18

Multidimensional Data Structures[1]

Hanan Samet
University of Maryland

18.1 Introduction

The representation of **multidimensional data** is an important issue in applications in diverse fields that include database management systems, computer graphics, computer vision, computational geometry, image processing, geographic information systems (GIS), pattern recognition, VLSI design, and others. The most common definition of multidimensional data is a collection of points in a higher dimensional space. These points can represent locations and objects in space as well as more general records. As an example of a record, consider an employee record that has attributes corresponding to the employee's name, address, sex, age, height, weight, and social security number. Such records arise in database management systems and can be treated as points in, for this example, a seven-dimensional space (i.e., there is one dimension for each attribute), albeit the different dimensions have different type units (i.e., name and address are strings of characters, sex is binary; while age, height, weight, and social security number are numbers).

When multidimensional data corresponds to locational data, we have the additional property that all of the attributes have the same unit, which is distance in space. In this case, we can combine the attributes and pose queries that involve proximity. For example, we may wish to find the closest city to Chicago within the two-dimensional space from which the locations of the cities are drawn. Another query seeks to find all cities within 50 miles of Chicago. In contrast, such queries are not very meaningful when the attributes do not have the same type. For example, it is not customary to seek the person with age–weight combination closest to John Jones, as we do not have a commonly accepted unit of year-pounds (year-kilograms) or

[1]All figures ©1998 by Hanan Samet.
Supported by the National Science Foundation under grant IRI-9712715.

definition thereof. It should be clear that we are not speaking of queries involving Boolean combinations of the different attributes (e.g., range queries), which are quite common.

When multidimensional data spans a continuous physical space (i.e., an infinite collection of locations), the issues become more interesting. In particular, we are no longer just interested in the locations of objects but, we are also interested in the space that they occupy (i.e., their extent). Some example objects include lines (e.g., roads, rivers), regions (e.g., lakes, counties, buildings, crop maps, polygons, polyhedra), rectangles, and surfaces. The objects may be disjoint or could even overlap. One way to deal with such data is to store it explicitly by parametrizing it and thereby reduce it to a point in a higher dimensional space. For example, a line in two-dimensional space can be represented by the coordinate values of its endpoints (i.e., a pair of x and a pair of y coordinate values) and then stored as a point in a four-dimensional space. Thus, in effect, we have constructed a transformation (i.e., mapping) from a two-dimensional space (i.e., the space from which the lines are drawn) to a four-dimensional space (i.e., the space containing the representative point corresponding to the line).

The transformation approach is fine if we are just interested in retrieving the data. It is appropriate for queries about the objects (e.g., determining all lines that pass through a given point or that share an endpoint, etc.) and the immediate space that they occupy. However, the drawback of the transformation approach is that it ignores the geometry inherent in the data (e.g., the fact that a line passes through a particular region) and its relationship to the space in which it is embedded.

For example, suppose that we want to detect if two lines are near each other, or, alternatively, to find the nearest line to a given line. This is difficult to do in the four-dimensional space, regardless of how the data in it is organized, since proximity in the two-dimensional space from which the lines are drawn is not necessarily preserved in the four-dimensional space. In other words, although the two lines may be very close to each other, the Euclidean distance between their representative points may be quite large.

Of course, we could overcome these problems by projecting the lines back to the original space from which they were drawn, but in such a case, we may ask what was the point of using the transformation in the first place? In other words, at the least, the representation that we choose for the data should allow us to perform operations on the data. Thus, we need special representations for spatial multidimensional data other than point representations. One solution is to use data structures that are based on spatial occupancy.

Spatial occupancy methods decompose the space from which the spatial data is drawn (e.g., the two-dimensional space containing the lines) into regions called *buckets.* They are also commonly known as **bucketing methods.** Traditional bucketing methods such as the grid file [45], BANG file [22], LSD trees [27], buddy trees [55], etc. have been designed for multidimensional point data that need not be locational. In the case of spatial data, these methods have usually been applied to the transformed data (i.e., the representative points). In contrast, we discuss their application to the actual objects in the space from which the objects are drawn (i.e., two dimensions in the case of a collection of line segments).

In this chapter, we explore a number of different representations of multidimensional data bearing the above issues in mind. In the case of point data, we examine representations of both locational and nonlocational data, as well as combinations of the two. While we cannot give exhaustive details of all of the data structures, we try to explain the intuition behind their development as well as to give literature pointers to where more information can be found. Many of these representations are described in greater detail in [50, 51], including an extensive bibliography. Our approach is primarily a descriptive one. Most of our examples are of two-dimensional spatial data, although we do touch briefly on three-dimensional data.

At times, we discuss bounds on execution time and space requirements. Nevertheless, this information is presented in an inconsistent manner. The problem is that such analyses are very difficult to perform for many of the data structures that we present. This is especially true for the data structures that are based on spatial occupancy (e.g., **quadtree** and **R-tree** variants). In particular, such methods have good observable average-case behavior but may have very bad worst cases which may only arise rarely in practice. Their analysis is beyond the scope of this chapter and usually we do not say anything about it. Nevertheless, these representations find frequent use in applications where their behavior is deemed acceptable, and is often found to be better than that of solutions whose theoretical behavior would appear to be superior. The problem is primarily attributed to the presence of large constant factors which are usually ignored in the *big O* and Ω analyses [38].

The rest of this chapter is organized as follows. Section 18.2 reviews a number of representations of point data of arbitrary dimensionality. Section 18.3 describes bucketing methods that organize collections of spatial objects (as well as multidimensional point data) by aggregating their bounding rectangles. Sections 18.2 and 18.3 are applicable to both spatial and nonspatial data, although all the examples that we present are of spatial data. Section 18.4 focuses on representations of region data, while Section 18.5 discusses a subcase of region data, which consists of collections of rectangles. Section 18.6 deals with curvilinear data, which also includes polygonal subdivisions and collections of line segments. Section 18.7 contains a summary and a brief indication of some research issues. Section 18.8 reviews some of the definitions of the terms used in this chapter. Note that although our examples are primarily from a two-dimensional space, the representations are applicable to higher dimensional spaces as well.

18.2 Point Data

Our discussion assumes that there is one record per data point, and that each record contains several attributes or keys (also frequently called fields, dimensions, coordinates, and axes). In order to facilitate retrieval of a record based on some of its attribute values, we also assume the existence of an ordering for the range of values of each of these attributes. In the case of locational attributes, such an ordering is quite obvious as the values of these attributes are numbers. In the case of alphanumeric attributes, the ordering is usually based on the alphabetic sequence of the characters making up the attribute value. Other data such as color could be ordered by the characters making up the name of the color or possibly the color's wavelength. It should be clear that finding an ordering for the range of values of an attribute is generally not an issue; the real issue is what ordering to use!

The representation that is ultimately chosen for the data depends, in part, on answers to the following questions:

1. What operations are to be performed on the data?
2. Should we organize the data or the embedding space from which the data is drawn?
3. Is the database static or dynamic (i.e., can the number of data points grow and shrink at will)?
4. Can we assume that the volume of data is sufficiently small so that it can all fit in core, or should we make provisions for accessing disk-resident data?

Disk-resident data implies grouping the data (either the underlying space based on the volume — that is, the amount — of the data it contains or the points, hopefully, by the proximity of their values) into sets (termed *buckets*) corresponding to physical storage units (i.e., pages). This leads to questions about their size, and how they are to be accessed.

1. Do we require a constant time to retrieve a record from a file or is a logarithmic function of the number of records in the file adequate? This is equivalent to asking if the access is via a directory in the form of an array (i.e., direct access) or a tree?
2. How large can the directories be allowed to grow before it is better to rebuild them?
3. How should the buckets be laid out on the disk?

Clearly, these questions are complex and we cannot address them all here. Some are answered in other sections. In this section, we focus primarily on dynamic data with an emphasis on two dimensions (i.e., attributes) and concentrate on the following queries:

1. Point queries — that is, if a particular point is present.

2. Range queries.
3. Boolean combinations of 1 and 2.

Most of the representations that we describe can be extended easily to higher dimensions, although some like the priority search tree are basically for two-dimensional data. Our discussion and examples are based on the fact that all of the attributes are locational or numeric and that they have the same range, although all of the representations can also be used to handle nonlocational and nonnumeric attributes. When discussing behavior in the general case, we assume a data set of N points and d attributes.

The simplest way to store point data is in a sequential list. Accesses to the list can be sped up by forming sorted lists for the various attributes which are known as *inverted lists* (e.g., [37]). There is one list for each attribute. This enables pruning the search with respect to the value of one of the attributes. In order to facilitate random access, the lists can be implemented using range trees [10].

It should be clear that the inverted list is not particularly useful for range searches. The problem is that it can only speed up the search for one of the attributes (termed the *primary* attribute). A number of solutions have been proposed. These solutions can be decomposed into two classes. One class of solutions enhances the range tree corresponding to the inverted list to include information about the remaining attributes in its internal nodes. This is the basis of the multidimensional range tree and variants of the priority search tree [15, 41] that are discussed at the end of this section.

The second class of solutions is more widely used and is exemplified by the *fixed-grid* method [9, 37]. It partitions the space from which the data is drawn into rectangular cells by overlaying it with a grid. Each grid cell c contains a pointer to another structure (e.g., a list) which contains the set of points that lie in c. Associated with the grid is an access structure to enable the determination of the grid cell associated with a particular point p. This access structure acts like a directory and is usually in the form of a d-dimensional array with one entry per grid cell or a tree with one leaf node per grid cell.

There are two ways to build a fixed grid. We can either subdivide the space into equal-sized intervals along each of the attributes (resulting in congruent grid cells) or place the subdivision lines at arbitrary positions that are dependent on the underlying data. In essence, the distinction is between organizing the data to be stored and organizing the embedding space from which the data is drawn [45]. In particular, when the grid cells are congruent (i.e., equal-sized when all of the attributes are locational with the same range and termed a *uniform grid*), use of an array access structure is quite simple and has the desirable property that the grid cell associated with point p can be determined in constant time. Moreover, in this case, if the width of each grid cell is twice the search radius for a rectangular range query, then the average search time is $O(F \cdot 2^d)$ where F is the number of points that have been found [11]. Figure 18.1 is an example of a uniform-grid representation for a search radius equal to 10 (i.e., a square of size 20×20[2]).

Use of an array access structure when the grid cells are not congruent requires us to have a way of keeping track of their size so that we can determine the entry of the array access structure corresponding to the grid cell associated with point p. One way to do this is to make use of what are termed *linear scales*, which indicate the positions of the grid lines (or partitioning hyperplanes in $d > 2$ dimensions). Given a point p, we determine the grid cell in which p lies by finding the "coordinate values" of the appropriate grid cell. The linear scales are usually implemented as one-dimensional trees containing ranges of values.

The use of an array access structure is fine as long as the data is static. When the data is dynamic, it is likely that some of the grid cells become too full while other grid cells are empty. This means that we need to rebuild the grid (i.e., further partition the grid or reposition the grid partition lines or hyperplanes) so that the various grid cells are not too full. However, this creates many more empty grid cells as a result

[2]Note that although the data consists of three attributes, one of which is nonlocational (i.e., name) and two of which are locational (i.e., the coordinate values), retrieval is only on the basis of the locational attribute values. Thus, there is no ordering on the name, and, therefore, we treat this example as two-dimensional locational data.

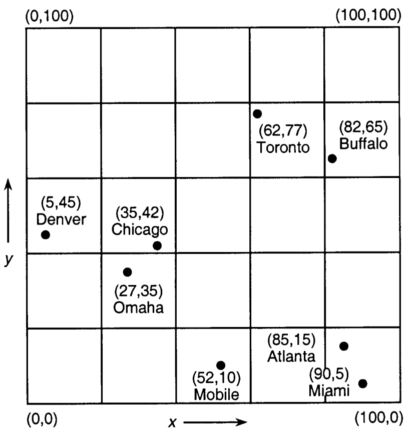

FIGURE 18.1 Uniform-grid representation corresponding to a set of points with a search radius of 20.

of repartitioning the grid (i.e., empty grid cells are split into more empty grid cells). In this case, we have two alternatives. The first is to assign an ordering to all the grid cells and to impose a tree access structure on the elements of the ordering that correspond to nonempty grid cells. The effect of this alternative is analogous to using a mapping from d dimensions to one dimension and then applying one of the one-dimensional access structures such as a B-tree, balanced binary tree, etc., to the result of the mapping. There are a number of possible mappings including row, Morton (i.e., bit interleaving or bit interlacing), and Peano–Hilbert (e.g.,[51])[3]. This alternative is applicable regardless of whether or not the grid cells are congruent. Of course, if the grid cells are not congruent, then we must also record their size in the element of the access structure.

The second alternative is to merge spatially adjacent empty grid cells into larger empty grid cells, while splitting grid cells that are too full, thereby making the grid adaptive. Again, the result is that we can no longer make use of an array access structure to retrieve the grid cell that contains query point p. Instead, we make use of a tree access structure in the form of a k-ary tree where k is usually 2^d. Thus, what we have done is marry a k-ary tree with the fixed-grid method. This is the basis of the point quadtree [17] and the PR quadtree [46, 51] which are multidimensional generalizations of binary trees.

The difference between the point quadtree and the PR quadtree is the same as the difference between *trees* and *tries* [20], respectively. The binary search tree [37] is an example of the former since the boundaries of different regions in the search space are determined by the data being stored. Address computation methods such as radix searching [37] (also known as digital searching) are examples of the latter, since region boundaries are chosen from among locations that are fixed regardless of the content of the data set. The process is usually a recursive halving process in one dimension, recursive quartering in two dimensions, etc., and is known as **regular decomposition.**

In two dimensions, a point quadtree is just a two-dimensional binary search tree. The first point that is inserted serves as the root, while the second point is inserted into the relevant quadrant of the tree rooted at the first point. Clearly, the shape of the tree depends on the order in which the points were inserted. For example, Fig. 18.2 is the point quadtree corresponding to the data of Fig. 18.1 inserted in the order `Chicago`, `Mobile`, `Toronto`, `Buffalo`, `Denver`, `Omaha`, `Atlanta`, and `Miami`.

In two dimensions, the PR quadtree is based on a recursive decomposition of the underlying space into four congruent (usually square in the case of locational attributes) cells until each cell contains no more

[3]These mappings have been investigated primarily for purely locational multidimensional point data. They cannot be applied directly to the key values for nonlocational point data.

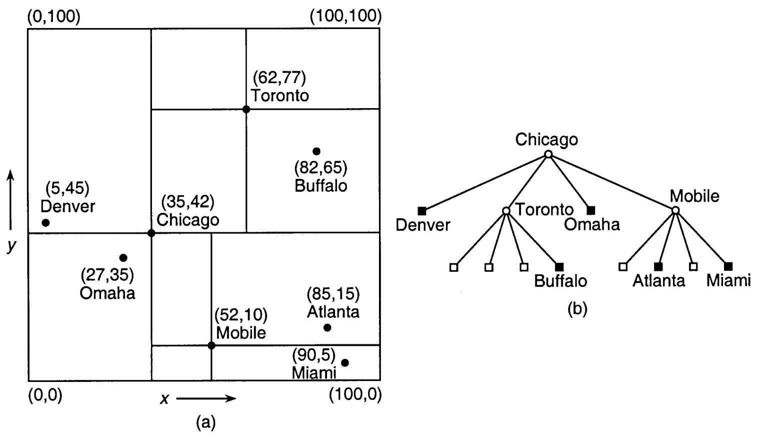

FIGURE 18.2 A point quadtree and the records it represents corresponding to Fig.18.1: (a) the resulting partition of space, and (b) the tree representation.

than one point. For example, Fig. 18.3 is the PR quadtree corresponding to the data of Fig. 18.1. The shape of the PR quadtree is independent of the order in which data points are inserted into it. The disadvantage of the PR quadtree is that the maximum level of decomposition depends on the minimum separation between two points. In particular, if two points are very close, then the decomposition can be very deep. This can be overcome by viewing the blocks or nodes as buckets with capacity c and only decomposing a block when it contains more than c points.

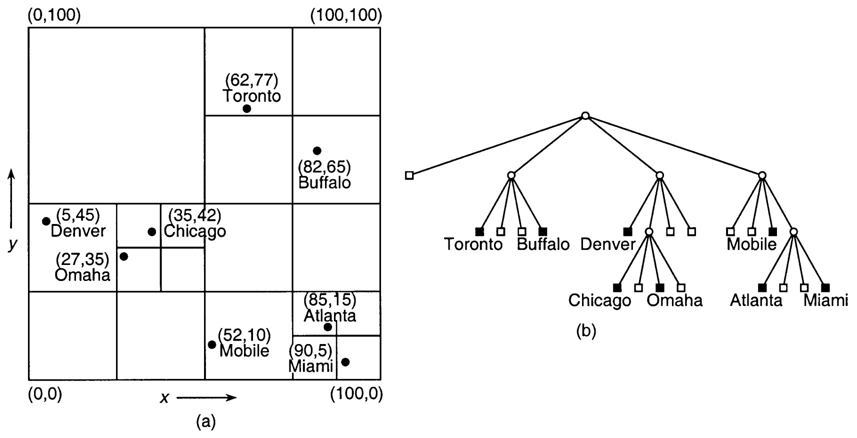

FIGURE 18.3 A PR quadtree and the records it represents corresponding to Fig.18.1: (a) the resulting partition of space, and (b) the tree representation.

As the dimensionality of the space increases, each level of decomposition of the quadtree results in many new cells as the fanout value of the tree is high (i.e., 2^d). This is alleviated by making use of a **k-d tree** [7]. The k-d tree is a binary tree where at each level of the tree, we subdivide along a different attribute so that, assuming d locational attributes, if the first split is along the x axis, then after d levels, we cycle back and again split along the x axis. It is applicable to both the point quadtree and the PR quadtree (in which case we have a *PR k-d tree,* or a **bintree** in the case of region data).

At times, in the dynamic situation, the data volume becomes so large that a tree access structure is inefficient. In particular, the grid cells can become so numerous that they cannot all fit into memory, thereby causing them to be grouped into sets (termed *buckets*) corresponding to physical storage units (i.e., pages) in secondary storage. The problem is that, depending on the implementation of the tree access structure, each time we must follow a pointer, we may need to make a disk access. This has led to a return to the use of an array access structure. The difference from the array used with the static fixed-grid method described earlier is that the array access structure (termed *grid directory*) may be so large (e.g., when d gets large) that it resides on disk as well, and the fact that the structure of the grid directory can be changed as the data volume grows or contracts. Each grid cell (i.e., an element of the grid directory) contains the address of a bucket (i.e., page) that contains the points associated with the grid cell. Notice that a bucket can correspond to more than one grid cell. Thus, any page can be accessed by two disk operations: one to access the grid cell and one more to access the actual bucket.

This results in *EXCELL* [59] when the grid cells are congruent (i.e., equal-sized for locational data), and *grid file* [45] when the grid cells need not be congruent. The difference between these methods is most evident when a grid partition is necessary (i.e., when a bucket becomes too full and the bucket is not shared among several grid cells). In particular, a grid partition in the grid file only splits one interval in two, thereby resulting in the insertion of a $(d-1)$-dimensional cross section. On the other hand, a grid partition in EXCELL means that all intervals must be split in two, thereby doubling the size of the grid directory.

Fixed-grids, quadtrees, k-d trees, grid file, EXCELL, as well as other hierarchical representations are good for range searching as they make it easy to implement the query. A typical query is one that seeks all cities within 80 miles of St. Louis, or, more generally, within 80 miles of the latitude position of St. Louis and within 80 miles of the longitude position of St. Louis.[4] In particular, these structures act as pruning devices on the amount of search that will be performed as many points will not be examined since their containing cells lie outside the query range. These representations are generally very easy to implement and have good expected execution times, although they are quite difficult to analyze from a mathematical standpoint. However, their worst cases, despite being rare, can be quite bad. These worst cases can be avoided by making use of variants of range trees [10] and priority search trees [41]. They are applicable to both locational and nonlocational attributes, although our presentation assumes that all the attributes are locational.

A one-dimensional range tree is a balanced binary search tree where the data points are stored in the leaf nodes and the leaf nodes are linked in sorted order by use of a doubly-linked list. A range search for $[L : R]$ is performed by searching the tree for the node with the smallest value that is $\geq L$, and then following the links until reaching a leaf node with a value greater than R. For N points, this process takes $O(\log_2 N + F)$ time and uses $O(N)$ storage. F is the number of points found.

A two-dimensional range tree is a binary tree of binary trees. It is formed in the following manner. First, sort all of the points along one of the attributes, say x, and store them in the leaf nodes of a balanced binary search tree, say T. With each non-leaf node of T, say I, associate a one-dimensional range tree, say T_I, of the points in the subtree rooted at I where now these points are sorted along the other attribute, say y. The range tree also can be adapted easily to handle d-dimensional data. In such a case, for N points, a d-dimensional range search takes $O(\log_2^d N + F)$ time, where F is the number of points found. The d-dimensional range tree uses $O(N \cdot \log_2^{d-1} N)$ storage.

The *priority search tree* is a related data structure that is designed for solving queries involving semi-infinite ranges in two-dimensional space. A typical query has a range of the form $([L_x : R_x], [L_y : \infty])$.

[4]The difference between these two formulations of the query is that the former admits a circular search region, while the latter admits a rectangular search region. In particular, the latter formulation is applicable to both locational and non-locational attributes, while the former is only applicable to locational attributes.

For example, Fig. 18.4 is the priority search tree for the data of Fig. 18.1. It is built as follows. Assume that no two data points have the same x coordinate value. Sort all the points along the x coordinate value and store them in the leaf nodes of a balanced binary search tree (a range tree in our formulation), say T. We proceed from the root node toward the leaf nodes. With each node I of T, associate the point in the subtree rooted at I with the maximum value for its y coordinate that has not already been stored at a shallower depth in the tree. If such a point does not exist, then leave the node empty. For N points, this structure uses $O(N)$ storage.

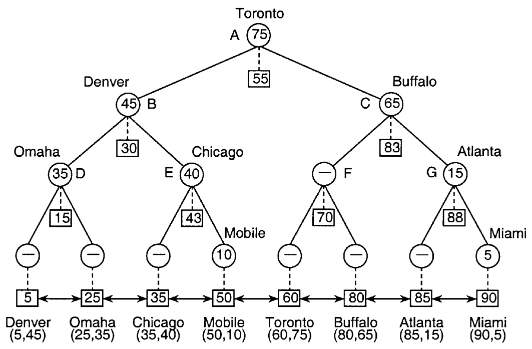

FIGURE 18.4 Priority search tree for the data of Fig.18.1. Each leaf node contains the value of its x coordinate in a square box. Each nonleaf node contains the appropriate x coordinate midrange value in a box using a link drawn with a broken line. Circular boxes indicate the value of the y coordinate of the point in the corresponding subtree with the maximum value for its y coordinate that has not already been associated with a node at a shallower depth in the tree.

It is not easy to perform a two-dimensional range query of the form $([L_x : R_x], [L_y : R_y])$ with a priority search tree. The problem is that only the values of the x coordinates are sorted. In other words, given a leaf node C that stores point (x_C, y_C), we know that the values of the x coordinates of all nodes to the left of C are smaller than x_C and the values of all those to the right of C are greater than x_C. On the other hand, with respect to the values of the y coordinates, we only know that all nodes below non-leaf node D with value y_D have values less than or equal to y_D; the y coordinate values associated with the remaining nodes in the tree that are not ancestors of D may be larger or smaller than y_D. This is not surprising, because a priority search tree is really a variant of a range tree in x and a heap (i.e., priority queue) [37] in y.

A heap enables finding the maximum (minimum) value in $O(1)$ time. Thus, it is easy to perform a semi-infinite range query of the form $([L_x : R_x], [L_y : \infty])$, as all we need do is descend the priority search tree and stop as soon as we encounter a y coordinate value that is less than L_y. For N points, performing a semi-infinite range query in this way takes $O(\log_2 N + F)$ time, where F is the number of points found.

The priority search tree is used as the basis of the *range priority tree* [15] to reduce the order of execution time of a two-dimensional range query to $O(\log_2 N + F)$ time (but still using $O(N \cdot \log_2 N)$ storage). Define an *inverse priority search tree* to be a priority search tree S such that with each node of S, say I, we associate the point in the subtree rooted at I with the minimum (instead of the maximum!) value for its y coordinate that has not already been stored at a shallower depth in the tree. The range priority tree is a balanced binary search tree (i.e., a range tree), say T, where all the data points are stored in the leaf nodes and are sorted by their y coordinate values. With each non-leaf node of T, say I, which is a left son of its father, we store a priority search tree of the points in the subtree rooted at I. With each non-leaf node of T, say I, which is a right son of its father we store an inverse priority search tree of the points in the subtree rooted at I. For N points, the range priority tree uses $O(N \cdot \log_2 N)$ storage.

Performing a range query for $([L_x : R_x], [L_y : R_y])$ using a range priority tree is done in the following manner. We descend the tree looking for the nearest common ancestor of L_y and R_y, say Q. The values of the y coordinates of all points in the left son of Q are less than R_y. We want to retrieve just the ones that are greater than or equal to L_y. We can obtain them with the semi-infinite range query $([L_x : R_x], [L_y : \infty])$. This can be done by using the priority tree associated with the left son of Q. Similarly, the values of the y coordinates of all points in the right son of Q are greater than L_y. We want to retrieve just the ones that are less than or equal to R_y. We can obtain them with the semi-infinite range query $([L_x : R_x], [-\infty : R_y])$. This can be done by using the inverse priority search tree associated with the right son of Q. Thus, for N points the range query takes $O(\log_2 N + F)$ time, where F is the number of points found.

18.3 Bucketing Methods

There are four principal approaches to decomposing the space from which the records are drawn. They are applicable regardless of whether the attributes are locational or nonlocational, although our discussion assumes that they are locational and that the records correspond to spatial objects. One approach makes use of an object hierarchy. It propagates the space occupied by the objects up the hierarchy with the identity of the propagated objects being implicit to the hierarchy. In particular, associated with each object is an object description (e.g., for region data, it is the set of locations in space corresponding to the cells that make up the object). Actually, since this information may be rather voluminous, it is often the case that an approximation of the space occupied by the object is propagated up the hierarchy rather than the collection of individual cells that are spanned by the object. For spatial data, the approximation is usually the minimum bounding rectangle for the object, while for nonspatial data it is simply the hyperrectangle whose sides have lengths equal to the ranges of the values of the attributes. Therefore, associated with each element in the hierarchy is a bounding rectangle corresponding to the union of the bounding rectangles associated with the elements immediately below it.

The R-tree (e.g., [6, 26]) is an example of an object hierarchy that finds use especially in database applications. The number of objects or bounding rectangles that are aggregated in each node is permitted to range between $m \leq \lceil M/2 \rceil$ and M. The root node in an R-tree has at least two entries unless it is a leaf node, in which case it has just one entry corresponding to the bounding rectangle of an object. The R-tree is usually built as the objects are encountered rather than waiting until all objects have been input. The hierarchy is implemented as a tree structure with grouping being based, in part, on proximity of the objects or bounding rectangles.

For example, consider the collection of line segment objects given in Fig. 18.5 shown embedded in a 4×4 grid. Figure 18.6(a) is an example R-tree for this collection with $m = 2$ and $M = 3$. Figure 18.6(b) shows the spatial extent of the bounding rectangles of the nodes in Fig. 18.6(a), with heavy lines denoting the bounding rectangles corresponding to the leaf nodes, and broken lines denoting the bounding rectangles corresponding to the subtrees rooted at the nonleaf nodes. Note that the R-tree is not unique. Its structure depends heavily on the order in which the individual objects were inserted into (and possibly deleted from) the tree.

Given that each R-tree node can contain a varying number of objects or bounding rectangles, it is not surprising that the R-tree was inspired by the B-tree [12]. Therefore, nodes are viewed as analogous to disk pages. Thus, the parameters defining the tree (i.e., m and M) are chosen so that a small number of nodes is visited during a spatial query (i.e., point and range queries), which means that m and M are usually quite large. The actual implementation of the R-tree is really a B$^+$-tree [12] as the objects are restricted to the leaf nodes.

The efficiency of the R-tree for search operations depends on its ability to distinguish between occupied space and unoccupied space (i.e., coverage), and to prevent a node from being examined needlessly due to a false overlap with other nodes. In other words, we want to minimize coverage and overlap. These goals

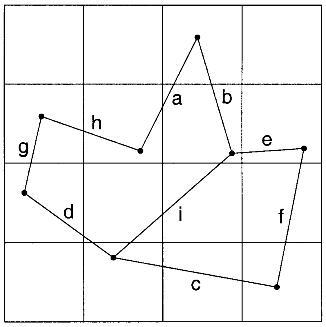

FIGURE 18.5 Example collection of line segments embedded in a 4×4 grid.

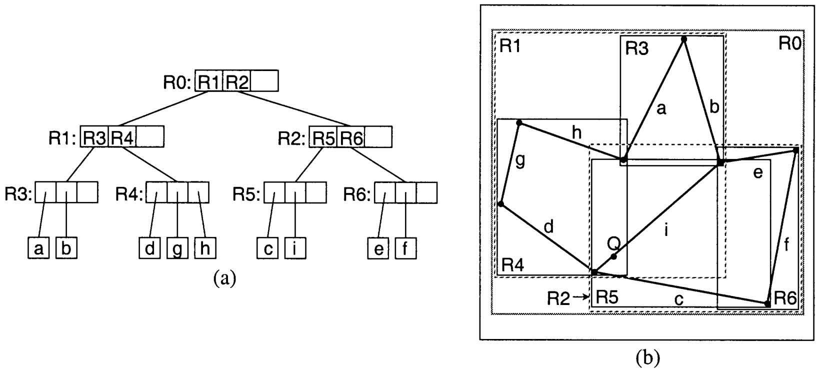

FIGURE 18.6 (a) R-tree for the collection of line segments with $m = 2$ and $M = 3$, in Fig.18.5, and (b) the spatial extents of the bounding rectangles. Notice that the leaf nodes in the index also store bounding rectangles, although this is only shown for the nonleaf nodes.

guide the initial R-tree creation process as well, subject to the previously mentioned constraint that the R-tree is usually built as the objects are encountered rather than waiting until all objects have been input.

The drawback of the R-tree (and any representation based on an object hierarchy) is that it does not result in a disjoint decomposition of space. The problem is that an object is only associated with one bounding rectangle (e.g., line segment `i` in Fig. 18.6 is associated with bounding rectangle R5, yet it passes through R1, R2, R4, and R5, as well as through R0 as do all the line segments). In the worst case, this means that when we wish to determine which object (e.g., an intersecting line in a collection of line segment objects, or a containing rectangle in a collection of rectangle objects) is associated with a particular point in the two-dimensional space from which the objects are drawn, we may have to search the entire collection. For example, in Fig. 18.6, when searching for the line segment that passes through point Q, we need to examine bounding rectangles R0, R1, R4, R2, and R5, rather than just R0, R2, and R5.

This drawback can be overcome by using one of three other approaches that are based on a decomposition of space into disjoint cells. Their common property is that the objects are decomposed into disjoint subobjects such that each of the subobjects is associated with a different cell. They differ in the degree of regularity imposed by their underlying decomposition rules, and by the way in which the cells are aggregated into buckets.

The price paid for the disjointness is that in order to determine the area covered by a particular object, we have to retrieve all the cells that it occupies. This price is also paid when we want to delete an object. Fortunately, deletion is not so common in such applications. A related costly consequence of disjointness is that when we wish to determine all the objects that occur in a particular region, we often need to retrieve some of the objects more than once. This is particularly troublesome when the result of the operation serves as input to another operation via composition of functions. For example, suppose we wish to compute the perimeter of all the objects in a given region. Clearly, each object's perimeter should only be computed once. Eliminating the duplicates is a serious issue (see [1] for a discussion of how to deal with this problem for a collection of line segment objects, and [2] for a collection of rectangle objects).

The first method based on disjointness partitions the embedding space into disjoint subspaces, and hence, the individual objects into subobjects, so that each subspace consists of disjoint subobjects. The subspaces are then aggregated and grouped in another structure, such as a B-tree, so that all subsequent groupings are disjoint at each level of the structure. The result is termed a k-d-B-tree [49]. The R^+-tree [56, 58] is a modification of the k-d-B-tree where at each level we replace the subspace by the minimum bounding rectangle of the subobjects or subtrees that it contains. The cell tree [25] is based on the same principle as the R^+-tree except that the collections of objects are bounded by minimum convex polyhedra instead of minimum bounding rectangles.

The R^+-tree (as well as the other related representations) is motivated by a desire to avoid overlap among the bounding rectangles. Each object is associated with all the bounding rectangles that it intersects. All bounding rectangles in the tree (with the exception of the bounding rectangles for the objects at the leaf nodes) are nonoverlapping.[5] The result is that there may be several paths starting at the root to the same object. This may lead to an increase in the height of the tree. However, retrieval time is sped up.

Figure 18.7 is an example of one possible R^+-tree for the collection of line segments in Fig. 18.5. This particular tree is of order (2,3) although in general it is not possible to guarantee that all nodes will always have a minimum of 2 entries. In particular, the expected B-tree performance guarantees are not valid (i.e., pages are not guaranteed to be m/M full) unless we are willing to perform very complicated record insertion and deletion procedures. Notice that line segment objects `c`, `h`, and `i` appear in two different nodes. Of course, other variants are possible since the R^+-tree is not unique.

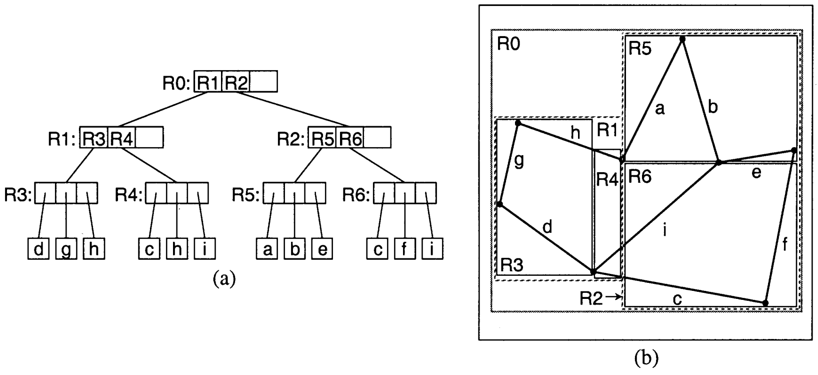

FIGURE 18.7 (a) R^+-tree for the collection of line segments in Fig.18.5 with $m = 2$ and $M = 3$, and (b) the spatial extents of the bounding rectangles. Notice that the leaf nodes in the index also store bounding rectangles, although this is only shown for the non-leaf nodes.

Methods such as the R^+-tree (as well as the R-tree) have the drawback that the decomposition is data-dependent. This means that it is difficult to perform tasks that require composition of different operations and data sets (e.g., set-theoretic operations such as overlay). The problem is that although these methods

[5]From a theoretical viewpoint, the bounding rectangles for the objects at the leaf nodes should also be disjoint However, this may be impossible (e.g., when the objects are line segments and if many of the line segments intersect at a point).

are good are distinguishing between occupied and unoccupied space in a particular image, they are unable to correlate occupied space in two distinct images, and likewise for unoccupied space in the two images.

In contrast, the remaining two approaches to the decomposition of space into disjoint cells have a greater degree of data-independence. They are based on a regular decomposition. The space can be decomposed either into blocks of uniform size (e.g., the uniform grid [19]) or adapt the decomposition to the distribution of the data (e.g., a quadtree-based approach such as [54]). In the former case, all the blocks are congruent (e.g., the 4×4 grid in Fig. 18.5). In the latter case, the widths of the blocks are restricted to be powers of two[6] and their positions are also restricted. Since the positions of the subdivision lines are restricted, and essentially the same for all images of the same size, it is easy to correlate occupied and unoccupied space in different images.

The uniform grid is ideal for uniformly distributed data, while quadtree-based approaches are suited for arbitrarily distributed data. In the case of uniformly distributed data, quadtree-based approaches degenerate to a uniform grid, albeit they have a higher overhead. Both the uniform grid and the quadtree-based approaches lend themselves to set-theoretic operations, and thus they are ideal for tasks that require the composition of different operations and data sets. In general, since spatial data are not usually uniformly distributed, the quadtree-based regular decomposition approach is more flexible. The drawback of quadtree-like methods is their sensitivity to positioning in the sense that the placement of the objects relative to the decomposition lines of the space in which they are embedded effects their storage costs and the amount of decomposition that takes place. This is overcome to a large extent by using a bucketing adaptation that decomposes a block only if it contains more than b objects.

18.4 Region Data

There are many ways of representing region data. We can represent a region either by its boundary (termed a **boundary-based representation**) or by its interior (termed an **interior-based representation**). In this section, we focus on representations of collections of regions by their interior. In some applications, regions are really objects that are composed of smaller primitive objects by use of geometric transformations and Boolean set operations. *Constructive solid geometry* (CSG) [48] is a term usually used to describe such representations. They are beyond the scope of this chapter. Instead, unless noted otherwise, our discussion is restricted to regions consisting of congruent cells of unit area (volume) with sides (faces) of unit size that are orthogonal to the coordinate axes.

Regions with arbitrary boundaries are usually represented by either using approximating bounding rectangles or more general boundary-based representations that are applicable to collections of line segments that do not necessarily form regions. In that case, we do not restrict the line segments to be perpendicular to the coordinate axes. Such representations are discussed in Section 18.6. It should be clear that although our presentation and examples in this section deal primarily with two-dimensional data, they are valid for regions of any dimensionality.

The region data is assumed to be uniform in the sense that all the cells that comprise each region are of the same type. In other words, each region is homogeneous. Of course, an image may consist of several distinct regions. Perhaps the best definition of a region is as a set of four-connected cells (i.e., in two dimensions, the cells are adjacent along an edge rather than a vertex) each of which is of the same type. For example, we may have a crop map where the regions correspond to the four-connected cells on which the same crop is grown. Each region is represented by the collection of cells that comprise it. The set of

[6]More precisely, for arbitrary attributes that can be locational and nonlocational, there exist $j \geq 0$ such that the product of w_i, the width of the block along attribute i, and 2^j is equal to the length of the range of values of attribute i.

collections of cells that make up all of the regions is often termed an *image array,* because of the nature in which they are accessed when performing operations on them. In particular, the array serves as an access structure in determining the region associated with a location of a cell as well as all remaining cells that comprise the region.

When the region is represented by its interior, then often we can reduce the storage requirements by aggregating identically valued cells into blocks. In the rest of this section we discuss different methods of aggregating the cells that comprise each region into blocks as well as the methods used to represent the collections of blocks that comprise each region in the image.

The collection of blocks is usually a result of a space decomposition process with a set of rules that guide it. There are many possible decompositions. When the decomposition is recursive, we have the situation that the decomposition occurs in stages and often, although not always, the results of the stages form a containment hierarchy. This means that a block b obtained in stage i is decomposed into a set of blocks b_j that span the same space. Blocks b_j are, in turn, decomposed in stage $i + 1$ using the same decomposition rule. Some decomposition rules restrict the possible sizes and shapes of the blocks as well as their placement in space. Some examples include

- Congruent blocks at each stage
- Similar blocks at all stages
- All sides of a block are of equal size
- All sides of each block are powers of two.

Other decomposition rules dispense with the requirement that the blocks be rectangular (i.e., there exist decompositions using other shapes such as triangles, etc.), while still others do not require that they be orthogonal, although, as stated before, we do make these assumptions here. In addition, the blocks may be disjoint or be allowed to overlap. Clearly, the choice is large. In the following, we briefly explore some of these decomposition processes. We restrict ourselves to disjoint decompositions, although this need not be the case (e.g., the field tree [18]).

The most general decomposition permits aggregation along all dimensions. In other words, the decomposition is arbitrary. The blocks need not be uniform or similar. The only requirement is that the blocks span the space of the environment. The drawback of arbitrary decompositions is that there is little structure associated with them. This means that it is difficult to answer queries such as determining the region associated with a given point, besides exhaustive search through the blocks. Thus, we need an additional data structure known as an index or an access structure. A very simple decomposition rule that lends itself to such an index in the form of an array is one that partitions a d-dimensional space having coordinate axes x_i into d-dimensional blocks by use of h_i hyperplanes that are parallel to the hyperplane formed by $x_i = 0$ $(1 \le i \le d)$. The result is a collection of $\prod_{i=1}^{d}(h_i + 1)$ blocks. These blocks form a grid of irregular-sized blocks rather than congruent blocks. There is no recursion involved in the decomposition process. We term the resulting decomposition an *irregular grid,* as the partition lines are at arbitrary positions in contrast to a *uniform grid* [19] where the partition lines are positioned so that all of the resulting grid cells are congruent.

Although the blocks in the irregular grid are not congruent, we can still impose an array access structure by adding d access structures termed *linear scales.* The linear scales indicate the position of the partitioning hyperplanes that are parallel to the hyperplane formed by $x_i = 0$ $(1 \le i \le d)$. Thus, given a location l in space, say (a,b) in two-dimensional space, the linear scales for the x and y coordinate values indicate the column and row, respectively, of the array access structure entry which corresponds to the block that contains l. The linear scales are usually represented as one-dimensional arrays although they can be implemented using tree access structures such as binary search trees, range trees, segment trees, etc.

Perhaps the most widely known decompositions into blocks are those referred to by the general terms *quadtree* and **octree** [50, 51]. They are usually used to describe a class of representations for two- and three-dimensional data (and higher as well), respectively, that are the result of a recursive decomposition of the environment (i.e., space) containing the regions into blocks (not necessarily rectangular) until the data in each block satisfies some condition (e.g., with respect to its size, the nature of the regions that comprise it, the number of regions in it, etc.). The positions and/or sizes of the blocks may be restricted or arbitrary. It is interesting to note that quadtrees and octrees may be used with both interior-based and boundary-based representations, although only the former are discussed in this section.

There are many variants of quadtrees and octrees (see also Sections 18.2, 18.5, and 18.6), and they are used in numerous application areas including high energy physics, VLSI, finite element analysis, and many others. Below, we focus on *region quadtrees* [35] and to a lesser extent on *region octrees* [32, 43] They are specific examples of interior-based representations for two- and three-dimensional region data (variants for data of higher dimension also exist), respectively, that permit further aggregation of identically-valued cells.

Region quadtrees and region octrees are instances of a restricted-decomposition rule where the environment containing the regions is recursively decomposed into four or eight, respectively, rectangular congruent blocks until each block is either completely occupied by a region or is empty (such a decomposition process is termed *regular*). For example, Fig. 18.8(a) is the block decomposition for the region quadtree corresponding to three regions A, B, and C. Notice that in this case, all the blocks are square, have sides whose size is a power of 2, and are located at specific positions. In particular, assuming an origin at the upper-left corner of the image containing the regions, then the coordinate values of the upper-left corner of each block (e.g., (a, b) in two dimensions) of size $2^i \times 2^i$ satisfy the property that $a \bmod 2^i = 0$ and $b \bmod 2^i = 0$.

The traditional, and most natural, access structure for a region quadtree corresponding to a d-dimensional image is a tree with a fanout of 2^d [e.g., Fig. 18.8(b)]. Each leaf node in the tree corresponds to a different block b and contains the identity of the region associated with b. Each non-leaf node f corresponds to a block whose volume is the union of the blocks corresponding to the 2^d sons of f. In this case, the tree is a containment hierarchy and closely parallels the decomposition in the sense that they are both recursive processes and the blocks corresponding to nodes at different depths of the tree are similar in shape.

Determining the region associated with a given point p is achieved by a process that starts at the root of the tree and traverses the links to the sons whose corresponding blocks contain p. This process has an $O(m)$ cost where the image has a maximum of m levels of subdivision (e.g., an image all of whose sides are of length 2^m).

Observe that using a tree with fanout 2^d as an access structure for a regular decomposition means that there is no need to record the size and location of the blocks as this information can be inferred

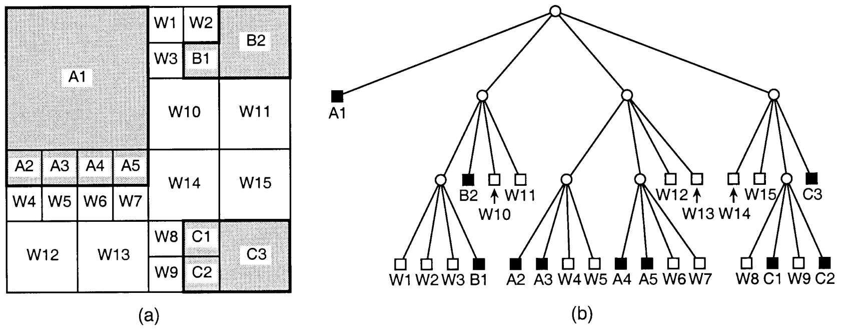

FIGURE 18.8 (a) Block decomposition and (b) its tree representation for the region quadtree corresponding to a collection of three regions A, B, and C.

from knowledge of the size of the underlying space. This is because the 2^d blocks that result from each subdivision step are congruent. For example, in two dimensions, each level of the tree corresponds to a quartering process that yields four congruent blocks. Thus, as long as we start from the root, we know the location and size of every block.

One of the motivations for the development of data structures such as the region quadtree is a desire to save space. The formulation of the region quadtree that we have just described makes use of an access structure in the form of a tree. This requires additional overhead to encode the internal nodes of the tree as well as the pointers to the subtrees. In order to further reduce the space requirements, a number of alternative access structures to the tree with fanout 2^d have been proposed. They are all based on finding a mapping from the domain of the blocks to a subset of the integers (i.e., to one dimension) and then using the result of the mapping as the index in one of the familiar tree-like access structures (e.g., a binary search tree, range tree, B^+-tree, etc.). The effect of these mappings is to provide an ordering on the underlying space. There are many possible orderings (e.g., Chapter 2 in [50]), with the most popular shown in Fig. 18.9. The domain of these mappings is the location of the cells in the underlying space, and thus we need to use some easily identifiable cell in each block such as the one in the block's upper-left corner. Of course, we also need to know the size of each block. This information can be recorded in the actual index as each block is uniquely identified by the location of the cell in its upper-left corner.

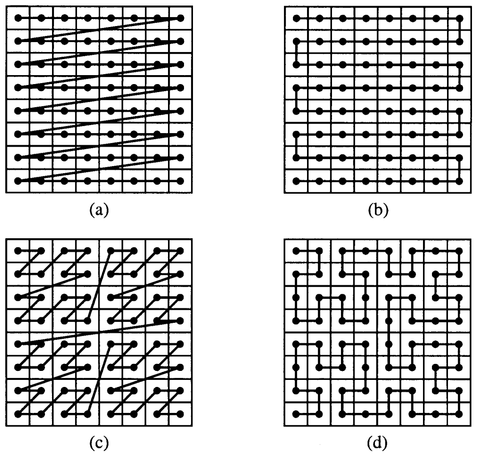

FIGURE 18.9 The result of applying four common different space-ordering methods to an 8×8 collection of cells whose first element is in the upper-left corner: (a) row order, (b) row-prime order, (c) Morton order, (d) Peano–Hilbert.

Since the size of each block b in the region quadtree can be specified with a single number indicating the depth in the tree at which b is found, we can simplify the representation by incorporating the size into the mapping. One mapping simply concatenates the result of interleaving the binary representations of the coordinate values of the upper-left corner (e.g., (a, b) in two dimensions) and i of each block of size 2^i so that i is at the right. The resulting number is termed a *locational code* and is a variant of the Morton order [Fig. 18.9(c)]. Assuming such a mapping and sorting the locational codes in increasing order yields an ordering equivalent to that which would be obtained by traversing the leaf nodes (i.e., blocks) of the tree representation [e.g., Fig. 18.8(b)] in the order NW, NE, SW, SE. The Morton ordering [as well as the Peano–Hilbert ordering shown in Fig. 18.9(d)] is particularly attractive for quadtree-like block decompositions, because all cells within a quadtree block appear in consecutive positions in the ordering. Alternatively, these two orders exhaust a quadtree block before exiting it. Therefore, once again, determining the region associated with point p consists of simply finding the block containing p.

As the dimensionality of the space (i.e., d) increases, each level of decomposition in the region quadtree results in many new blocks as the fanout value 2^d is high. In particular, it is too large for a practical implementation of the tree access structure. In this case, an access structure termed a *bintree* [36, 53, 60] with a fanout value of 2 is used. The bintree is defined in a manner analogous to the region quadtree except that at each subdivision stage, the space is decomposed into two equal-sized parts. In two dimensions, at odd stages we partition along the y axis and at even stages we partition along the x axis. In general, in the case of d dimensions, we cycle through the different axes every d levels in the bintree.

The region quadtree, as well as the bintree, is a regular decomposition. This means that the blocks are congruent—that is, at each level of decomposition, all of the resulting blocks are of the same shape and size. We can also use decompositions where the sizes of the blocks are not restricted in the sense that the only restriction is that they be rectangular and be a result of a recursive decomposition process. In this case, the representations that we described must be modified so that the sizes of the individual blocks can be obtained. An example of such a structure is an adaptation of the point quadtree [17] to regions. Although the point quadtree was designed to represent points in a higher dimensional space, the blocks resulting from its use to decompose space do correspond to regions. The difference from the region quadtree is that in the point quadtree, the positions of the partitions are arbitrary, whereas they are a result of a partitioning process into 2^d congruent blocks (e.g., quartering in two dimensions) in the case of the region quadtree.

As in the case of the region quadtree, as the dimensionality d of the space increases, each level of decomposition in the point quadtree results in many new blocks since the fanout value 2^d is high. In particular, it is too large for a practical implementation of the tree access structure. In this case, we can adapt the k-d tree [7], which has a fanout value of 2, to regions. As in the point quadtree, although the k-d tree was designed to represent points in a higher dimensional space, the blocks resulting from its use to decompose space do correspond to regions. Thus, the relationship of the k-d tree to the point quadtree is the same as the relationship of the bintree to the region quadtree. In fact, the k-d tree is the precursor of the bintree and its adaptation to regions is defined in a similar manner in the sense that for d-dimensional data we cycle through the d axes every d levels in the k-d tree. The difference is that in the k-d tree, the positions of the partitions are arbitrary, whereas they are a result of a halving process in the case of the bintree.

The k-d tree can be further generalized so that the partitions take place on the various axes at an arbitrary order, and, in fact, the partitions need not be made on every coordinate axis. The k-d tree is a special case of the *BSP tree* (denoting *binary space partitioning*) [23] where the partitioning hyperplanes are restricted to be parallel to the axes, whereas in the BSP tree they have an arbitrary orientation. The BSP tree is a binary tree. In order to be able to assign regions to the left and right subtrees, we need to associate a direction with each subdivision line. In particular, the subdivision lines are treated as separators between two halfspaces.[7] Let the subdivision line have the equation $a \cdot x + b \cdot y + c = 0$. We say that the right subtree is the "positive" side and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c \geq 0$. Similarly, we say that the left subtree is "negative" and contains all subdivision lines formed by separators that satisfy $a \cdot x + b \cdot y + c < 0$. As an example, consider Fig. 18.10(a), which is an arbitrary space decomposition whose BSP tree is given in Fig. 18.10(b). Notice the use of arrows to indicate the direction of the positive halfspaces. The BSP tree is used in computer graphics to facilitate viewing.

[7] A (linear) *halfspace* in d-dimensional space is defined by the inequality $\Sigma_{i=0}^{d} a_i \cdot x_i \geq 0$ on the $d+1$ homogeneous coordinates ($x_0 = 1$). The halfspace is represented by a column vector a. In vector notation, the inequality is written as $a \cdot x \geq 0$. In the case of equality, it defines a hyperplane with a as its normal. It is important to note that halfspaces are volume elements; they are not boundary elements.

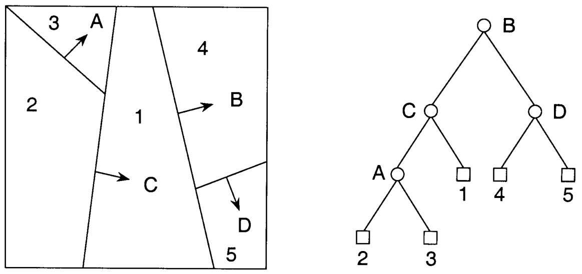

FIGURE 18.10 (a) An arbitrary space decomposition and (b) its BSP tree. The arrows indicate the direction of the positive halfspaces.

As mentioned before, the various hierarchical data structures that we described can also be used to represent regions in three dimensions and higher. As an example, we briefly describe the region octree which is the three-dimensional analog of the region quadtree. It is constructed in the following manner. We start with an image in the form of a cubical volume and recursively subdivide it into eight congruent disjoint cubes (called octants) until blocks are obtained of a uniform color or a predetermined level of decomposition is reached. Figure 18.11(a) is an example of a simple three-dimensional object whose region octree block decomposition is given in Fig. 18.11(b) and whose tree representation is given in Fig. 18.11(c).

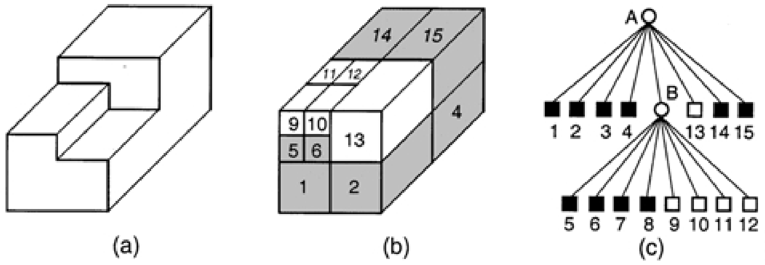

FIGURE 18.11 (a) Example three-dimensional object; (b) its region octree block decomposition; and (c) its tree representation.

The aggregation of cells into blocks in region quadtrees and region octrees is motivated, in part, by a desire to save space. Some of the decompositions have quite a bit of structure, thereby leading to inflexibility in choosing partition lines, etc. In fact, at times, maintaining the original image with an array access structure may be more effective from the standpoint of storage requirements. In the following, we point out some important implications of the use of these aggregations. In particular, we focus on the region quadtree and region octree. Similar results could also be obtained for the remaining block decompositions.

The aggregation of similarly valued cells into blocks has an important effect on the execution time of the algorithms that make use of the region quadtree. In particular, most algorithms that operate on images represented by a region quadtree are implemented by a preorder traversal of the quadtree, and thus their execution time is generally a linear function of the number of nodes in the quadtree. A key to the analysis of the execution time of quadtree algorithms is the **Quadtree Complexity Theorem** [32], which states that the number of nodes in a region quadtree representation for a simple polygon (i.e., with nonintersecting edges and without holes) is $O(p + q)$ for a $2^q \times 2^q$ image with perimeter p measured in terms of the width of unit-sized cells (i.e., pixels). In all but the most pathological cases (e.g., a small square of unit width centered in a large image), the q factor is negligible, and thus, the number of nodes is $O(p)$.

The Quadtree Complexity Theorem also holds for three-dimensional data [42] (i.e., represented by a region octree) where perimeter is replaced by surface area, as well as for objects of higher dimensions d for which it is proportional to the size of the $(d-1)$-dimensional interfaces between these objects. The most important consequence of the Quadtree complexity theorem is that it means that most algorithms that execute on a region quadtree representation of an image, instead of one that simply imposes an array access structure on the original collection of cells, usually have an execution time that is proportional to the number of blocks in the image rather than the number of unit-sized cells. In its most general case, this means that the use of the region quadtree, with an appropriate access structure, in solving a problem in d-dimensional space will lead to a solution whose execution time is proportional to the $(d-1)$-dimensional space of the surface of the original d-dimensional image. On the other hand, use of the array access structure on the original collection of cells results in a solution whose execution time is proportional to the number of cells that comprise the image. Therefore, region quadtrees and region octrees act like dimension-reducing devices.

18.5 Rectangle Data

The rectangle data type lies somewhere between the point and region data types. It can also be viewed as a special case of the region data type in the sense that it is a region with only four sides. Rectangles are often used to approximate other objects in an image for which they serve as the minimum rectilinear enclosing object. For example, bounding rectangles are used in cartographic applications to approximate objects such as lakes, forests, hills, etc. In such a case, the approximation gives an indication of the existence of an object. Of course, the exact boundaries of the object are also stored; but they are only accessed if greater precision is needed. For such applications, the number of elements in the collection is usually small, and most often the sizes of the rectangles are of the same order of magnitude as the space from which they are drawn.

Rectangles are also used in VLSI design rule checking as a model of chip components for the analysis of their proper placement. Again, the rectangles serve as minimum enclosing objects. In this application, the size of the collection is quite large (e.g., millions of components) and the sizes of the rectangles are several orders of magnitude smaller than the space from which they are drawn.

It should be clear that the actual representation that is used depends heavily on the problem environment. At times, the rectangle is treated as the Cartesian product of two one-dimensional intervals with the horizontal intervals being treated in a different manner than the vertical intervals. In fact, the representation issue is often reduced to one of representing intervals. For example, this is the case in the use of the plane-sweep paradigm [47] in the solution of rectangle problems such as determining all pairs of intersecting rectangles. In this case, each interval is represented by its left and right endpoints. The solution makes use of two passes.

The first pass sorts the rectangles in ascending order on the basis of their left and right sides (i.e., x coordinate values) and forms a list. The second pass sweeps a vertical scan line through the sorted list from left to right halting at each one of these points, say p. At any instant, all rectangles that intersect the scan line are considered *active* and are the only ones whose intersection needs to be checked with the rectangle associated with p. This means that each time the sweep line halts, a rectangle either becomes active (causing it to be inserted in the set of active rectangles) or ceases to be active (causing it to be deleted from the set of active rectangles). Thus, the key to the algorithm is its ability to keep track of the active rectangles (actually just their vertical sides) as well as to perform the actual one-dimensional intersection test.

Data structures such as the segment tree [8], interval tree [14], and the priority search tree [41] can be used to organize the vertical sides of the active rectangles so that, for N rectangles and F intersecting pairs of rectangles, the problem can be solved in $O(N \cdot \log_2 N + F)$ time. All three data structures enable intersection detection, insertion, and deletion to be executed in $O(\log_2 N)$ time. The difference between them is that the segment tree requires $O(N \cdot \log_2 N)$ space while the interval tree and the priority search tree only need $O(N)$ space.

The key to the use of the priority search tree to solve the rectangle intersection problem is that it treats each vertical side (y_B, y_T) as a point (x, y) in a two-dimensional space (i.e., it transforms the corresponding interval into a point as discussed in Section 18.1). The advantage of the priority search tree is that the storage requirements for the second pass only depend on the maximum number M of vertical sides that can be active at any one time. This is achieved by implementing the priority search tree as a red-black balanced binary tree [24], thereby guaranteeing updates in $O(\log_2 M)$ time. This also has an effect on the execution time of the second pass which is $O(N \cdot \log_2 M + F)$ instead of $O(N \cdot \log_2 N + F)$. Of course, the first pass which must sort the endpoints of the horizontal sides still takes $O(N \cdot \log_2 N)$ time for all three representations.

Most importantly, the priority search tree enables a more dynamic solution than either the segment or interval trees as only the endpoints of the horizontal sides need to be known in advance. On the other hand, for the segment and interval trees, the endpoints of both the horizontal and vertical sides must be known in advance. Of course, in all cases, all solutions based on the plane-sweep paradigm are inherently not dynamic as the paradigm requires that we examine all of the data one by one. Thus, the addition of even one new rectangle to the database forces the re-execution of the algorithm on the entire database.

In this chapter, we are primarily interested in dynamic problems. The data structures that are chosen for the collection of the rectangles are differentiated by the way in which each rectangle is represented. One representation discussed in Section 18.1 reduces each rectangle to a point in a higher dimensional space, and then treats the problem as if we have a collection of points [28]. Again, each rectangle is a Cartesian product of two one-dimensional intervals where the difference from its use with the plane-sweep paradigm is that each interval is represented by its centroid and extent. Each set of intervals in a particular dimension is, in turn, represented by a grid file [45], which is described in Section 18.2.

The second representation is region-based in the sense that the subdivision of the space from which the rectangles are drawn depends on the physical extent of the rectangle — not just one point. Representing the collection of rectangles, in turn, with a tree-like data structure has the advantage that there is a relation between the depth of node in the tree and the size of the rectangle(s) that is (are) associated with it. Interestingly, some of the region-based solutions make use of the same data structures that are used in the solutions based on the plane-sweep paradigm.

There are three types of region-based solutions currently in use. The first two solutions use the R-tree and the R$^+$-tree (discussed in Section 18.3) to store rectangle data (in this case the objects are rectangles instead of arbitrary objects). The third is a quadtree-based approach and uses the MX-CIF quadtree [34].

In the *MX-CIF quadtree*, each rectangle is associated with the quadtree node corresponding to the smallest block which contains it in its entirety. Subdivision ceases whenever a node's block contains no rectangles. Alternatively, subdivision can also cease once a quadtree block is smaller than a predetermined threshold size. This threshold is often chosen to be equal to the expected size of the rectangle [34]. For example, Fig. 18.12 is the MX-CIF quadtree for a collection of rectangles. Rectangles can be associated with both terminal and nonterminal nodes.

It should be clear that more than one rectangle can be associated with a given enclosing block, and thus, often we find it useful to be able to differentiate between them. This is done in the following manner [34]. Let P be a quadtree node with centroid (CX,CY), and let S be the set of rectangles that are associated with P. Members of S are organized into two sets according to their intersection (or collinearity of their sides) with the lines passing through the centroid of P's block—that is, all members of S that intersect the line $x = CX$ form one set and all members of S that intersect the line $y = CY$ form the other set.

If a rectangle intersects both lines (i.e., it contains the centroid of P's block), then we adopt the convention that it is stored with the set associated with the line through $x = CX$. These subsets are implemented as binary trees (really tries), which in actuality are one-dimensional analogs of the MX-CIF quadtree. For

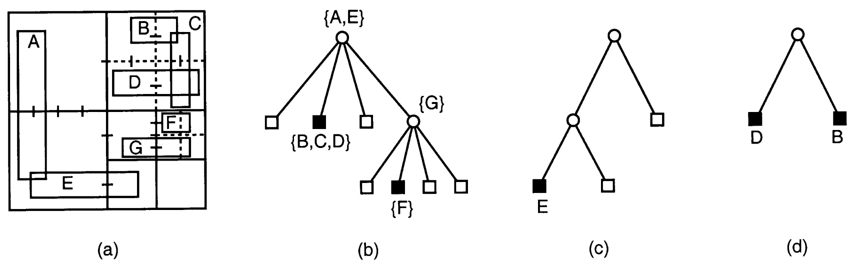

FIGURE 18.12 (a) Collection of rectangles and the block decomposition induced by the MX-CIF quadtree; (b) the tree representation of (a); (c) the binary trees for the y axes passing through the root of the tree in (b), and (d) the NE son of the root of the tree in (b).

example, Fig. 18.12(b) and Fig. 18.12(d) illustrate the binary trees associated with the y axes passing through the root and the NE son of the root, respectively, of the MX-CIF quadtree of Fig. 18.12(c). Interestingly, the MX-CIF quadtree is a two-dimensional analog of the interval tree described above. More precisely, the MX-CIF is a two-dimensional analog of the tile tree [40] which is a regular decomposition version of the interval tree. In fact, the tile tree and the one-dimensional MX-CIF quadtree are identical when rectangles are not allowed to overlap.

18.6 Line Data and Boundaries of Regions

Section 18.4 was devoted to variations on hierarchical decompositions of regions into blocks, an approach to region representation that is based on a description of the region's interior. In this section, we focus on representations that enable the specification of the boundaries of regions, as well as curvilinear data and collections of line segments. The representations are usually based on a series of approximations which provide successively closer fits to the data, often with the aid of bounding rectangles. When the boundaries or line segments have a constant slope (i.e., linear and termed *line segments* in the rest of this discussion), then an exact representation is possible.

There are several ways of approximating a curvilinear line segment. The first is by digitizing it and then marking the unit-sized cells (i.e., pixels) through which it passes. The second is to approximate it by a set of straight line segments termed a *polyline*. Assuming a boundary consisting of straight lines (or polylines after the first stage of approximation), the simplest representation of the boundary of a region is the polygon. It consists of vectors which are usually specified in the form of lists of pairs of x and y coordinate values corresponding to their start and end points. The vectors are usually ordered according to their connectivity. One of the most common representations is the chain code [21], which is an approximation of a polygon's boundary by use of a sequence of unit vectors in the four (and sometimes eight) principal directions.

Chain codes, and other polygon representations, break down for data in three dimensions and higher. This is primarily due to the difficulty in ordering their boundaries by connectivity. The problem is that in two dimensions connectivity is determined by ordering the boundary elements $e_{i,j}$ of boundary b_i of object o so that the end vertex of the vector v_j corresponding to $e_{i,j}$ is the start vertex of the vector v_{j+1} corresponding to $e_{i,j+1}$. Unfortunately, such an implicit ordering does not exist in higher dimensions as the relationship between the boundary elements associated with a particular object are more complex.

Instead, we must make use of data structures that capture the topology of the object in terms of its faces, edges, and vertices. The winged-edge data structure is one such representation that serves as the basis of the boundary model (also known as *BRep* [5]). Such representations are not discussed further here.

Polygon representations are very local. In particular, if we are at one position on the boundary, we don't know anything about the rest of the boundary without traversing it element by element. Thus, using such representations, given a random point in space, it is very difficult to find the nearest line to it as the lines are not sorted. This is in contrast to hierarchical representations which are global in nature. They are primarily based on rectangular approximations to the data as well as on a regular decomposition in two dimensions. In the rest of this section, we discuss a number of such representations.

In Section 18.3 we already examined two hierarchical representations (i.e., the R-tree and the R^+-tree) that propagate object approximations in the form of bounding rectangles. In this case, the sides of the bounding rectangles had to be parallel to the coordinate axes of the space from which the objects are drawn. In contrast, the *strip tree* [4] is a hierarchical representation of a single curve that successively approximates segments of it with bounding rectangles that does not require that the sides be parallel to the coordinate axes. The only requirement is that the curve be continuous; it need not be differentiable.

The strip tree data structure consists of a binary tree whose root represents the bounding rectangle of the entire curve. The rectangle associated with the root corresponds to a rectangular strip, that encloses the curve, whose sides are parallel to the line joining the endpoints of the curve. The curve is then partitioned in two at one of the locations where it touches the bounding rectangle (these are not tangent points as the curve only needs to be continuous; it need not be differentiable). Each subcurve is then surrounded by a bounding rectangle and the partitioning process is applied recursively. This process stops when the width of each strip is less than a predetermined value.

In order to be able to cope with more complex curves such as those that arise in the case of object boundaries, the notion of a strip tree must be extended. In particular, closed curves and curves that extend past their endpoints require some special treatment. The general idea is that these curves are enclosed by rectangles which are split into two rectangular strips, and from now on the strip tree is used as before.

The strip tree is similar to the point quadtree in the sense that the points at which the curve is decomposed depend on the data. In contrast, a representation based on the region quadtree has fixed decomposition points. Similarly, strip tree methods approximate curvilinear data with rectangles of arbitrary orientation, while methods based on the region quadtree achieve analogous results by use of a collection of disjoint squares having sides of length power of two. In the following we discuss a number of adaptations of the region quadtree for representing curvilinear data.

The simplest adaptation of the region quadtree is the MX quadtree [32, 33]. It is built by digitizing the line segments and labeling each unit-sized cell (i.e., pixel) through which it passes as of type `boundary`. The remaining pixels are marked `WHITE` and are merged, if possible, into larger and larger quadtree blocks. Figure 18.13(a) is the MX quadtree for the collection of line segment objects in Fig. 18.5. A drawback of the MX quadtree is that it associates a thickness with a line. Also, it is difficult to detect the presence of a vertex whenever five or more line segments meet.

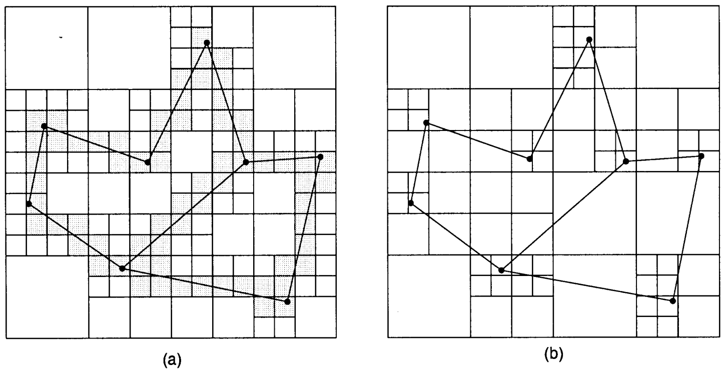

FIGURE 18.13 (a) MX quadtree and (b) edge quadtree for the collection of line segments of Fig.18.5.

The edge quadtree [57, 61] is a refinement of the MX quadtree based on the observation that the number of squares in the decomposition can be reduced by terminating the subdivision whenever the square contains a single curve that can be approximated by a single straight line. For example, Fig. 18.13(b) is the edge quadtree for the collection of line segment objects in Fig. 18.5. Applying this process leads to quadtrees in which long edges are represented by large blocks or a sequence of large blocks. However, small blocks are required in the vicinity of corners or intersecting edges. Of course, many blocks will contain no edge information at all.

The PM quadtree family [44, 54] (see also edge-EXCELL [59]) represents an attempt to overcome some of the problems associated with the edge quadtree in the representation of collections of polygons (termed *polygonal maps*). In particular, the edge quadtree is an approximation because vertices are represented by pixels. There are a number of variants of the PM quadtree. These variants are either vertex-based or edge-based. They are all built by applying the principle of repeatedly breaking up the collection of vertices and edges (forming the polygonal map) until obtaining a subset that is sufficiently simple so that it can be organized by some other data structure.

The PM_1 quadtree [54] is an example of a vertex-based PM quadtree. Its decomposition rule stipulates that partitioning occurs as long as a block contains more than one line segment unless the line segments are all incident at the same vertex which is also in the same block [e.g., Fig. 18.14(a)]. Given a polygonal map whose vertices are drawn from a grid (say $2^m \times 2^m$), and where edges are not permitted to intersect at points other than the grid points (i.e., vertices), it can be shown that the maximum depth of any leaf node in the PM_1 quadtree is bounded from above by $4m + 1$ [52]. This enables a determination of the maximum amount of storage that will be necessary for each node.

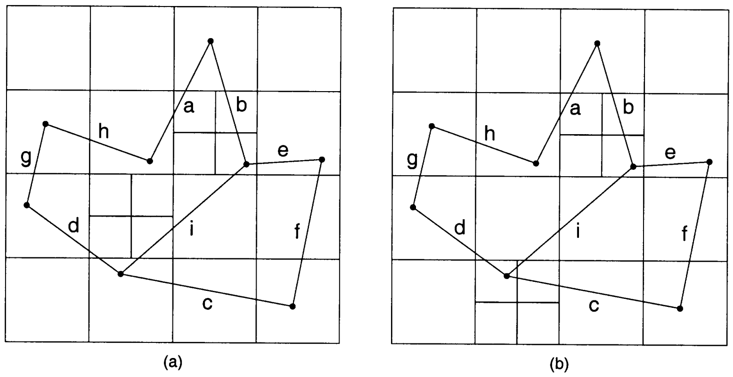

FIGURE 18.14 (a) PM_1 quadtree and (b) PMR quadtree for the collection of line segments of Fig. 18.5.

A similar representation has been devised for three-dimensional images (e.g., [3] and the references cited in [51]). The decomposition criteria are such that no node contains more than one face, edge, or vertex unless the faces all meet at the same vertex or are adjacent to the same edge. This representation is quite useful, since its space requirements for polyhedral objects are significantly smaller than those of a region octree.

The PMR quadtree [44] is an edge-based variant of the PM quadtree. It makes use of a probabilistic splitting rule. A node is permitted to contain a variable number of line segments. A line segment is stored in a PMR quadtree by inserting it into the nodes corresponding to all the blocks that it intersects. During this process, the occupancy of each node that is intersected by the line segment is checked to see if the insertion causes it to exceed a predetermined *splitting threshold.* If the splitting threshold is exceeded, then the node's block is split *once,* and only once, into four equal quadrants.

For example, Fig. 18.14(b) is the PMR quadtree for the collection of line segment objects in Fig. 18.5 with a splitting threshold value of 2. The line segments are inserted in alphabetic order (i.e., a–i). It should be clear that the shape of the PMR quadtree depends on the order in which the line segments are inserted. Note the difference from the PM_1 quadtree in Fig. 18.14(a)—that is, the NE block of the SW quadrant is decomposed in the PM_1 quadtree while the SE block of the SW quadrant is not decomposed in the PM_1 quadtree.

On the other hand, a line segment is deleted from a PMR quadtree by removing it from the nodes corresponding to all the blocks that it intersects. During this process, the occupancy of the node and its siblings is checked to see if the deletion causes the total number of line segments in them to be less than the predetermined splitting threshold. If the splitting threshold exceeds the occupancy of the node and its siblings, then they are merged and the merging process is reapplied to the resulting node and its siblings. Notice the asymmetry between the splitting and merging rules.

The PMR quadtree is very good for answering queries such as finding the nearest line to a given point [29, 30] (see [31] for an empirical comparison with hierarchical object representations such as the R-tree and R^+-tree). It is preferred over the PM_1 quadtree (as well as the MX and edge quadtrees) as it results in far fewer subdivisions. In particular, in the PMR quadtree there is no need to subdivide in order to separate line segments that are very "close" or whose vertices are very "close," which is the case for the PM_1 quadtree. This is important, since four blocks are created at each subdivision step. Thus, when many subdivision steps that occur in the PM_1 quadtree result in creating many empty blocks, the storage requirements of the PM_1 quadtree will be considerably higher than those of the PMR quadtree. Generally, as the splitting threshold is increased, the storage requirements of the PMR quadtree decrease while the time necessary to perform operations on it will increase.

Using a random image model and geometric probability, it has been shown [39], theoretically and empirically using both random and real map data, that for sufficiently high values of the splitting threshold (i.e., ≥ 4), the number of nodes in a PMR quadtree is asymptotically proportional to the number of line segments and is independent of the maximum depth of the tree. In contrast, using the same model, the number of nodes in the PM_1 quadtree is a product of the number of lines and the maximal depth of the tree (i.e., n for a $2^n \times 2^n$ image). The same experiments and analysis for the MX quadtree confirmed the results predicted by the Quadtree complexity theorem (see Section 18.4), which is that the number of nodes is proportional to the total length of the line segments.

Observe that although a bucket in the PMR quadtree can contain more line segments than the splitting threshold, this is not a problem. In fact, it can be shown [51] that the maximum number of line segments in a bucket is bounded by the sum of the splitting threshold and the depth of the block (i.e., the number of times the original space has been decomposed to yield this block).

18.7 Research Issues and Summary

A review has been presented of a number of representations of multidimensional data. Our focus has been on multidimensional spatial data with extent rather than just multidimensional point data. Moreover, the multidimensional data was not restricted to locational attributes in that the handling of nonlocational attributes for point data was also described. There has been a particular emphasis on hierarchical representations. Such representations are based on the "divide-and-conquer" problem-solving paradigm. They are of interest because they enable focusing computational resources on the interesting subsets of data. Thus, there is no need to expend work where the payoff is small. Although many of the operations for which they are used can often be performed equally as efficiently, or more so, with other data structures, hierarchical data structures are attractive because of their conceptual clarity and ease of implementation.

When the hierarchical data structures are based on the principle of regular decomposition, we have the added benefit that different data sets (often of differing types) are in registration. This means that they are partitioned in known positions that are often the same or subsets of one another for the different data sets. This is true for all the features including regions, points, rectangles, lines, volumes, etc. This means that a query such as "finding all cities with more than 20,000 inhabitants in wheat growing regions within 30 miles of the Mississippi River" can be executed by simply overlaying the region (crops), point (i.e., cities), and river maps even though they represent data of different types. Alternatively, we may extract regions such as those within 30 miles of the Mississippi River. Such operations find use in applications involving spatial data such as geographic information systems.

Current research in multidimensional representations is highly application-dependent in the sense that the work is driven by the application. Many of the recent developments have been motivated by the interaction with databases. The choice of a proper representation plays a key role in the speed with which responses are provided to queries. Knowledge of the underlying data distribution is also a factor and research is ongoing to make use of this information in the process of making a choice. Most of the initial applications in which the representation of multidimensional data has been important have involved spatial data of the kind described in this chapter. Such data is intrinsically of low dimensionality (i.e., two and three). Future applications involve higher dimensional data for applications such as image databases where the data are often points in feature space. The incorporation of the time dimension is also an important issue that confronts many database researchers.

18.8 Defining Terms

Bintree: A regular decomposition k-d tree for region data.

Boundary-based representation: A representation of a region that is based on its boundary.

Bucketing methods: Data organization methods that decompose the space from which spatial data is drawn into regions called buckets. Some conditions for the choice of region boundaries include the number of objects that they contain or on their spatial layout (e.g., minimizing overlap or coverage).

Fixed-grid method: Space decomposition into rectangular cells by overlaying a grid on it. If the cells are congruent (i.e., of the same width, height, etc.), then the grid is said to be uniform.

Interior-based representation: A representation of a region that is based on its interior (i.e., the cells that comprise it).

K-d tree: General term used to describe space decomposition methods that proceed by recursive decomposition across a single dimension at a time of the space containing the data until some condition is met such as that the resulting blocks contain no more than b objects (e.g., points, lines, etc.) or that the blocks are homogeneous. The k-d tree is usually a data structure for points which cycles through the dimensions as it decomposes the underlying space.

Multidimensional data: Data that has several attributes. It includes records in a database management system, locations in space, and also spatial entities that have extent such as lines, regions, volumes, etc.

Octree: A quadtree-like decomposition for three dimensional data.

Quadtree: General term used to describe space decomposition methods that proceed by recursive decomposition across all the dimensions (technically two dimensions) of the space containing the data until some condition is met such as that the resulting blocks contain no more than b objects (e.g., points, lines, etc.) or that the blocks are homogeneous (e.g., region data). The underlying space is not restricted to two-dimensions although this is the technical definition of the term. The result is usually a disjoint decomposition of the underlying space.

Quadtree complexity theorem: The number of nodes in a quadtree region representation for a simple polygon (i.e., with nonintersecting edges and without holes) is $O(p+q)$ for a $2^q \times 2^q$ image with perimeter p measured in pixel widths. In most cases, q is negligible, and thus, the number of nodes is proportional to the perimeter. It also holds for three-dimensional data where the perimeter is replaced by surface area, and in general for d-dimensions where instead of perimeter we have the size of the $(d-1)$-dimensional interfaces between the d-dimensional objects.

R-tree: An object hierarchy where associated with each element of the hierarchy is the minimum bounding rectangle of the union of the minimum bounding rectangles of the elements immediately below it. The elements at the deepest level of the hierarchy are groups of spatial objects. The result is usually a nondisjoint decomposition of the underlying space. The objects are aggregated on the basis of proximity and with the goal of minimizing coverage and overlap.

Regular decomposition: A space decomposition method that partitions the underlying space by recursively halving it across the various dimensions instead of permitting the partitioning lines to vary.

Acknowledgments

The assistance of Gisli Hjaltason in the preparation of the figures is greatly appreciated.

References

[1] Aref, W.G. and Samet. H., Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the Fifth International Symposium on Spatial Data Handling,* 178–189, Charleston, SC, Aug. 1992.

[2] Aref, W.G. and Samet, H., Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the Third International Conference on Information and Knowledge Management,* 347–354, Gaithersburg, MD, ACM Press. Dec. 1994.

[3] Ayala, D., Brunet, P., Juan, R., and Navazo, I., Object representation by means of nonminimal division quadtrees and octrees. *ACM Transactions on Graphics,* 4(1), 41–59, Jan. 1985.

[4] Ballard, D.H., Strip trees: a hierarchical representation for curves. *Communications of the ACM,* 24(5), 310–321, May 1981. (Also corrigendum, *Communications of the ACM,* 25, 3, Mar. 1982, 213.)

[5] Baumgart, B.G., A polyhedron representation for computer vision. In *Proceedings of the National Computer Conference 44,* 589–596, Anaheim, CA, May 1975.

[6] Beckmann, N., Kriegel, H.P., Schneider, R., and Seeger, B., The R^*-tree: an efficient and robust access method for points and rectangles. In *Proceedings of the ACM SIGMOD Conference,* 322–331, Atlantic City, NJ, Jun. 1990.

[7] Bentley, J.L., Multidimensional binary search trees used for associative searching. *Communications of the ACM,* 18(9), 509–517, Sep. 1975.

[8] Bentley, J.L., Algorithms for Klee's rectangle problems. (unpublished), 1977.

[9] Bentley, J.L. and Friedman, J.H., Data structures for range searching. *ACM Computing Surveys,* 11(4), 397–409, Dec. 1979.

[10] Bentley, J.L. and Mauer, H.A., Efficient worst-case data structures for range searching. *Acta Informatica,* 13, 155–168, 1980.

[11] Bentley, J.L., Stanat, D.F., and Williams, Jr., E.H., The complexity of finding fixed-radius near neighbors. *Information Processing Letters,* 6(6), 209–212, Dec. 1977.

[12] Comer, D., The ubiquitous B-tree. *ACM Computing Surveys,* 11(2), 121–137, Jun. 1979.

[13] de Berg, M., van Kreweld, M., Overmars, M., and Schwarzkopf, O., *Computational geometry: algorithms and applications.* Springer-Verlag, Berlin, Germany, 1997.

[14] Edelsbrunner, W., Dynamic rectangle intersection searching. Institute for Information Processing 47, Technical University of Graz, Graz, Austria, Feb. 1980.

[15] Edelsbrunner, H., A note on dynamic range searching. *Bulletin of the EATCS,* (15), 34–40, Oct. 1981.

[16] Edelsbrunner, H., *Algorithms in Combinatorial Geometry.* Springer-Verlag, Berlin, 1987.

[17] Finkel, R.A. and Bentley, J.L., Quad trees: a data structure for retrieval on composite keys. *Acta Informatica,* 4(1), 1–9, 1974.

[18] Frank, A.U. and Barrera, R., The fieldtree: a data structure for geographic information systems. In *Design and Implementation of Large Spatial Databases — First Symposium, SSD'89,* Buchmann, A., Günther, O., Smith, T.R., and Wang, Y.F., Eds., 29–44, Santa Barbara, Jul. 1989. (Also Springer-Verlag Lecture Notes in Computer Science 409.)

[19] Franklin, W.R., Adaptive grids for geometric operations. *Cartographica,* 21(2&3), 160–167, Summer & Autumn, 1984.

[20] Fredkin, E., Trie memory. *Communications of the ACM,* 3(9), 490–499, Sep. 1960.

[21] Freeman, H., Computer processing of line-drawing images. *ACM Computing Surveys,* 6(1), 57–97, Mar. 1974.

[22] Freeston, M., The BANG file: a new kind of grid file. In *Proceedings of the ACM SIGMOD Conference,* 260–269, San Francisco, CA, May 1987.

[23] Fuchs, H., Kedem, Z.M., and Naylor, B.F., On visible surface generation by a priori tree structures. *Computer Graphics,* 14(3), 124–133, Jul. 1980. (Also *Proceedings of the SIGGRAPH'80 Conference,* Seattle, WA, Jul. 1980).

[24] Guibas, L.J. and Sedgewick, R., A dichromatic framework for balanced trees. In *Proceedings of the Nineteenth Annual IEEE Symposium on the Foundations of Computer Science,* 8–21, Ann Arbor, MI, Oct. 1978.

[25] Günther, O., *Efficient structures for geometric data management.* Ph.D. Thesis, University of California at Berkeley, Berkeley, CA, 1987. (Also Lecture Notes in Computer Science 337, Springer-Verlag, Berlin, 1988).

[26] Guttman, A., R-trees: a dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference,* 47–57, Boston, MA, Jun. 1984.

[27] Henrich, A., Six, H.W., and Widmayer, P., The LSD tree: spatial access to multidimensional point and non-point data. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases,* Apers, P.M.G. and Wiederhold, G., Eds., 45–53, Amsterdam, The Netherlands, Aug. 1989.

[28] Hinrichs, K. and Nievergelt, J., The grid file: a data structure designed to support proximity queries on spatial objects. In *Proceedings of the WG'83 (International Workshop on Graphtheoretic Concepts in Computer Science),* Nagl, M. and Perl, J., Eds., 100–113, Linz, Austria, 1983. Trauner Verlag.

[29] Hjaltason, G.R. and Samet, H., Ranking in spatial databases. In *Advances in Spatial Databases — Fourth International Symposium, SSD'95,* Egenhofer, M.J. and Herring, J.R., Eds., 83–95, Portland, ME, Aug. 1995. (Also Springer-Verlag Lecture Notes in Computer Science 951).

[30] Hoel, E.G. and Samet, H., Efficient processing of spatial queries in line segment databases. In *Advances in Spatial Databases — Second Symposium, SSD'91,* Günther, O. and Schek, H.J., Eds., 237–256, Zurich, Switzerland, Aug. 1991. (Also Springer-Verlag Lecture Notes in Computer Science 525).

[31] Hoel, E.G. and Samet, H., A qualitative comparison study of data structures for large line segment databases. In *Proceedings of the ACM SIGMOD Conference,* 205–214, San Diego, CA, Jun. 1992.

[32] Hunter, G.M., *Efficient computation and data structures for graphics.* Ph.D. Thesis, Princeton University, Princeton, NJ, 1978.

[33] Hunter, G.M. and Steiglitz, K., Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 1(2), 145–153, Apr. 1979.

[34] Kedem, G., The quad-CIF tree: a data structure for hierarchical on-line algorithms. In *Proceedings of the Nineteenth Design Automation Conference,* 352–357, Las Vegas, Jun. 1982.

[35] Klinger, A., Patterns and search statistics. In *Optimizing Methods in Statistics,* Rustagi, J.S., Ed., 303–337. Academic Press, New York, 1971.

[36] Knowlton, K., Progressive transmission of grey-scale and binary pictures by simple efficient, and lossless encoding schemes. *Proceedings of the IEEE,* 68(7), 885–896, Jul. 1980.

[37] Knuth, D.E., *The Art of Computer Programming vol. 3, Sorting and Searching.* Addison-Wesley, Reading, MA, 1973.

[38] Knuth, D.E., Big omicron and big omega and big theta. *SIGACT News,* 8(2), 18–24, Apr.-Jun. 1976.

[39] Lindenbaum, M. and Samet, H., A probabilistic analysis of trie-based sorting of large collections of line segments. Computer Science Department TR-3455, University of Maryland, College Park, MD, Apr. 1995.

[40] McCreight, E.M., Efficient algorithms for enumerating intersecting intervals and rectangles. Technical Report CSL-80-09, Xerox Palo Alto Research Center, Palo Alto, CA, Jun. 1980.

[41] McCreight, E.M., Priority search trees. *SIAM Journal on Computing,* 14(2), 257–276, May 1985.

[42] Meagher, D., Octree encoding: a new technique for the representation, the manipulation, and display of arbitrary 3-d objects by computer. Electrical and Systems Engineering IPL-TR-80-111, Rensselaer Polytechnic Institute, Troy, NY, Oct. 1980.

[43] Meagher, D., Geometric modeling using octree encoding. *Computer Graphics and Image Processing,* 19(2), 129–147, Jun. 1982.

[44] Nelson, R.C. and Samet, H., A consistent hierarchical representation for vector data. *Computer Graphics,* 20(4), 197–206, Aug. 1986. (Also *Proceedings of the SIGGRAPH'86 Conference,* Dallas, Aug. 1986).

[45] Nievergelt, J., Hinterberger, H., and Sevcik, K.C., The grid file: an adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems,* 9(1), 38–71, Mar. 1984.

[46] Orenstein, J.A., Multidimensional tries used for associative searching. *Information Processing Letters,* 14(4), 150–157, Jun, 1982.

[47] Preparata, F.P. and Shamos, M.I., *Computational Geometry: An Introduction.* Springer-Verlag, New York, 1985.

[48] Requicha, A.A.G., Representations of rigid solids: theory, methods, and systems. *ACM Computing Surveys,* 12(4), 437–464, Dec. 1980.

[49] Robinson, J.T., The *k–d–b*–tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference,* 10–18, Ann Arbor, MI, Apr. 1981.

[50] Samet, H., *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[51] Samet, H., *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[52] Samet, H., Shaffer, C.A., and Webber, R.E., Digitizing the plane with cells of non–uniform size. *Information Processing Letters,* 24(6), 369–375, Apr. 1987.

[53] Samet, H. and Tamminen, M., Efficient component labeling of images of arbitrary dimension represented by linear bintrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence,* 10(4), 579–586, Jul. 1988.

[54] Samet, H. and Webber, R.E., Storing a collection of polygons using quadtrees. *ACM Transactions on Graphics,* 4(3), 182–222, Jul. 1985. (Also *Proceedings of Computer Vision and Pattern Recognition 83,* Washington, DC, Jun. 1983, 127–132, and University of Maryland Computer Science TR–1372).

[55] Seeger, B. and Kriegel, H.P., The buddy-tree: an efficient and robust access method for spatial data base systems. In *Proceedings of the 16th International Conference on Very Large Databases (VLDB),* McLeod, D., Sacks-Davis, R., and Schek, H., Eds., 590–601, Brisbane, Australia, Aug. 1990.

[56] Sellis, T., Roussopoulos, N., and Faloutsos, C., The R^+–tree: a dynamic index for multi–dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB),* Stocker, P.M. and Kent, W., Eds., 71–79, Brighton, England, Sep. 1987. (Also University of Maryland Computer Science TR–1795).

[57] Shneier, M., Two hierarchical linear feature representations: edge pyramids and edge quadtrees. *Computer Graphics and Image Processing,* 17(3), 211–224, Nov. 1981.

[58] Stonebraker, M., Sellis, T., and Hanson, E., An analysis of rule indexing implementations in data base systems. In *Proceedings of the First International Conference on Expert Database Systems,* 353–364, Charleston, SC, Apr. 1986.

[59] Tamminen, M., The EXCELL method for efficient geometric access to data. *Acta Polytechnica Scandinavica,* 1981. (Mathematics and Computer Science Series No. 34).

[60] Tamminen, M., Comment on quad– and octtrees. *Communications of the ACM,* 27(3), 248–249, Mar. 1984.

[61] Warnock, J.E., A hidden surface algorithm for computer generated half tone pictures. Computer Science Department TR 4–15, University of Utah, Salt Lake City, Jun. 1969.

Further Information

It is impossible to give a complete enumeration of where research on multidimensional data structures is published, since it is often mixed with the application. Hands-on experience with some of the representations described in this chapter can be obtained by looking at the JAVA applets on `http://www.cs.umd.edu/~hjs/quadtree/index.html`. Multidimensional spatial data is covered in the texts by Samet [50, 51]. Their perspective is one from computer graphics, image processing, geographic information systems (GIS), databases, solid modeling, as well as VLSI design and computational geometry. A more direct computational geometry perspective can be found in the books by Edelsbrunner [16], Preparata and Shamos [47], and Overmars et al. [13].

New developments in the field of multidimensional data structures are reported in many different conferences, again since it is so application-driven. Some good starting pointers from the GIS perspective are the Symposium on Spatial Databases and the International Workshop on Spatial Data Handling, which are held in alternating years. From the standpoint of computational geometry, the annual ACM Symposium on Computational Geometry and the annual ACM-SIAM Symposium on Discrete Algorithms are good sources. From the perspective of databases, the annual ACM Conference on the Management of Data (SIGMOD) and the Very Large Database Conference (VLDB) usually contain a few papers dealing with the application of such representation. Other useful sources are the proceedings of the annual SIGGRAPH Conference.

Journals where such research appears are as varied as the applications. Theoretical results can be found in *SIAM Journal of Computing* while those from the GIS perspective may be found in a new journal called *GeoInformatica.* Many related articles are also found in the computer graphics and computer vision journals such as *ACM Transactions on Graphics,* the old *Computer Vision, Graphics and Image Processing,* which has been renamed *Graphical Models and Image Processing* and *Image Understanding,* and *IEEE Transactions on Pattern Analysis and Machine Intelligence.*