Chapter T2.3 in Geographical Information Systems Principles, Technical Issues, Management Issues, and Applications (edited by ongley, Goodchild, Maguire en Rhind), Wiley pages 385-400 (vol.1), 1999.

Spatial Access Methods

Peter van Oosterom, Senior Information Manager, Company Staff, Cadastre Netherlands, P.O. Box 9046, 7300 GH Apeldoorn, The Netherlands
(Research interests: spatial databases, spatial algorithms, and map generalization. Telephone +31 55 5285163, fax +31 55 3557931; email oosterom@kadaster.nl)

1 Introduction

This chapter first recaptures why spatial access methods are needed. It is important to note that spatial access methods are not only useful for spatial data. In a database that contains information about persons with income and age, one could select all persons with income between $10,000 and $15,000 and age between 20 and 25 years. This can be regarded as a spatial query on point data (x=income, y=age), which could be supported by a spatial access method. Some early main memory spatial access methods are described (Section 3), followed by an overview of space filling curves (Section 4). As it is impossible to present all spatial access methods described in recent literature, only the following characteristic families are presented: Quadtree, Grid based methods, and R-tree (Sections 5-7). Special attention is paid to spatial access methods taking multiple scales into account (Section 8). Besides the theory of spatial access methods, the issue of using them in a database in practice is treated in the conclusion of this Chapter (Section 9).

2 Why are spatial access methods needed?

The main purpose is to support efficient spatial selection, for example range queries or nearest neighbour queries, of spatial objects. Further, spatial access methods are also used to implement efficient spatial analysis such as map overlay, and other types of spatial joins. A characteristic of spatial data sets is that they are usually large and that the data is quite often distributed in an irregular manner. A spatial access method needs to take into account both spatial indexing and clustering techniques. Without a spatial index, every object in the database has to be checked whether it meets the spatial selection criterion; a 'full table scan' in a relational database. As spatial data sets are usually very large, this is unacceptable in practice for interactive use and most other applications. Therefore, a spatial index is required, which enables to find the required object addresses efficiently without looking at every object. In case the whole spatial data set resides in main memory it is sufficient to know the addresses of the requested objects, as main memory storage allows random access and does not introduce significant delays. However, most spatial data sets are so large they can not reside in the main memory of the computer and must be stored on secondary memory; e.g. hard disk. Clustering is needed to group those objects which are often requested together. Otherwise, many different disk pages will have to be fetched, resulting in slow response. For spatial selecting the clustering implies storing objects which are close together in reality also close together in the computer memory (instead of scattered over the whole memory). Solutions for this are storing the objects in the order implied by the index traversal (of the whole domain) or try to achieve some kind of spatial clustering based on a space filling curve.

In traditional database systems sorting (or ordering) the data is the basis for efficient searching later on, as used in the B-tree approach. Higher dimensional data can not be sorted in an obvious manner, as it is possible for text strings, numbers, or dates (one-dimensional data). Basically, computer memory is one-dimensional. However, spatial data is 2D, 3D (or even higher) and must be organized somehow in the memory. An intuitive solution to organize the data is using a regular grid just as on a paper map. Each grid cell has a unique name; e.g. 'A3', 'C6', or 'D5'. The cells are stored in some order in the memory and can each contain a (fixed) number of object references. In a grid cell, a reference is stored to an object whenever the object (partially) overlaps the cell. However, this will not be very efficient due to the irregular data distribution of spatial data: many cells will be empty, e.g. in the ocean, while many other cells will be overfull, e.g. in the city center. Therefore, more advanced techniques have been developed.

3 Main memory access methods

Though originally not designed for handling very large data sets, the main memory data structures show several interesting techniques with respect to handling spatial data. In this section the KD-tree (adaptive, bintree) and the BSP-tree will be illustrated.

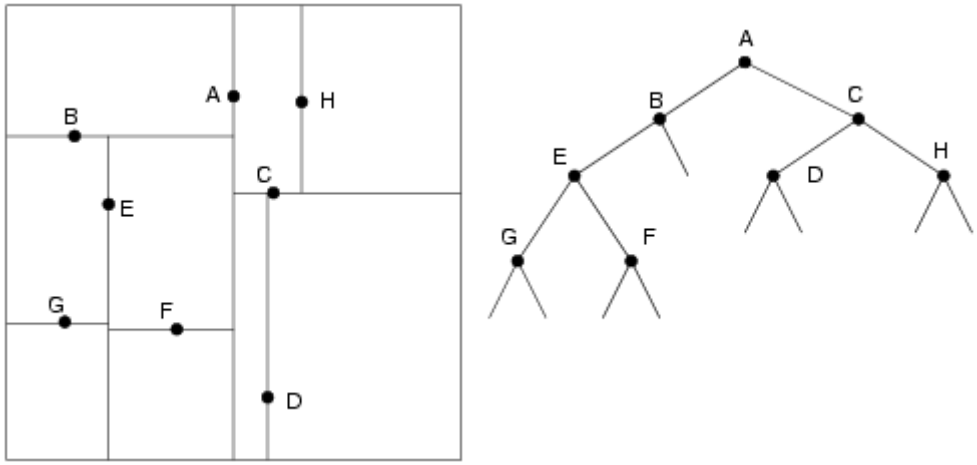

Fig. 1: KD-tree

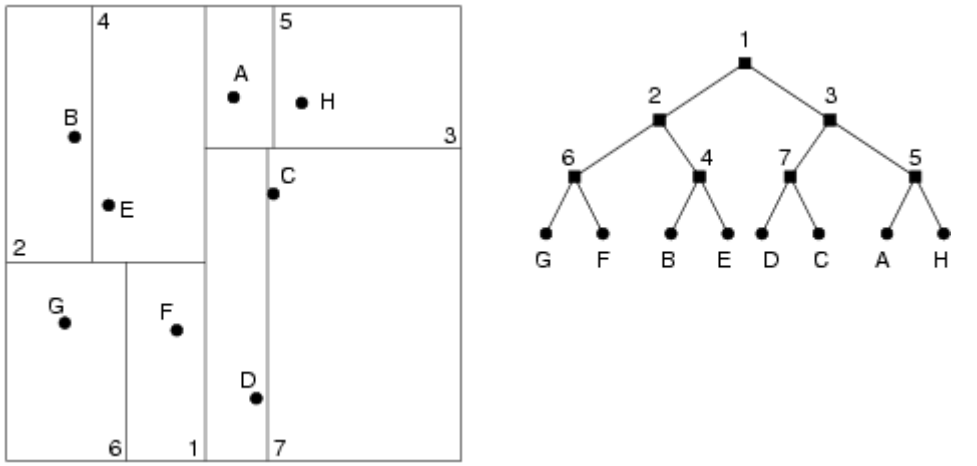

Fig. 2: Adaptive KD-tree

3.1 The KD-tree

The basic form of the KD-tree stores K-dimensional points (Bentley 1975). This section concentrates on the two-dimensional (2D) case. Each internal node of the KD-tree contains one point and also corresponds to a rectangular region. The root of the tree corresponds to the whole region of interest. The rectangular region is divided into two parts by the x-coordinate of the stored point on the odd levels and by the y-coordinate on the even levels in the tree; see Fig. 1. A new point is inserted by descending the tree until a leaf node is reached. At each internal node the value of the proper coordinate of the stored point is compared with the corresponding coordinate of the new point and the proper path is chosen. This continues until a leaf node is reached. This leaf also represents a rectangular region, which in turn will be divided into two parts by the new point. The insertion of a new point results in one new internal node. Range searching in the KD-tree starts at the root, checks whether the stored node (split point) is included in the search range and whether there is overlap with the left and/or right subtree. For each subtree which overlaps the search region, the procedure is repeated until the leaf level is reached.

A disadvantage of the KD-tree is that the shape of the tree depends on the order in which the points are inserted. In the worst-case, a KD-tree of n points has n levels. The adaptive KD-tree (Bentley and Friedman 1979) solves this problem by choosing a splitting point (which is not an element of the input

set of data points), which divides the set of points into two sets of (nearly) equal size. This process is repeated until each set contains one point at the most; see Fig. 2. The adaptive KD-tree is not dynamic: it is hard to insert or delete points while keeping the tree balanced. The adaptive KD-tree for n points can be built in O(n log n) time and takes O(n) space for K=2. A range query takes O(sqrt(n)+t) time in 2D where t is the number of points found. Another variant of the KD-tree is the bintree (Tamminen 1984). Here the space is divided into two equal-sized rectangles instead of two rectangles with equal numbers of points. This is repeated until each leaf contains one point at the most.

The modification that makes the KD-tree suitable for secondary memory is described in (Robinson 1981) and is called the KDB-tree. For practical use, it is more convenient to use leaf nodes containing more than one data point. The maximum number of points that a leaf may contain is called the bucket size. The bucket size is chosen in such a way that it fits within one disk page. Also, internal nodes are grouped and each group is stored on one page in order to minimize the number of disk accesses. Robinson describes algorithms for deletions and insertions under which the KDB-tree remains balanced. Unfortunately, no reasonable upper bound for memory usage can be guaranteed.

Matsuyama **et al** (1984) show how the geometric primitives polyline and polygon may be incorporated using the centroids of a bounding box in the 2D-tree. Rosenberg (1985) uses a 4D-tree to store a bounding box by putting the minimum and maximum points together in one 4D point. This technique can be used to generalize other geometric data structures that are originally suited only for storing and retrieving points. The technique works well for exact match-queries, but is often more complicated in the case of range queries. In general, geometrically close 2D rectangles do not map into geometrically close 4D points (Hutflesz, Six, and Widmayer 1990). The ranges are transformed into complex search regions in the higher-dimensional space, which in turn result in slow query responses.

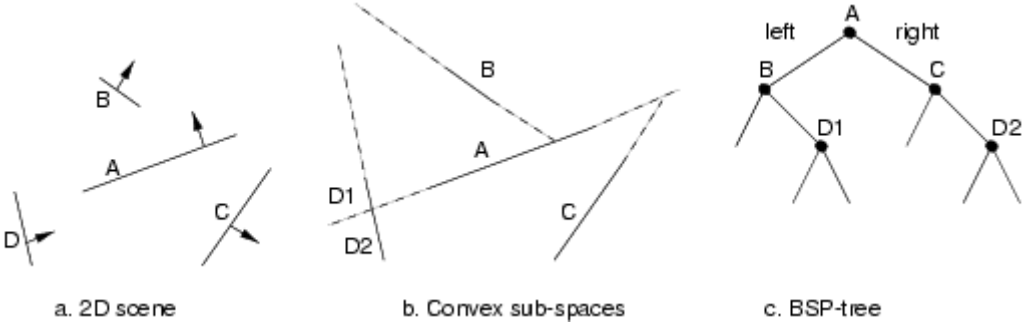

Fig. 3: The Building of a BSP-tree

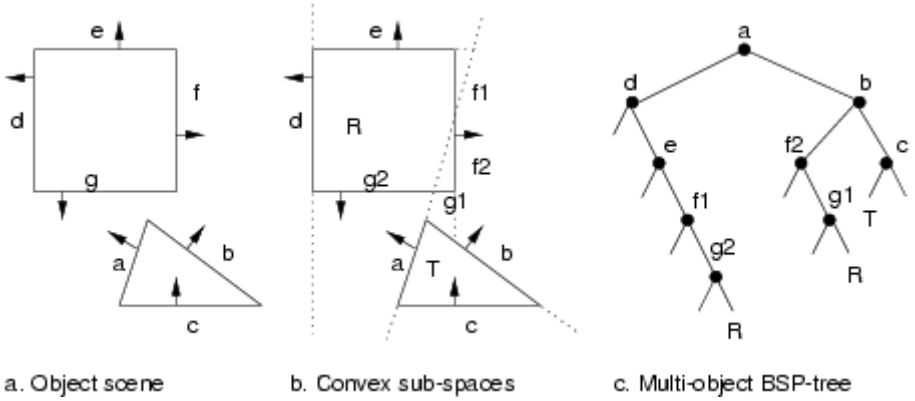

Fig. 4: The Building of a Multi-Object BSP-tree

3.2 The BSP-tree

This section first describes the Binary Space Partitioning (BSP)-tree, then a variant for GIS applications is given: the multi-object BSP-tree for storing polylines and polygons. The original use of the BSP-tree was in three-dimensional (3D) computer graphics (Fuchs, Abram, and Grant 1983,

Fuchs, Kedem, and Naylor 1980). The BSP-tree was used by Fuchs to produce a hidden surface image of a static three-dimensional scene. After a pre-processing phase it is possible to produce an image from any view-angle in O(n) time, with n the number of polygons in the BSP-tree.

In this chapter the two-dimensional BSP-tree is used for the structured storage of geometric data. It is a data structure that is not based on a rectangular division of space. It uses the line segments of the polylines and the edges of the polygons to divide the space in a recursive manner. The BSP-tree reflects this recursive division of space. Each time a (sub) space is divided into two subspaces by a so-called splitting primitive, a corresponding node is added to the tree. The BSP-tree represents an organization of space by a set of convex subspaces in a binary tree. This tree is useful during spatial search and other spatial operations. Fig. 3 [a] shows a scene with some directed line segments. The `left' side of the line segment is marked with an arrow. From this scene, line segment a is selected and space is split into two parts by the supporting line of a, indicated by a dashed line in Fig. 3 [b]. This process is repeated for each of the two subspaces with the other line segments. The splitting of space continues until there are no line segments left. Note that sometimes the splitting of a space implies that a line segment (which has not yet been used for splitting itself) is split into two parts. Line d, for example, is split into d1 and d2. Fig. 3 [b] shows the resulting organization of the space, as a set of (possibly open) convex subspaces. The corresponding BSP-tree is drawn in Fig. 3 [c].

The BSP-tree, as discussed so far, is suited only for storing a collection of (unrelated) line segments. In GIS it must be possible to represent objects, for example a polygon. The multi-object BSP-tree (van Oosterom 1990) is an extension to the BSP-tree to cater for object representation. It stores the line segments that together make up the boundary of the polygon. The multi-object BSP-tree has explicit leaf nodes which correspond to the convex subspaces created by the BSP-tree. Fig. 4 [a] presents a 2D scene with two objects, triangle T with sides abc, and rectangle R with sides defg. The method divides the space in the convex subspaces of Fig. 4 [b]. The BSP-tree of Fig. 4 [c] is extended with explicit leaf nodes, each representing a convex part of the space. If a convex subspace corresponds to the `outside' region, no label is drawn in the figure. If no more than one identification tag per leaf is allowed, only mutually exclusive objects can be stored in the multi-object BSP-tree, otherwise it would be possible also to deal with objects that overlap. A disadvantage of this BSP-tree is that the representation of one object is scattered over several leaves, e.g. rectangle R in Fig. 4. The (multi-object) BSP-tree allows efficient implementation of spatial operations, such as pick and rectangle search.

The choice of which line segment to use for dividing the space very much influences the building of the tree. It is preferable to have a balanced BSP-tree with as few nodes as possible. This is a very difficult requirement to fulfill, because balancing the tree requires that line segments from the middle of the data set be used to split the space. These line segments will probably split other line segments. Each split of a line segment introduces an extra node in the BSP-tree. However, in (Paterson and Yao 1989) it is proven that, if the original line segments are disjoint, then it is possible to build a BSP-tree with O(n log n) nodes and depth O(log n) using an algorithm requiring only O(n log n) time.

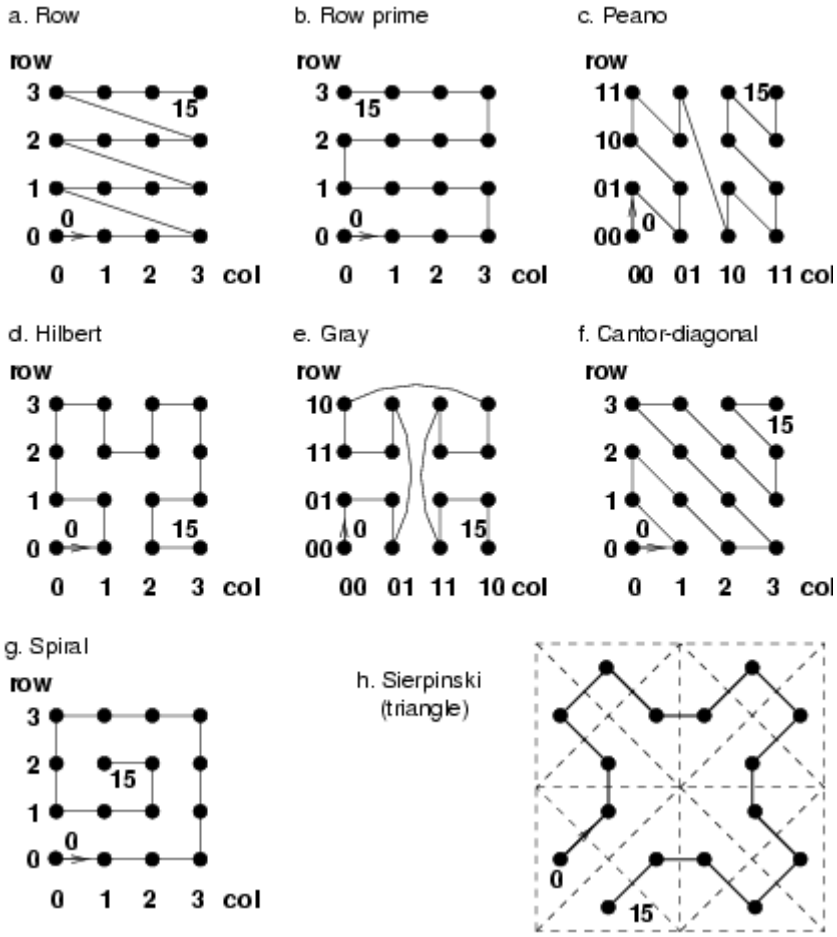

Fig. 5: Eight Different Orderings

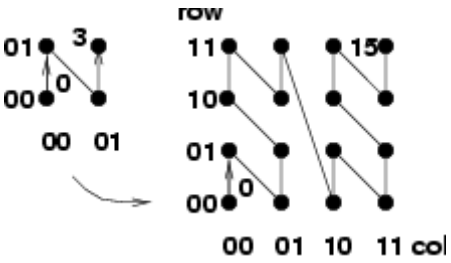

Fig. 6: Geometric Construction of the Peano Curve

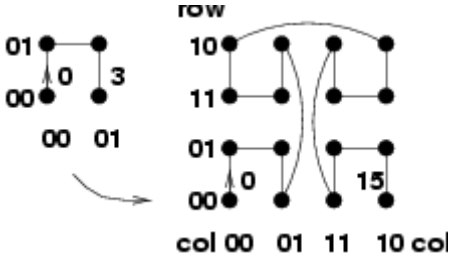

Fig. 7: Geometric Construction of the Gray Curve

4 Space filling curves

This section presents an overview and some properties of space filling curves. Space filling curves order the points in a discrete two-dimensional space. This technique is also called tile indexing. It transforms a two-dimensional problem into a one-dimensional one, so it can be used in combination with a well-known data structure for one-dimensional storage and retrieval, e.g. the B-tree (Bayer and McCreight 1973). The presentation in this subsection is based on several papers (Abel and Mark 1990, Goodchild and Grandfield 1983, Jagadish 1990, Nulty and Barholdi, III, 1994) and on the book of Samet (Samet 1989).

The row ordering simply numbers the cells row by row, and within each row the points are numbered from left to right; see Fig. 5 [a]. The row prime (or snake-like, or boustrophedon) ordering is a variant in which alternate rows are traversed in opposite directions; see Fig. 5 [b]. Obvious variations are column and column prime orderings in which the roles of row and column are transposed. Bitwize interleaving of the two coordinates results in a one-dimensional key, called the Morton key (Orenstein and Manola 1988). The Morton key is also known as Peano key, or N-order, or Z-order. For example, row 2 = 10_{bin} column 3 = 11_{bin} has Morton key 13 = 1101_{bin}; see Fig. 5 [c]. Hilbert ordering is based on the classic Hilbert-Peano curve, as drawn in Fig. 5 [d]. Gray ordering is obtained by bitwize interleaving the Gray-codes of the x and y coordinates. As Gray codes have the property that successive codes differ in exactly one bit position, a 4-neighbour cell only differs in one bit; see Fig. 5 [e] (Faloutsos 1988). In Fig. 5 [f] the Cantor-diagonal ordering is shown. Note that the numbering of the points is adapted to the fact that we are dealing with a space that is bounded in all directions; e.g., point (1,3) got order number 10 instead of 11 and point (3,3) got 15 instead of 24. The spiral ordering is depicted in Fig. 5 [g]. Finally, Fig. 5 [h] shows the Sierpinski curve, which is based on a recursive triangle subdivision.

Fig. 6 shows the geometric construction of the Peano curve: at each step of the refinement each vertex of the basic curve is replaced by the previous order curve. A similar method, but now also including reflection is used to geometrically construct the reflected binary gray curve; see Fig. 7. The Hilbert curve is constructed by rotating the previous order curves at vertex 0 with -90 degrees and at vertex 3 with 90 degrees; see Fig. 8. The Sierpinski curve starts with two triangles, each triangle is again split into two new triangles and this is repeated until the required resolution is obtained. Note that the orientation and ordering of the triangles is important; see Fig. 9. Abel and Mark (1990) have identified desirable qualitative properties of spatial orderings:

1. An ordering is continuous if, and only if, the cells in every pair with consecutive keys are 4-neighbours.
2. An ordering is quadrant-recursive if the cells in any valid Quadtree subquadrant of the matrix are assigned a set of consecutive integers as keys.
3. An ordering is monotonic if, and only if, for every fixed x, the keys vary monotonically with y in some particular way, and vice versa.
4. An ordering is stable if the relative order of points is maintained when the resolution is doubled.

Ordering techniques are very efficient for exact match queries for points, but there is quite a difference in their efficiency for other types of geometric queries, e.g. a range query. Abel and Mark (1990) conclude from their practical comparative analysis of five orderings (they do not consider the Cantor-diagonal, the Spiral and the Sierpinski orderings) that the Morton ordering and the Hilbert ordering are, in general, the best alternatives. Some quantitative properties of curves are: the total length of the curve, the variability in unit lengths (path between two cells next in order), the average of the average distance between 4 neighbours, and the average of the maximum distance between 4 neighbours. Goodchild (1989) proved that the expected difference of 4-neighbour keys of an n by n matrix is (n+1)/2 for Peano, Hilbert, row, and row-prime orderings. This does not seam to be a very discriminating property. Therefore, Faloutsos and Roseman (1989) try to give a better measure for spatial clustering: average (Manhattan) maximum distance of all cells within N/2 key value of a given cell on a N*N grid; see Table 1. Another measure for clustering is the average number of clusters for all possible range queries. Note that a cluster is defined as a group of cells with consecutive key value; see Table 2 and Fig. 10.

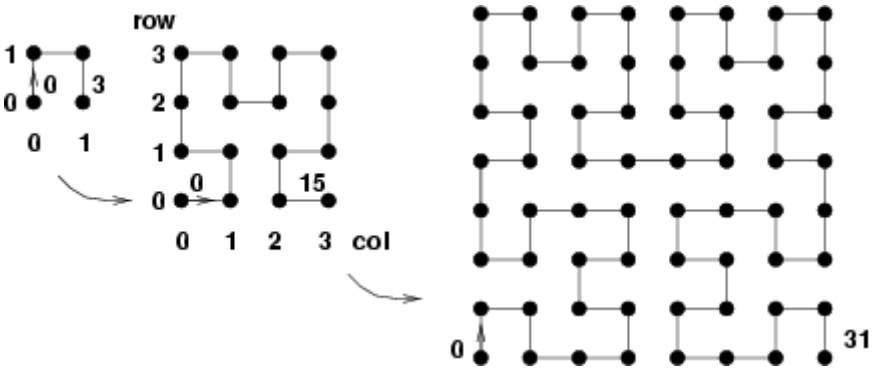

Fig. 8: Geometric Construction of the Hilbert Curve

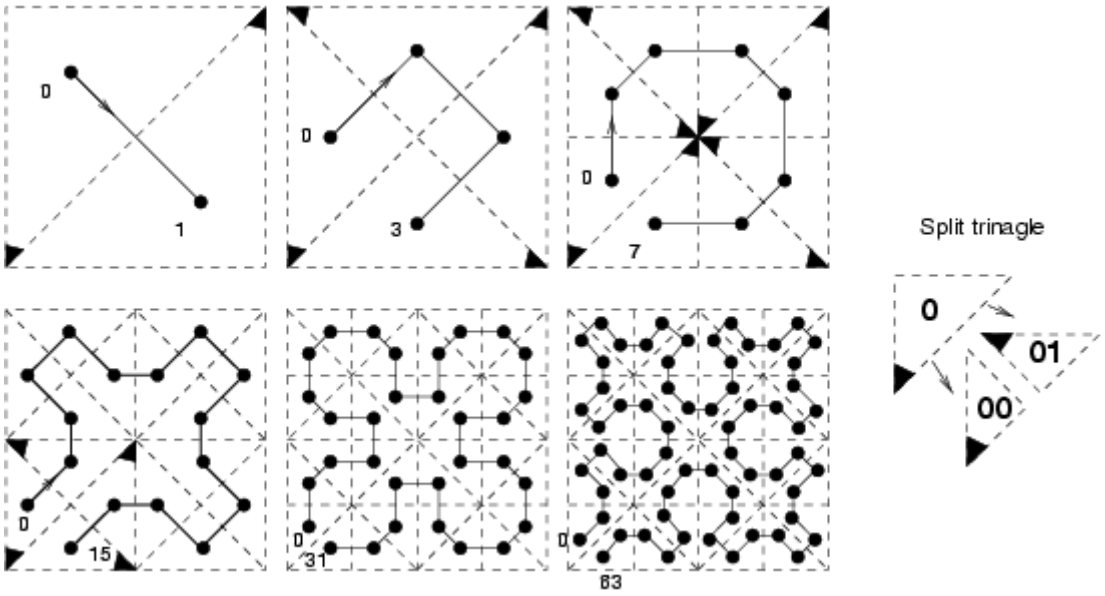

Fig. 9: Geometric Construction of the Sierpinski Curve

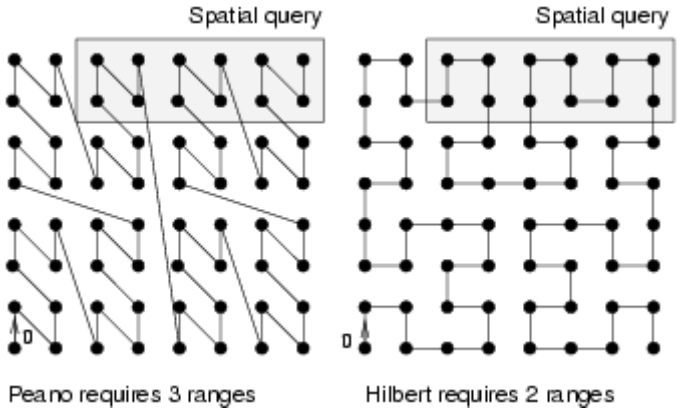

Fig. 10: Number of Clusters for a Given Range Query

N*N GRID	HILBERT	PEANO	I
2*2	1.00	1.50	.50
4*4	2.00	2.75	.75
8*8	3.28	4.84	1.56
16*16	4.89	7.91	3.02

Table 1: Average Manhattan maximum distance of cell within N/2 key value (results from Faloutsos and Roseman 1989)

N*N GRID	HILBERT	PEANO	I
2*2	1.11	1.22	.11
4*4	1.64	2.16	.52
8*8	2.93	4.41	1.48
16*16	5.60	9.29	3.69

Table 2: Average number of clusters for all possible range queries (results from Faloutsos and Roseman 1989)

5 Quadtree-family

The quadtree is a generic name for all kinds of trees that are built by recursive division of space into four quadrants. Several different variants have been described in literature of which the following will be presented here: Point quadtree, PR quadtree, Region quadtree, and PM quadtree. Samet (1984, 1989) gives an excellent overview, of which this section is a partial abstract.

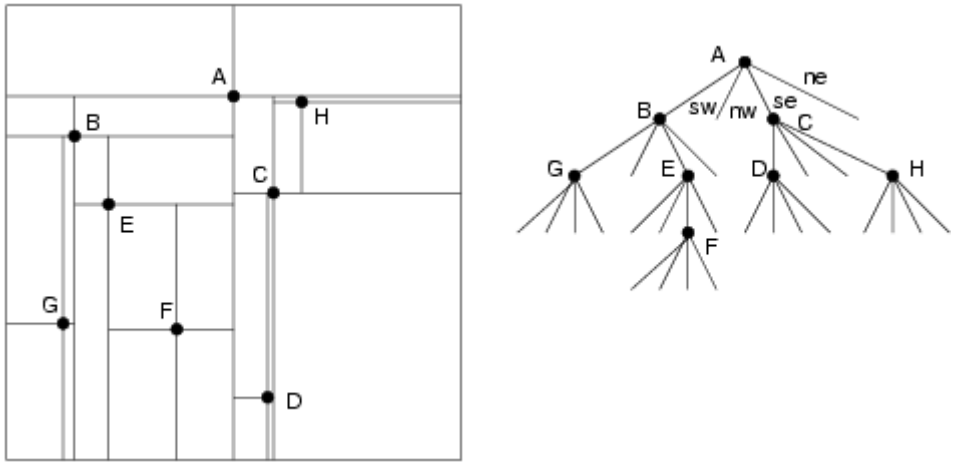

Fig. 11: Point Quadtree

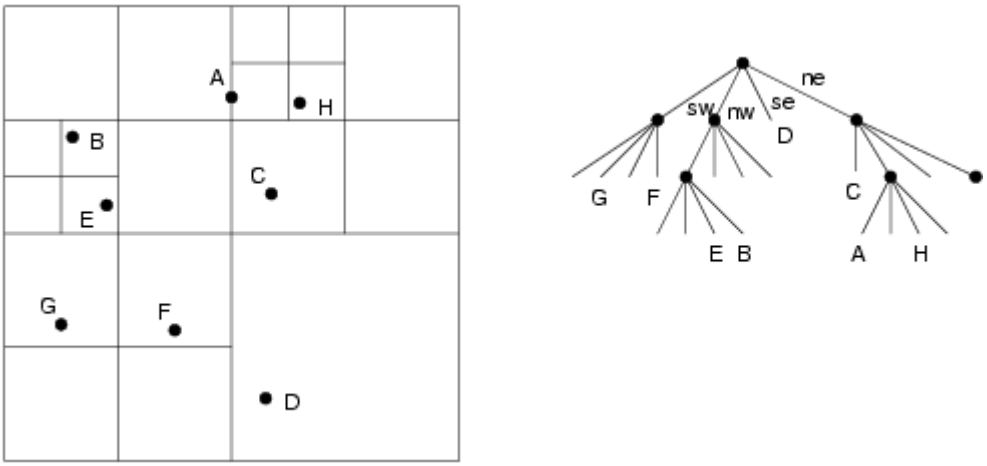

Fig. 12: PR Quadtree

5.1 Point quadtree and PR quadtree

The point Quadtree resembles the KD-tree described in Section 3. The difference is that the space is divided into four rectangles instead of two; see Fig. 11. The input points are stored in the internal nodes of the tree. The four different rectangles are typically referred to as SW (south-west), NW (north-west), SE (south-east), and NE (north-east). Searching in the Quadtree is very similar to the KD-tree: whenever a point is included in the search range it is reported and whenever a subtree overlaps with the search range it is traversed.

A small variation of the point Quadtree is the PR Quadtree (point region), which does not use the points of the data set to divide the space. Each time, it divides the space, a square, into four equal subsquares, until each contains no more than the given bucket size (e.g. 1 object). Note that dense data regions require more partitions and therefore the quadtree will not be balanced in this situation; see Fig. 12.

5.2 Region quadtree

A very well-known Quadtree is the region Quadtree, which is used to store a rasterized approximation of a polygon. First, the area of interest is enclosed by a square. A square is repeatedly divided into four squares of equal size until it is completely inside (a black leaf) or outside (a white leaf) the polygon or until the maximum depth of the tree is reached (dominant colour is assigned to the leaf); see Fig. 13. The main drawback is that it does not contain an exact representation of the polygon. The same applies if the region Quadtree is used to store points and polylines. This kind of Quadtree is useful for storing raster data.

5.3 PM quadtree

A polygonal map, a collection of polygons, can be represented by the PM Quadtree. The vertices are stored in the tree in the same way as in the PR Quadtree. The edges are segmented into q-edges which completely fall within the squares of the leaves. There are seven classes of q-edges. The first class of q-edges are those that intersect one boundary of the square and meet at a vertex within that square. The other six classes intersect two boundaries and are named after the boundaries they intersect: NW, NS, NE, EW, SW and SE. For each non-empty class, the q-edges are stored in a balanced binary tree. The first class is ordered by an angular measure and the other six classes are ordered by their intercepts along the perimeter. Fig. 14 shows a polygonal map and the corresponding PM Quadtree. The PM Quadtree provides a reasonably efficient data structure for performing various operations: inserting an edge, point-in-polygon testing, overlaying two maps, range searching and windowing.

The extensive research efforts on Quadtrees in the last decade resulted in more variants and algorithms to manipulate these Quadtrees efficiently. For example, the CIF Quadtree which is a Quadtree particularly suited for rectangles. Another interesting example is the linear Quadtree: in this

representation there is no explicit Quadtree, but only an enumeration of the quadcodes belonging to the object; e.g. 0, 10, 12, and 300 in Fig. 13. Descriptions of them can be found in Rosenberg (1985) and Samet (1984).

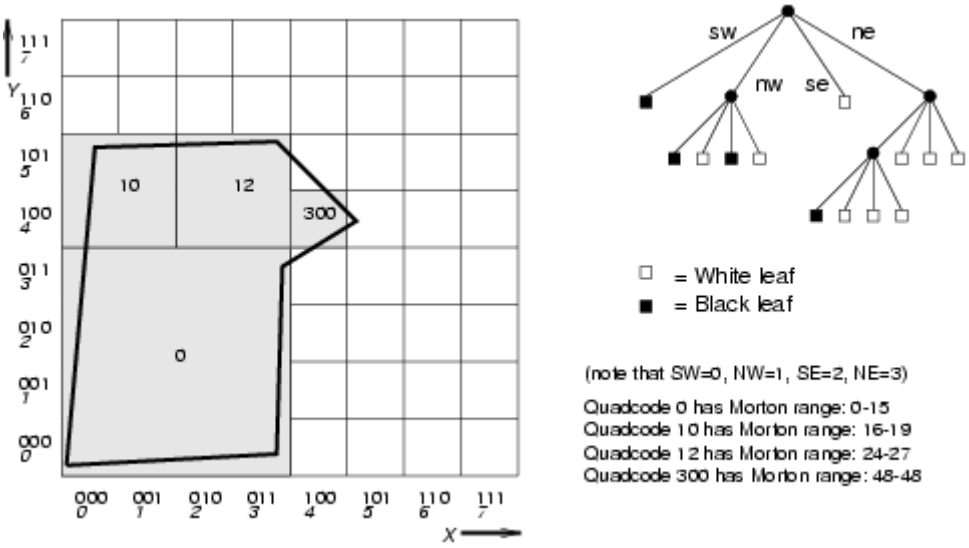

Fig. 13: Region Quadtree

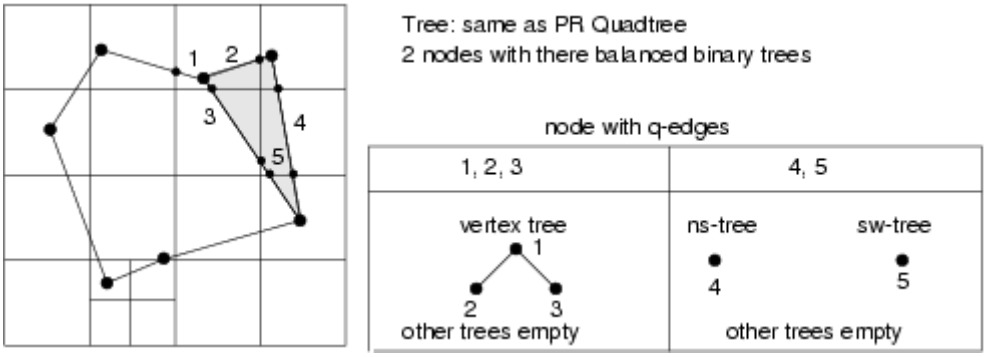

Fig. 14: PM Quadtree

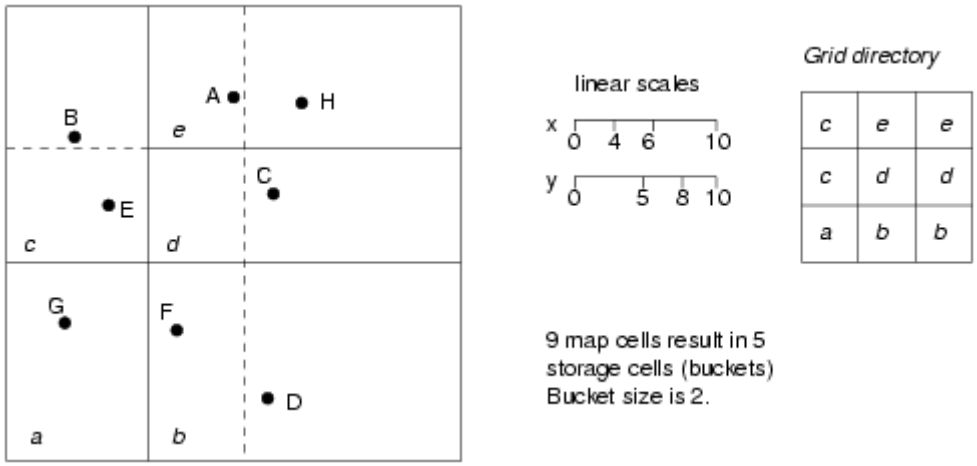

Fig. 15: The Grid File

6 Grid based methods

The intuitively attractive approach to organize space by imposing a regular grid has been refined in several different manners in order to avoid the problems when dealing with irregular distributed data. In this section two different approaches are described: the grid file and the Field-tree.

6.1 The Grid File

The principle of a Grid File is the division of the space into rectangles (regular tiles, grids, squares, cells) that can be identified by two indices, one for the x-direction and one for the y-direction. The Grid File is a non-hierarchical structure. The geometric primitives are stored in the grids, which are not necessarily of equal size. There are several variants of this technique. In this subsection the file structure as defined by Nievergelt **et al** (1984) is described.

The advantage of the Grid File as defined by Nievergelt is that it adjusts itself to the density of the data unlike most other Grid Files, but it is also more complicated. The cell division lines need not be equidistant; for x and y there is a one-dimensional array (in main memory) with the linear scales, the actual sizes of the cells. Neighbouring cells may be joined into one bucket if the resulting area is a rectangle. The buckets have a fixed size and are stored on a disk page. The grid directory is a two-dimensional array, with a pointer for each cell to the correct bucket. Fig. 15 shows a Grid File with linear scales and grid directory. The Grid File has good dynamic properties. If a bucket is too full to store a new primitive and it is used for more than one cell, then the bucket may be divided into two buckets. This is a minor operation. If the bucket is used for only one cell, then a division line is added to one of the linear scales. This is a little more complex but still a minor operation. In case of a deletion of primitives, the merging process is performed analogous to the splitting process for insertion.

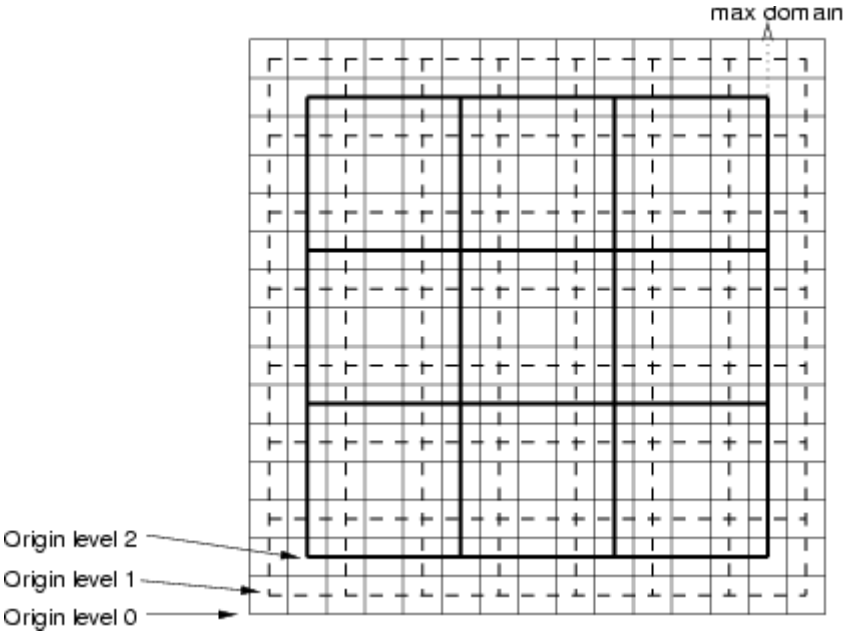

Fig. 16: The Positioning of Geometric Objects in the Field-tree

6.2 The Field-tree

The Field-tree is suited to store points, polylines, and polygons in a non-fragmented manner. Several variants of the Field-tree have been published by Frank **et al** (Frank 1983, Frank and Barrera 1989, Kleiner and Brassel 1986). In this subsection attention will be focused on the Partition Field-tree. Conceptually, the Field-tree consists of several levels of grids, each with a different resolution and a different displacement/origin; see Fig. 16. A grid cell is called a field. Actually, the Field-tree is not a hierarchical tree, but a directed acyclic graph, as each field can have one, two, or, four ancestors. At one level the fields form a partition and therefore never overlap. In another variant, the Cover Field-tree (Frank and Barrera 1989), the fields may overlap. It is not necessary that at each level the entire grid explicitly be present as fields.

A newly inserted object is stored in the smallest field in which it completely fits (unless its importance requires it to be stored at a higher level). As a result of the different displacements and grid resolutions, an object never has to be stored more than 3 levels above the field size that corresponds to the object size. Note that this is not the case in a Quadtree-like structure, because there the edges at different levels are collinear. The insertion of a new object may cause a field to become too full. In that case an attempt is made to create one or more new descendants and to reorganize the field by moving objects down. This is not always possible. A drawback of the Field-tree is that an overflow page is sometimes required, as it is not possible to move relatively large or important objects of an overfull field to a lower level field.

7 R-tree family

Instead of dividing space in some manner, it is also possible to group the objects in some hierarchic organization based on (a rectangular approximation of) their location. This is the approach of the R-tree and in this section several variants are also described: R+-tree, R*-tree, Hilbert R-tree, and the Sphere-tree.

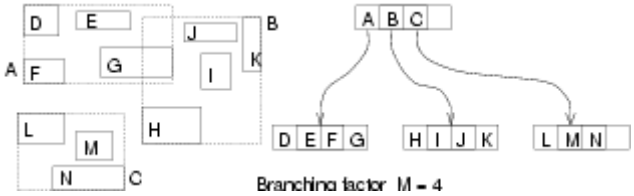

Fig. 17: R-tree (taken from van Oosterom 1994)

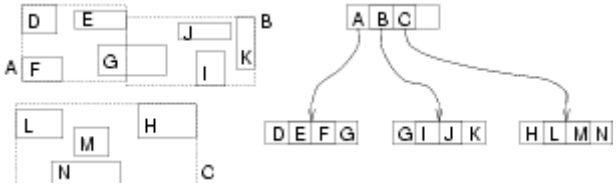

Fig. 18: R+-tree (taken from van Oosterom 1994)

7.1 The R-tree

The R-tree is an index structure that was defined by Guttman in 1984. The leaf nodes of this multi-way tree contain entries of the form: (I, object-id), where object-id is a pointer to a data object and I is a bounding box (or an axes-parallel minimal bounding rectangle, MBR. The data object can be of any type: point, polyline, or polygon. The internal nodes contain entries of the form: (I, child-pointer), where child-pointer is a pointer to a child and I is the MBR of that child. The maximum number of entries in each node is called the branching factor M and is chosen to suit paging and disk I/O buffering. The Insert and Delete algorithms assure that the number of entries in each node remains between m and M, where $m \leq \lceil M/2 \rceil$ is the minimum number of entries per node. An advantage of the R-tree is that pointers to complete objects (e.g. polygons) are stored; so the objects are never fragmented.

When inserting a new object, the tree is traversed from the root to a leaf choosing each time the child which needs the least enlargement to enclose the object. If there is still space, then the object is stored in that leaf. Otherwise, the leaf is split into two leaves in such a way that it is tried to minimize the total area of the two leaves. A new leaf may cause the parent to become overfull, so it has to be split also. This process may be repeated upto the root. During the reverse operation, delete, a node may become underfull. In this situation the node is removed (all other entries are saved and reinserted at the proper level later on). Again, this may cause the parent to become underfull, and the same technique is applied at the next level. This process may have to be repeated upto the root. Of course, the MBRs of all affected nodes have to be updated during an insert or a delete operation.

Fig. 17 shows an R-tree with two levels and M = 4. The lowest level contains three leaf nodes and the highest level contains one node with pointers and MBRs of the leaf nodes. Coverage is defined as the total area of all the MBRs of all leaf R-tree nodes, and overlap is the total area contained within two or

more leaf MBRs (Faloutsos, Sellis, and Roussopoulos 1987). In Fig. 17 the coverage is A U B U C and the overlap is A $\cap$ B. It is clear that efficient searching demands both low coverage and overlap.

7.2 Some R-tree variants

Roussopoulos and Leifker (Roussopoulos and Leifker 1985) describe the Pack algorithm which creates an initial R-tree that is more efficient than the R-tree created by the Insert algorithm. The Pack algorithm requires all data to be known a priori. The R+-tree (Faloutsos **et al** 1987), a modification of the R-tree, avoids overlap at the expense of more nodes and multiple references to some objects; see Fig. 18. Therefore, point queries always corresponds to a single-path tree traversal. A drawback of the R+-tree is that no minimum space utilization per node can be given. Analytical results indicate that R+-trees allow more efficient searching, in the case of relative large objects.

The R*-tree (Beckmann **et al** 1990) is based on the same structure as the R-tree, but is applies a different insert algorithm. When a node overflows, it is not split right away, but first it is tried to remove p entries and reinsert these in the tree. The parameter p can vary. In the original paper it is suggested to set p to be 30 per cent of the maximum number of entries per node. In some cases this will solve the node overflow problem without splitting the node. In general this will result in a fuller tree. However, this reinsert technique will not always solve the problem and sometimes a real node split is required. Instead of only minimizing the total area, it is now also tried to minimize overlap between the nodes, and to make the nodes as square as possible.

The Hilbert R-tree uses the centre point Hilbert value of the MBR to organize the objects (Kamel and Faloutsos 1994). When grouping objects (based on their Hilbert value), they form an entry in their parent node which contains both the union of all MBRs of the objects and the largest Hilbert value of the objects. Again, this is repeated on the higher levels until a single root is obtained. Inserting and deleting is basically done using the (largest) Hilbert value and applying B-tree (Bayer and McCreight 1973) techniques. Searching is done using the MBR and applying R-tree techniques. Due to the B-tree technique it is possible to get fuller nodes, because `2-to-3' or `3-to-4' (or higher) split policies can be used. This results in a more compact tree, which is again beneficial for the performance. The drawback of the Hilbert R-tree is that not the real spatial aspects of the objects are used to organize them, but their Hilbert value. It is possible that two objects which are very close in reality have Hilbert values which are very different. Therefore, these two objects will not end up in the same node in spite of the fact that they are very close, each of them is grouped with other objects (further away in reality, but with closer Hilbert value). This will result in larger MBRs and therefore reduced performance.

The Sphere-tree (van Oosterom and Claassen 1990) is very similar to the R-tree (Guttman 1984), with the exception that is uses minimal bounding circles (or spheres in higher dimensions, MBSs) instead of MBRs; see Fig. 19. Besides being orientation-insensitive, the Sphere-tree has the advantage over the R-tree in that it requires less storage space. The operations on the Sphere-tree are very similar to those on the R-tree, with the exception of the computation of the minimal bounding circle, which is more difficult (Sylvester 1857, Elzinga and Hearn 1972 and Megiddo 1983).

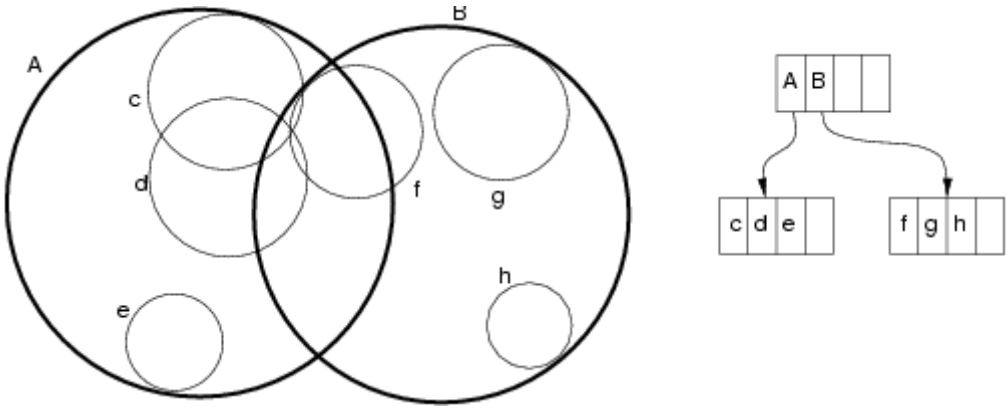

Fig. 19: The Sphere-tree

8 Multi-scale spatial access methods

When not only taking spatial location, but also importance (resolution, scale) into account when designing an access method, the interactive GIS applications can be supported even better. Think of a user who is panning and zooming in a certain data set. Just enlarging the objects when the user zooms in, will result in a poor map. Not only must the objects be enlarged, but they must also be displayed with more detail (because of the higher resolution), and also less significant objects must be displayed. A simple solution is to store the map at different scales (or level of details). This would introduce redundancy with all related drawbacks: possible inconsistency and increased memory usage. Therefore, the geographic data should be stored in an integrated manner without redundancy, and if required, supported by a special data structure. Detail levels are closely related to cartographic map generalization techniques. Besides being suited for map generalization, these multi-scale data structures must also provide spatial properties; e.g., it must be possible to find all objects within a specified region efficiently. The name of these types of data structures is reactive data structures (van Oosterom 1989, 1991, 1994).

The simplification part of the generalization process is supported by the Binary Line Generalization-tree (van Oosterom and van den Bos 1989) based on the Douglas-Peucker algorithm (Douglas and Peucker 1973); see Fig. 20. The Reactive-tree (van Oosterom 1991) is a spatial index structure, that also takes care of the selection part of the generalization. The Reactive-tree is based on the R-tree (Guttman 1984) with the difference that important objects are not stored at leaf level, but are stored at higher levels according to their importance; see Fig. 21 and 22. The further one zooms in, the more tree levels must be addressed. Roughly stated, during map generation based on a selection from the Reactive-tree, one should try to choose the required importance value such that a constant number of objects will be selected. This means that if the required region is large only the more important objects should be selected and if the required region is small, then the less important objects must also be selected. When using the Reactive-tree and the BLG-tree for the generalization of an area partitioning, some problems are encountered: gaps may be introduced by omitting small features and mismatches may occur as a result of independent simplification of common boundaries. These problems can be solved by additionally using the GAP-tree. Using the Reactive-tree, BLG-tree, and the GAP-tree, it is possible to interactively browse through large geographic data sets ay very different scales (van Oosterom and Schenkelaars 1995).

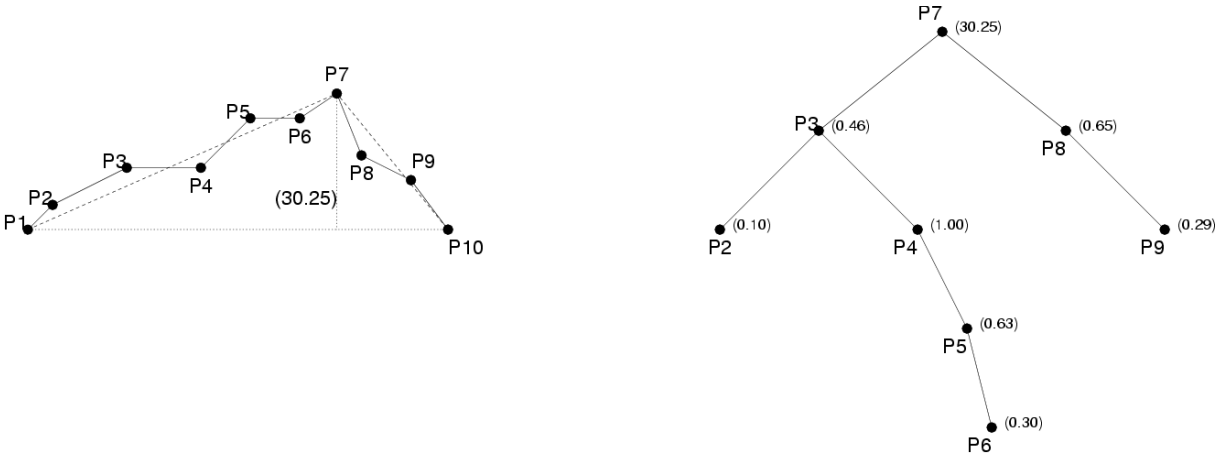

Fig. 20: Binary Line Generalization tree (taken from van Oosterom 1994)

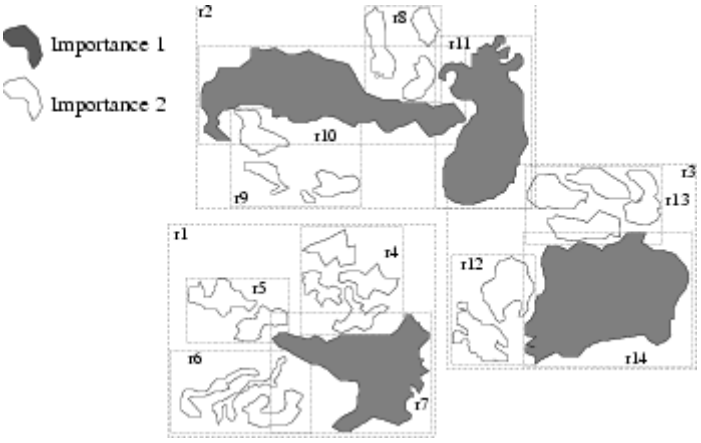

Fig. 21: An Example of Reactive-tree Rectangles (taken from van Oosterom and Schenkelaaars 1995)

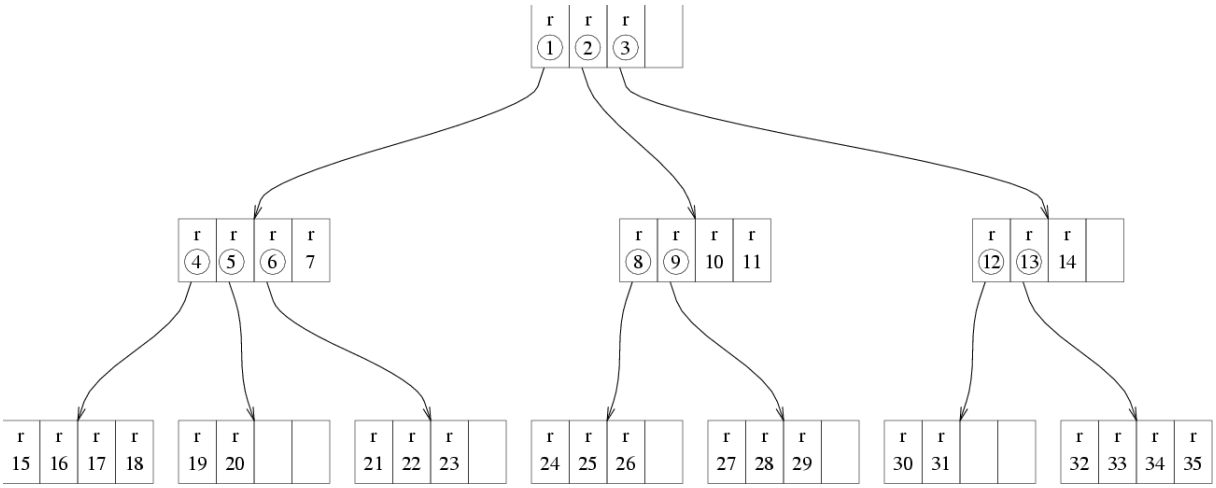

Fig. 22: The Reactive-tree (taken from van Oosterom and Schenkelaars 1995)

9 Conclusion

Though many spatial access methods have been described in the literature, the sad situation is that only a few have been implemented within the kernel of a (commercial) databases and are ready to be used: e.g. the R-tree in Illustra (Informix). At the moment several layered `middleware' solutions are provided; e.g. the Spatial Data Engine (SDE) of ERSI. The drawback of the layered approaches is that the database query optimizer does not know anything about the spatial data, so it can not generate an optimal query plan. Further, all access should be through the layer, but already many database applications do exist which have direct access to the database. In this situation the consistency may become a serious problem. So, in practice the possibilities are quite limited and users have to develop their own solutions. An approach for this is the Spatial Location Code (SLC) (van Oosterom and Vijlbrief 1996), which designed to enable efficient storage and retrieval of spatial data in a standard (relational) DBMS. It is used for indexing and clustering geographic objects in a database and it combines the strong aspects of several known spatial access methods (Quadtree, Field-tree, and Morton code) into one SLC value per object. The unique aspect of the SLC is that both location and extend of possibly non-zero-sized objects are approximated by this single value. The SLC is quite general and can be applied in higher dimensions.

References

Abel, D. J., and Mark, D. M. (1990). `A Comparative Analysis of Some Two-Dimensional Orderings', International Journal of Geographical Information Systems, 4 (1), 21-31.

Bayer, R., and McCreight, E. (1973). `Organization and Maintenance of Large Ordered Indexes', Acta Informatica, 1, 173-189.

Beckmann, N., Kriegel, H.-P., Schneider, R., and Seeger, B. (1990). `The R -tree: An Efficient and Robust Access Method for Points and Rectangles', in ACM/SIGMOD, Atlantic City, pp. 322-331 New York. ACM.

Bentley, J. L., and Friedman, J. H. (1979). `Data Structures for Range Searching', Computing Surveys, 11 (4), 397-409.

Bentley, J. L. (1975). `Multidimensional Binary Search Trees Used for Associative Searching', Communications of the ACM, 18 (9), 509-517.

Douglas, D. H., and Peucker, T. K. (1973). `Algorithms for the Reduction of Points Required to Represent a Digitized Line or its Caricature', Canadian Cartographer, 10, 112-122.

Elzinga, J., and Hearn, D. W. (1972). `Geometrical Solutions for Some Minimax Location Problems', Transportation Science, 6, 379-394.

Faloutsos, C., and Roseman, S. (1989). `Fractals for Secondary Key Retrieval', in Eight ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS), pp. 247-252.

Faloutsos, C., Sellis, T., and Roussopoulos, N. (1987). `Analysis of Object Oriented Spatial Access Methods', ACM SIGMOD, 16 (3), 426-439.

Faloutsos, C. (1988). `Gray Codes for Partial Match and Range Queries', IEEE Transactions on Software Engineering, SE-14 (10), 1381-1393.

Frank, A. U., and Barrera, R. (1989). `The Field-tree: A Data Structure for Geographic Information System', in Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California, pp. 29-44 Berlin. Springer-Verlag.

Frank, A. (1983). `Storage Methods for Space Related Data: The Field-tree', Tech. rep. Bericht no. 71, Eidgenössische Technische Hochschule Zürich.

Fuchs, H., Kedem, Z. M., and Naylor, B. F. (1980). `On Visible Surface Generation by A Priori Tree Structures', ACM Computer Graphics, 14 (3), 124-133.

Fuchs, H., Abram, G. D., and Grant, E. D. (1983). `Near Real-Time Shaded Display of Rigid Objects', ACM Computer Graphics, 17 (3), 65-72.

Goodchild, M. F., and Grandfield, A. W. (1983). `Optimizing Raster Storage: An Examination of Four Alternatives', in Auto-Carto 6, pp. 400-407.

Goodchild, M. F. (1989). `Tiling Large Geographical Databases', in Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California, pp. 137-146 Berlin. Springer-Verlag.

Guttman, A. (1984). `R-Trees: A Dynamic Index Structure for Spatial Searching', ACM SIGMOD, 13, 47-57.

Hutflesz, A., Six, H.-W., and Widmayer, P. (1990). `The R-File: An Efficient Access Structure for Proximity Queries', in Proceedings IEEE Sixth International Conference on Data Engineering, Los

Angeles, California, pp. 372-379 Los Alamitos, CA. IEEE Computer Society Press.

Jagadish, H. V. (1990). `Linear Clustering of Objects with Multiple Attributes', in ACM/SIGMOD, Atlantic City, pp. 332-342 New York. ACM.

Kamel, I., and Faloutsos, C. (1994). `Hilbert R-tree: An improved R-tree using fractals', in VLDB Conference.

Kleiner, A., and Brassel, K. E. (1986). `Hierarchical Grid Structures for Static Geographic Data Bases', in Auto-Carto London, pp. 485-496 London. Auto Carto.

Matsuyama, T., Hao, L. V., and Nagao, M. (1984). `A File Organization for Geographic Information Systems Based on Spatial Proximity', Computer Vision, Graphics and Image Processing, 26, 303-318.

Megiddo, N. (1983). `Linear-Time Algorithms for Linear Programming in R3 and Related Problems', SIAM Journal on Computing, 12 (4), 759-776.

Nievergelt, J., Hinterberger, H., and Sevcik, K. C. (1984). `The Grid File: An Adaptable, Symmetric Multikey File Structure', ACM Transactions on Database Systems, 9 (1), 38-71.

Nulty, W. G., and Barholdi, III, J. J. (1994), `Robust Multidimensional searching with spacefilling curves', in Proceedings of the 6th International Symposium on Spatial Data Handling, Edinburgh, Scotland, pp. 805-818.

Orenstein, J. A., and Manola, F. A. (1988). `PROBE Spatial Data Modeling and Query Processing in an Image Database Application', IEEE Transactions on Software Engineering, 14 (5), 611-629.

Paterson, M. S., and Yao, F. F. (1989). `Binary Partitions with Applications to Hidden-Surface Removal and Solid Modelling', in Proceedings 5th ACM Symposium on Computational Geometry, pp. 23-32 New York. ACM.

Robinson, J. T. (1981). `The K-D-B-Tree: A Search Structure for Large Multidimensional Dynamic Indexes', ACM SIGMOD, 10, 10-18.

Rosenberg, J. B. (1985). `Geographical Data Structures Compared: A Study of Data Structures Supporting Region Queries', IEEE Transactions on Computer Aided Design, CAD-4 (1), 53-67.

Roussopoulos, N., and Leifker, D. (1985). `Direct Spatial Search on Pictorial Databases Using Packed R-Trees', ACM SIGMOD, 14 (4), 17-31.

Samet, H. (1984). `The Quadtree and Related Hierarchical Data Structures', Computing Surveys, 16 (2), 187-260.

Samet, H. (1989). The Design and Analysis of Spatial Data Structures. Addison-Wesley, Reading, Mass.

Sylvester, J. J. (1857). `A Question in the Geometry of Situation', Quarterly Journal of Mathematics, 1, 79.

Tamminen, M. (1984). `Comment on Quad- and Octtrees', Communications of the ACM, 27 (3), 248-249.

van Oosterom, P., and Claassen, E. (1990). `Orientation Insensitive Indexing Methods for Geometric Objects', in 4th International Symposium on Spatial Data Handling, Zürich, pp. 1016-1029.

van Oosterom, P., and Schenkelaars, V. (1995). `The Development of an Interactive Multi-Scale GIS', International Journal of Geographical Information Systems, 9 (5), 489-507.

van Oosterom, P., and van den Bos, J. (1989). `An Object-Oriented Approach to the Design of Geographic Information Systems', Computers & Graphics, 13 (4), 409-418.

van Oosterom, P., and Vijlbrief, T. (1996). `The Spatial Location Code', in Proceedings of the 7th International Symposium on Spatial Data Handling, Delft, The Netherlands.

van Oosterom, P. (1989). `A Reactive Data Structure for Geographic Information Systems', in Auto-Carto 9, pp. 665-674.

van Oosterom, P. (1990). `A modified binary space partitioning tree for Geographic Information Systems', International Journal of Geographical Information Systems, 4 (2), 133-146.

van Oosterom, P. (1991). `The Reactive-Tree: A Storage Structure for a Seamless, Scaleless Geographic Database', in Auto-Carto 10, pp. 393-407.

van Oosterom, P. (1994). Reactive Data Structures for Geographic Information Systems. Oxford University Press, Oxford.