R4 and S1,..,S3, respectively.

The maximum number of records (entries) in a page is determined by the size of the records (entries) and the transfer untit between main memory and secondary storage supported by the operating system. Thus, an insertion of a record may result in an overfilled data page. Such an overflow is immediately eliminated by transferring some of the records from the original page to a new allocated page. Thereafter, the rectangle of the original directory entry is updated and a new directory entry (pointing to the new data page) is inserted in the parent directory page. An overflow in a directory page is eliminated in the same way. Thus, a split is always propagated up to the root, but never down to the leaves.

Retrieval operations, such as an exact match query, can be easily implemented for the buddy-tree. Starting with the root page, the buddy-tree computes the rectangle containing the query point. If such a rectangle exists, the root of the corresponding subtree will be investigated. In the other case, there is no answer for the query and the algorithm returns with an appropriate message. Eventually, the algorithm might access a data page where an answer might be found. This brief example shows that an exact match query is restricted to a single path.

Despite of this simplistic data structure, the directory rectangles of a page have to fulfill a rather complicated condition: the entries can always be mapped into an adaptive

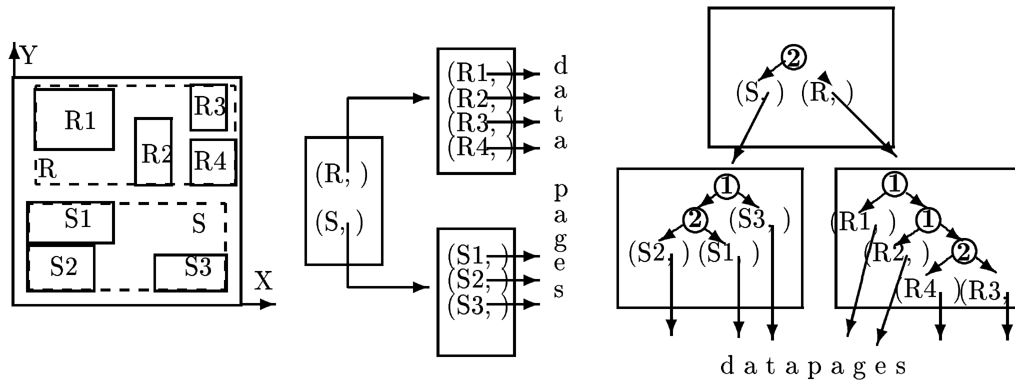

Figure 1: *On the left side, the page regions R1,..,R3, S1,..,S4 are depicted; in the middle, the corresponding directory of the list representation is shown; and on the right side, the directory is shown based on the kd-trie implementation*

kd-trie An adaptive kd-trie generalizes the concept of an ordinary kd-trie [Ore 82]. It is a binary digital tree where the internal nodes comprise a name of an axis and two pointers referring to (local) subtrees stored in the same directory page of the buddy-tree. A leaf node contains either a directory entry or a NIL-value. Each node of the adaptive kd-trie belongs to a rectangle, in the following called *B-rectangle*. B-rectangles are generated by successive halving of the data space using hyperplanes perpendicular to the axes of the data space. Let us emphasize that there is no demand on a special order of the axes. For example, $[0, 1] \times [0, .25]$ is a B-rectangle in the unit square, because it can be generated by halfing the square twice with respect to the 2nd axis. Let us first discuss how the B-rectangle of the root node is derived. If the kd-trie is stored in the root page of the buddy-tree, the B-rectangle is the complete data space. Otherwise, the B-rectangle is inherited from the corresponding leaf of the kd-trie stored in the parent page. Now let us consider an internal node of the kd-trie. If the node is the left (right) child of its parent node, the corresponding rectangle will be the left (right) half of the rectangle of the parent node with respect to the axis denoted in the parent node. A set of directory entries can be mapped into an adaptive kd-trie, if each of these entries can be uniquely stored in a leaf node of the kd-trie in such a way that the B-rectangle cover the directory rectangle. An example is given in Figure 1. The directory rectangles R1,..,R4 can be represented in the kd-trie illustrated in the most right box. The B-rectangle belonging to the root of the kd-trie is the upper half of the data space.

In this paper, we present two different implementations of the buddy-tree. Both are used later for the discussion of SAMs. The first approach is based on the representation of a directory page as an unsorted list of tuples (R,p) where R is a rectangle and p is a pointer to a subtree. This approach is illustrated in the middle of Figure 1. Obviously, this approach has a serious drawback in consuming a lot of CPU-time. For a retrieval operation, at least half of the list must be traversed on the average. Moreover, after merging pages (because of low occupation) the partition in the parent page must be checked whether it can be still represented in a kd-trie or not. Also this testing requires a lot of CPU-time. Since the speed of main memory improves with a much higher rate than the speed of secondary storage, this problem will become less important in future computer systems. The advantage of this approach is its compact representation. Instead of the exact representation, approximations of the directory rectangles are stored based on Z-ordering, see [Ore 86]. This technique is comparable to the prefix technique for B-trees as it improves the fan-out in the directory pages. Mostly not more than 8 bytes are required to represent a directory entry.

The other approach for the implementation stores explicitly adaptive kd-tries in the directory pages. It is illustrated on the right side of Figure 1. A pointer to a (directory or data) page and the minimum bounding rectangle of the child page are stored in the leaf nodes of the kd-tries. Keeping the minimum bounding rectangle is not necessary for the implementation as a point access method. However, because we will use this property for the implementation of SAMs, we already mention it here. In the current implementation, these rectangles can be left out of the data structure during initialization of a file. Now, the drawback is that the fan-out will be lower than in case of the first approach, and therefore the height of the tree might increase. However, we can expect a decline in CPU-time. Let us emphasize that the other parts of the implementation are not influenced by the representation of the directory pages.

We briefly sum up the most important properties of the buddy-tree. Insertions, deletions and exact match queries are restricted to a single path of the directory. This property is essential for a dynamic data structure. However, there is no guarantee of an upper bound for the storage utilization. When an overflow occurs, we will choose a split axis. According to the hyperplane (perpendicular to the split axis) the records are separated into two groups. Because this process depends mainly on the data space (and not on the data records), it might be possible that only one entry is in the one page and all the other entries are in the other page. Due to the organization of the data space,

the height of the tree is only linearly restricted in the number of records. However, if w denotes the resolution of the data space, i.e. the data space consists of 2^w different elements, and if we assume that the keys identify the data records, the height of the buddy-tree is bounded by

$$\lceil \frac{w - \lfloor \log_2 b \rfloor}{\lfloor \log_2 c \rfloor} \rceil$$

Here b and c denote the capacity in the data and directory pages, respectively. Nevertheless, the buddy-tree guarantees at least a linear growth of the directory. The advantage of relying on a data-space-oriented split algorithm is that the performance of the buddy-tree is much less influenced by the sequence of insertions. Although worst-case performance of the buddy-tree is not outstanding, it offers an excellent average case performance. In a performance comparison [KSSS 89], including highly skewed data files, the buddy-tree has been shown to be superior to its competitors (grid file, BANG-file, hB-tree, ZB+-tree).

3 Segment Access Methods

In this section we give an overview of access methods organizing one- and multidimensional segments. Segments are used to index data sets containing one- or multidimensional points. There are various applications for such access methods, such as spatial data processing [Sam 89], time and version management [LS 89] and rule indexing [HCKW 90].

In the following we restrict our considerations to spatial data processing. For that let us assume that we have to organize a set of polygons in a 2-dimensional data space D. Each polygon ρ can be represented by an ordered sequence of points $P_1, .., P_m$ with $P_i = (x_i, y_i) \in D$, $1 \leq i \leq m$. A very common way to provide an index on these polygons is to use their (rectilinear) minimum bounding rectangles. Given a polygon ρ, the minimum bounding rectangle $R(\rho) = (X_L(\rho), Y_L(\rho), X_U(\rho), Y_U(\rho))$ is given by

$$X_L(\rho) = \min_{i=1,..,m} x_i \qquad X_U(\rho) = \max_{i=1,..,m} x_i$$
$$Y_L(\rho) = \min_{i=1,..,m} y_i \qquad Y_U(\rho) = \max_{i=1,..,m} y_i$$

We use the notation $R = (X_L, Y_L, X_U, Y_U)$ for short. For given sets of polygons Φ and Ψ, a point $P \in D$ and a rectangle $S \subseteq D$, an index on each of these sets can be used to support the following operations:

point query: Find all polygons $\rho \in \Phi$ with $P \in \rho$.

rectangle intersection query: Find all polygons $\rho \in \Phi$ with $\rho \cap S \neq \emptyset$.

rectangle containment query: Find all polygons $\rho \in \Phi$ with $\rho \subseteq S$.

rectangle enclosure query: Find all polygons $\rho \in \Phi$ with $\rho \supseteq S$.

spatial join: Find all polygons $\rho \in \Phi$ and $\sigma \in \Psi$ with $\rho \cap \sigma \neq \emptyset$

As discussed in [Ore 86], a query can be answered in two steps. In the *filter step*, a candidate set is computed which includes all possible answers. This can be achieved by checking the minimum bounding rectangles against the query condition. All the polygons whose minimum bounding rectangle match the condition are stored in the candidate set. This filter step can be efficiently supported by multidimensional segment access methods. Afterwards, in the *refinement step*, each polygon of the candidate set is checked whether it really fulfills the query. If the polygon is simple (for example convex) or if it comprises only a few points and edges, the test can be easily realized. However, such polygons are seldom found in real applications. For more complex polygons, the decomposition of polygons into small and simple objects seems to be a good solution. One possibility might be to store triangulations of the polygons in the data base. Assuming a triangulation, each triangle again can be accessed by its minimum bounding rectangle. Thus, segment access methods can be used in both steps of spatial query processing. A detailed discussion of this idea can be found in [KHHSS 91].

In order to maintain low insertion costs, exact match queries should also be supported efficiently. In particular, a SAM should be dynamic, i.e. insertions and deletions should not reduce retrieval performance. In analogy to PAMs it is important that retrieval performance should be independent of the distribution of the spatial objects. The distribution of the rectangles is characterized by the distribution of the locations and extensions.

In the following we will describe three techniques to implement a SAM on top of a PAM. A more detailed description of these techniques can be found in [SK 88]. Our main purpose is to present the implementation of these techniques on top of the buddy-tree.

3.1 Clipping

The technique of clipping can be best explained by describing the insertion of a new rectangle into the buddy-tree. An insertion of a rectangle S requires the following: At

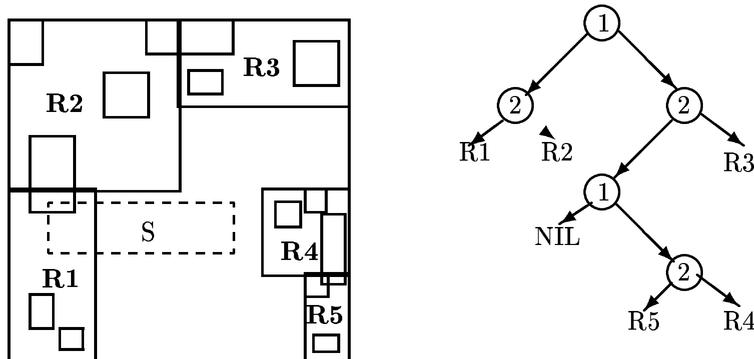

Figure 2: Before insertion of the data rectangle S

first we have to compute a subset of page regions $R_1, .., R_m$ of the buddy-tree such that

$$(\cup_{1 \leq i \leq m} R_i) \supseteq S \quad \text{and} \quad S \cap R_i \neq \emptyset,\ i = 1, .., m$$

Then we insert the rectangle S in every page whose page region is in $\{R_1, .., R_m\}$.

As long as the rectangle does not intersect with empty data space, the insertion is straightforward. In the other case, the insert algorithm has to modify the buddy-tree to fulfill the demand $(\cup_{1 \leq i \leq m} R_i) \supseteq S$. An example is illustrated in Figure 2. The buddy-tree actually stores 13 rectangles in 5 pages with page regions $R_1, .., R_5$. The hatched part of the data space does not intersect with a spatial object and therefore, it is not represented in the directory. The new (dotted) rectangle S, which should be inserted, intersects with empty data space. This problem is well known from the implementation of the R^+-tree [SRF 87], a segment access method based on the KDB-tree [Rob 81] using clipping. However, it turns out that the solution is much easier for the buddy-tree than for the R^+-tree. From Section 2, we recall that each directory page of the buddy-tree can be represented by at least one adaptive kd-trie. Using one of these kd-tries, the buddy-tree introduces new regions which must be created to cover the inserted rectangle. After that, the buddy-tree tries to merge these regions with the already existing regions by maintaining the general properties of the directory. The example of Figure 2 illustrates the solution. On the right side we have represented the partition by an adaptive kd-trie. As mentioned before, each node of the kd-trie corresponds to a B-rectangle. The data rectangle S intersects with two B-rectangles that correspond to leaves. In these leaves, either page regions are expanded or (if a NIL-value is in the leaf) new page regions are created so that page regions cover the new data rectangle. In our example the page region R_1 is expanded and a new page region R_6 is created. Thereafter, we merge the

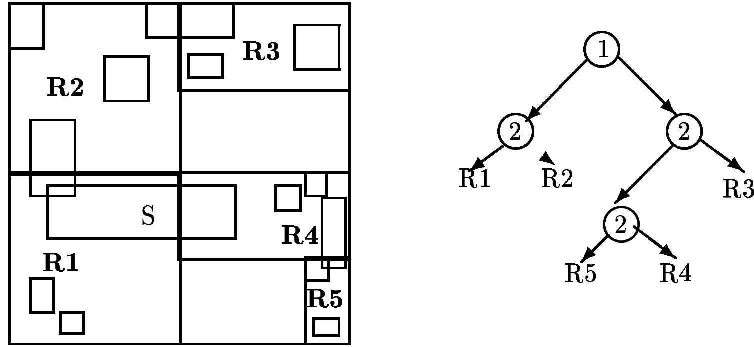

Figure 3: After insertion of the data rectangle S

page region R_6 with the page region R_4 without violating the demands on the directory. The situation after the insertion is illustrated in Figure 3.

Another problem of the R$^+$-tree was that an insertion may continuously entail up and down updates in the tree structure. Until now, there is no solution to control this process. However, this problem does not occur for the buddy-tree at all. The reason is that insertions are restricted to a single path whenever the buddy-tree is used as a PAM. The insertion process of a data rectangle traverse the buddy-tree from the leaves to the root and not in the other direction.

The implementation of the queries is straightforward and is comparable to a simple range query in the buddy-tree. However, one problem has not gained enough attention. Let us consider a rectangle intersection query. In general, the answers are not uniquely represented in the response set. In particular, for a very large response set, an external sort must be performed to eliminate duplicates. Large response sets might be very likely in case of spatial join operations. However, external sorting can be easily avoided, as we demonstrate it by considering a rectangle intersection query. For each candidate rectangle the buddy-tree has found in a data page, the center of the intersection of the query rectangle and the candidate rectangle is computed. If this point is in the page region (of the actual data page) we accept the candidate as an answer of the query. Otherwise, we reject the candidate rectangle. Let us recall that because of the disjoint page regions only one page region contains this point.

Another important change must be made to prevent the buddy-tree from a deadlock situation. Let us assume that the capacity of a data page is 4, and that 5 rectangles were inserted covering a common point of the data space. Then a split of the data page cannot

eliminate the overflow. Thus, it is necessary to allow overflow records. The buddy-tree uses the most common technique, called overflow chaining, where another page (overflow page) is chained to the original page. One of the records is stored in the overflow page and the primary page is locked against further split operations.

The properties of the clipping technique can be summarized as follows. The redundancy of the objects results in low storage utilization and in expensive insert and delete operations. Point and rectangle enclosure queries require access to exactly a path of the buddy-tree. The efficiency of rectangle intersection and rectangle containment queries declines with an increasing size of the query region. Finally, the implementation of the scheme requires much effort.

These properties indicate that applications are efficiently supported where mainly point queries occur. Typical applications include rule-based systems [HCKW 90] and version management [LS 89]. However, spatial applications favor more the spatial join operation and the rectangle intersection query. Because of the uncontrolled replication of data, clipping cannot be recommended for spatial applications under any circumstances.

It is possible to reduce redundancy. Instead of clipping minimum bounding rectangles, the spatial objects themselves can be clipped. However, because of complicated insert algorithms, the spatial objects must be simple. We introduced this approach for line segments, because we have expect the best possible improvement for this type of data. The reduction of redundancy was less than expected: only 10% less redundancy was gained on the average by clipping the lines in comparison to clipping the minimum bounding rectangles.

3.2 Overlapping Regions

The technique of overlapping regions is implemented in several SAMs, like R-trees [Gut 84, BKSS 90] and spatial kd-trees [Ooi 87]. For the implementation of this technique on top of the buddy-tree, we will make use of the kd-trie representation in the directory pages. To build up the structure, we use the centers of the rectangles as a 2-dimensional key. Then the rectangles are inserted in a 2-dimensional buddy-tree. There is one basic difference to the buddy-tree used as a pure point access method. The directory rectangles stored in the leaves of the kd-trie are not the minimum bounding rectangles of the centers, but of the whole rectangles stored in the corresponding subtree. Therefore, directory rectangles in a page may have a common intersection. However, the B-rectangles of the leaf nodes are disjoint to each other. They cover only the centers,

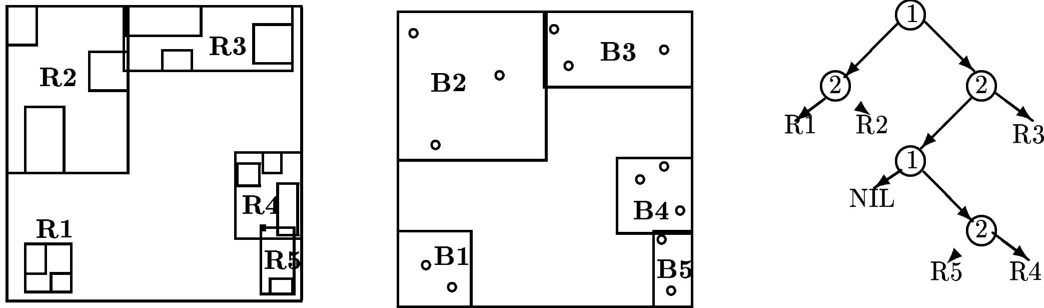

Figure 4: Overlapping regions on top of the buddy-tree

but not the complete rectangles stored in the corresponding subtree. An example of the structure is given in figure 4 where 13 rectangles are already inserted. We assume that the rectangles are stored in five data pages whose minimum bounding rectangles are $R_1, \ldots, R_5$. These rectangles are stored in the leaf nodes of the kd-trie, as illustrated on the right side. Let us emphasize that R_4 and R_5 have a common intersection. The B-rectangles $B_1, \ldots, B_5$ of the data pages and the centers of the data rectangles are illustrated in the middle of figure 4. The B-rectangles are exclusively used for insertion. A rectangle is inserted in a page when the center of the rectangle is covered by the B-rectangle of the page.

Spatial queries are performed without making use of the internal nodes of the kd-tries. Instead, queries require only the directory rectangles. For a point query, we follow the pointer to the subtree whenever the directory rectangle cover the point.

The advantage of overlapping regions implemented on the buddy-tree is that insertions and deletions are very efficiently supported. Let us mention that, in general, an insertion assumes first an unsuccessful exact match query. This query is restricted to a single path for the buddy-tree, while the R-tree has to traverse almost several paths of the directory. The drawback of the buddy-tree is that the fan-out in the directory pages is lower than in case of the R-tree. However, the internal nodes of a kd-trie does not require as much space as the directory entries that are stored in the leaves of the kd-trie.

Let us summarize the most important properties of SAMs based on overlapping regions. First of all, there is no redundancy. The SAM will have a high storage utilization and the costs for insertions and deletions are rather low. For a simple point query, it might be necessary to traverse several paths of the directory. Therefore, for point queries

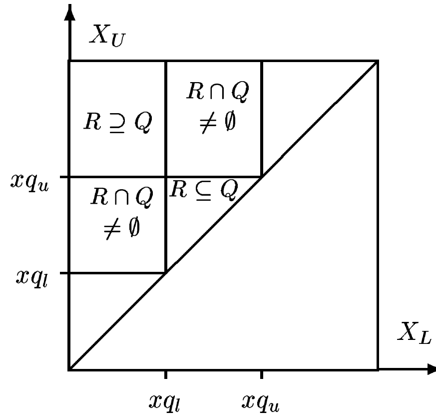

Figure 5: Query regions projected on the (X_L, X_U) plane of the data space

we expect that this approach will be inferior to the SAMs using clipping. However with an increasing size of the query region, rectangle intersection queries will be very efficiently supported.

3.3 Transformation

The basic idea of transformation-based schemes is to represent the minimum bounding rectangle by higher dimensional points. For instance, a 2-dimensional rectangle can be represented by a 4-dimensional point (X_L, Y_L, X_U, Y_U). This representation is called the corner-representation. An other possibility is the center-representation where the rectangle is characterized by its center and the distance from the center to the edge of the rectangle. Independent of the representation, the buddy-tree (or any other multidimensional access method) is able to store these 4-dimensional points.

A survey of the transformation technique can be found in [Hin 85, SK 88, Sam 89]. In this paper we restrict our considerations to the corner-representation. Additionally to the transformation of the rectangles, the original query regions must be also transformed into higher dimensional query regions. As an example, the query regions of the different types of queries are shown in Figure 5. The regions are projected on the (X_L, X_U) plane of the data space. The query segment is specified by $Q = (xq_l, xq_u)$. We classify the regions of the data space depending on the condition the segments R fulfill, see Figure 5.

The data distribution of these transformed data is very skewed. Because of $X_U > X_L$ ($Y_U > Y_L$) the transformed data occurs only above the diagonal. Moreover, under the realistic assumption that data rectangles are rather small in comparison to the complete data space, the transformed points are very close to the diagonal. For the grid file, this

distribution is close to the worst case. The buddy-tree should not be influenced by skew distributions. Moreover, the transformed query regions will mostly cover a lot of empty data space. Thus, the distribution of the query regions does not follow the distribution of the data points. This should not reduce the performance of the buddy-tree. One design goal of the buddy-tree was actually to fulfill these requirements.

The properties of the transformation technique can be summarized as follows. Similar to the overlapping region techniques there is no redundancy of rectangles, thus, insertion and deletions are inexpensive, and storage utilization is high. In general, the point query is not restricted to a single path of the tree. In contrast to previously discussed techniques, the transformation technique favors the rectangle containment query. For a given query rectangle, the query region of the rectangle containment query is smaller than the query region of the rectangle intersection query, see Figure 5. Thus, the rectangle containment query requires almost always less disk accesses than the rectangle intersection query. This is not true for the other techniques, that is, both types of queries require always the same number of disc accesses. A serious drawback for the transformation technique is that rectangles originally close together might be far away in the transformed data space. In particular, this behavior can be observed in case of rectangles with a high variance in size. Obviously, the implementation of the transformation technique is simple since the different types of queries can be easily expressed by range queries. The other techniques require more effort for the implementation.

4 Performance Comparison

In order to compare the performance of the different techniques based on the buddy-tree, we decided to realize an experimental performance comparison based on our implementations. All the methods were implemented in Modula-2 and we ran the comparisons on SUN workstations. For tree-based access methods, like the buddy-tree, we kept the last accessed path in main memory. In order to keep the performance comparison manageable, we have chosen relatively small data and directory page sizes of 1024 bytes. Using smaller page sizes, we obtain similar performance results as for much larger file sizes. The maximum number of data entries per page was the same for all SAMs in the performance comparison. In contrast to that, there were no demands on the organization of the directory pages.

In addition to the various versions of buddy-trees, we included two other structures in

our comparison. The one is the grid file implemented with the transformation technique (corner-transformation). The reason to take the grid file is simply its popularity. The comparison should demonstrate how much the special features of the buddy-tree pay off in comparison to an "ordinary" point access method such as the grid file. Let us emphasize that we used the 2-level implementation of the grid file [Hin 85], that is, a main memory resident grid directory contains references to the grid directories which are stored on secondary storage. We assume the main memory part of the directory can be really kept in main memory although the size might be very large.

The other structure participating in the comparison is the R*-tree. The data structure of the R*-tree is the same as for any other R-tree [Gut 84, Gre 89]. However, the R*-tree has completely different insertion and splitting algorithms. A survey of the algorithms and an comprehensive comparison of the different R-trees can be found in [BKSS 90]. The paper concluded that the R*-tree performs essentially better than any other R-tree structure.

To compare the performance of the five SAMs we selected six data files. Because the data chosen in [BKSS 90] is considered to be realistic (though they were mostly artificially generated), we decided to take these files for our performance comparison.

Each rectangle is assumed to be in the unit square $[0,1)^2$. The distribution of the rectangles is described by the distribution of their centers and by the triple $(n, \mu_{area}, \sigma_{area})$. Here n denotes the number of rectangles, μ_{area} and σ_{area} are the mean and variance value of the area of the the rectangles, respectively. A description of the data files follows:

F1 Uniform The centers of the rectangles follow a 2-dimensional independent uniform distribution. ($n = 100,000$, $\mu_{area} = .0001$, $\sigma_{area} = .9505$)

F2 Cluster The centers follow a distribution with 640 clusters, each cluster contains about 1600 objects. ($n = 99,968$, $\mu_{area} = .00002$, $\sigma_{area} = 1.538$)

F3 Parcel The unit square is decomposed into 100,000 disjoint rectangles. Then we consider these rectangles increased by a factor of 2.5.
($n = 100,000$, $\mu_{area} = .000025$, $\sigma_{area} = 30.3458$)

F4 Real-data These rectangles are the minimum bounding rectangles of some elevation lines taken from a real cartography map.
($n = 120,576$, $\mu_{area} = .000093$, $\sigma_{area} = 1.504$)

F5 Gaussian The centers follow a 2-dimensional (independent) Gaussian distribution. ($n = 100,000$, $\mu_{area} = .00008$, $\sigma_{area} = .8988$)

F6 Mixed-Uniform The centers of the rectangles follow a 2-dimensional independent uniform distribution. First we generate a set of 99,000 *small* rectangles and $\mu_{area} = .0000101$. Then we build up a set of 1,000 *large* rectangles and $\mu_{area} = .001$. Finally, we merge these sets together in one file.
($n = 100,000$, $\mu_{area} = .00002$, $\sigma_{area} = 6.778$)

4.1 Build up the files

At first, we give some performance measures for building up the files. The most important numbers are the average number of disk accesses for an insertion and the storage utilization (SU). Let us recall that the storage utilization is the number of rectangles actually stored in the data pages divided by the maximum number of rectangles which can be stored in the data pages. Due to replication of the data rectangles, clipping offers in general low storage utilization. The directory pages and the other overhead of the access method do not affect the storage utilization. Thus, we decided to add two additional parameters. One parameter gives the ratio of directory to data pages. The other parameter called real storage utilization (real SU) is the ratio of the number of bytes occupied by the rectangles to the number of bytes actually allocated by the SAM.

The results of the different data files are presented in the Tables 1-6. The first column indicates the type of SAM. The second column shows the average insertion costs measured by the number of disk accesses. In the tables and in the rest of the paper, we use the abbreviations CL, OR and TR for Clipping, Overlapping Regions and Transformation, respectively. As shown in the tables, some experiments were not performed for the grid file and the buddy-tree with CL. The reasons are given below:

- The grid file generated for the distribution Cluster and Parcel very large data files. After the grid files grow to more than 20 Mbytes (recall that the pure data files are about 1.6 Mbytes) we decided to quit the experiments. The reason for these large sizes can only be explained by a very large directory. We expect that the query performance is far away from all the other schemes.

- The buddy-tree in combination with CL was nearly a disaster for each of the data files. Aside from very large data files, the insertion costs were incredibly high. Finally, it was not possible for us to keep hold of a SUN/60 for several days only

SAM	avg. insertion	$\frac{\text{\# directory pages}}{\text{\# data pages}}$	SU in %	real SU in %
Buddy & OR	3.39	4.53	69.76	52.14
Buddy & CL	–	–	–	–
Buddy & TR	2.96	1.52	63.41	48.80
Grid File	3.08	5.08	38.02	28.27
R*-tree	4.42	2.73	75.84	57.68

Table 1: Build up the file F1 (Uniform)

SAM	avg. insertion	$\frac{\text{\# directory pages}}{\text{\# data pages}}$	SU in %	real SU in %
Buddy & OR	3.72	4.55	45.28	33.84
Buddy & CL	–	–	–	–
Buddy & TR	3.01	1.72	46.46	35.70
Grid File	–	–	–	–
R*-tree	3.77	2.71	72.15	54.09

Table 2: Build up the file F2 (Cluster)

to insert 100,000 rectangles into a file structure. There was only one successful run of a file which was tailored for clipping.

Now, let us discuss the insertion and storage costs for the SAMs. Surprisingly, the grid file requires on the average the fewest insertion costs. However, the storage utilization of the grid file is always lower than the one of the other schemes and the directory is large in relation to the data file, see for example Table 5. For the R*-tree, the buddy-tree with OR and the buddy-tree with TR, we averaged the performance over the six distributions. The results are shown in figure 7. The buddy-tree with TR requires the lowest insertion costs, but it has also the lowest storage utilization. And the R*-tree presents the other extreme: it offers the highest storage utilization, but requires the highest insertion costs. The buddy-tree with OR can be seen as a good compromise offering both, acceptable

SAM	avg. insertion	$\frac{\text{\# directory pages}}{\text{\# data pages}}$	SU in %	real SU in %
Buddy & OR	3.91	5.34	43.57	32.32
Buddy & CL	–	–	–	–
Buddy & TR	3.72	4.63	40.09	29.93
Grid File	–	–	–	–
R*-tree	10.73	2.61	72.52	55.21

Table 3: Build up the file F3 (Parcel)

SAM	avg. insertion	$\frac{\text{\# directory pages}}{\text{\# data pages}}$	SU in %	real SU in %
Buddy & OR	3.50	3.92	71.60	53.45
Buddy & CL	–	–	–	–
Buddy & TR	2.99	2.07	60.93	46.63
Grid File	3.08	6.51	45.35	33.26
R*-tree	4.22	2.51	70.51	53.74

Table 4: Build up the file F4 (Real)

SAM	avg. insertion	$\frac{\text{\# directory pages}}{\text{\# data pages}}$	SU in %	real SU in %
Buddy & OR	3.74	5.97	64.56	47.59
Buddy & CL	–	–	–	–
Buddy & TR	3.36	4.29	56.39	42.24
Grid File	3.12	7.13	35.11	25.60
R*-tree	9.15	2.47	73.82	56.29

Table 5: Build up the file F5 (Gaussian)

SAM	avg. insertion	$\frac{\text{\# directory pages}}{\text{\# data pages}}$	SU in %	real SU in %
Buddy & OR	3.42	4.63	69.76	52.08
Buddy & CL	5.68	1.57	48.44	37.26
Buddy & TR	2.97	2.36	58.36	44.54
Grid File	3.07	5.49	41.45	30.70
R*-tree	4.46	2.63	73.13	55.66

Table 6: Build up the file F6 (Mixed Uniform)

SAM	avg. insertion	SU in %	real SU in %
Buddy & OR	3.61	60.67	45.24
Buddy & TR	3.17	54.27	41.31
R*-tree	6.13	73.00	55.45

Table 7: Average performance for building up the files F1-F6

storage utilization and low insertion costs. Moreover, a high variance of the insertion costs appears in case of the R*-tree, but not in case of the buddy-trees. Thus, the buddy-tree offers a better dynamic behavior than the R*-tree. In addition to the access costs, CPU-time influences the total costs. The numbers we measured are not acceptable for publishing until now (because the implementations used different low level modules). However, the trend is that the buddy-tree with CL uses by far most CPU-time, followed by the buddy-tree with TR.

4.2 Queries

For the files (F1)-(F6) we generated a single query file containing 1,000 point queries, 400 rectangle intersection queries, 400 rectangle containment queries and 400 rectangle enclosure queries. The 1,000 point queries are uniformly distributed in the data space. The three groups of rectangle queries are performed with the same set of 400 rectangles. The center of the query rectangles are uniformly distributed in the data space. The ratio of the x-extension to the y-extension uniformly varies from 0.25 to 2.25. The 400 query rectangles consist of 4 groups of 100 rectangles where the area of the query rectangles varies from 0.001%, 0.01%, 0.1% to 1% relative to the area of the data space.

For each file (F1)-(F6) we measured the number of disc accesses per query. These numbers are averaged for the different types of queries. Moreover, for the rectangle intersection queries we averaged our results depending on the size of the query rectangles. Let us mention that the numbers for the queries are normalized by the numbers we obtained for the R*-tree. In the following the performance of the R*-tree is always given by 100%. In our tables we included an extra row with the average number of accesses required by the R*-tree. Hence, it is possible to calculate the average number of accesses for all the other SAMs. The results are presented in the Tables 8-13.

A summary of all the results is given in Table 14. The numbers are the (unweighted) average of the results presented in the other tables. For the grid file we averaged only over the four tables available.

Let us first discuss the performance of the three techniques implemented on top of the buddy-tree. Overall, OR is the winner in our performance comparison. For point queries, OR requires about 90% less disc accesses than TR, see Table 14. For intersection queries the performance gain of OR in comparison to TR declines with an increasing size of the query rectangles. This indicates that OR cluster the rectangles in a better fashion than TR. Obviously, the effect of clustering is more visible for queries with high

SAM	point	intersection				enclosure	containment
		0.001	0.01	0.1	1.0		
Buddy & OR	100.15	95.20	102.23	110.99	108.48	83.66	107.35
Buddy & CL	–	–	–	–	–	–	–
Buddy & TR	161.20	151.99	159.37	158.48	132.64	144.93	81.80
Grid File	934.89	832.95	772.61	597.92	349.66	1241.08	128.50
R*-tree	100.00	100.00	100.00	100.00	100.00	100.00	100.00
# accesses	5.26	6.04	7.63	13.92	53.42	2.86	20.25

Table 8: Queries on the file F1 (Uniform)

SAM	point	intersection				enclosure	containment
		0.001	0.01	0.1	1.0		
Buddy & OR	80.35	87.61	103.05	136.18	153. 72	66.20	144.89
Buddy & CL	–	–	–	–	–	–	–
Buddy & TR	103.75	119.47	129.15	151.33	155.53	63.56	117.54
Grid File	–	–	–	–	–	–	–
R*-tree	100.00	100.00	100.00	100.00	100.00	100.00	100.00
# accesses	2.00	2.26	2.95	7.13	36.00	1.42	12.09

Table 9: Queries on the file F2 (Cluster)

SAM	point	intersection				enclosure	containment
		0.001	0.01	0.1	1.0		
Buddy & OR	101.75	106.70	108.97	117.46	142.89	87.77	130.08
Buddy & CL	–	–	–	–	–	–	–
Buddy & TR	289.52	276.52	267.39	215.95	197.52	315.74	88.08
Grid File	–	–	–	–	–	–	–
R*-tree	100.00	100.00	100.00	100.00	100.00	100.00	100.00
# accesses	5.67	6.26	7.36	13.29	36.76	4.13	15.92

Table 10: Queries on the file F3 (Parcel)

SAM	point	intersection				enclosure	containment
		0.001	0.01	0.1	1.0		
Buddy & OR	93.93	96.98	97.82	103.82	101.97	81.12	101.65
Buddy & CL	–	–	–	–	–	–	–
Buddy & TR	183.79	175.24	174.56	162.80	132.23	178.11	91.75
Grid File	666.40	619.28	527.62	386.69	237.95	937.45	130.77
R*-tree	100.00	100.00	100.00	100.00	100.00	100.00	100.00
# accesses	4.78	5.29	7.35	14.65	60.84	2.49	22.03

Table 11: Queries on the file F4 (Real)

SAM	point	intersection				enclosure	containment
		0.001	0.01	0.1	1.0		
Buddy & OR	95.94	92.16	103.77	102.11	115.22	82.04	110.02
Buddy & CL	–	–	–	–	–	–	–
Buddy & TR	176.65	168.99	175.03	161.21	149.45	173.24	90.19
Grid File	907.62	723.68	674.12	604.41	372.46	1078.70	122.38
R*-tree	100.00	100.00	100.00	100.00	100.00	100.00	100.00
# accesses	4.83	5.87	7.69	10.88	46.19	2.70	17.66

Table 12: Queries on the file F5 (Gaussian)

SAM	point	intersection				enclosure	containment
		0.001	0.01	0.1	1.0		
Buddy & OR	94.24	93.10	95.74	102.47	103.44	80.08	101.83
Buddy & CL	41.15	45.01	53.23	81.03	113.47	70.18	99.44
Buddy & TR	172.26	169.51	160.39	153.71	135.67	162.11	95.06
Grid File	655.86	589.11	519.53	404.87	259.72	832.63	135.75
R*-tree	100.00	100.00	100.00	100.00	100.00	100.00	100.00
# accesses	4.86	5.51	7.27	13.76	52.06	2.85	19.65

Table 13: Queries on the file F6 (Mixed Uniform)

SAM	point	intersection				enclosure	containment
		0.001	0.01	0.1	1.0		
Buddy & OR	94.39	95.29	103.05	112.17	120.95	80.14	115.97
Buddy & TR	181.20	176.95	177.65	167.25	150.51	172.95	94.07
Grid File	791.19	691.26	623.47	498.47	304.95	1022.47	129.35
R*-tree	100.00	100.00	100.00	100.00	100.00	100.00	100.00

Table 14: Average query performance overall files

selectivity. As mentioned TR favors the containment query. Thus, it is not surprising that TR outperforms OR for this type of query. Due to high variance in the size of the rectangles, TR performs worst for file (F4), see Table 11. For the point queries in this experiment, TR performs three times worse than OR. The performance of CL is reported in Table 13. As mentioned above, the file (F6) is tailored for this technique. Therefore, it is not surprising that CL outperforms the other techniques. However, the gain in performance is less than expected. As mentioned, the number of disc accesses required for a point query corresponds to the height of the tree. Because the tree has height two, a point query is answered in two disk accesses. With an increasing size of the query rectangle, the performance gain for rectangle intersection queries becomes less in comparison to the other methods. For the 1% rectangle intersection query it is even worse than the buddy-tree with OR. This behavior indicates that many duplicates are in the response set.

Because query performance of TR is not too far from the one of OR and TR can be easily implemented, TR still might be used until more sophisticated techniques are implemented. However, our results in Table 14 reveal the need of a robust PAM. Although the worst files are omitted, the performance of the grid file with TR is up to six times worse than the buddy-tree with TR, see for example Table 8. Obviously, the special properties of the buddy-tree pay off. The most important one is that the buddy-tree does not reflect empty space in its directory. Overall we state that the grid file with the transformation technique (using the corner-representation) is not an appropriate SAM. Let us mention that the center-representation was originally introduced for the grid file to smooth out the skew of the data distribution.

Eventually, let us compare the R*-tree and the buddy-tree with OR. Overall, our comparison demonstrates that for small query regions the buddy-tree with OR is superior to the R*-tree, see Table 14. With increasing size of the query region the R*-tree becomes better due to its higher storage utilization. The worst performance of the buddy-tree with OR occur for the file (F2), see Table 9. For this file, the 1% rectangle intersection query requires 50% more accesses than in case of the R*-tree. However, with respect to its dynamic behavior, the buddy-tree with OR is a serious competitor of the R*-tree. Let us mention that both methods performs essentially better than the previous versions of R-trees (see [BKSS 90] for details).

5 Conclusion

In this paper we investigated segment access methods (SAMs) suitable for the organization of one- and multidimensional segments. We presented several SAMs implemented on top of the buddy-tree, an access method originally designed for the organization of multidimensional points. Each of the SAMs used one of the following techniques: clipping, transformation and overlapping regions. The implementation issues were discussed in detail and furthermore, design alternatives were revealed. Finally, we made an experimental performance comparison with some popular competitors, such as the grid-file and the R*-tree.

The performance comparison allows to judge the basic three different techniques because they were implemented on top of a single point access method (in our case the buddy-tree). The technique of overlapping regions seems to be not suitable: low storage utilization and high insertion cost have to be paid without gaining (essentially) in query performance. Overall, the technique of overlapping regions was the most efficient one. However, the performance of the transformation technique is not far off. With respect to its simple implementation, this technique can be used until more sophisticated algorithms are implemented. However, the choice of the underlying point access method is important for the transformation technique. The grid file offers worse performance in order of magnitudes in comparison to the buddy-tree. Furthermore, we have shown that the buddy-tree with the technique of overlapping regions offers a similar retrieval performance and an essentially better dynamic behavior (i.e. less costs for insertions and deletions) than the R*-tree.

After the implementation of the basic data and storage structure we are going to implement a spatial data base system on top of our data structures. Until now, we developed a spatial query processor and a very general storage model. The next step will be the design of a spatial object algebra with a graphical interface on top.

Acknowledgements

This work was supported by a grant of the German Research Society (DFG). I thank H.-P. Kriegel for his support and the students H.-J. Forst, T. Otten and O. Reins for their implementation efforts in realizing some of the segment access methods. Furthermore, I wish to thank D. Bradshaw for reviewing a first version of the paper.

References

[BKSS 90] N. Beckmann, H. P. Kriegel, R. Schneider, B. Seeger: 'The R*-tree: an efficient and robust access method for points and rectangles', Proc. ACM SIGMOD Int. Conf. on Management of Data, 322-331, 1990

[Fre 87] M. Freeston: 'The BANG file: a new kind of grid file', Proc. ACM SIGMOD Int. Conf. on Management of Data, 260-269, 1987

[FOR 90] H.-J. Frost, T. Otten and O. Reins: 'Improvements on the buddy-tree', master thesis (in German), University of Bremen, 1990

[Gre 89] D. Greene: 'An implementation and performance analysis of spatial data access methods', Proc. 5th Int. Conf. on Data Engineering, 606-615, 1989

[Gut 84] A. Guttman: 'R-trees: a dynamic index structure for spatial searching', Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984

[HCKW 90] E. N. Hanson, M. Chaabouni, C.-H. Kim, Y.-W. Wang: 'A predicate matching algorithm for database rule systems', Proc. ACM SIGMOD Int. Conf. on Management of Data, 271-280, 1990

[HSW 89] A. Henrich, H.-W. Six and P. Widmayer: 'The LSD-tree: Spatial access to multidimensional point- and non-point objects', 15th Int. Conf. on Very Large Data Bases, 45-53, 1989

[Hin 85] K. Hinrichs: 'The grid file system: implementation and case studies for applications', Dissertation Nr. 7734, Eidgenössische technische Hochschule (ETH) Zürich, 1985

[HSW 90] A. Hutflesz, H.-W. Six and P. Widmayer: 'The R-file: An efficient access structure for proximity queries', Proc. 6th Int. Conf. on Data Engineering, 372-379, 1990

[KSSS 89] H.P. Kriegel, M. Schiwietz, R. Schneider, B. Seeger: 'A performance comparison of multidimensional point and spatial access methods', Proc. of the Symp. on Large Spatial Data Bases, Lecture Notes in Comp. Science No. 409, Springer-Verlag, 1990

[KHHSS 91] H.P. Kriegel, P. Heep, S. Heep, M. Schiwietz, R. Schneider: 'An access method based query processor for spatial database systems', technical report, University of Bremen, 1991

[LS 90] D.B. Lomet, B. Salzberg: 'The hB-tree: a robust multi-attribute indexing method', ACM Trans. on Database Systems, Vo. 15, 4, 1989

[LS 89] D.B. Lomet, B. Salzberg: 'Access Methods for multiversion data', Proc. ACM SIGMOD Int. Conf. on Management of Data, 315-324, 1989

[NHS 84] J. Nievergelt, H. Hinterberger, K.C. Sevcik: 'The grid file: an adaptable, symmetric multikey file structure', ACM Trans. on Database Systems, Vol. 9, 1, 38-71, 1984

[Ooi 87] B.C. Ooi: 'A data structure for geographic database', Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Informatik Fachbericht 136, 247-258, 1987

[Ore 82] J. A. Orenstein: 'Multidimensional tries used for associative searching', Information Processing Letters, 14, 4, 150-157, 1982

[Ore 86] J. A. Orenstein: 'Query processing in object-oriented data base systems', Proc. ACM SIGMOD Int. Conf. on Management of Data, 1986

[Rob 81] J. T. Robinson: 'The K-D-B-tree: a search structure for large multidimensional dynamic indexes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 10-18, 1981

[Sam 89] H. Samet: 'The design and analysis of spatial data structures', Addison Wesley, 1989

[SK 88] B. Seeger, H. P. Kriegel: 'Design and implementation of spatial access methods', Proc. 14th Int. Conf. on Very Large Data Bases, 360-371, 1988

[SK 90] B. Seeger, H. P. Kriegel: 'The buddy-tree: an efficient and robust access method for spatial data base systems', Proc. 16th Int. Conf. on Very Large Data Bases, 590-601, 1990

[SRF 87] T. Sellis, N. Roussopoulos, C. Faloutsos: 'The R^+-tree: a dynamic index for multidimensional objects', Proc. 13th Int. Conf. on Very Large Database, 1987

[SW 88] H.-W. Six, P. Widmayer: 'Spatial Searching in Geometric Databases', Proc. 4th Int. Conf. on Data Engineering, 496-503, 1988

[TS 82] M. Tamminen, R. Sulonen: 'The Excell method for efficient geometric access to data', Proc. 19th ACM Design Automation Conf. 345-351, 1982