CAR-TR-990 September 2003
CS-TR-4526
UMIACS 2003-94

# Object-based and Image-based Object Representations

Hanan Samet

Computer Science Department
Center for Automation Research
Institute for Advanced Computer Studies
University of Maryland
College Park, Maryland 20742

## Abstract

An overview is presented of object-based and image-based representations of objects by their interiors. The representations are distinguished by the manner in which they can be used to answer two fundamental queries in database applications: (1) Feature query: given an object, determine its constituent cells (i.e., their locations in space). (2) Location query: given a cell (i.e., a location in space), determine the identity of the object (or objects) of which it is a member as well as the remaining constituent cells of the object (or objects). Regardless of the representation that is used, the generation of responses to the feature and location queries is facilitated by building an index (i.e., the result of a sort) either on the objects or on their locations in space, and implementing it using an access structure that correlates the objects with the locations. Assuming the presence of an access structure, implicit (i.e., image-based) representations are described that are good for finding the objects associated with a particular location or cell (i.e., the location query), while requiring that all cells be examined when determining the locations associated with a particular object (i.e., the feature query). In contrast, explicit (i.e., object-based) representations are good for the feature query, while requiring that all objects be examined when trying to respond to the location query. The goal is to be able to answer both types of queries with one representation and without possibly having to examine every cell. Representations are presented that achieve this goal by imposing containment hierarchies on either space (i.e., the cells in the space in which the objects are found), or objects. In the former case, space is aggregated into successively larger-sized chunks (i.e., blocks), while in the latter, objects are aggregated into successively larger groups (in terms of the number of objects that they contain). The former is applicable to image-based interior-based representations of which the space pyramid is an example. The latter is applicable to object-based interior-based representations of which the R-tree is an example. The actual mechanics of many of these representations are demonstrated in the VASCO JAVA applets found at `http://www.cs.umd.edu/~hjs/quadtree/index.html`.

The support of the National Science Foundation under Grants EIA-99-00268, IIS-00-86162, and EIA-00-91474, and Microsoft Research is gratefully acknowledged.

# 1 Introduction

The representation of spatial objects and their environment is an important issue in building and maintaining databases (e.g., [135]) to support applications in computer graphics, game programming, computer vision, image processing, robotics, pattern recognition, computational geometry, and geographic information systems (GIS). In this survey our goal is to introduce practitioners and researchers in these areas to these representations. Bearing these goals in mind, whenever there are several ways of explaining a concept we use the terminology and notation that is common to these fields rather than that which is more commonly used in spatial databases (e.g., [121, 148]). Thus the survey is not necessarily aimed at database researchers (as is for example [41]) although we hope that they will also find it useful. Please note also that we are focussing on the representation of spatial objects which means that the objects have extent (e.g., [13, 133, 134, 153]) rather than being merely points. The representation of multidimensional points has been much studied in the database literature (e.g., [26, 61, 134]).

We assume that the objects are connected[1] although their environment need not be. The objects and their environment are usually decomposed into collections of more primitive elements (termed *cells*) each of which has a location in space, a size, and a shape. These elements can either be subobjects of varying shape (e.g., a table consists of a flat top in the form of a rectangle and four legs in the form of rods whose lengths dominate their cross-sectional areas), or can have a uniform shape. The former yields an *object-based* decomposition while the latter yields an *image-based* or *cell-based* decomposition. Another way of characterizing these two decompositions is that the former decomposes the objects while the latter decomposes the environment in which the objects lie. This distinction is commonly used to characterize algorithms in computer graphics (e.g., [55]).

Each of the decompositions has its advantages and disadvantages. They depend primarily on the nature of the queries that are posed to the database. The most general queries ask *where*, *what*, *who*, *why*, and *how*. The ones that are relevant to our application are *where* and *what*. They are stated more formally as follows:

1. Feature query: given an object, determine its constituent cells (i.e., their locations in space).
2. Location query: given a cell (i.e., a location in space), determine the identity of the object (or objects) of which it is a member as well as the remaining constituent cells of the object (or objects).

Not surprisingly, the queries can be classified using the same terminology that we used in the characterization of the decomposition. In particular, we can either try to find the cells (i.e., their locations in space) occupied by an object or find the objects that overlap a cell (i.e., a location in space). If objects are associated with cells so that a cell contains the identity of the relevant object (or objects), then the feature query is analogous to retrieval by contents while the location query is analogous to retrieval by location.

The feature and location queries are the basis of two more general classes of queries. In particular, the feature query is a member of a broader class of queries described collectively as being *feature-based* (also *object-based*), while the location query is a member of a broader class of queries described collectively as

[1] Intuitively, this means that a $d$-dimensional object cannot be decomposed into disjoint subobjects so that the subobjects are not adjacent in a $(d-1)$-dimensional sense.

being *location-based* (also *image-based* or *cell-based*). The location query is also commonly referred to as a point query in databases (e.g., [96]), the point location problem in computational geometry (e.g., [25, 119]), and a "pick" operation in computer graphics (e.g., [55]) which is actually used to find the nearest object to a given location. The class of location-based queries include the numerous variants of the *window query* which retrieves the objects that cover an arbitrary region (often rectangular). All of these queries are used in several applications including geographic information systems (e.g., [6, 138]) and spatial data mining (e.g., [162]). It is important to note that the only reason that we discuss these particular queries is that they are the motivation for the different representations that we present. Of course, there are many other possible queries such as a more generalized formulation of neighbor finding, Boolean set operations, spatial joins, etc. However, their discussion is beyond the scope of this survey.

The most common representation of the objects and their environment is as a collection of cells of uniform size and shape (termed *pixels* and *voxels* in two and three dimensions, respectively) all of whose boundaries (with dimensionality one less than that of the cells) are of unit size. Such a representation is used in many applications in image processing, computer graphics, remote sensing, and geographic information systems (GIS) where it is known as a *raster* representation. In fact, it forms the basis of the map algebra system of Tomlin [159] which has been implemented in a number of GIS systems (e.g., ARC/INFO). Since the cells are uniform, there exists a way of referring to their locations in space relative to a fixed reference point (e.g., the origin of the coordinate system). An example of a location of a cell in space is a set of coordinate values that enable us to find it in the $d$-dimensional space of the environment in which it lies. Note that the concept of the *location* of a cell in space is quite different from that of the *address* of a cell, which is the physical location (e.g., in memory, on disk, etc.), if any, where some of the information associated with the cell is stored. This distinction between the location in space of a cell and the address of a cell is important and we shall make use of it often.

In most applications, the boundaries (i.e., edges and faces in two and three dimensions, respectively) of the cells are parallel to the coordinate axes. In our discussion, we assume that the cells comprising a particular object are contiguous (i.e., adjacent), and that a different unique value is associated with each distinct object, thereby enabling us to distinguish between the objects. Depending on the underlying representation, this value may be stored with the cells. For example, Figure 1 contains three two-dimensional objects A, B, and C and their corresponding cells. Note that in this example there exist two hyperplanes that will separate the objects; however, this is not crucial to our discussion. We also observe that, although not the case in this example, objects are allowed to overlap which means that a cell may be associated with more than one object. Here we assume, without loss of generality, that the volume of the overlap must be an integer multiple of the volume of a cell (i.e., pixels, voxels, etc.).

The shape of an object $o$ can be represented either by the interiors of the cells comprising $o$, or by the subset of the boundaries of those cells comprising $o$ that are adjacent to the boundary of $o$. In particular, interior-based methods represent an object $o$ by using the locations in space of the cells that comprise $o$, while boundary-based methods represent $o$ by using the locations in space of the cells that are adjacent to the boundary of $o$. In general, interior-based representations make it very easy to calculate properties of an object such as its mass, and, depending on the nature of the aggregation process, to determine the value associated

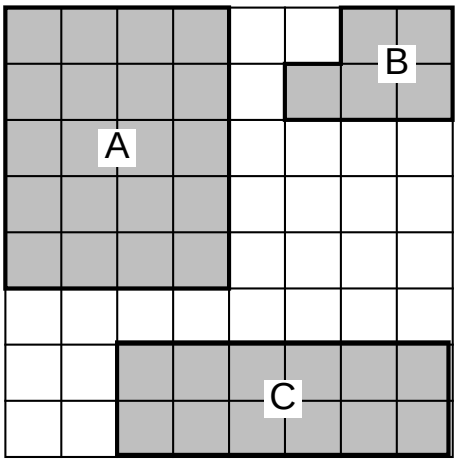


Figure 1: Example collection of three objects and the cells that they occupy.

with any point (i.e., location) in the space covered by a cell in the object. On the other hand, boundary-based representations make it easy to obtain the boundary of an object. The first boundary representation was the chain code [57], which is a digitized version of a vector representation. Other more commonly used boundary representations are those known collectively as the boundary model (also referred to as *BRep*) among which are included a number of variants of the winged-edge data structure (e.g., [17, 70, 87, 134]), as well as representations such as the BSP tree (denoting *Binary Space Partitioning*) [60] which is mentioned briefly in Section 3. In this paper the focus is on interior-based representations and thus further discussion of these representations is beyond our scope.

The simplest representation represents the objects by use of collections of unit-size cells. The collection can be represented either explicitly or implicitly. The representation is *explicit* if the identities of all the contiguous cells that form the object are hardwired into the representation (characterized as being *object-based*), while the representation is *implicit* if we can only determine the cells that make up the object by examining the contiguous cells of a given cell and determining if they are associated with the same object (characterized as being *image-based*).

These representations can be made more compact by aggregating similar elements (i.e., unit-size cells). These elements are usually identically-valued contiguous cells, or even objects which, ideally, are in proximity. The result is that the cells that make up the object collection no longer need to be of unit size and their sizes can vary. The varying-sized cells are termed *blocks*.

Regardless of the representation that is used, the generation of responses to the feature and location queries is facilitated by building an index (i.e., the result of a sort) either on the objects or on their locations in space, and implementing it using an access structure that correlates the objects with the locations. Assuming the presence of an access structure, the implicit (i.e., image-based) representations described in Sections 2 and 3 are good for finding the objects associated with a particular location or cell (i.e., the location query), while requiring that all cells be examined when determining the locations associated with a particular object (i.e., the feature query). In contrast, the explicit (i.e., object-based) representations described in Sections 2 and 3 are good for the feature query, while requiring that all objects be examined when trying to respond to

the location query. Our goal is to be able to answer both types of queries with one representation and without possibly having to examine every cell. This is the main focus of this paper.

We achieve our goal by imposing containment hierarchies on the representations. The hierarchies differ depending on whether the hierarchy is of space (i.e., the cells in the space in which the objects are found), or of objects. In the former case, we aggregate space into successively larger-sized chunks (i.e., blocks), while in the latter, we aggregate objects into successively larger groups (in terms of the number of objects that they contain). The former is applicable to implicit (i.e., image-based) interior-based representations, while the latter is applicable to explicit (i.e., object-based) interior-based representations.

The basic idea is that in image-based representations, we propagate objects up the hierarchy, with the occupied space being implicit to the representation. Thus we retain the property that associated with each cell is an identifier indicating the object of which it is a part. In fact, it is this information that is propagated up the hierarchy so that each element in the hierarchy contains the union of the objects that appear in the elements immediately below it.

On the other hand, in the object-based representations, we propagate the space occupied by the objects up the hierarchy, with the identities of the objects being implicit to the representation. Thus we retain the property that associated with each object is a 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. The approximation is usually the minimum bounding box for the object that is customarily stored with the explicit representation. Therefore, associated with each element in the hierarchy is a bounding box corresponding to the union of the bounding boxes associated with the elements immediately below it. The bounding box is quite general in the sense that it can be used to approximate all types of data rather than just objects with axis-parallel boundaries as in this paper.

The use of the bounding box approximation has the drawback that the bounding boxes at a given level in the hierarchy are not necessarily disjoint, which means that responding to the location query may require all of the bounding boxes to be visited as the space spanned by an object may be included in several bounding boxes; however, the object is only associated with one of the bounding boxes. This can be overcome by decomposing the bounding boxes so that disjointness holds. The drawback of this solution is that an object may be associated with more than one bounding box, which may result in the object being reported as satisfying a particular query more than once. For example, suppose that we want to retrieve all the objects that overlap a particular region (i.e., a window query) rather than a point as is done in the location query.

It is very important to note that the presence of the hierarchy does not mean that the alternative query (i.e., the feature query in the case of a space hierarchy and the location query in the case of an object hierarchy) can be answered immediately. Instead, obtaining the answer usually requires that the hierarchy be descended. The effect is that the order of the execution time needed to obtain the answer is reduced from linear to logarithmic. Of course, this is not always the case. For example, the fact that we are using bounding boxes for the space spanned by the objects rather than the exact space occupied by them means that we do not always have a complete answer when reaching the bottom of the hierarchy. In particular, at this point, we may have to

resort to a more expensive point-in-polygon test [55].

It is worth repeating that the only reason for imposing the hierarchy is to facilitate responding to the alternative query (i.e., the feature query in the case of a space hierarchy on the implicit representation, and the location query in the case of an object hierarchy on the explicit representation). Thus the base representation of the hierarchy is still usually used to answer the original query, because often, when using the hierarchy, the inherently logarithmic overhead incurred by the need to descend the hierarchy may be too expensive (e.g., when using the implicit representation with an array access structure to respond to the location query). Of course, other considerations such as space requirements may cause us to modify the base representation of the hierarchy, with the result that it will take longer to respond to the original query (e.g., the use of a tree-like access structure with an implicit representation). Nevertheless, as a general rule, in the case of the space hierarchy, we use the implicit representation (which is the basis of this hierarchy) to answer the location query, while in the case of the object hierarchy, we use the explicit representation (which is the basis of this hierarchy) to answer the feature query.

The rest of this paper is organized as follows. Section 2 examines both image-based and object-based representations that consist of collections of unit-size cells, while Section 3 reviews how these representations are made more compact by aggregating similar elements into blocks. Sections 4 and 5 describe how to modify the image-based and object-based representations, respectively, to be hierarchical. Section 6 briefly discusses some disjoint object-based representations, while concluding remarks are drawn in Section 7. Note that all of the representations that we discuss can be used in a dynamic environment although some more easily in the sense that updates are less costly to process as less of the representation needs to be rebuilt in the case of an update. The actual mechanics of many of these representations are demonstrated in the VASCO JAVA applets found at `http://www.cs.umd.edu/~hjs/quadtree/index.html` [28].

In order to simplify matters, the representations described in Sections 2 and 3 assume that the objects can be decomposed into cells whose boundaries are parallel to the coordinate axes. Moreover, it is assumed that each unit-size cell or block is contained entirely in one or more objects — that is, a cell or block cannot be partially contained in two objects. This means that either each cell in a block belongs to the same object or objects, or all of the cells in the block do not belong to any of the objects. Of course, as we pointed out before, more complex objects are possible (e.g., arbitrary polyhedra as well as collections of objects whose boundaries do not coincide with the boundaries of the underlying blocks) and also a cell or a block could be allowed to overlap several objects without being completely contained in them. In this case, the hierarchical object-based representations which are presented in Section 5 are the most appropriate and the discussion therein is applicable.

## 2 Unit-size Cells

Interior-based representations aggregate identically-valued cells by recording their locations in space. When the aggregation is explicit, the identities of the contiguous cells that form the object are hardwired into the representation. An example of an explicit aggregation is one that associates a set with each object *o* that

contains the location in space of each cell that comprises $o$. In this case, no identifying information (e.g., the object identifier corresponding to $o$) is stored in the cells. Thus there is no need to allocate storage for the cells (i.e., no addresses are associated with them). One possible implementation of this set is a list. For example, consider Figure 1 and assume that the origin $(0,0)$ is at the upper-left corner. Assume further that this is also the location of the pixel that abuts this corner. Therefore, the explicit representation of object B is the set of locations $\{(5,1)\ (6,0)\ (6,1)\ (7,0)\ (7,1)\}$. It should be clear that using the explicit representation, given an object $o$, it is easy to determine the cells (i.e., locations in space) that comprise it (the feature query).

Of course, even when using an explicit representation, we must still be able to access object $o$ from a possibly large collection of objects, which may require an additional data structure such as an index on the objects (e.g., a table of object-value pairs where *value* indicates the entry in the explicit representation corresponding to *object*). This index does not make use of the spatial coverage of the objects and thus may be implemented using conventional searching techniques such as hashing [96]. In this case, we will need $O(N)$ additional space for the index, where $N$ is the number of different objects. We do not discuss such indexes here.

The fact that no identifying information as to the nature of the object is stored in the cell means that the explicit representation is not suited for answering the inverse query of determining the object associated with a particular cell at location $l$ in space (i.e., the location query). Using the explicit representation, the location query can be answered only by checking for the presence of location $l$ in space in the various sets associated with the different objects. This will be time-consuming, as it may require that we examine all cells in each set. In other words, the explicit representation is primarily suited to retrieval on the basis of knowledge of the objects rather than of the locations of the cells in space and this is the rationale for characterizing it as being *object-based*.

Note that since the explicit representation consists of sets, there is no particular order for the cells within each set although an ordering could be imposed based on spatial proximity of the locations of the cells in space, etc. For example, the list representation of a set already presupposes the existence of an ordering. Such an ordering could be used to obtain a small, but not insignificant, decrease in the time (in an expected sense) needed to answer the location query. In particular, now whenever cell $c$ is not associated with object $o$, we will be able to cease searching the list associated with $o$ after having inspected half of the cells associated with $o$ instead of all of them, which is the case when no ordering exists.

An important shortcoming of the use of the explicit representation, which has an effect somewhat related to the absence of an ordering, is the inability to distinguish between occupied and unoccupied cells. In particular, in order to detect that a cell $c$ is not occupied by any object we must examine the sets associated with each object, which is quite time-consuming. Of course, we could avoid this problem by forming an additional set which contains all of the unoccupied cells, and examine this set first whenever processing the location query. The drawback of such a solution is that it slows down all instances of the location query that involve cells that are occupied by objects.

We can avoid examining every cell in each object set, thereby speeding up the location query in certain cases, by storing a simple approximation of the object with each object set $o$. This approximation should

be of a nature that makes it easy to check if it is impossible for a location *l* in space to be in *o*. One such approximation is a minimum bounding box whose sides are parallel to the coordinate axes of the space in which the object is embedded. For example, for object B in Figure 1 such a box is anchored at the lower-left corner of cell (5,1) and the upper-right corner of cell (7,0). The existence of a box *b* for object *o* means that if *b* does not contain *l*, then *o* does not contain *l* either, and we can proceed with checking the other objects. This bounding box is usually a part of the explicit representation.

There are several ways of increasing the quality of the approximation. For example, the minimum bounding box may be rotated by an arbitrary angle so that the sides are still orthogonal while no longer having to be parallel to the coordinate axes and known as an OBB-tree denoting *oriented bounding box* (e.g., [29, 67, 120]). These representations adaptations of the strip tree [16] and the Douglas-Peucker generalization algorithm [45, 46, 130] for curves in two dimensions. The number of sides as well as the number of their possible orientations may be expanded so that it is arbitrary (e.g., a convex hull [29]), or bounded although greater than the dimensionality of the underlying space (e.g., the P-tree [86] and k-DOP [93] where the number of possible orientations is bounded, and the minimum bounding polybox [30] which attempts to find the optimal orientations). The most general solution is the convex hull which is often approximated by a minimum bounding polygon of a fixed number of sides having either an arbitrary orientation (e.g., the minimum bounding n-corner [44, 140, 141]) or a fixed orientation usually parallel to the coordinate axes (e.g., [51]). Restricting the sides (i.e., faces in dimension higher than 2) of the polyhedron to be parallel to the coordinate axes (termed an *axis-parallel polygon*) enables simpler point-in-object tests (e.g., the vertex representation [49, 50, 51]). The minimum bounding box may also be replaced by a circle, sphere (e.g., [81, 82, 110, 111, 112, 164]), ellipse, intersection of the minimum bounding box with the minimum bounding sphere (e.g., [91]), as well as other shapes (e.g., [29]). Interestingly, many of these solutions arose in applications in collision detection (e.g., [67, 93]).

In the rest of this paper we restrict our discussion to minimum bounding boxes that are rectangles with sides parallel to the coordinate axes, although, of course, the techniques we describe are applicable to other more general bounding objects. Nevertheless, in the interest of brevity, we often use the term *bounding box* even though the terms *minimum bounding box* or *minimum bounding object* would be more appropriate.

The location query can be answered more directly if we allocate an address *a* in storage for each cell *c* where an identifier is stored that indicates the identity of the object (or objects) of which *c* is a member. Recall that such a representation is said to be implicit as in order to determine the rest of the cells that comprise the object associated with *c* (and thus complete the response to the location query), we must examine the identifiers stored in the addresses associated with the contiguous cells and then aggregate the cells whose associated identifiers are the same. However, in order to be able to use the implicit representation, we must have a way of finding the address *a* corresponding to *c*, taking into account that there is possibly a very large number of cells, and then retrieving *a*.

Finding the right address requires an additional data structure, termed an *access structure*, such as an index on the locations in space. An example of such an index is a table of cell-address pairs where *address* indicates the physical location where the information about the object associated with the location in space corresponding to *cell* is stored. The table is indexed by the location in space corresponding to *cell*. The

index is really an ordering and hence its range is usually the integers (i.e., one-dimensional). When the data is multidimensional (i.e., cells in $d$-dimensional space where $d > 0$), it may not be convenient to use the location in space corresponding to the cell as an index since its range spans data in several dimensions. Instead, we employ techniques such as laying out the addresses corresponding to the locations in space of the cells in some particular order and then making use of an access structure in the form of a mapping function to enable the quick association of addresses with the locations in space corresponding to the cells. Retrieving the address is more complex in the sense that it can be a simple memory access or it may involve an access to secondary or tertiary storage if virtual memory is being used. In most of our discussion, we assume that all data is in main memory, although, as we will see, several representations do not rely on this assumption.

Such an access structure enables us to obtain the contiguous cells (as we know their locations in space) without having to examine all of the cells. Therefore, we will know the identities of the cells that comprise an object thereby enabling us to complete the response to the location query with an implicit representation. In other words, the implicit representation lends itself to retrieval on the basis of knowledge only of the cells rather than of the objects, and this is the rationale for characterizing it as being *image-based*. In the rest of this section, we discuss several such access structures.

The existence of an access structure also enables us to answer the feature query with the implicit representation, although this is quite inefficient. In particular, given an object $o$, we must exhaustively examine every cell (i.e., location $l$ in space) and check if the address where the information about the object associated with $l$ is stored contains $o$ as its value. This will be time-consuming, as it may require that we examine all the cells.

There are many ways of laying out the addresses corresponding to the locations in space of the cells each having its own mapping function. Some of the most important ones for a two-dimensional space are illustrated in Figure 2 for an $8 \times 8$ portion of the space and are described briefly below. To repeat, in essence, what we are doing is providing a mapping from the $d$-dimensional space containing the locations of the cells to the one-dimensional space of the range of index values (i.e., integers) which are used to access a table whose entries contain the addresses where information about the contents of the cells is stored. The result is an ordering of the space, and the curves shown in Figure 2 are termed *space-filling curves* (e.g., [131]). Choosing among the space-filling curves illustrated in Figure 2 is not easy as each one has its advantages and disadvantages. Below, we review a few of their desirable properties, and show how some of the two--dimensional orderings satisfy them.

- The curve should pass through each location in space once and only once.
- The mapping from the higher-dimensional space to the integers should be relatively simple and likewise for the inverse mapping. This is the case for all but the Peano-Hilbert order (Figure 2d). For the Morton order (Figure 2c), the mapping is obtained by interleaving the binary representations of the coordinate values of the location of the cell. The number associated with each cell is known as its *Morton number*. The Gray order (Figure 2g) is obtained by applying a Gray code [68] to the result of bit interleaving, while the double Gray order (Figure 2h) is obtained by applying a Gray code to the result of bit interleaving the Gray code of the binary representation of the coordinate values. The U order

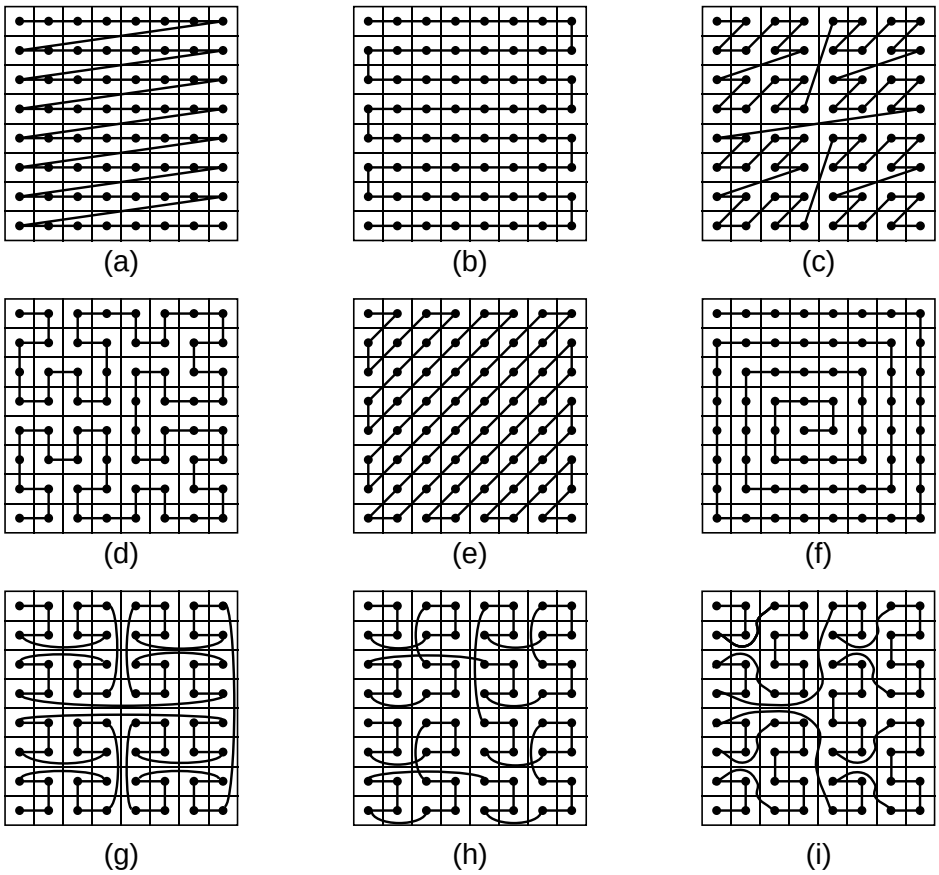


Figure 2: The result of applying several different space-ordering methods to an 8×8 collection of cells whose first element is in the upper-left corner: (a) row order, (b) row-prime order, (c) Morton order, (d) Peano-Hilbert order, (e) Cantor-diagonal order, (f) spiral order, (g) Gray order, (h) double Gray order, and (i) U order.

(Figure 2i) is obtained in a similar manner to the Z order except for an intermediate application of $d-1$ 'exclusive or' ($\oplus$) operations on the binary representation of selected combinations of the coordinate values prior to the application of bit interleaving [102, 143]. Thus the difference in cost between the Z order and the U order in $d$ dimensions is just the performance of additional $d-1$ 'exclusive or' operations. This is in contrast with the Peano-Hilbert order where the mapping and inverse mapping processes are considerably more complex.

- The ordering should be stable. This means that the relative ordering of the individual locations is preserved when the resolution is doubled (e.g., when the size of the two-dimensional space in which the cells are embedded grows from $8 \times 8$ to $16 \times 16$) or halved assuming that the origin stays the same. The Morton, U, Gray, and double Gray orders are stable, while the row (Figure 2a), row-prime (Figure 2b), Cantor-diagonal (Figure 2e), and spiral (Figure 2f) orders are not stable. The Peano-Hilbert order is also not stable as can be seen by its definition. In particular, in two dimensions, the Peano-Hilbert order of resolution $i+1$ (i.e., a $2^i \times 2^i$ image) is constructed by taking the Peano-Hilbert curve of resolution $i$ and rotating the NW, NE, SE, and SW quadrants by 90 degrees clockwise, 0 degrees, 0 degrees, and

90 degrees counterclockwise, respectively. For example, Figures 3a, 3b, and 3c give the Peano-Hilbert curves of resolutions 1, 2, and 3, respectively.

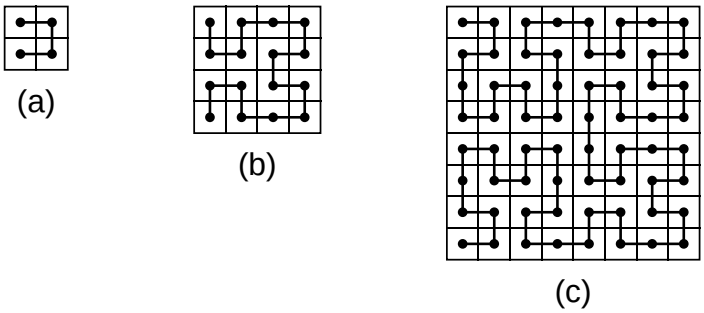


Figure 3: Peano-Hilbert curves of resolution (a) 1, (b) 2, and (c) 3.

- Two locations that are adjacent (i.e., in the sense of a $(d-1)$-dimensional adjacency also known as 4-adjacent) in space are neighbors along the curve and vice versa. In two dimensions, this means that the locations share an edge or a side. This is impossible to satisfy for all locations at all space sizes. However, for the row-prime, Peano-Hilbert, and spiral orders, every element is a 4-adjacent neighbor of the previous element in the sequence while this is not the case for the other orders. This means that the row-prime, Peano-Hilbert, and spiral orders have a slightly higher degree of locality than the other orders.

- The process of retrieving the neighbors of a location in space should be simple.

- The order should be admissible. This means that at each position in the ordering, at least one 4-adjacent neighbor in each of the lateral directions (i.e., horizontal and vertical) must have already been encountered. This is useful in several algorithms (e.g., connected component labeling [42][2]). The row and Morton orders are admissible while the Peano-Hilbert, U, Gray, and double Gray orders are not admissible. The row-prime, Cantor-diagonal, and spiral orders are admissible only if we permit the direction of the 4-adjacent neighbors to vary from position to position along the curve. For example, for the row-prime order, at positions on odd rows, the previously encountered 4-adjacent neighbors are the western and northern neighbors, while at positions on even rows, it is the eastern and northern neighbors.

The row order (Figure 2a) is of special interest to us because its mapping function is the one most frequently used by the multidimensional array, which is the most common access structure. The Morton order [108] has a long history having been first mentioned by Peano [118], and has been used by many researchers (e.g., [1, 65, 113, 165]). It is also known as a Z order [113] and as an N order [165]. The Peano-Hilbert order was first mentioned soon afterwards by Hilbert [78], and has also been used by a number of researchers (e.g., [52, 85]).

[2] A region or object four-connected component, is a maximal four-connected set of locations belonging to the same object, where a set $S$ of locations is said to be four-connected if for any locations $p$, $q$ in $S$ there exists a sequence of locations $p = p_0, p_1, \cdots, p_n = q$ in $S$, such that $p_{i+1}$ is 4-adjacent (8-adjacent) to $p_i$, $0 \leq i < n$. The process of assigning the same label to all 4-adjacent locations that belong to the same object is called connected component labeling (e.g., [116, 125]).

Although conceptually very simple, the U order introduced by Schrack and Liu [102, 143] is relatively recent. It is a variant of the Morton order, while also resembling the Peano-Hilbert order. The primitive shape is a 'U' which is the same as that of the Peano-Hilbert order. However, unlike the Peano-Hilbert order, and like the Morton order, the ordering is applied recursively with no rotation thereby enabling it to be stable. The U order has a slight advantage over the Morton order in that more of the locations that are adjacent (i.e., in the sense of a $(d-1)$-dimensional adjacency) along the curve are also neighbors in space. This is directly reflected in the lower average distance between two successive positions in the order (i.e., for a $2^{16} \times 2^{16}$ image, it is 1.4387 for the U order while it is 2.0000, 1.6724, and 1.5000 for the double Gray, Morton, and Gray orders, respectively, and 1 for the row-prime, Peano-Hilbert, and spiral orders [137]). However, the price of this is that like the Peano-Hilbert order, the U order is also not admissible. Nevertheless, like the Morton order, the process of retrieving the neighbors of a location in space is simple when the space is ordered according to the U order. Asano et al. [14] describe an order which has the same properties as the Peano-Hilbert order except that for any square region, there are at most three breaks in the continuity of the curve in contrast to four for the Peano-Hilbert order (i.e., at most three out of four 4-adjacent subblocks of a square block are not neighbors along the curve in contrast with a possibility that all four 4-adjacent subblocks are not neighbors in the Peano-Hilbert order). This property is useful for retrieval of square like regions in the case of a range query when the data is stored on disk in this order as each break in the continuity can result in a disk seek operation.

The multidimensional array (having a dimension equal to the dimensionality of the space in which the objects and the environment are embedded) is an access structure which, given a cell *c* at a location *l* in space, enables us to calculate the address *a* containing the identifier of the object associated with *c*. The array is only a conceptual multidimensional structure (it is not a multidimensional physical entity in memory) in the sense that it is a mapping of the locations in space of the cells into sequential addresses in memory. The actual addresses are obtained by the array access function (see e.g., [95] as well as the above discussion on space orderings) which is based on the extents of the various dimensions (i.e., coordinate axes). The array access function is usually the mapping function for the row order (Figure 2a). Thus the array enables us to implement the implicit representation with no additional storage except for what is needed for the array's descriptor. The descriptor contains the bounds and extents of each of the dimensions which are used to define the mapping function (i.e., they determine the values of its coefficients) so that the appropriate address can be calculated given the cell's location in space.

The array is called a *random access structure* because the address associated with a location in space can be retrieved in constant time independent of the number of elements in the array and does not require any search. Note that we could store the object identifier *o* in the array element itself instead of allocating a separate address *a* for *o* thereby saving some space.

The array is an implicit representation because we have not explicitly aggregated all the contiguous cells that comprise a particular object. They can be obtained given a particular cell *c* at a location *l* in space belonging to object *o* by recursively accessing the array elements corresponding to the locations in space that are adjacent to *l* and checking if they are associated with object *o*. This process is known as depth-first connected component labeling.

Interestingly, depth-first connected component labeling could also be used to answer the feature query efficiently with an implicit representation if we add a data structure such as an index on the objects (e.g., a table of object-location pairs where *location* is one of the locations in space that comprise *object*). Thus given an object $o$ we use the index to find a location in space that is part of $o$, and then proceed with the depth-first connected component labeling as before. This index does not make use of the spatial coverage of the objects and thus it can be implemented using conventional searching techniques such as hashing [96]. In this case, we will need $O(N)$ additional space for the index, where $N$ is the number of different objects. We do not discuss such indexes here.

Of course, we could also answer the location query with an explicit representation by adding an index which associates objects with locations in space (i.e., having the form location-objects). However, this would require $O(S)$ additional space for the index, where $S$ is the number of cells. The $O(S)$ bound assumes that only one object is associated with each cell. If we take into account that a cell could be associated with more than one object, then the additional storage needed is $O(NS)$, if we assume $N$ objects. Since the number of cells $S$ is usually much greater than the number of objects $N$, the addition of an index to the explicit representation is not as practical as extending the implicit representation with an index of the form object-location as described above. Thus it would appear that the implicit representation is more useful from the point of view of flexibility when taking storage requirements in to account.

The implicit representation can be implemented with access structures other than the array. This is an important consideration when many of the cells are not in any of the objects (i.e., they are empty). The problem is that using the array access structure is wasteful of storage, as the array requires an element for each cell regardless of whether the cell is associated with any of the objects. In this case, we choose to keep track of only the nonempty cells.

We have two ways to proceed. The first is to use one of several multidimensional access structures such as a point quadtree, k-d tree, MX quadtree, etc. as described in [134]. The second is to make use of one of the orderings of space shown in Figure 2 to obtain a mapping from the nonempty contiguous cells to the integers. The result of the mapping serves as the index in one of the familiar tree-like access structures (e.g., binary search tree, range tree, $B^+$-tree, etc.) to store the address which indicates the physical location where the information about the object associated with the location in space corresponding to the nonempty cell is stored.

# 3 Blocks

An alternative class of representations of the objects and their environment removes the stipulation that cells making up the object collection be of a unit size and permits their sizes to vary. The resulting cells are termed *blocks* and are usually rectangular with sides parallel to the coordinate axes (this is assumed in our discussion unless explicitly stated otherwise). The volume (e.g., area in two dimensions) of the blocks need not be an integer multiple of that of the unit-size cells, although this is often the case. Observe that when the volumes of the blocks are integer multiples of that of the unit-size cells, then we have two levels of aggregation in

the sense that an object consists of an aggregation of blocks which are themselves aggregations of cells. We assume that all the cells in a block belong to the same object or objects. In other words, the situation that some of the cells in the block belong to object $o_1$ while the others belong to object $o_2$ (and not to $o_1$) is not allowed.

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 but one side of a block are unit-sized
- all sides of a block are of equal size
- all sides of each block are powers of two
- etc.

Other decomposition rules dispense with the requirement that the blocks be rectangular, while still others do not require that they be orthogonal. 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.

The simplest decomposition rule is one that permits aggregation of identically-valued cells in only one dimension. It assigns a priority ordering to the various dimensions and then fixes the coordinate values of all but one of the dimensions, say $i$, and then varies the value of the $i^{th}$ coordinate and aggregates all adjacent cells belonging to the same object into a one-dimensional block. This technique is commonly used in image processing applications where the image is decomposed into rows which are scanned from top to bottom, and each row is scanned from left to right while aggregating all adjacent pixels with the same value into a block. It is useful in image transmission as we only have to transmit the pixels where a change in value takes place. The aggregation into one-dimensional blocks is the basis of *runlength encoding* [129]. Similar techniques are applicable to higher-dimensional data where, for example in the case of three-dimensional data, one would scan the image one 2-dimensional hyperplane at a time where each hyperplane would be scanned in raster scan order. Techniques analogous to runlength encoding form the basis of the vertex representation for representing axis-parallel polygons of arbitrary dimension [49, 50].

The drawback of the decomposition into one-dimensional blocks described above is that all but one side of each block must be of unit width. The most general decomposition removes this restriction along all of the dimensions, thereby permitting 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. This general decomposition has the potential of requiring less space. However, its drawback is that the determination of optimal partition points may be a computationally expensive procedure. We assume that the blocks are disjoint although this need not be the case. We also assume that the blocks are rectangular as well as orthogonal (e.g., Figure 4). although again this is not absolutely necessary as there exist decompositions using other shapes as well (e.g., triangles, etc.).

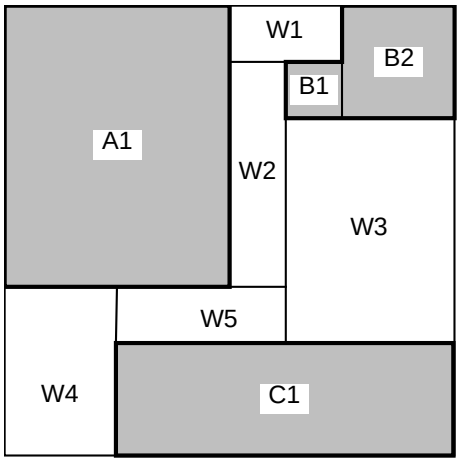


Figure 4: Arbitrary block decomposition for the collection of objects and cells in Figure 1. Blocks corresponding to object O are labeled Oi and the blocks that are not in any of the objects as Wi using the suffix i to distinguish between them in both cases.

It is easy to adapt the explicit representation to deal with blocks resulting from an arbitrary decomposition (which also includes the one that yields one-dimensional blocks). In particular, instead of associating a set with each object *o* that contains the location in space of each cell that comprises *o*, we need to associate with each object *o* the locations in space and size of each block that comprises *o*. This can be done by specifying the coordinate values of the upper-left corner of each block and the sizes of its sides. Without loss of generality, we use this format for the explicit representation of all of the block decompositions described in this section.

Using the explicit representation of blocks, both the feature and location queries are answered in essentially the same way as they were for unit-sized cells. The only difference is that for the location query instead of checking if a particular location *l* in space is a member of one of the sets of cells associated with the various objects, we must check if *l* is covered by one of the blocks in the sets of blocks of the various objects. This is a fairly simple process as we know the location in space and size of each of the blocks.

Implementing an arbitrary decomposition (which also includes the one that results in one-dimensional blocks) using an implicit representation is also quite easy. We build an index based on an easily identifiable location in each block such as its upper-left corner. We make use of the same techniques that were presented in the discussion of the implicit representation for unit-sized cells in Section 2. The only difference is that we must also record the size of each block along with the address indicating the physical location where the information about the object associated with the locations in space corresponding to the block is stored.

As in the case of unit-size cells, regardless of which access structure is used to implement the index, we determine the object $o$ associated with a cell at location $l$ by finding the block $b$ that covers $l$. If $b$ is an empty block, then we exit. Otherwise, we return the object $o$ associated with $b$. Notice that the search for the block that covers $l$ may be quite complex in the sense that the access structures may not necessarily achieve as much pruning of the search space as in the case of unit-sized cells. In particular, this is the case whenever the space ordering and the block decomposition method to whose results the ordering is being applied do not have the property that all of the cells in each block appear in consecutive order. In other words, given the cells in the block $e$ with minimum and maximum values in the ordering, say $u$ and $v$, there exists at least one cell in block $f$ distinct from $e$ which is mapped to a value $w$ where $u < w < v$. Thus, supposing that the index is implemented using a tree-like access structure, a search for the block $b$ that covers $l$ may require that we visit several subtrees of a particular node in the tree.

As we saw in the description of the algorithm for responding to query 2, the drawback of the arbitrary decomposition into blocks is that since there is no rule for the formation of the blocks, there is also no easy rule for accessing them. The *irregular grid* is one way to overcome this drawback by making use of a very simple decomposition rule 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 $x_i = 0$ $(1 \leq i \leq d)$. The result is a collection of $\prod_{i=1}^{d}(h_i + 1)$ blocks. These blocks form a grid of irregular-sized blocks as the partition lines are at arbitrary positions in contrast to the *uniform grid* [56] where the partition lines are positioned so that all of the resulting grid cells are congruent. Observe that there is no recursion involved in the decomposition process. For example, Figure 5a is an example block decomposition using hyperplanes parallel to the $x$ and $y$ axes for the collection of objects and cells given in Figure 1.

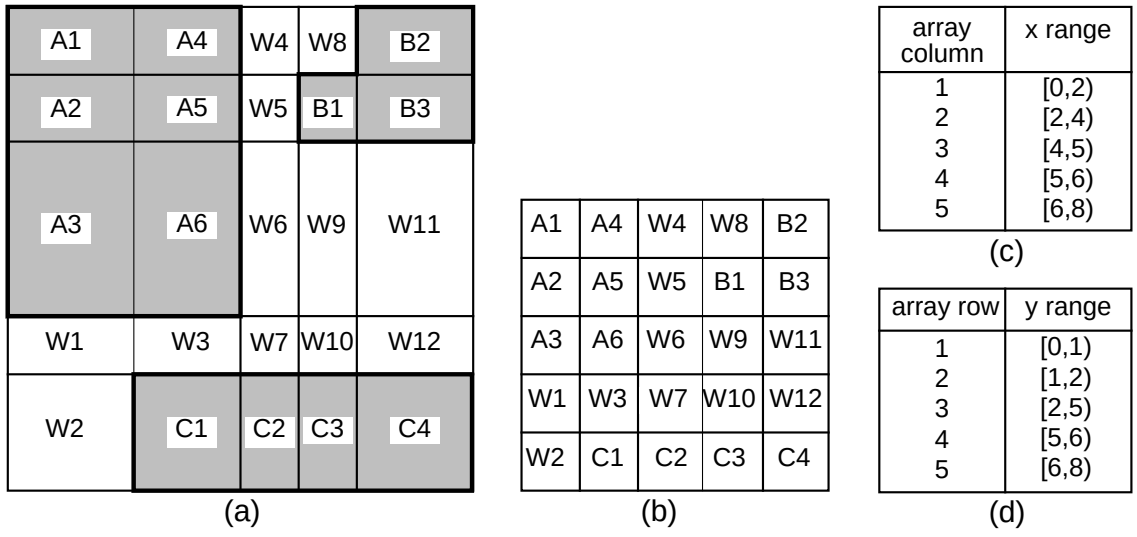


Figure 5: (a) Block decomposition resulting from the imposition of a grid with partition lines at arbitrary positions on the collection of objects and cells in Figure 1 yielding an irregular grid, (b) the array access structure, (c) the linear scale for the x coordinate values, and (d) the linear scale for the y coordinate values. Blocks corresponding to object O are labeled Oi and the blocks that are not in any of the objects as Wi using the suffix i to distinguish between them in both cases.

The block decomposition resulting from the use of an irregular grid is handled by an explicit representation in the same way as the arbitrary decomposition. Finding a suitable implicit representation is a bit more complex as we must define an appropriate access structure. Although the blocks are not congruent, we can still impose an array access structure on them 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 $x_i = 0$ ($1 \leq i \leq d$). 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 $l$.

For example, Figure 5b is the array access structure corresponding to the block decomposition in Figure 5a, while Figures 5c and 5d are the linear scales for the $x$ and $y$ axes, respectively. In this example, the linear scales are shown as tables (i.e., array access structures). In fact, they can be implemented using tree access structures. The representation described here is an adaptation for regions of the *grid file* [109] data structure for points.

Our implementation of the access structures for the irregular grid yields a representation that is analogous to an *indirect uniform grid* in the sense that given a cell at location $l$ we need to make $d+1$ array-like accesses (analogous to the two memory references involved with indirect addressing in computer instruction formats) to obtain the object $o$ associated with it instead of just one array access when the grid is uniform (i.e., all the blocks are congruent and cell-sized). The first $d$ accesses find the identity of the array element (i.e., block $b$) that contains $l$, while the last access determines the object $o$ associated with $b$. Once we have found block $b$, we examine the adjacent blocks to obtain the rest of the cells comprising object $o$, thereby completing the response to the location query, by employing the same methods as we used for the array access structure for the uniform-sized cells. The only difference is that every time we find a block $b$ in the array access structure associated with $o$, we must examine $b$'s corresponding entries in the linear scales to determine $b$'s size so that we can report the cells that comprise $b$ as parts of object $o$.

Perhaps the most widely known decompositions into blocks are those referred to by the general terms *quadtree* and *octree* [133, 134]. 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 objects 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 objects that comprise it, the number of objects 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. Moreover, both explicit and implicit aggregations of the blocks are possible.

There are many variants of quadtrees and octrees, 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* [92] and *region octrees* [83, 105]. 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 environ-

ment containing the objects is recursively decomposed into four or eight, respectively, rectangular congruent blocks until each block is either completely occupied by an object or is empty (such a decomposition process is termed *regular*). For example, Figure 6a is the block decomposition for the region quadtree corresponding to Figure 1. 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 corresponding to the environment containing the objects, then the coordinate values of the upper-left corner of each block (e.g., $(i, j)$ in two dimensions) of size $2^s \times 2^s$ satisfy the property that $a \bmod 2^s = 0$ and $b \bmod 2^s = 0$.

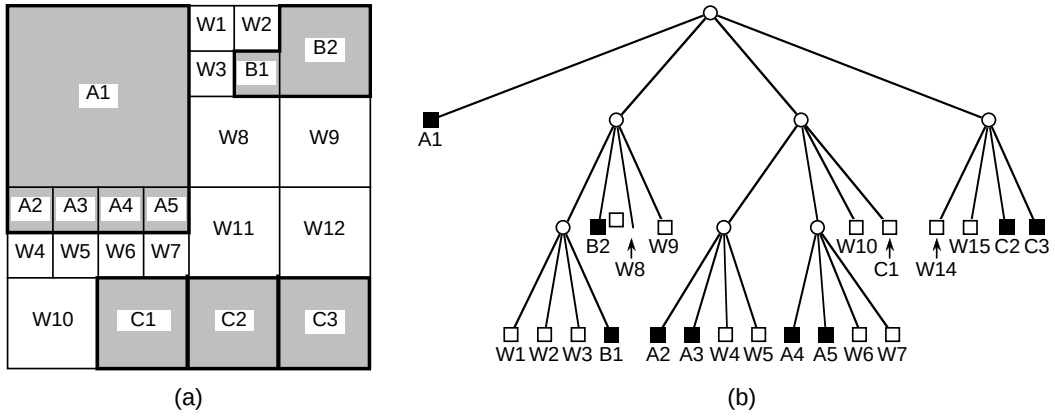


Figure 6: (a) Block decomposition and (b) its tree representation for the collection of objects and cells in Figure 1. Blocks corresponding to object O are labeled Oi and the blocks that are not in any of the objects as Wi using the suffix i to distinguish between them in both cases.

A region quadtree can be implemented using an explicit representation by associating a set with each object *o* that contains its constituent blocks. Each block is specified by numbers corresponding to the coordinate values of its upper-left corner and the size of one of its sides. These numbers are stored in the set in the form $(i, j) : k$ where $(i, j)$ and $k$ correspond to the coordinate values of the upper-left corner and depth, respectively, of the block. For example, the explicit representation of the collection of blocks n Figure 1 is given by the sets A={(0,0):2,(0,4):0,(1,4):0,(2,4):0,(3,4):0}, B={(5,1):0,(6,0):1}, and C={(2,6):1,(4,6):1,(6,6):1}, which correspond to blocks {A1,A2,A3,A4,A5}, {B1,B2}, and {C1,C2,C3}, respectively.

A region quadtree implementation that makes use of an implicit representation is quite different. First, we allocate an address *a* in storage for each block *b* which stores an identifier that indicates the identity of the object (or objects) of which *b* is a member. Second, it is necessary to impose an access structure on the collection of blocks in the same way as the array was imposed on the collection of unit-sized cells. Such an access structure enables us to determine easily the value associated with any point in the space covered by a cell without resorting to exhaustive search. Note that depending on the nature of the access structure, it's not always necessary to store the location and size of each block with *a*.

There are many possible access structures. Interestingly, using an array as an access structure is not particularly useful as it defeats the rationale for the aggregation of cells into blocks unless, of course, all the blocks are of a uniform size in which case we have the analog of a two-level grid.

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., Figure 6b corresponding to the collection of two-dimensional objects in Figure 1 whose quadtree block decomposition is given in Figure 6a). Each leaf node in the tree corresponds to a different block $b$ and contains the address $a$ in storage where an identifier is stored that indicates the identity of the object (or objects) of which $b$ is a member. As in the case of the array, where we could store the object identifier $o$ in the array element itself instead of allocating a separate address $a$ for $o$, we could achieve the same savings by storing $o$ in the leaf node of the tree. Each nonleaf node $f$ corresponds to a block whose volume is the union of the blocks corresponding to the $2^d$ children 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.

Answering the location query using the tree structure is different from using an array where it is usually achieved by a table lookup having an $O(1)$ cost (unless the array is implemented as a tree, which is a possibility [38]). In contrast, the location query is usually answered in a tree by locating the block that contains the location in space corresponding to the desired cell. This is achieved by a process that starts at the root of the tree and traverses the links to the children whose corresponding blocks contain the desired location. This process has an $O(n+F)$ cost where the environment has a maximum of $n$ levels of subdivision (e.g., an environment all of whose sides are of length $2^n$), and $F$ is the cardinality of the answer set.

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. This information can be inferred from knowledge of the size of the underlying space as 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 (rectangular here, although a triangular decomposition process could also be defined which yields four equilateral triangles; however, in such a case, we are no longer dealing with rectangular cells). Thus as long as we start from the root, we know the location and size of every block.

There are a number of alternative access structures to the tree with fanout $2^d$. 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 applying one of the familiar tree-like access structures (e.g., a binary search tree, range tree, $B^+$-tree, etc.). There are many possible mappings (e.g., [133]). The simplest is to use the same technique that we applied to the collection of blocks of arbitrary size. In particular, we can apply one of the orderings of space shown in Figure 2 to obtain a mapping from the coordinate values of the upper-left corner $u$ of each block to the integers.

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 (Figure 2c). 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., Figure 6b) in the order NW, NE, SW, SE.

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* [94, 139, 154] 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. Of course, in $d$ dimensions, the depth of the tree may increase by a factor of $d$.

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 [54] 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 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. Therefore, we use a k-d tree [21] which is an access structure having a fanout of 2 that has the same relationship to the point quadtree as the bintree has to the region quadtree. 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. In other words, the bintree is a regular decomposition k-d tree.

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. In this case, at each nonleaf node of the k-d tree, we must also record the identity of the axis that is being split. We use the term *generalized k-d tree* to describe this structure. The generalized k-d tree is really an adaptation to regions of the *adaptive k-d tree* [59] and the *LSD tree* [77] which were originally developed for points. It can also be regarded as a special case of the *BSP tree* (denoting *Binary Space Partitioning*) [60]. In particular, in the generalized k-d tree, 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 used in computer graphics to facilitate viewing.

One of the shortcomings of the generalized k-d tree is the fact that we can only decompose the space into two parts along a particular dimension at each step. If we wish to partition a space into $p$ parts along a dimension $i$, then we must perform $p-1$ successive partitions on dimension $i$. Once these $p-1$ partitions are complete, we partition along another dimension. The *puzzletree* [39] is a further generalization of the k-d tree that decomposes the space into two or more parts along a particular dimension at each step so that no two successive partitions use the same dimension. In other words, the puzzletree compresses all successive partitions on the same dimension in the generalized k-d tree.

The puzzletree is motivated by a desire to overcome the rigidity in the shape, size, and position of the blocks that result from the bintree (and to an equivalent extent, the region quadtree) partitioning process (because of its regular decomposition). In particular, in many cases, the decomposition rules ignore the homogeneity present in certain regions on account of the need to place the partition lines in particular positions as well as a possible limit on the number of permissible partitions along each dimension at each decomposition step. Often, it is desirable for the block decomposition to follow the perceptual characteristics of the objects as well as reflect their dominant structural features.

For example, consider a front view of a scene containing a table and two chairs. Figures 7a and 7b are the block decompositions resulting from the use of a bintree and a puzzletree, respectively, for this scene, while Figure 7c is the tree access structure corresponding to the puzzletree in Figure 7b. Notice the natural decomposition in the puzzletree of the chair into the legs, seat, and back, and of the table into the top and legs. On the other hand, the blocks in the bintree (and to a greater extent in the region quadtree, although not shown here) do not have this perceptual coherence. Of course, we are aided here by the separability of the objects; however, this does not detract from the utility of the representation as it only means that the objects can be decomposed into fewer parts.

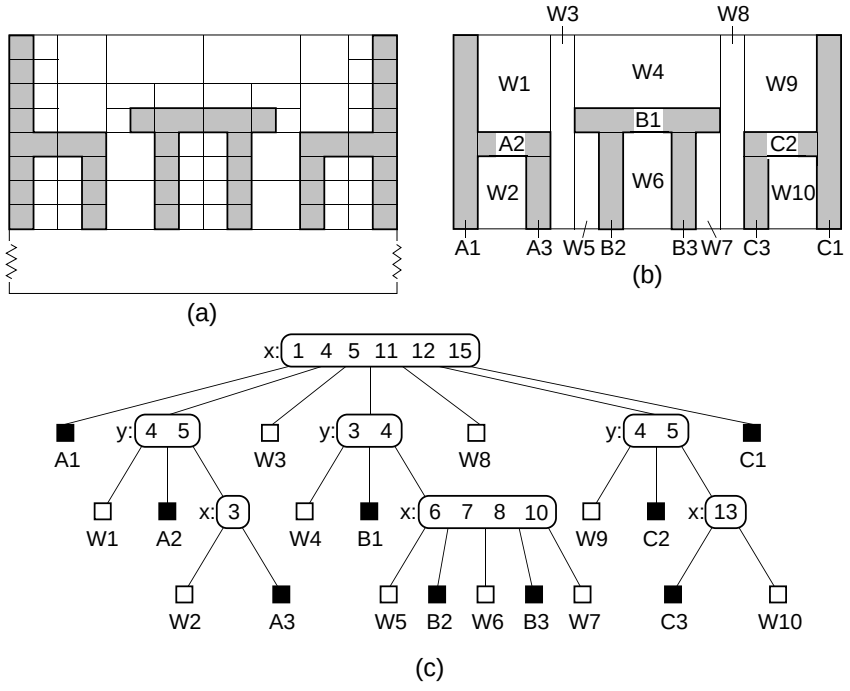


Figure 7: Block decomposition for the (a) bintree and (b) puzzletree corresponding to the front view of a scene containing a table and two chairs; (c) is the tree access structure for the puzzletree in (b).

# 4 Image-based Hierarchical Interior-based Representations (Pyramids)

Our goal here is to be able to take an object $o$ as input and return the cells that it occupies (the feature query) when using a representation that stores with each cell the identities of the objects of which it is a part. The most natural hierarchy that can be imposed on the cells to enable us to answer this query is one that aggregates every $q$ cells regardless of the values associated with them into larger congruent blocks (unlike the aggregation of identically-valued cells into multi-dimensional blocks as in the region quadtree). This process is repeated recursively so that groups of $q$ blocks are repeatedly aggregated into one block until there is just one block left. The value associated with the block $b$ is the union of the names (i.e., object identifiers) of the objects associated with the cells or blocks that comprise block $b$. The identities of the cells and blocks that are aggregated depends, in part, on how the collection of the cells is represented. For example, assuming a two-dimensional underlying space, if the cells are represented as one long list consisting of the cells of the first row, followed by those of the second row, etc., then one possible aggregation combines every successive $q$ cells. In this case, the blocks are really one-dimensional entities.

The process that we have just outlined can be described more formally as follows. We make the following assumptions:

- The blocks are rectangular with sides parallel to the coordinate axes.
- Each block contains $q$ cells or $q$ blocks so that, assuming $d$ dimensions, $q = \prod_{j=1}^{d} r_j$ where the block has width $r_j$ for dimension $j$ ($1 \leq j \leq d$) measured in cells or blocks depending on the level in the hierarchy at which the block is found.
- All blocks at a particular level in the hierarchy are congruent with the different levels forming a containment hierarchy.
- There are $S$ cells in the underlying space, and let $n$ be the smallest power of $q$ such $q^n \geq S$.
- The underlying space can be enlarged by adding $L$ empty cells so that $q^n = S + L$ and that each side of the underlying space along dimension $j$ is of width ${r_j}^n$.

The hierarchy consists of the set of sets $\{C_i\}$ ($0 \leq i \leq n$) where $C_n$ corresponds to the original collection of cells having $S + L$ elements, $C_{n-1}$ contains $(S + L)/q$ elements corresponding to the result of the initial aggregation of $q$ cells into $(S + L)/q$ congruent blocks, and $C_0$ is a set consisting of just one element corresponding to a block of size $S + L$. Each element $e$ of $C_i$ ($0 \leq i \leq n - 1$) is a congruent block whose value is the union of the values (i.e., sets of object identifiers) associated with the blocks of the $q$ elements of $C_{i+1}$. The value of each element of $C_n$ is the object identifier(s) corresponding to the object(s) of which its cell is a part.

The resulting hierarchy is known as a *cell pyramid* (e.g., [6, 31, 32, 47, 48, 84, 106, 123, 124, 146, 151, 150, 155][3] and is frequently characterized as a *multiresolution* representation since the original collection of

[3] Actually, the qualifier *cell* is rarely used. However, we use it here to avoid confusion with other variants of the pyramid which are based on a hierarchy of objects rather than cells as discussed in Section 5.

objects is described at several levels of detail by using cells that have different sizes, although similar in shape. It is important to distinguish the cell pyramid from the region quadtree which, as we recall, is an example of an aggregation into square blocks where the basis of the aggregation is that the cells have identical values (i.e., are associated with the same object, or objects if object overlap is permitted). The region quadtree is an instance of what is termed a *variable-resolution* representation, which, of course, is not limited to blocks that are square. In particular, it can be used with a limited number of nonrectangular shapes (most notably, triangles in two dimensions [20, 134]).

It is quite difficult to use the cell pyramid, in the form that we have described, to respond to the feature query and to the complete location query (i.e., to obtain all of the contiguous cells that make up the object associated with the query location) due to the absence of an access structure. This can be remedied by implementing a set of arrays $A_i$ in a one-to-one correspondence to $C_i$ ($0 \leq i \leq n$) where $A_i$ is a $d$-dimensional array of side length ${r_j}^i$ for dimension $j$ ($1 \leq j \leq d$). Each of the elements of $A_i$ corresponds to a $d$-dimensional block of side length ${r_j}^{n-i}$ for dimension $j$ ($1 \leq j \leq d$) assuming a total underlying space of side length ${r_j}^n$. The result is a stack of arrays $A_i$, termed an *array pyramid*, which serves as an access structure to collections $C_i$ ($0 \leq i \leq n$). The array pyramid is an instance of an implicit interior-based representation consisting of array access structures. Of course, other representations are possible through the use of alternative access structures (e.g., different types of trees).

We illustrate the array pyramid for two dimensions with $r_1 = 2$ and $r_2 = 2$. Assume that the space in which the original collection of cells is found is of size $2^n \times 2^n$. Let $C_n$ correspond to the original collection of cells. The hierarchy of arrays consists of the sequence $A_i$ ($0 \leq i \leq n$) so that elements of $A_i$ access the corresponding elements in $C_i$. We obtain $C_{n-1}$ by forming an array of size $2^{n-1} \times 2^{n-1}$ with $2^{2n-2}$ elements so that each element $e$ in $C_{n-1}$ corresponds to a $2 \times 2$ square consisting of 4 elements (i.e., cells) in $C_n$ and has a value consisting of the union of the names (i.e., labels) of the objects that are associated with these 4 cells. This process is applied recursively to form $C_i$ ($0 \leq i \leq n-1$) where $C_0$ is a collection consisting of just one element whose value is the set of names of all the objects associated with at least one cell. The arrays are assumed to be stored in memory using sequential allocation with conventional orderings (e.g., lexicographically), and are accessed by use of the $d$-dimensional coordinate values of the cells. For example, Figure 8 is the array pyramid for the collection of objects in Figure 1.

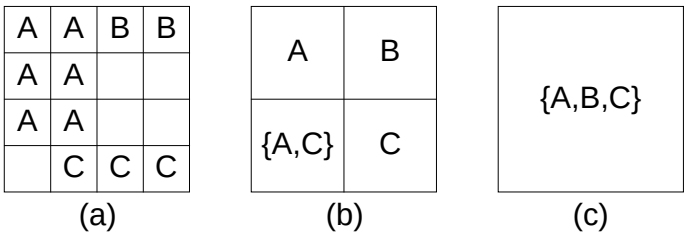


Figure 8: Array pyramid for the collection of objects and cells in Figure 1. (a) Array $A_2$. (b) Array $A_1$. (c) Array $A_0$. The block decomposition in Figure 1 corresponds to Array $A_3$.

Using the array pyramid, it is very easy to respond to the feature query, as we just examine the relevant parts of the stack of arrays. For example, suppose that we want to determine the locations that comprise

object $o$, and we use the array pyramid consisting of arrays $A_i$ ($0 \leq i \leq n$) in a two-dimensional space of size $2^n \times 2^n$ where the blocks are squares of side length $2^{n-i}$. We start with $A_0$, which consists of just one element $e$, and determine if $o$ is a member of the set of values associated with $e$. If it is not, then we exit and the answer is negative. If it is, then we examine the four elements in $A_1$ that correspond to $e$ and repeat the test. At this point, we know that $o$ is a member of at least one of them as otherwise $o$ could not have been a member of the set of values associated with element $e$ of $A_0$. This process is applied recursively to elements of $A_j$ that contained $o$ (i.e., the appropriate elements of $A_{j+1}$ are examined for $1 \leq j \leq n-1$) until encountering $A_n$ at which time the process stops. The advantage of this method is that elements of $A_{j+1}$ are not examined unless object $o$ is guaranteed to be a member of the set of values associated with at least one of them.

The array pyramid uses a sequence of arrays as an access structure. An alternative implementation is one that imposes an access structure in the form of a tree $T$ on the elements of the hierarchy $\{C_i\}$. One possible implementation is a tree of fanout $q$ where the root $T_0$ corresponds to $C_0$, nodes $\{T_{ij}\}$ at depth $i$ to $C_i$ ($1 \leq i \leq n-1$), while the leaf nodes $\{T_{nj}\}$ correspond to $C_n$. In particular, element $t$ in the tree at depth $j$ corresponds to element $e$ of $C_j$ ($0 \leq j \leq n-1$) and $t$ contains $q$ pointers to its $q$ children in $T_{j+1}$ corresponding to the elements of $C_{j+1}$ that are contained in $e$. The result is termed a *cell-tree pyramid*. Figure 9 shows the cell-tree pyramid corresponding to the collection of objects in Figure 1 where the cells are labeled as in Figure 6a. This example makes use of two-dimensional data with $r_1 = 2$ and $r_2 = 2$. In this case, notice the similarity between the cell-tree pyramid and the region quadtree implementation that uses an access structure which is a tree with a fanout of 4 (Figure 6b).

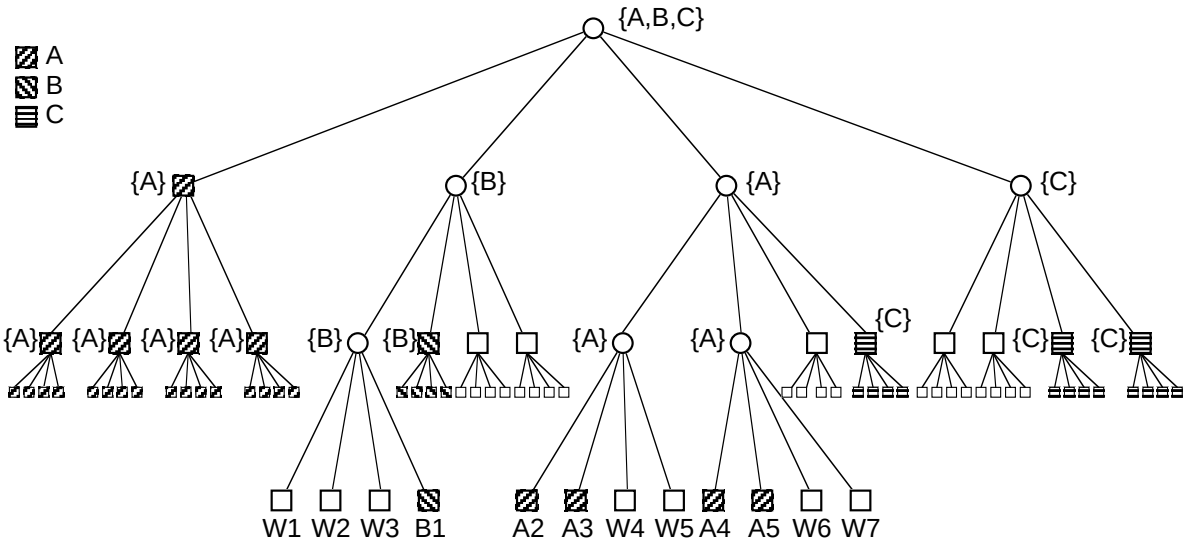


Figure 9: Cell-tree pyramid for the collection of objects and cells in Figure 1.

Using the term *quadtree* in its most general sense (i.e., $d$-dimensional blocks whose sides need not be powers of two nor be of the same length), the cell-tree pyramid can be viewed as a complete quadtree (i.e., where no aggregation takes place at the deepest level, or, equivalently, all leaf nodes with no children are at the maximum depth of the tree). Nevertheless, there are some very important differences. The first difference, as we pointed out before, is that the quadtree is a variable-resolution representation, while the cell-tree pyramid

is a multiresolution representation. The second, and most important, difference is that in the case of the quadtree, the nonleaf nodes serve only as an access structure. They do not include any information about the objects present in the nodes and cells below them. This is why the quadtree, like the array, is not useful for answering the feature query. Of course, we could also devise a variant of the quadtree (termed a *truncated-tree pyramid* [135]) which uses the nonleaf nodes to store information about the objects present in the cells and nodes below them (e.g., Figure 10). Note that both the cell-tree pyramid and the truncated-tree pyramid are instances of an implicit interior-based representation with a tree access structure.

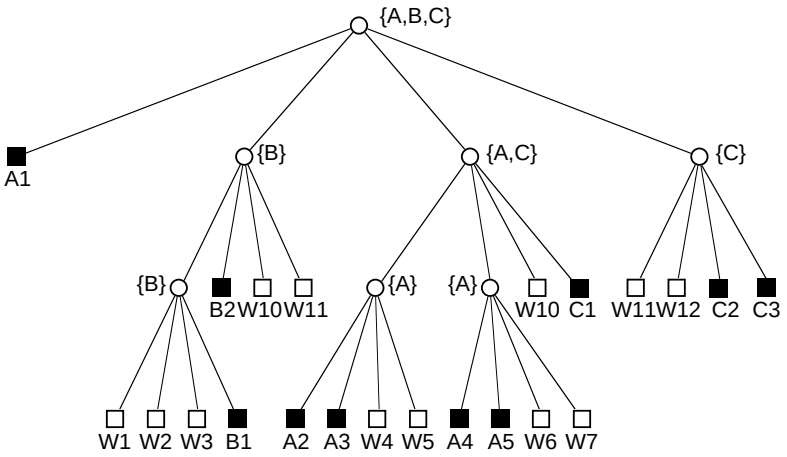


Figure 10: Truncated-tree pyramid for the collection of objects and cells in Figure 1.

Our definition of the pyramid was made in a bottom-up manner in the sense that we started with a block size and an underlying space size. Next, we expanded the size of the underlying space so that a containment hierarchy of congruent blocks at each level and similar blocks at different levels could be formed. We can also define a variant of the pyramid where the requirements of block congruence at each level and block similarity at different levels are relaxed. This is a bit easier if we define the pyramid in a top-down manner as we can calculate the number of cells by which the underlying space needs to be expanded as the block sizes at the different levels are defined. It should be clear that the congruence requirement is more restrictive than the similarity requirement. If we relax the requirement that the blocks at different levels are similar, but retain the requirement that the blocks at the same level are congruent, then we must store at each level $i$ the size of the block $q_i$ (i.e., the values of the individual components $r_{ij}$ of $q_i = \prod_{j=1}^{d} r_{ij}$ for dimension $j$ ($1 \leq j \leq d$)).

If we relax both the requirement that the blocks at the different levels are similar and the requirement that the blocks at each level are congruent while still requiring that they form a containment hierarchy, then we are in effect permitting partitioning hyperplanes (i.e., lines in two dimensions) at arbitrary positions. In this case, we get a more general pyramid if we use a top-down definition as now we can have a different partition at each level. In this case, we have an irregular grid at each level, and thus we must store the positions of the partitioning hyperplanes (i.e., lines in two dimensions) at each level. We call the result an *irregular grid pyramid*. If the irregular grid is implemented with an array access structure, then the result is called an *irregular grid array pyramid*.

Other pyramid variants are also possible. For example, the *dynamically quantized pyramid (DQP)* [114,

149] is a two-dimensional containment hierarchy where the blocks at the different levels are neither similar nor congruent. It differs from the irregular pyramid in that the there is a possibly different $2 \times 2$ grid partition at each block at each level rather than one grid partition at each level. Notice the close similarity to a complete point quadtree [54]. The DQP finds use in cluster detection as well as multidimensional histogramming. Of course, even more general variants are possible. In particular, we could use any one of the other recursive and nonrecursive decompositions described in Section 3 at each block with the appropriate access structure.

# 5 Object-based Hierarchical Interior-based Representations (R-trees)

Our goal here is to be able to take a location *a* as input and return the objects in which *a* is a member (the location query) when using a representation that stores with each object the addresses of the cells that comprise it (i.e., an explicit representation). The most natural hierarchy that can be imposed on the objects that would enable us to answer this query is one that aggregates every *M* objects (that are hopefully in close spatial proximity, although this is not a requirement) into larger objects. This process is repeated recursively until there is just one aggregated object left. Since the objects may have different sizes and shapes, it is not easy to compute and represent the aggregate object. Moreover, it is similarly difficult to test each one of them (and their aggregates) to determine if they contain *a* since each one may require a different test by virtue of the different shapes. Thus, it is useful to use a common aggregate shape and point-inclusion test to prune the search.

The common aggregate shape and point-inclusion test that we use assumes the existence of a minimum enclosing box (termed a *bounding box*) for each object. This bounding box is part of the data associated with each object and aggregate of objects. In this case, we reformulate our object hierarchy to be in terms of bounding boxes. In particular, we aggregate the bounding boxes of every *M* objects into a box (i.e., block) of minimum size that contains them. This process is repeated recursively until there is just one block left. The value associated with the bounding box *b* is its location (e.g., the coordinate values of its diagonally opposite corners for two-dimensional data). It should be clear that the bounding boxes serve as a filter to prune the search for an object that contains *a*.

In this section we expand on hierarchies of objects which actually aggregate the bounding boxes of the objects. Section 5.1 gives an overview of object hierarchies and introduces the general concepts of an object pyramid and an object-tree pyramid, which provides a tree access structure for the object pyramid. Sections 5.2–5.4 present several aggregation methods. In particular, Section 5.2 discusses ordering-based aggregation methods. Section 5.3 discusses extent-based aggregation techniques which result in the R-tree representation, while Section 5.4 describes the R*-tree which is the best of the extent-based aggregation methods. Next, Section 5.5 discusses methods of updating or loading an object-tree pyramid with a large number of objects at once, termed bulk insertion and bulk loading, respectively. Section 5.6 concludes the presentation by reviewing some of the shortcomings of the object-tree pyramid and discussing some of the solutions that have been proposed to overcome them. For a comparative look at these different aggregation methods, see the VASCO JAVA applets found at `http://www.cs.umd.edu/~hjs/quadtree/index.html` [27]. The VASCO system also includes many other indexing techniques for points, lines, rectangles, and regions that

are based on space decomposition (see also the sp-GiST system [5]), Other libraries that are based on object hierarchies include GiST [75] and XXL [23].

## 5.1 Overview

The nature of the aggregation (i.e., using bounding boxes), the number of objects that are being aggregated at each step (as well as whether it can be varied), and, most importantly, deciding which objects to aggregate is quite arbitrary although an appropriate choice can make the search process much more efficient. The decision as to which objects to aggregate assumes that we have a choice in the matter. It could be that the objects have to be aggregated in the order in which they are encountered. This could lead to poor search performance when the objects are not encountered in an order that correlates with spatial proximity. Of course, this is not an issue as long as we just have $\leq M$ objects.

It should be clear that the issue of choice only arises if we know the identities of all the objects before starting the aggregation process (unless we are permitted to rebuild the hierarchy each time we encounter a new object or delete an object), and if we are permitted to reorder them so that objects in aggregate $i$ need not necessarily have been encountered prior to the objects in aggregate $i+1$, and vice versa. This is not always the case (i.e., a dynamic versus a static database), although for the moment we do assume that we know the identities of all of the objects before starting the aggregation, and that we may aggregate any object with any other object. Observe also that the bounding boxes in the hierarchy are not necessarily disjoint. In fact, the objects may be configured in space in such a way that no disjoint hierarchy is possible. By the same reasoning, the objects themselves need not be disjoint.

The process that we have just outlined can be described more formally as follows. Assume that there are $N$ objects in the space and let $n$ be the smallest power of $M$ such that $M^n \geq N$. Assume that all aggregates contain $M$ elements with the exception of the last one at each level which may contain less than $M$ as $M^n$ is not necessarily equal to $N$. The hierarchy of objects consists of the set $D$ of sets $\{D_i\}$ $(0 \leq i \leq n)$ where $D_n$ corresponds to the set of bounding boxes of the individual objects, $D_{n-1}$ corresponds to the result of the initial aggregation of the bounding boxes of $M$ objects into $N/M$ aggregates of objects and consists of $N/M$ bounding boxes, and $D_0$ is a set containing just one element corresponding to the aggregations of all of the objects and is a bounding box that encloses all of the objects. We term the resulting hierarchy an *object pyramid*. Once again, we have a *multiresolution* representation as the original collection of objects is described at several levels of detail by virtue of the number of objects whose bounding boxes are grouped at each level. This is in contrast with the cell pyramid where the different levels of detail are distinguished by the sizes of the cells that comprise the elements at each level.

Searching an object pyramid consisting of sets $D_i$ $(0 \leq i \leq n)$ for the object containing a particular location $a$ (i.e., the location query) proceeds as follows. We start with $D_0$, which consists of just one bounding box $b$, and determine if $a$ is inside $b$. If it is not, then we exit and the answer is negative. If it is, then we examine the $M$ elements in $D_1$ that are covered by $b$ and repeat the test using their bounding boxes. Note that unlike the cell pyramid, at this point, $a$ is not necessarily included in the $M$ bounding boxes in $D_1$ as these $M$ bounding boxes are not required to cover the entire space spanned by $b$. In particular, we exit if $a$ is not covered by

at least one of the bounding boxes at this level. This process is applied recursively to all elements of $D_j$ for $0 \leq j \leq n$ until all elements of $D_n$ have been processed at which time the process stops. The advantage of this method is that elements of $D_j$ $(1 \leq j \leq n)$ are not examined unless $a$ is guaranteed to be covered by at least one of the elements of $D_{j-1}$.

The bounding boxes serve to distinguish between occupied and unoccupied space, thereby indicating whether the search for the objects that contain a particular location (i.e., the location query) should proceed further. At a first glance, it would appear that the object pyramid is rather inefficient for responding to the location query as in the worst case all of the bounding boxes at all levels must be examined. However, the maximum number of bounding boxes in the object pyramid, and hence the maximum number that will have to be inspected, is $\sum_{j=0}^{n} M^j \leq 2N$.

Of course, we may also have to examine the actual sets of locations associated with each object when the bounding box does not result in any of the objects being pruned from further consideration since the objects are not necessarily rectangular in shape (i.e., boxes). Thus using the hierarchy provided by the object pyramid results in at most an additional factor of two in terms of the number of bounding box tests while possibly saving many more tests. Therefore, the maximum amount of work to answer the location query with the hierarchy is of the same order of magnitude to that which would have been needed had the hierarchy not been introduced.

As we can see, the way in which we introduced the hierarchy to form the object pyramid did not necessarily enable us to make more efficient use of the explicit interior-based representation to respond to the location query. The problem was that once we determined that location $a$ was covered by one of the bounding boxes, say $b$, in $D_j$ $(0 \leq j \leq n-1)$, we had no way to access the bounding boxes comprising $b$ without examining all of the bounding boxes in $D_{j+1}$. This is easy to rectify by imposing an access structure in the form of a tree $T$ on the elements of the hierarchy $D$. One possible implementation is a tree of fanout $M$ where the root $T_0$ corresponds to the bounding box in $D_0$. $T_0$ has $M$ links to its $M$ children $\{T_{1k}\}$ which correspond to the $M$ bounding boxes in $D_1$ that comprise $D_0$. The set of nodes $\{T_{ik}\}$ at depth $i$ correspond to the bounding boxes in $D_i$ $(0 \leq i \leq n)$, while the set of leaf nodes $\{T_{nk}\}$ correspond to $D_n$. In particular, node $t$ in the tree at depth $j$ corresponds to bounding box $b$ in $D_j$ $(0 \leq j \leq n-1)$, and $t$ contains $M$ pointers to its $M$ children in $T_{j+1}$ corresponding to the bounding boxes in $D_{j+1}$ that are contained in $b$. We use the term *object-tree pyramid* to describe this structure.

Figure 11a is an example object-tree pyramid for a simple collection of 9 rectangle objects with $M = 3$ (and thus $n = 2$). Figure 11b shows the spatial extents of the objects and the bounding boxes of the nodes in Figure 11a, with broken lines denoting the bounding boxes corresponding to the leaf nodes. Note that the object-tree pyramid is not unique. Its structure depends heavily on the order in which the individual objects and their corresponding bounding boxes are aggregated.

The object-tree pyramid that we have just described still has a worst case where we may have to examine all of the bounding boxes in $D_j$ $(1 \leq j \leq n)$ when executing the location query or its variants (e.g., a window query). This is the case if query location $a$ is contained in every bounding box in $D_{j-1}$. Such a situation, although rare, can arise in practice because $a$ may be included in the bounding boxes of many objects (termed

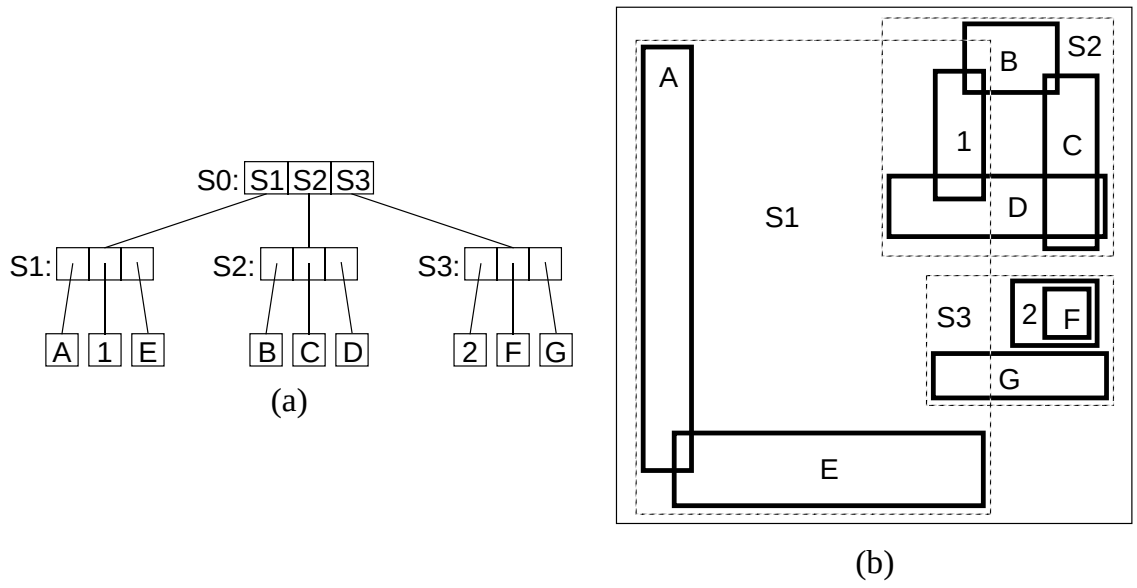


Figure 11: (a) Object-tree pyramid for a collection of rectangle objects with M=3, and (b) the spatial extents of the objects and the bounding boxes of the nodes in (a) with broken lines denoting the bounding boxes of the corresponding leaf nodes. Notice that the leaf nodes in the index also store bounding boxes although this is shown only for the nonleaf nodes.

a *false hit*), as the bounding boxes are not disjoint, while *a* is contained in a much smaller number of objects. Equivalently, false hits are caused by the fact that a spatial object may be spatially contained in full or in part in several bounding boxes or nodes while being associated with just one node or bounding box.

However, unlike the object pyramid, the object-tree pyramid does guarantee that only the bounding boxes that contain *a* will be examined and no others. Thus we have not improved on the worst-case of the object pyramid in that we may still have to examine $2N$ bounding boxes, although we have reduced its likelihood. It is interesting to observe that the object pyramid and the object-tree pyramid are instances of an explicit interior-based representation since it is still the case that associated with each object *o* is a set containing the addresses of the cells that comprise it. Note also that the access structure facilitates only the determination of the object associated with a particular cell and not which cells are contiguous. Thus the object-tree pyramid is not an instance of an implicit interior-based representation.

The decision as to which objects to aggregate plays an important factor in the efficiency of the object-tree pyramid in responding to the location query. The efficiency of the object-tree pyramid for search operations depends on its abilities to distinguish between occupied space and unoccupied space, and to prevent a node from being examined needlessly due to a false overlap with other nodes.

The extent to which these efficiencies are realized is a direct result of how well our aggregation policy is able to satisfy the following two goals. The first goal is to minimize the number of aggregated nodes that must be visited by the search. This goal is accomplished by minimizing the area common to sibling aggregated nodes (termed *overlap*). The second goal is to reduce the likelihood that sibling aggregated nodes are visited by the search. This is accomplished by minimizing the total area spanned by the bounding boxes of the sibling aggregated nodes (termed *coverage*). A related goal to that of minimizing the coverage is one of minimizing

the area in sibling aggregated nodes that is not spanned by the bounding boxes of any of the children of the sibling aggregated nodes (termed *dead area*). Dead area is usually decreased by minimizing coverage and thus minimizing dead area is often not taken into account explicitly. Another way of interpreting these goals is that they are designed to ensure that objects that are spatially close to each other are stored in the same node. Of course, at times, these goals may be contradictory.

For example, consider the four bounding boxes in Figure 12a. The first goal is satisfied by the aggregation in Figure 12c, while the second goal is satisfied by the aggregation in Figure 12b. The dead area is shown shaded in Figures 12b and 12c. Note that the dead area in Figure 12b is considerably smaller than the dead area in Figure 12c on account of the smaller amount of coverage in the children in Figure 12b. Also observe that the dead area for the bounding box of one aggregated node is not part of the bounding boxes of children a sibling aggregated node as seen in Figure 12b.

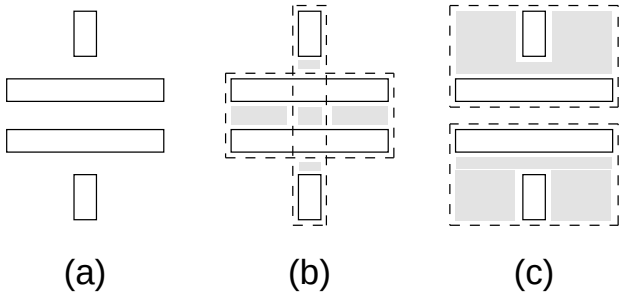


Figure 12: (a) Four bounding boxes and the aggregations that would be induced, (b) by minimizing the total area (i.e., coverage) of the covering bounding boxes of the two nodes and (c) by minimizing the area common (i.e., overlap) to the covering bounding boxes of the two nodes. The dead area for the two possible aggregations is shown shaded.

These goals could be satisfied by using trial-and-error methods that examine all possible aggregations and choose the one that yields the minimum amount of overlap or coverage among the constituent bounding boxes of the nodes as well as among the nodes at a given level. The cost is clearly prohibitive. These trial-and-error methods can be made more intelligent by use of iterative optimization [66]. However, the cost is still too high.

The aggregation techniques described above take the space (i.e., volume) occupied by (termed *extent of*) the bounding boxes of the individual spatial objects into account. They are described in Sections 5.3 and 5.4. An alternative is to order the objects prior to performing the aggregation. However, in this case, the only choice that we may possibly have with respect to the identities of the objects which are aggregated is when the number of objects (or bounding boxes) that are being aggregated at each step is permitted to vary. The most obvious order, although not particularly interesting or useful, is one that preserves the order in which the objects were initially encountered (i.e., objects in aggregate $i$ have been encountered before those in aggregate $i+1$). The more common orders are based on proximity or on the values of a small set of parameters describing a common property that is hopefully related to the proximity (and to a lesser degree to the shape and extent) of the objects or their bounding boxes in one or all of the dimensions of the space in which they lie [88, 107, 128]. Ordering-based aggregation techniques are discussed in Section 5.2.

## 5.2 Ordering-based Aggregation Techniques

The most frequently used ordering technique is based on mapping the bounding boxes of the objects to a representative point in a lower, the same, or a higher-dimensional space and then applying one of the space-ordering techniques described in Section 2 and shown in Figure 2. We use the term *object number* to refer to the result of the application of space ordering[4]. Some possible representative points for two-dimensional rectangle objects include the following (e.g., [134]):

1. The centroid.
2. The centroid and the horizontal and vertical extents (i.e., the horizontal and vertical distances from the centroid to the relevant sides).
3. The $x$ and $y$ coordinate values of the two diagonally opposite corners of the rectangle (e.g., the upper-left and lower-right corners).
4. The $x$ and $y$ coordinate values of the lower-right corner of the rectangle and its height and width.

For example, consider the collection of 22 rectangle objects given in Figure 13 where the numbers associated with the rectangles denote the relative times at which they were created. Figure 14 shows the result of applying a Morton order (Figure 14a) and Peano-Hilbert order (Figure 14b) to the collection of rectangle objects in Figure 13 using their centroids as the representative points.

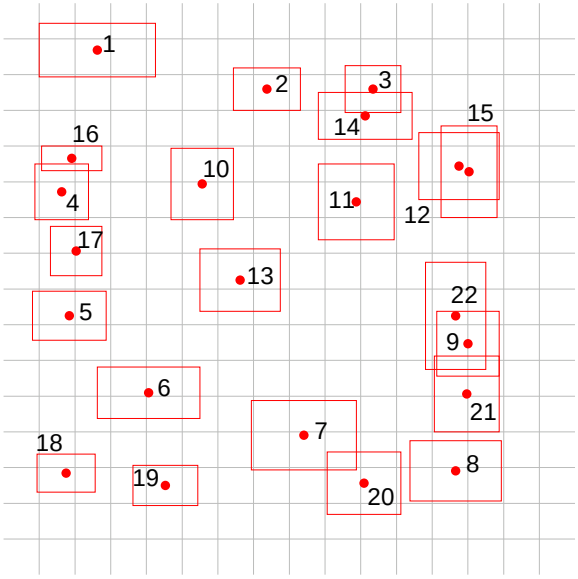


Figure 13: A collection 22 rectangle objects where the numbers associated with the rectangles denote the relative times at which they were created.

[4]Interestingly, we will see that no matter which of the implementations of the object-tree pyramid is being deployed, the ordering is used primarily to build the object-tree pyramid, although it is used for splitting in some cases such as the Hilbert R-tree [89]. The actual positions of the objects in the ordering (i.e., the object numbers) are not usually recorded in the object-tree pyramid which is somewhat surprising as this could be used to speed up operations such as point location, etc.

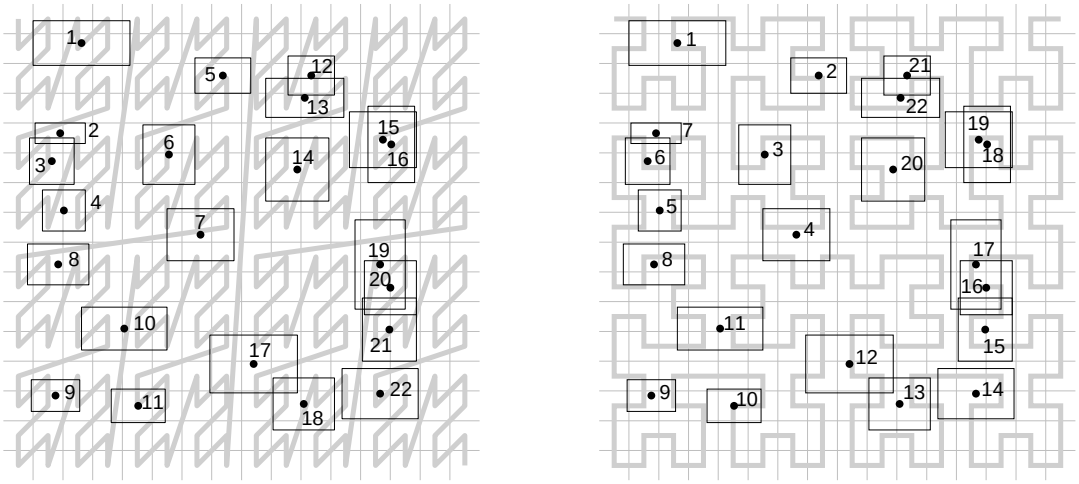


Figure 14: The result of applying (a) a Morton order, and (b) a Peano-Hilbert order to the collection of 22 rectangle objects in Figure 13 using their centroids as the representative points.

Once the $N$ objects have been ordered, the hierarchy $D$ is built in the order $D_n$, $D_{n-1}$,$\cdots$, $D_1$, $D_0$ where $n$ is the smallest power of $M$ such that $M^n \geq N$. $D_n$ consists of the set of original objects and their bounding boxes. There are two ways of grouping the items to form the hierarchy $D$: one-dimensional and multidimensional.

In the one-dimensional grouping method, $D_{n-1}$ is formed as follows. The first $M$ objects and their corresponding bounding boxes form the first aggregate, the second $M$ objects and their corresponding bounding boxes form the second aggregate, etc. $D_{n-2}$ is formed by applying this aggregation process again to the set $D_{n-1}$ of $N/M$ objects and their bounding boxes. This process is continued recursively until we obtain the set $D_0$ containing just one element corresponding to a bounding box that encloses all of the objects. Note however, that when the process is continued recursively, the elements of the sets $D_i (0 \leq i \leq n-1)$ are not necessarily ordered in the same manner as the elements of $D_n$.

There are several implementations of the object-tree pyramid using the one-dimensional grouping methods. For example, the *Hilbert packed R-tree* [88] is an object-tree pyramid that makes use of a Peano-Hilbert order. It is important to note that only the leaf nodes of the Hilbert packed R-tree are ordered using the Peano-Hilbert order. The nodes at the remaining levels are ordered according to the time at which they were created. For example, Figure 15a shows the bounding boxes corresponding to the first level of aggregation for the Hilbert packed R-tree for the collection of 22 rectangle objects in Figure 13 with $M = 6$. Similarly, Figure 15b shows the same result were we to build the same structure using a Morton order (i.e., a Morton packed R-tree), again with $M = 6$. Notice that there is quite a bit of overlap among the bounding boxes as the aggregation does not take the extent of the bounding boxes into account when forming the structure.

A slightly different approach is employed in the *packed R-tree* [128] which is another instance of an object-tree pyramid. The packed R-tree is based on ordering the objects on the basis of some criterion such as increasing value of the $x$ coordinate or any of the space-ordering methods shown in Figure 2. Once this order has been obtained, the leaf nodes in the packed R-tree are filled by examining the objects in increasing

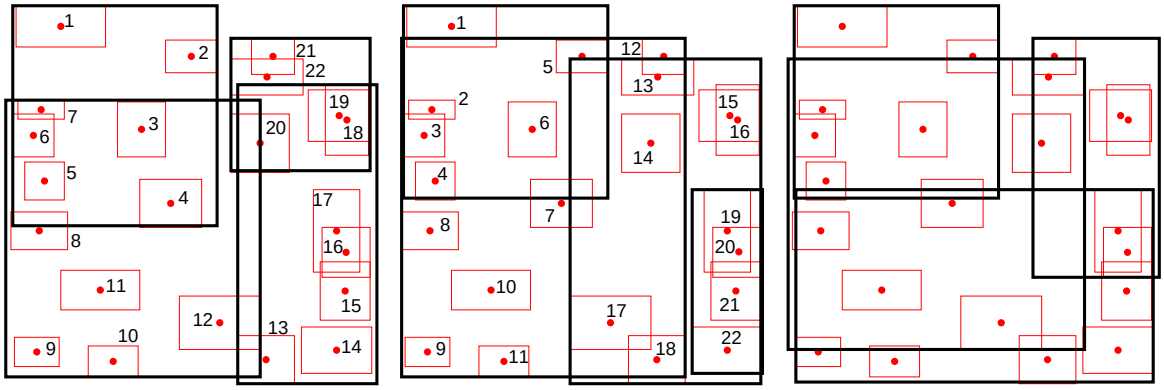


Figure 15: The bounding boxes corresponding to the first level of aggregation for the (a) Hilbert packed R-tree, (b) Morton packed R-tree, and (c) packed R-tree (using a Peano-Hilbert order for the initial ordering) for the collection of 22 rectangle objects in Figure 13 with M=6. The numbers associated with the rectangle objects in (a) and (b) denote the positions of their corresponding centroids in the order.

order where each leaf node is filled with the first unprocessed object and its $M-1$ nearest neighbors which have not yet been inserted in other leaf nodes. Once an entire level of the packed R-tree has been obtained, the algorithm is reapplied to add nodes at the next level using the same nearest neighbor criterion, terminating when a level contains just one node. The only difference between the ordering that is applied at the levels containing the nonleaf nodes from that used at the level of the leaf nodes is that in the former case we are ordering the bounding boxes while in the latter case we are ordering the actual objects.

Besides the difference in the way nonleaf nodes are formed, we point out that the packed R-tree construction process makes use of a proximity criterion in the domain of the actual data rather than the domain of the representative points which is the case of the Hilbert packed R-tree. This distinction is quite important as it means that the Hilbert packed R-tree construction process makes no attempt to reduce or minimize coverage and overlap which, as we shall soon see, are the real cornerstones of the R-tree data structure [74]. Therefore, as we point out below, this makes the Hilbert packed R-tree (and, to a lesser extent, the packed R-tree) much more like a B-tree that is constructed by filling each node to capacity. For example, Figure 15c shows the bounding boxes corresponding to the first level of aggregation for the packed R-tree for the collection of 22 rectangle objects in Figure 13. In this case, the objects were initially ordered using a Peano-Hilbert order.

The *STR* method (denoting sort-tile-recurse) of Leutenegger, López, and Edgington [101] is an example of the multidimensional grouping method. Our explanation assumes, without loss of generality, that the underlying space is two-dimensional although the extension of the method to higher dimensions is straightforward. Assuming a total of $N$ rectangles and a node capacity of $M$ rectangles per leaf node, $D_{n-1}$ is formed by constructing a tiling of the underlying space consisting of $s$ vertical slabs where each slab contains $s$ tiles. Each tile corresponds to an object-tree pyramid leaf node which is filled to capacity. Note that the result of this process is that the underlying space is being tiled with rectangular tiles thereby resembling a grid,

but, most importantly, unlike a grid, the horizontal edges of horizontally adjacent tiles (i.e., with a common vertical edge) do not form a straight line (i.e., are not connected). Using this process means that the underlying space is tiled with approximately $\sqrt{N/M} \times \sqrt{N/M}$ tiles and results in approximately $N/M$ object-tree pyramid leaf nodes. The tiling process is applied recursively to these $N/M$ tiles to form $D_{n-2}$, $D_{n-3}$, ... etc. until obtaining just one node.

The STR method builds the object-tree pyramid in a bottom-up manner. The actual mechanics of the STR method are as follows. Sort the rectangles on the basis of one coordinate value of some easily identified point that is associated with them, say the $x$ coordinate value of their centroid. Aggregate the sorted rectangles into $\sqrt{N/M}$ groups of $\sqrt{NM}$ rectangles each of which forms a vertical slab containing all rectangles whose centroid's $x$ coordinate value lies in the slab. Next, for each vertical slab $v$, sort all rectangles in $v$ on the basis of their centroid's $y$ coordinate value. Aggregate the $\sqrt{NM}$ sorted rectangles in each slab $v$ into $\sqrt{N/M}$ groups of $M$ rectangles each. Recall that the elements of these groups form the leaf nodes of the object-tree pyramid. Notice that the minimum bounding boxes of the rectangles in each tile are usually larger than the tiles. The process of forming a grid-like tiling is now applied recursively to the $N/M$ minimum bounding boxes of the tiles with $N$ taking on the value of $N/M$ until the number of tiles is no larger than $M$, in which case all of the tiles fit in the root node and we are done.

A couple of items are worthy of further note. First, the minimum bounding boxes of the rectangles in each tile are usually larger than the tiles. This means that the tiles at each level will overlap. Thus we do not have a true grid in the sense that the elements at each level of the object-tree pyramid are usually not disjoint. Second, the ordering that is applied is quite similar to a row order (actually column order to be precise) as illustrated in Figure 2a where the $x$ coordinate value serves as a primary key to form the vertical slabs while the $y$ coordinate value serves as the secondary key to form the tiles from the vertical slabs. Nevertheless, the ordering serves only to determine the partitioning lines to form the tiles but is not used to organize the collection of tiles.

Notice that the STR method is a bottom-up technique. However, the same idea could also be applied in a top-down manner so that we originally start with $M$ tiles which are then further partitioned. In other words, we start with $\sqrt{M}$ vertical slabs containing $\sqrt{M}$ tiles apiece. This is instead of the initial $\sqrt{N/M}$ vertical slabs containing $\sqrt{N/M}$ tiles in the bottom-up method. The disadvantage of the top-down method is that it requires that we make roughly $2\log_M N$ passes over all of the data whereas the bottom-up method has the advantage of making just two passes over the data (one for the $x$ coordinate value and one for the $y$ coordinate value) since all recursive invocations of the algorithm deal with centroids of the tiles.

The top-down method can be viewed as an ordering technique in the sense that the objects are partitioned, thereby creating a partial ordering, according to their relative position with respect to some criterion such as a value of a statistical measure for the set of objects as a whole. For example, in the VAMSplit R-tree of White and Jain [163], which is applied to point data, the split axis (i.e., $x$ or $y$ or $z$, etc.) is chosen on the basis of having the maximum variance from the mean in the distribution of the point data. Once the axis is chosen, the objects are split is into two equally-sized sets constrained so that the resulting nodes are as full as possible. This process is applied recursively to the resulting sets.

The top-down method is also used by García, López, and Leutenegger [62] with a different partitioning strategy. For each dimension, this strategy applies a user-defined function to decide on the quality or penalty incurred by the split (e.g., coverage, overlap, etc.). Assuming $N$ objects and a bucket capacity $M$ (which is also the fanout of packed nonleaf nodes), the partitioning algorithm uses a heuristic that considers $O(M)$ split positions and selects the one among those that yields the minimum cost or penalty.

The algorithm proceeds as follows. At the initial step, it sorts the $N$ objects along each dimension, and then groups the sorted objects into $M$ groups of $l = \lceil N/M \rceil$ objects each. It then constructs the minimum bounding box of each group in $O(N)$ time. Next, it processes the bounding boxes of the groups in increasing order and forms a bounding box for the first two groups, the first three groups, ..., up to the first $l-1$ groups. The algorithm considers many different orderings, and chooses the best split among all the different orderings. The suggested orderings are the min, max, and center of the bounding boxes for each dimension — that is, for two-dimensional data, the suggested algorithm may consider between two, four, or six different orderings at each pass. The same process is applied to the bounding boxes in decreasing order. This can be done in $O(M)$ time. At this point, the algorithm finds the optimal split position by considering all possible $O(M)$ split positions, which can also be done in $O(M)$ time. This results in two buckets containing $i \cdot l$ and $N - i \cdot l$ where $i$ is between 1 and $M-1$. This process is then applied to each bucket that contains more than $l$ objects, which may also require that the objects be sorted again. Once all buckets have $\leq l$ objects, we have completed the first level of the R-tree. Next, this process is applied recursively to the subtrees at the next level.

If at each step of the algorithm, the nodes resulting from the split contain approximately the same number of bounding boxes, then the sorting component of the algorithm performs a minimum number of comparisons as the maximum sizes of the buckets are minimized. In order to analyze the execution time of the algorithm, we assume a worst-case scenario for each split (which means that $i = 1$ or equivalently $i = M-1$) at each stage for each level. It can be shown that in such a case the total execution time for sorting (which is the dominant cost factor in the algorithm) is $O(c \cdot d \cdot N \cdot (\log N)^2 \cdot M/\log M)$ where $c$ is the number of possible orderings and $d$ is the dimensionality of the data [3]. It is important to note that use of this method does not necessarily result in a minimum cost partition since it does not take into account all of the possible groupings of the $N$ objects, which is exponential in $N$ (i.e., $O(2^N)$).

Regardless of how the objects are aggregated, the object-tree pyramid is analogous to a height-balanced $M$-ary tree where only the leaf nodes contain data (objects in this case), and all of the leaf nodes are at the same level. Thus the object-tree pyramid is good for static data sets. However, in a dynamic environment where objects are added and deleted at will, the object-tree pyramid needs to be rebuilt either entirely or partially to maintain the balance, order, and node size constraints. In the case of binary trees, this issue is addressed by making use of a B-tree, or a $B^+$-tree if we wish to restrict the data (i.e., the objects) to the leaf nodes as is the case in our application. Below, we show how to use the $B^+$-tree to make the object-tree pyramid dynamic.

When the aggregation in the object-tree pyramid is based on ordering the objects, the objects and their bounding boxes can be stored directly in the leaf nodes of the $B^+$-tree. We term the result an *object $B^+$-tree*. The key difference between the object $B^+$-tree and the object-tree pyramid is that the $B^+$-tree (and likewise the object $B^+$-tree) permits the number of objects and nodes that are aggregated at each step to vary (i.e.,

the number of children per node). This is captured by the order of the B$^+$-tree, where for an order $(m,M)$ B$^+$-tree, this number usually ranges between $m \geq \lceil M/2 \rceil$ and $M$ with the root having at least 2 children unless it is a leaf node. The only modification to the B$^+$-tree definition is in the format of the nodes of the object B$^+$-tree. In particular, the format of each nonleaf node $p$ is changed so that if $p$ has $j$ children, then $p$ contains the following 3 items of information for each child $s$:

1. A pointer to $s$.
2. The maximum object number associated with any of the children of $s$ (analogous to a key in the conventional B$^+$-tree).
3. The bounding box $b$ for $s$ (e.g., the coordinate values of a pair of diagonally opposite corners of $b$).

Notice that $j$ bounding boxes are stored in each node corresponding to the $j$ children instead of just one bounding box as called for in the definition of the object-tree pyramid. This is done to speed up the point-inclusion tests necessary to decide which child to descend when executing the location query. In particular, it avoids a disk access when the nodes are stored on disk.

A leaf node $p$ in the object B$^+$-tree has a similar format with the difference that instead of having pointers to $j$ children which are nodes in the tree, $p$ has $j$ pointers to records corresponding to the $j$ objects that it represents. Therefore, $p$ contains the following 3 items of information for each object $s$:

1. A pointer to the actual object corresponding to $s$.
2. The object number associated with $s$.
3. The bounding box $b$ for $s$ (e.g., the coordinate values of a pair of diagonally opposite corners of $b$).

Observe that unlike the object-tree pyramid, the object B$^+$-tree does store object numbers in both the leaf and nonleaf nodes in order to facilitate updates. The update algorithms (i.e., data structure creation, insertion, and deletion) for an object B$^+$-tree are identical to those for a B$^+$-tree with the added requirement of maintaining the bounding box information, while the search algorithms (e.g., the location query, window queries, etc.) are identical to those for an object-tree pyramid. The performance of the object B$^+$-tree for answering range queries is enhanced if the initial tree is built by inserting the objects in sorted order filling each node to capacity, subject to the minimum occupancy constraints, thereby resulting in a tree with minimum depth. Of course, such an initialization will cause subsequent insertions to be more costly as they will inevitably result in node split operations whereas this would not necessarily be the case if the nodes were not filled to capacity initially. The *Hilbert R-tree* [89] is an instance of an object B$^+$-tree that applies a Peano-Hilbert space ordering (Figure 2d) to the centroid of the bounding boxes of the objects. The Hilbert R-tree is closely related to the *Hilbert tree* [99] which applies the same ordering to a set of points and then stores the result in a height-balanced binary tree (see also [160] which makes use of a Morton order and a 1-2 brother tree [115]).

Figure 16a shows the bounding boxes corresponding to the first level of aggregation for the Hilbert R-tree for the collection of 22 rectangle objects in Figure 13 with $m = 3$ and $M = 6$ when the objects are inserted

in the order in which they were created (i.e., their corresponding number in Figure 13. Similarly, Figure 16b shows the corresponding result when using a Morton order instead of a Peano-Hilbert order. Notice that for pedagogical reasons, the trees were not created by inserting the objects in sorted order as suggested above as in this case the resulting trees would be the same as the Hilbert packed R-tree and Morton packed R-tree in Figures 15a and 15b, respectively.

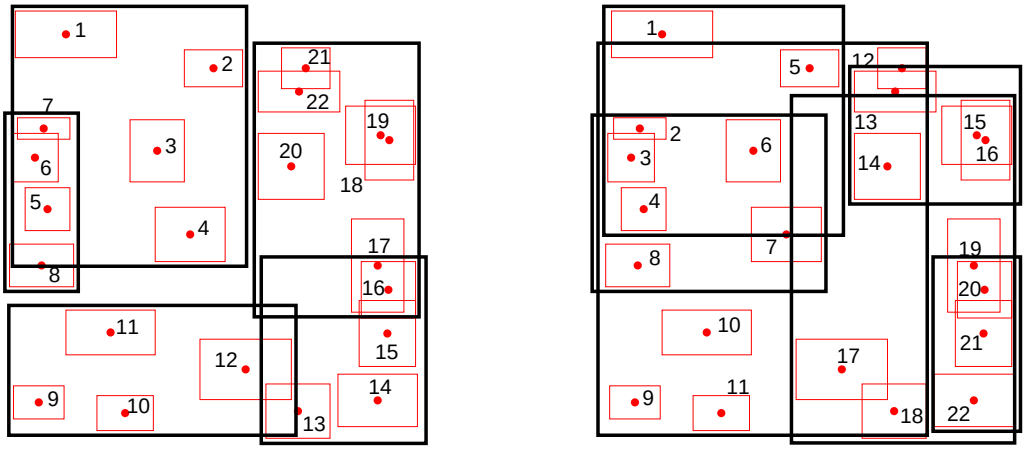


Figure 16: The bounding boxes corresponding to the first level of aggregation for the (a) Hilbert R-tree, and (b) Morton R-tree for the collection of 22 rectangle objects in Figure 13 with m=3 and M=6.

Observe that use of ordering-based aggregation methods can lead to substantial overlap between the bounding boxes of the nodes. Rearranging the objects that are aggregated in each node can alleviate this problem, but only to a very limited extent as the order of the leaf nodes must be maintained — that is, all elements of leaf node $i$ must have a Peano-Hilbert (Morton) order number that is less than all elements of leaf node $i+1$. Thus all we can do is change the number of elements that are aggregated in the node subject to the node capacity constraints. Of course, this means that the resulting trees are not unique. For example, in Figure 16a we could aggregate objects 13–17 into one nonleaf node and objects 18–22 into another nonleaf node which results in less overlap. However, the real shortcoming is that it could be the case that objects 2 and 20 should be aggregated (actually object 2 with objects 18–22) but this is impossible as their corresponding positions in the Peano-Hilbert order are so far apart. The problem is caused, in part, by the presence of objects with nonzero extent and the fact that neither the extent of the objects nor their proximity is taken into account in the ordering-based aggregation techniques (i.e., they do not try to minimize coverage and/or overlap which are the cornerstones of the R-tree). This deficiency was also noted earlier for the Hilbert packed R-tree.

## 5.3 Extent-based Aggregation Techniques

When the objects are to be aggregated on the basis of their extent (i.e., the space occupied by their bounding boxes), then good dynamic behavior is achieved by making use of an *R-tree* [74]. An R-tree is a generalization of the object-tree pyramid where, for an order $(m, M)$ R-tree, the number of objects or bounding boxes that are aggregated in each node is permitted to range between $m \leq \lceil M/2 \rceil$ and $M$ while it is always $M$ for the object-tree pyramid. 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 box of an object. The R-tree is usually built as the objects are encountered rather than waiting until all objects have been input. Of the different variations on the object-tree pyramid that we discussed, the R-tree is the one that is used most frequently, especially in database applications.

Figure 17a is an example R-tree for the same collection of 9 rectangle objects given in Figure 11 with $m = 2$ and $M = 3$. Figure 17b shows the spatial extents of the objects and the bounding boxes of the nodes in Figure 17 with broken lines denoting the bounding boxes corresponding to the leaf nodes, and gray lines denoting the bounding boxes 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.

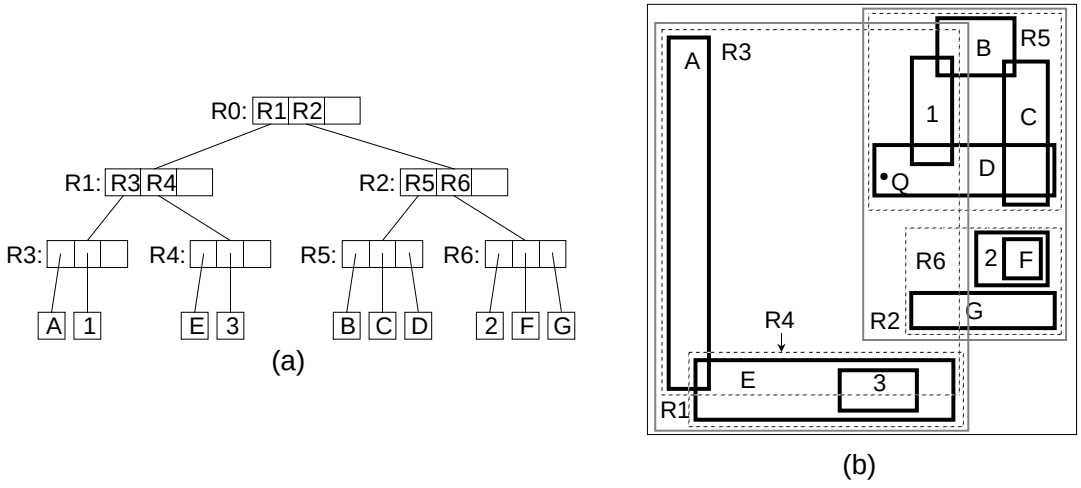


Figure 17: (a) R-tree for the same collection of rectangle objects given in Figure 11 with m=2 and M=3, and (b) the spatial extents of the objects and the bounding boxes of the nodes in (a) with broken lines denoting the bounding boxes of the corresponding leaf nodes. Notice that the leaf nodes in the index also store bounding boxes, although this is shown only for the nonleaf nodes.

Given that each R-tree node can contain a varying number of objects or bounding boxes, it is not surprising that the R-tree was inspired by the B-tree. This means that 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., variants of the location query), which means that $m$ and $M$ are usually quite large.

The need to minimize the number of disk accesses also effects the format of each R-tree node. Recall that in the definition of the object-tree pyramid, each node $p$ contains $M$ pointers to $p$'s children and one bounding

box corresponding to the union of the bounding boxes of $p$'s children. This means that in order to decide which of node $p$'s children should be descended, we must access the nodes corresponding to these children to perform the point-inclusion test. Each such access requires a disk I/O operation. In order to avoid these disk I/O operations, the format of R-tree node $p$ is modified so that $p$ contains $k$ ($m \leq k \leq M$) pointers to $p$'s children and the $k$ bounding boxes of $p$'s children instead of containing just one bounding box corresponding to the union of the bounding boxes of $p$'s children as is the case for the object-tree pyramid[5]. Recall that this format is also used in the definition of a node in the object $B^+$-tree. Once again, we observe that the $k$ point-inclusion tests do not require any disk I/O operations at the cost of being able to aggregate a smaller number of objects in each node since $m$ and $M$ are now smaller assuming that the page size is fixed.

As long as the number of objects in each R-tree leaf node is between $m$ and $M$, no action needs to be taken on the R-tree structure other than adjusting the bounding boxes when inserting or deleting an object. If the number of objects in a leaf node decreases below $m$, then the node is said to *underflow*. In this case, the objects in the underflowing nodes must be reinserted, and bounding boxes in nonleaf nodes must be adjusted. If these nonleaf nodes also underflow, then the objects in their leaf nodes must also be reinserted. If the number of objects in a leaf node increases above $M$, then the node is said to *overflow*. In this case, it must be split and the $M+1$ objects that it contains must be distributed in the two resulting nodes. Splits are propagated up the tree.

Underflows in an R-tree are handled in an analogous manner to the way they are dealt with in a B-tree. In contrast, the overflow situation points out a significant difference between an R-tree and a B-tree. Recall that overflow is a result of attempting to insert an item $t$ in node $p$ and determining that node $p$ is too full. In a B-tree, we usually don't have a choice as to the node $p$ that is to contain $t$ since the tree is ordered. Thus once we determine that $p$ is full, we must either split $p$ or apply a rotation (also known as *deferred splitting*) process. On the other hand, in an R-tree, we can insert $t$ in any node $p$, as long as $p$ is not full. However, once $t$ is inserted in $p$, we must expand the bounding box associated with $p$ to include the space spanned by the bounding box $b$ of $t$. Of course, we can also insert $t$ in a full node $p$, in which case we must also split $p$.

The need to expand the bounding box of $p$ has an effect on the future performance of the R-tree, and thus we must make a wise choice with respect to $p$. As in the case of the object-tree pyramid, the efficiency of the R-tree for search operations depends on its abilities to distinguish between occupied space and unoccupied space, and to prevent a node from being examined needlessly due to a false overlap with other nodes. Again, as in the object-tree pyramid, the extent to which these efficiencies are realized is a direct result of how well we are able to satisfy our goals of minimizing coverage and overlap. These goals 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.

In the original definition of the R-tree [74] the goal of minimizing coverage is the one that is followed. In particular, an object $t$ is inserted by a recursive process that starts at the root of the tree and chooses the

[5]The A-tree [132] is somewhat of a compromise in that it stores quantized approximations of the $k$ bounding boxes of $p$'s children where the locations of the bounding boxes of $p$'s children are specified relative to the location of the bounding box of $p$ thereby enabling them to be encoded with just a small number of bits. This idea was first proposed by Henrich [76] and is also used in the hybrid tree of Chakrabarti and Mehrotra [33, 34].

child whose corresponding bounding box needs to be expanded by the smallest amount to include $t$. As we will see, other researchers make use of other criteria such as minimizing overlap with adjacent nodes and even perimeter (e.g., in the R*-tree [19] as described in Section 5.4). Theodoridis and Sellis [156, 157] try to minimize the value of an objective function consisting of a linear combination of coverage, overlap, and dead area with equal weights. García, and López, and Leutenegger [62] also make use of a similar objective function to build the entire R-tree in a top-down manner.

Not surprisingly, these same goals also guide the node-splitting process. In this situation, one goal is to distribute the objects among the nodes so that the likelihood that the two nodes will be visited in subsequent searches will be reduced. This is accomplished by minimizing the total area spanned by the bounding boxes of the resulting nodes (equivalent to what we termed *coverage*). The second goal is to reduce the likelihood that both nodes are examined in subsequent searches. This goal is accomplished by minimizing the area common to both nodes (equivalent to what we termed *overlap*). Again, we observe that, at times, these goals may be contradictory.

Several node-splitting policies have been proposed that take these goals into account. They are differentiated on the basis of their execution-time complexity and by the number of these goals that they attempt to meet. An easy way to see the different complexities is to look at the following three algorithms [74], all of which are based on minimizing the coverage. The simplest is an exhaustive algorithm [74] that tries all possibilities. In such a case, the number of possible partitions is $2^M - 1$. This is unreasonable for most values of $M$ (e.g., $M = 50$ for a page size of 1024 bytes).

The exhaustive approach can be applied to obtain an optimal node split according to an arbitrary cost function that can take into account coverage, overlap, and other factors. Interestingly, although we pointed out earlier that there are $O(2^M)$ possible cases to be taken into account, the exhaustive algorithm can be implemented in such a way that it need not require $O(2^M)$ time. In particular, Becker et al. [18] present an implementation that takes only $O(M^3)$ time for two-dimensional data and $O(dM \log M + d^2 M^{2d-1})$ time for $d$-dimensional data.

García, and López, and Leutenegger [63] present an implementation of the exhaustive approach that uses the same insight as the implementation of Becker et al. [18], which is that some of the boundaries of the two resulting minimum bounding boxes are shared with the minimum bounding box of the overflowing node. This insight constrains the number of possible groupings of the $M$ objects in the node that is being split. The algorithm is flexible in that it can use different cost functions for evaluating the appropriateness of a particular node split. However, the cost function is restricted to being "extent monotone" which means that the cost function increases monotonically as the extent of one of the sides of the two bounding rectangles is increased (this property is also used by Becker et al. [18], although the property is stated somewhat differently).

Although the implementations of Becker et al. [18] and García, and López, and Leutenegger [63] both find optimal node splits, the difference between them is that the former has the added benefit of guaranteeing that the node split satisfies some balancing criteria, which is a requirement in most R-tree implementations. The rationale, as we recall, is that in this way the nodes are not too full, which would cause them to overflow again. For example, in many R-tree implementations there is a requirement that the split be such that

each node receives exactly half the rectangles, or that each receives at least 40% of the rectangles. Satisfying the balancing criteria is more expensive, as could be expected, and in two dimensions, the cost of the algorithm of Becker et al. [18] is $O(M^3)$ as opposed to $O(M^2)$ for the algorithm of García, and López, and Leutenegger [62].

García, and López, and Leutenegger [63] found that identifying optimal node splits yielded only modest improvements in query performance which led them to introduce another improvement to the insertion process. This improvement is based on trying to fit one of the two groups resulting from a split of node *e* into one of *e*'s siblings instead of creating a new node for every split. In particular, one of the groups is inserted into the sibling *s* for which the cost increase, using some predefined cost function, resulting from movement into *s* is minimized. Once a sibling *s* has been chosen, we move the appropriate group and reapply the node splitting algorithm if the movement caused *s* to overflow. This process is applied repeatedly as long as there is overflow while requiring that we choose among the siblings that have not been modified by this process. If we find that there is overflow in node *i* and there is no unmodified sibling left, then a new node is created containing one of the new groups resulting from the split of *i*. Even if each node overflows, this process is guaranteed to terminate as at each step there is one less sibling candidate for motion.

The process described above is somewhat similar to what is termed *forced reinsertion* in the R*-tree (see Section 5.4) with the difference that forced reinsertion results in reinsertion of the individual entries (i.e., objects in the case of leaf nodes and minimum bounding boxes in the case of nonleaf nodes) at the root instead of as a group into one of the siblings. This reinsertion into siblings is also reminiscent of rotation (i.e., "deferred splitting") in conventional B-trees with the difference being that there is no order in the R-tree which is why motion into all unmodified siblings had to be considered. This strategy was found to increase the node utilization and thereby improve query performance (by as much as 120% in experiments [63] compared to the Hilbert R-tree [89]).

The remaining two node-splitting algorithms have a common control structure that consists of two stages. The first stage "picks" a pair of bounding boxes $j$ and $k$ to serve as "seeds" for the two resulting nodes, while the second stage redistributes the remaining bounding boxes into the nodes corresponding to $j$ and $k$. The redistribution process tries to minimize the "growth" of the area spanned by $j$ and $k$. Thus the first and second stages can be described as "seed-picking" and "seed-growing", respectively.

The first of these "seed-picking" algorithms is a quadratic cost algorithm [74] that initially finds the two bounding boxes that would waste the most area were they to be in the same node. This is determined by subtracting the sum of the areas of the two bounding boxes from the area of the covering bounding box. These two bounding boxes are placed in the separate nodes, say $j$ and $k$. Next, the remaining bounding boxes are examined, and for each bounding box, say $i$, $d_{ij}$ and $d_{ik}$ are computed, which correspond to the increases in the area of the covering bounding boxes of nodes $j$ and $k$, respectively, when $i$ is added to them. Now, the bounding box $r$ such that $|d_{rj} - d_{rk}|$ is a maximum is found, and $r$ is added to the node with the smallest increase in area. This process is repeated for the remaining bounding boxes. The motivation for selecting the maximum difference $|d_{rj} - d_{rk}|$ is to find the bounding box having the greatest preference for a particular node $j$ or $k$.

The second of of these "seed-picking" algorithms is a linear cost algorithm [74] that examines each dimension and finds the two bounding boxes with the greatest separation. Recalling that each bounding box has a low and a high edge along each axis, these two bounding boxes are the one whose high edge is the lowest along the given axis and the one whose low edge is the highest along the same axis. The separations are normalized by dividing the actual separation by the width of the bounding box of the overflowing node along the corresponding axis. The final "seeds" are the two bounding boxes having the greatest normalized separation among the $d$ pairs that we found. The remaining bounding boxes are processed in arbitrary order and placed in the node whose bounding box (i.e., of the entries added so far) is increased the least in area as a result of their addition. Empirical tests [74] showed that there was not much difference between the three node-splitting algorithms in the performance of a window search query (i.e., in CPU time and in the number of disk pages accessed). Thus, the faster linear cost node-splitting algorithm was found preferable for this query even though the quality of the splits was somewhat inferior.

An alternative node splitting policy is based on minimizing the overlap. One technique which has a linear cost [4] applies $d$ partitions (one for each of the $d$ dimensions) to the bounding boxes in the node $t$ being split thereby resulting in $2d$ sets of bounding boxes. In particular, we have one set for each face of the bounding box $b$ of $t$. The partition is based on associating each bounding box $o$ in $t$ with the set corresponding to the closest face along dimension $i$ of $b$[6]. Once the $2d$ partitions have been constructed (i.e., each bounding box $o$ has been associated with $d$ sets), select the partition that ensures the most even distribution of bounding boxes. In case of a tie, choose the partition with the least overlap. In case of another tie, choose the partition with the least coverage. For example, consider the four bounding boxes in Figure 18a. The partition along the $x$ axis yields the sets $\{1,2\}$ and $\{3,4\}$ (Figure 18b) while the partition along the $y$ axis yields the sets $\{1,3\}$ and $\{2,4\}$ (Figure 18c). Since both partitions yield sets that are evenly distributed, we choose the one that minimizes overlap (i.e., along the $y$ axis).

The algorithm is linear as it examines each bounding box once along each dimension (actually, it is $O(dM)$ for $M$ objects but $d$ is usually much smaller than $M$). Experiments with randomly generated rectangles [4] resulted in lower coverage and overlap than the linear and quadratic algorithms described above [74] that are based on minimizing the coverage. The window search query was also found to be about 16% faster with the linear algorithm based on minimizing overlap than the quadratic algorithm based on minimizing coverage. The drawback of this linear algorithm (i.e., [4]) is that it does not guarantee that the two nodes resulting from the partition will contain an equal number of bounding boxes. This is because the partitions are based on proximity to the borders of the bounding box of the node being split. In particular, when the data is not uniformly distributed, although the resulting nodes are likely to have little overlap (as they are likely to partition the underlying space into two equal areas), they will most likely contain an uneven number of bounding boxes.

[6]Formally, each bounding box $o$ has two faces $f_{oil}$ and $f_{oih}$ that are parallel to the respective faces $f_{bil}$ and $f_{bih}$ of $b$ where $l$ and $h$ correspond to the low and high values of coordinate or dimension $i$. For each dimension $i$, there are two sets $S_{il}$ and $S_{ih}$ corresponding to faces $f_{bil}$ and $f_{bih}$ of $b$, and the algorithm inserts $o$ into $S_{il}$ if $x_i(f_{oil}) - x_i(f_{bil}) < x_i(f_{bih}) - x_i(f_{oih})$ and into $S_{ih}$ otherwise where $x_i(f)$ is the $i^{th}$ coordinate value of face $f$.

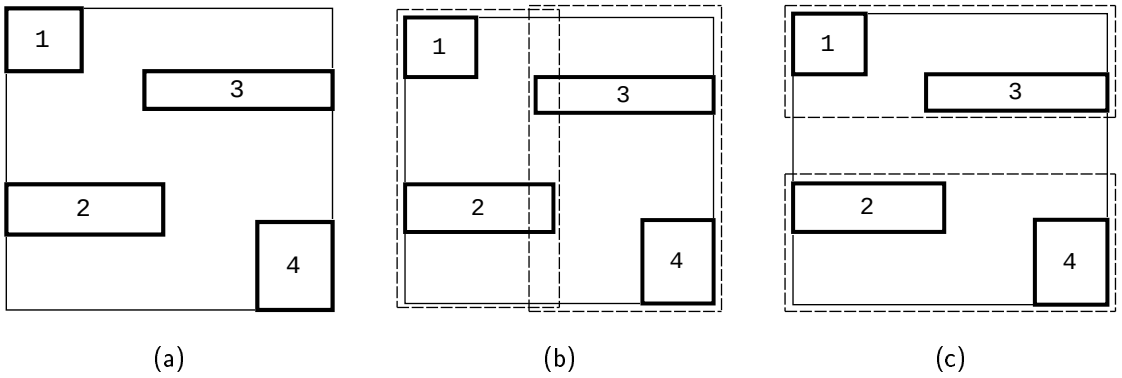


Figure 18: (a) Example collection of rectangles demonstrating a linear node splitting algorithm that ensures the most even distribution of bounding boxes, and the two possible splits caused by associating each bounding box with its closest face caused by a partition along the (b) x axis, and (c) y axis.

## 5.4 R*-tree

Better decompositions in terms of less node overlap and lower storage requirements than those achieved by the linear and quadratic node-splitting algorithms have also been reported in [19] where three significant changes have been made to the R-tree construction algorithm including a different node-splitting strategy. An R-tree that is built using these changes is termed an *R*-tree* [19][7]. These changes are described below. Interestingly, these changes also involve using a node splitting policy that, at times, tries to minimize both coverage and overlap.

The first change is the use of an intelligent object insertion procedure that is based on minimizing overlap in the case of leaf nodes, while minimizing the increase in area (i.e., coverage) in the case of nonleaf nodes. The distinction between leaf and nonleaf nodes is necessary as the insertion algorithm starts at the root and must process nonleaf nodes before encountering the leaf node where the object will ultimately be inserted. Thus we see that the bounding box $b$ for an object $o$ is inserted into the leaf node $p$ for whom the resulting bounding box has the minimum increase in the amount of overlap with the bounding boxes of $p$'s siblings (children of nonleaf node $s$). This is in contrast to the R-tree where $b$ is inserted into the leaf node $p$ for whom the increase in area is a minimum (i.e., based on minimizing coverage). This part of the R*-tree object insertion algorithm is quadratic in the number of entries in each node (i.e., $O(M^2)$ for an order $(m, M)$ R*-tree where the number of objects or bounding boxes that are aggregated in each node is permitted to range between $m \leq \lceil M/2 \rceil$) as the overlap must be checked for each leaf node child $p$ of the selected nonleaf node $s$ with all of $p$'s $O(M)$ siblings.

The second change is that when a node $p$ is found to overflow in an R*-tree, instead of immediately splitting $p$ as is done in the R-tree, first, an attempt is made to see if some of the objects in $p$ could possibly be more suited to being in another node. This is achieved by reinserting a fraction (30% has been found to yield good performance [19]) of these objects in the tree (termed *forced reinsertion*). Forced reinsertion is

[7] The '*' is used to signify its "star"-like performance [144] in comparison with R-trees built using the other node-splitting algorithms as can be seen in examples such as Figures 19 and 20.

similar in spirit to rotation (also known as "deferred splitting") in a conventional B-tree, which was also a technique developed to avoid splitting a node.

There are several ways of determining the objects to be reinserted. One suggestion is to sort the bounding boxes in $p$ according to the distance of the centers of their bounding boxes from the center of the bounding box of $p$, and to reinsert the designated fraction that are the farthest. Once we have determined the objects to be reinserted, we need to choose an order in which to reinsert them. There are two obvious choices: from farthest to closest (termed *far-reinsert*) or from closest to farthest (termed *close-reinsert*). Becker et al. [19] make a case for using close-reinsert on the basis of results of experiments. One possible explanation is that if the reinsertion procedure places the first object to be reinserted in $p$, then the size of the bounding box of $p$ is likely to be increased more if 'far-reinsert' was used rather than 'close-reinsert' thereby increasing the likelihood of the remaining objects being reinserted in $p$ as well. This has the effect of defeating the motivation for the introduction of the reinsertion process which is to try to reorganize the nodes. However, it could also be argued that using 'far-reinsert' is more likely to result in the farthest object being reinserted in a node other than $p$, on account of the smaller amount of overlap, which is one of the goals of the reinsertion process. Thus the question of which method to use is not completely settled.

The sorting step in forced reinsertion takes $O(M \log M)$ time. However, this cost is greatly overshadowed by the fact that each invocation of forced reinsertion can result in the reinsertion of $O(M)$ objects thereby increasing the cost of insertion by a factor of $O(M)$. One problem with forced reinsertion is that it could lead to overflow in the same node $p$ again when all of the bounding boxes are reinserted in $p$, or even to overflow in another node $q$ at the same depth. This could lead to an infinite loop. In order to prevent the occurrences of such a situation, forced reinsertion is applied only once at each depth for a given object. Note also that forced reinsertion is applied in a bottom-up manner in the sense that resolving overflow in the leaf nodes may also lead to overflow of the nonleaf nodes, in which case we apply forced reinsertion to the nonleaf nodes as well. When applying forced reinsertion to a nonleaf node $p$ at depth $l$, we reinsert only the elements in $p$ and at depth $l$.

Forced reinsertion is quite important as usually an R-tree is built by inserting the objects one by one as they are encountered in the input. Thus we don't usually have the luxury of processing the objects in sorted order. This could lead to some bad decompositions in the sense that the redistribution stage may prefer one of the "seed" nodes over the other in a consistent manner. Of course, this can be overcome by taking into account the bounding boxes of all of the objects before building the R-tree; but now the representation is no longer dynamic. Forced reinsertion is a compromise in the sense that it permits us to periodically rebuild part of the R-tree as a means of compensating for some bad node placement decisions.

The third change involves the manner in which an overflowing node $p$ is split. Again, as in the original R-tree node-splitting algorithm, a two-stage process is used. The difference is in the nature of the stages. The process follows closely an approach presented in an earlier study of the R-tree [69] which did not result in the coining of a new name for the data structure! In particular, in contrast to the original R-tree node-splitting strategy [74] where the first stage "picks" two "seeds" for the two resulting nodes which are subsequently "grown" by the second stage, in the R*-tree (as well as in the approach described in [69]), the first stage determines the axis (i.e., hyperplane) along which the split is to take place, while the second stage determines

the position of the split. In two dimensions, for example, the split position calculated in the second stage serves as the boundary separating the left (or an equivalent alternative is the right) sides of the bounding boxes of the objects that will be in the left and right nodes resulting from the split.

Note that the result of the calculation of the split position in the second stage has the same effect as the redistribution step in the linear and quadratic cost R-tree node-splitting algorithms as it indicates which bounding boxes are associated with which node. In particular, as we will see below, the first stage makes use of the result of sorting the faces of the bounding boxes along the various dimensions. Moreover, it would appear that the first and last bounding boxes in the sort sequence play a somewhat similar role to that of the "seeds" in the original R-tree node-splitting algorithms. However, this comparison is false as there is no "growing" process in the second stage. In particular, these "seeds" do not "grow" in an independent manner in the sense that the bounding boxes $b_i$ that will be assigned to their groups are determined by the relative positions of the corresponding faces of $b_i$ (e.g., in two dimensions, the sorted order of their left, right, top, or bottom sides).

This two-stage process is implemented by performing $2d$ sorts (two per axis) of the bounding boxes of the objects in the overflowing node $p$. For each axis $a$, the bounding boxes are sorted according to their two opposite faces that are perpendicular to $a$. The positions of the faces of the bounding boxes in the sorted lists serve as the candidate split positions for the individual axes. There are several ways of using this information to determine the split axis and split position along the axis.

Becker et al. [19] choose the split axis as the axis $a$ for which the average perimeter of the bounding boxes of the two resulting nodes for all of the possible splits along $a$ is the smallest while still satisfying the constraint posed by $m$ and $M$. An alternative approach (although not necessarily yielding the desired result as shown below) is one that chooses the split axis as the axis $a$ for which the perimeter of the two resulting nodes is a minimum. Basing the choice on the value of the perimeter is related to the goal of minimizing coverage by favoring splits that result in nodes whose bounding boxes have a square-like shape. Basing the choice of the split axis on the minimum average perimeter results in giving greater weight to the axis where the majority of the possible splits result in nodes whose bounding boxes have square-like shapes. This stage takes $O(dM \log M)$ time as the sort takes $O(M \log M)$ time for each axis while the average perimeter computation can be done in $O(M)$ time for each axis when scanning the faces of the bounding boxes in sorted order.

The position of the split along the axis $a$ selected by the first stage is calculated by examining the two sorted lists of possible split positions (i.e., faces of the bounding boxes) for $a$ and choosing the split position for which the amount of overlap between the bounding boxes of the two resulting nodes is the smallest while still satisfying the constraint posed by $m$ and $M$. Ties are resolved by choosing the position which minimizes the total area of the resulting bounding boxes thereby reducing the coverage. Minimizing the overlap reduces the likelihood that both nodes will be visited in subsequent searches. Thus we see that the R*-tree's node-splitting policy tries to address the issues of minimizing both coverage and overlap. Determining the split position requires $O(M)$ overlap computations when scanning the bounding boxes in sorted order. Algorithms that employ this sort-and-scan paradigm are known as plane-sweep techniques [15, 119, 147].

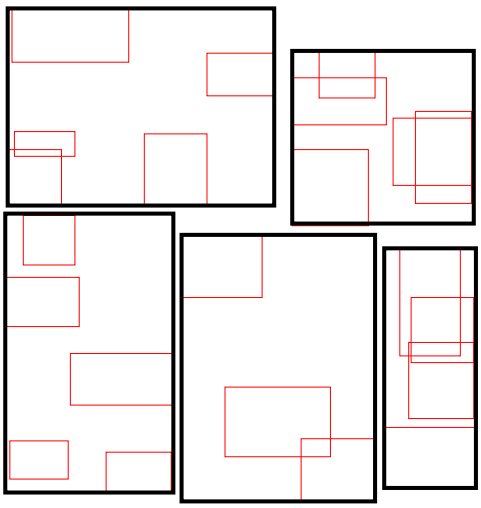

Figure 19: The bounding boxes corresponding to the first level of aggregation for an R*-tree for the collection of 22 rectangles in Figure 14.

Figure 19 shows the bounding boxes corresponding to the first level of aggregation for an R*-tree in comparison to that resulting from the use of an R-tree that deploys the exhaustive (Figure 20a), linear cost (Figure 20b), quadratic cost (Figure 20c), and the linear cost of [4] (Figure 20d) node-splitting algorithms for the collection of 22 rectangles in Figure 14. It is quite clear from the figure, at least for this example data set, that the combined criterion used by the R*-tree node-splitting algorithm that chooses the split which minimizes the sum of the perimeters of the bounding boxes of the two resulting nodes, as well as their overlap, seems to be working. Whether this is indeed the change in the definition that leads to this behavior is unknown.

Empirical studies have shown that use of the R*-tree node-splitting algorithm instead of the conventional linear and quadratic cost R-tree node-splitting algorithms leads to a reduction in the space requirements (i.e., improved storage utilization) ranging from 10 to 20% [19, 80] while requiring significantly more time to build the R*-tree [80]. The effect of the R*-tree node-splitting algorithms *vis-a-vis* the conventional linear and quadratic cost node-splitting algorithms on query execution time is not so clear due to the need to take factors such as paging activity, node occupancy, etc. into account [19, 80, 107].

Although the definition of the R*-tree makes three changes to the original R-tree definition [74], it can be argued that the main distinction, from a conceptual point of view rather than from its effect on performance, is in the way an overflowing node is split, and in the way the bounding boxes are redistributed in the two resulting nodes[8]. In particular, the original R-tree node splitting algorithms [74] determine "seeds" while the R*-tree algorithm determines a split axis and an axis split value. The bounding boxes of the objects are redistributed about these "seeds" and axis, respectively. At this point, it is important to re-emphasize that the motivation for these redistribution strategies is to avoid the exhaustive search solution which looks at all

[8]On the other hand, it could also be argued that forced reinsertion is the most important distinction as it has the ability to undo the effect of some insertions which may have caused undesired increases in overlap and coverage.

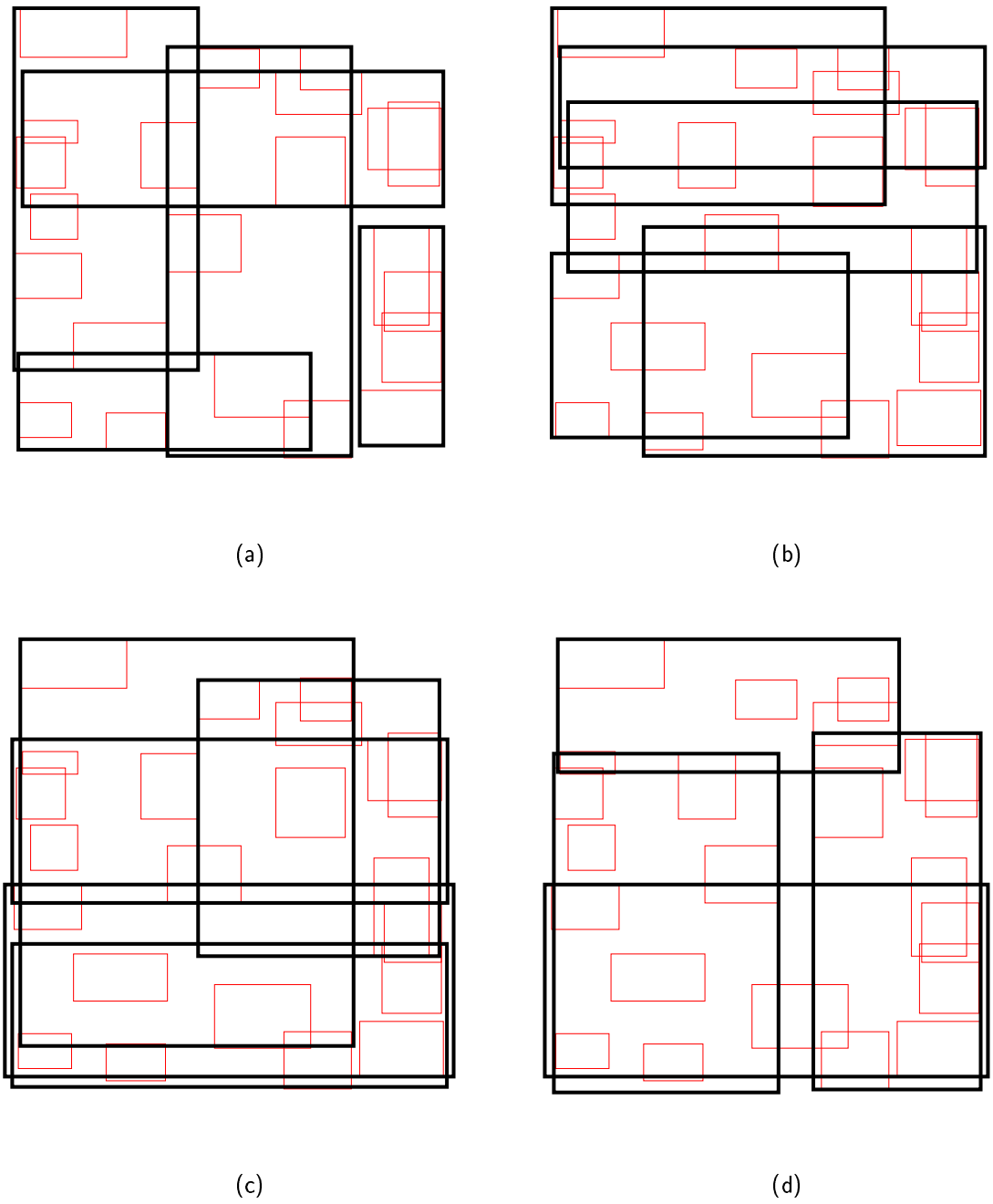


Figure 20: The bounding boxes corresponding to the first level of aggregation for an R-tree built using different node splitting policies: (a) exhaustive, (b) linear, (c) quadratic), and (d) the linear algorithm of [4] for the collection of 22 rectangles in Figure 14.

possible partitions.

The R*-tree redistribution method first sorts the boundaries of the bounding boxes along each of the axes and then uses this information to find the split axis $a$ (with respect to the minimum average perimeter of the bounding boxes of the resulting nodes) and split position (with respect to the minimal overlap once the

split axis was chosen). This is a heuristic that attempts to approximate the solution to the $d$-dimensional problem (i.e., optimal partitioning with minimal coverage or overlap) with an approximation of the optimal one-dimensional solution along one of the axes. Intuitively, the validity of this approximation would appear to decrease as $d$ (i.e., the dimensionality of the underlying space) increases since more and more splits are eliminated from consideration. However, the opposite conclusion might be true as it could be argued that although the number of eliminated splits grows exponentially with $d$, the majority of the eliminated splits are bad anyway. This is a problem for further study.

The remaining changes involving forced reinsertion and intelligent object insertion could have also been used in the R-tree construction algorithms. In particular, although the original R-tree definition [74] opts for minimizing coverage in determining the subtree in which an object is to be inserted, it does leave it open as to whether minimizing coverage or overlap is best. Similarly, using forced reinsertion does not change the R-tree definition. It can be applied regardless of how a node is split and which policy is used to determine the node into which the object is to be inserted. The evaluation of the R*-tree conducted in [19] involves all three of these changes. An evaluation of R-trees constructed using these remaining changes is also of interest.

The node splitting rules that form the basis of the R*-tree have also been used in conjunction with some of the methods for constructing instances of the object-tree pyramid such as the Hilbert packed R-tree. In particular, DeWitt et al. [40] suggest that it is not a good idea to fill each leaf node of the Hilbert packed R-tree to capacity. Instead, they pack each leaf node $i$, say up to 75% of capacity, and then for each additional object $x$ to be placed in $i$, they check if the bounding rectangle of $i$ needs to be enlarged by too much (e.g., more than 20% in area [40]) in order to contain $x$, in which case they start packing another node. In addition, whenever a node has been packed, the contents of a small number (e.g., 3 [40]) of the most recently created nodes are combined into a large node which is then resplit using the R$^*$-tree splitting methods. Although experiments show that these modifications lead to a slower construction time than that for the conventional Hilbert packed R-tree, the query performance is often improved (e.g., up to 20% in the experiments [40]).

## 5.5 Bulk Insertion and Bulk Loading

Up to now, our discussion of building the object-tree pyramid has been differentiated on the basis of whether it is done in a static or a dynamic environment. The static methods exemplified by the various packing methods such as the packed R-tree, Hilbert packed R-tree, and the STR method were primarily motivated by a desire to build the structure as fast as possible. This is in addition to the secondary considerations of maximizing storage utilization and possibly faster query performance as a result of a shallower structure since each node is filled to capacity thereby compensating for the fact that these methods may result in more coverage and overlap. The dynamic methods exemplified by the R-tree and the R*-tree were motivated equally by a desire to avoid rebuilding the structure as updates occur (primarily as objects are added and, to a lesser extent, deleted), and by a desire for faster query performance due to a reduction of coverage and overlap.

At times, it is desired to update an existing object-tree pyramid with a large number of objects at once. Performing these updates one object at a time using the implementations of the dynamic methods described above can be expensive. The CPU and I/O costs can be lowered by grouping the input objects prior to the

insertion. This technique is known as *bulk insertion*. It can also be used to build the object-tree pyramid from scratch in which case it is also known as *bulk loading*. In fact, we have seen several such techniques already in our presentation of the static methods which employ packing. The difference is that although the bulk loading methods that we discuss below are based on grouping the input objects prior to using one of the dynamic methods of constructing the object-tree pyramid, the grouping does not involve sorting the input objects which is a cornerstone of the bulk loading methods that employ packing. We discuss the bulk insertion methods first.

A simple bulk insertion idea is to sort all of the $m$ new objects to be inserted according to some order (e.g., Peano-Hilbert) and then insert them into an existing object-tree pyramid in this order [90]. This approach is used in the cubetree [127], a packed R-tree like structure for data warehousing and OLAP (denoting *online analytic processing* [35]) applications. The rationale for sorting the new objects is to have each new object be relatively close to the previously inserted object so that most of the time the nodes on the insertion path are likely to be the same, which is even more likely to be the case if some caching mechanism is employed. Thus, the total number of I/O operations is reduced. This technique works fine when the number of objects being inserted is small relative to the total number of objects. Also, it may be the best choice when the collection of new objects is spread over a relatively large portion of the underlying space, as in such cases the use of other methods (see below) may lead to excessive overlap (but see discussion of GBI [37] below). It can be used with any of the methods of building an object-tree pyramid.

Another related bulk insertion method, due to Kamel, Khalil, and Kouramajian [90], first orders the new objects being inserted according to the Peano-Hilbert order, and then aggregates them into leaf nodes where each node is filled to a predetermined percentage of the capacity (e.g., 70%) as if we are building just the leaf nodes of a Hilbert packed R-tree for the new objects. These leaf nodes are inserted into an object-tree pyramid in the order in which they were built.

The STLT (denoting *Small-Tree-Large-Tree*) method of Chen, Choubey, and Rundensteiner [36] can be viewed as a generalized variant of the method of Kamel, Khalil, and Kouramajian [90] in that instead of inserting the leaf nodes of the object-tree pyramid $T$ of the new data (which has been built using any construction algorithm) in the existing tree $E$, it just inserts the root of $T$ so that the leaf nodes of $T$ will be at the same depth as the leaf nodes of $E$. Although this method will lead to a faster insertion time than dynamic insertion, it will result in poorer query performance due to a significant overlap between the nodes in $T$ and $E$. In order to overcome this problem Choubey, Chen, and Rundensteiner [37] introduce a new method (termed *Generalized Bulk Insertion (GBI)*) that uses cluster analysis to divide the new data into clusters. Small clusters (e.g., containing just one point) are inserted using a regular dynamic insertion method, whereas for larger clusters, a tree is built and inserted using the STLT method. In other words, the STLT method is really a sub-component of the GBI method. This reduces the amount of overlap, which can be very high for the STLT method.

The bulk loading methods that we describe [11, 24] insert the individual objects using dynamic insertion methods. In particular, as we pointed out above, the objects are not preprocessed (e.g., via an explicit sorting step or aggregation into a distinct object-tree pyramid) prior to insertion as is the case for the bulk insertion methods. In particular, the sorting is deferred as much as possible although at the end of the bulk loading

process, the data is ordered on the basis of the underlying tree structure, and hence can be considered to be sorted. These bulk loading methods are general in that they are not just applicable to the R-tree; instead, they are applicable to most balanced tree data structures which resemble B-trees. They are based on the general concept of the buffer tree [10], wherein each internal node of the buffer tree contains a buffer of records stored on disk.

The basic idea behind the methods based on the buffer tree is that insertions into each nonleaf node of the buffer tree are batched. In particular, insertions occur into the buffer associated with the root node and slowly trickle down the tree as buffers are emptied when they are full. The buffers enable the effective use of available main memory, thereby resulting in large savings in I/O cost over the regular dynamic insertion method (although the CPU cost may be higher, in part, due to the large fanout when using one of the methods [24] as we point out below). Nevertheless, it could be the case that the actual execution could be slower in comparison to a non bulk-loading method in the case that many overflowing buffers need to be trickled down.

In the method proposed by van den Bercken, Seeger, and Widmayer [24], the R-tree is built recursively bottom-up. At each stage, an intermediate tree structure is built where the lowest level corresponds to the next level of the final R-tree. The nonleaf nodes in the intermediate tree structures have a high fanout (determined by available internal memory) as well as a buffer that receives insertions. Arge et al. [11] achieve a similar effect by using a regular R-tree structure (i.e., where the nonleaf nodes have the same fanout as the leaf nodes which is the size of a disk page) and only attaching buffers to nodes at certain levels of the tree. The advantages of the method of Arge et al. [11] over the method of van den Bercken, Seeger, and Widmayer [24] are that it is more efficient as it does not build intermediate structures, and it results in a better space partition. Moreover, the method of Arge et al. [11] yields the same R-tree as would have been obtained using conventional dynamic insertion methods without buffering (with the exception of the R*-tree where the use of forced re-insertion is difficult to incorporate in the buffering approach), while this is not the case for the method of van den Bercken, Seeger, and Widmayer [24]. In addition, the method of [11] supports bulk-insertions (as opposed to just initial bulk-loading as in [24]) and other bulk-queries including intermixed insertions and queries.

## 5.6 Shortcomings and Solutions

In this section we point out some of the shortcomings of the object-tree pyramid as well as point out some of the solutions. As we are dealing with the representations of objects, which are inherently of low dimension, we do not discuss the shortcomings and solutions for high-dimensional data (e.g., the X-tree [22] which attempts to address the problem arising when there is much overlap among the nodes corresponding to the partitions that result from a node split). One of the drawbacks of the object-tree pyramid (i.e., the R-tree as well as its variants such as the R*-tree) is that as the node size (i.e., page size — that is, $M$) gets large, the performance starts to degrade. This is somewhat surprising as according to conventional wisdom, performance should increase with node size as the depth of the tree decreases thereby requiring fewer fewer disk accesses. The problem is that as the node size increases, operations on each node take more CPU time. This

is especially true if the operation involves search (e.g., finding the nearest object to a point) as the bounding boxes in each node are not ordered [79].

This problem can be overcome by ordering the bounding boxes in each node using the same ordering-based aggregation techniques that were used to make the object-tree pyramid more efficient in responding to the location query. For example, we could order the bounding boxes by applying a Morton or Peano-Hilbert space ordering to their centroids. We term the result an *ordered R-tree*. Interestingly, the ordered R-tree can be viewed as a hybrid between an object $B^+$-tree and an R-tree in the sense that nodes are ordered internally (i.e., their constituent bounding box elements) using the ordering of the object $B^+$-tree, while they are ordered externally (i.e., vis-a-vis each other) using an R-tree.

Although the R-tree is height-balanced, the branching factor of each node is not the same. Recall that each node contains between $m$ and $M$ objects or bounding boxes. This has several drawbacks. First, it means that the nodes are not fully occupied thereby causing the tree structure to be deeper than it would be had the nodes been completely full. Therefore, the number of data objects in the leaf nodes of the descendants of sibling nonleaf nodes is not the same and, in fact, can vary quite greatly, thereby leading in imbalance in terms of the number of objects stored in different subtrees. This could have a detrimental effect on the efficiency of retrieval. Second, satisfying the branching factor condition often requires compromising the goal of minimizing total coverage, overlap, and perimeter. The packed R-tree and the Hilbert packed R-tree are some ways to overcome this problem as they initially have a branching factor of $M$ at all but the last node at each level. However, they are not necessarily designed to meet our goals of minimizing total coverage, overlap, and perimeter.

The S-tree [2] is an approach to overcome the above drawbacks of the R-tree and its packed variants by trading off the height-balanced property in return for reduced coverage, overlap, and perimeter in the resulting minimum bounding boxes. The S-tree has the property that each node that is not a leaf node or a penultimate node (i.e., a node whose children are all leaf nodes) has $M$ children. In addition, for any pair of sibling nodes (i.e., with the same parent) $s_1$ and $s_2$ with $N_{s_1}$ and $N_{s_2}$ objects in their descendants, respectively, we have that $p \leq N_{s_1}/N_{s_2} \leq 1/p$ $(0 < p \leq 0.5)$, where $p$, termed the *skew factor*, is a parameter that is related to the skewness of the data and governs the amount of tradeoff thereby providing a worst-case guarantee on the skewness of the descendants of the node. In particular, the number of objects in the descendants of each of a pair of sibling nodes is at least a fraction $p$ of the total number of objects in the descendants of both nodes. This guarantee is fairly tight when $p$ is close to 0.5, while it is quite loose when $p$ is small. In other words, when $p = 0.5$, the difference in the number of objects that will be found in the subtrees of a pair of sibling nodes is within a factor of 2, whereas this ratio can get arbitrarily large in a conventional R-tree.

The cost-based unbalanced R-tree (*CUR-tree*) of Ross, Sitzmann, and Stuckey [126] is another variant of an R-tree where the height-balanced requirement is relaxed in order to improve the performance of point and window queries in an environment where all the data is in main memory. The CUR-tree makes use of a cost model for the data structure (i.e., the R-tree) that accounts for operations such as reading the node and making the comparisons needed to continue the search. In particular, upon every insertion and deletion (rather than just upon overflow), every node on the insertion path is examined to determine if its entries should be rearranged to lower the cost function. The result is that nodes can be split, and their entries can be promoted

or demoted, at the cost of a slower update time.

García. and López, and Leutenegger [64] propose to improve the query performance of R-trees by restructuring the tree. Such restructuring can be performed after an R-tree has been built or dynamically as insertions take place. The key idea is to select a node *e* to be restructured and then apply the restructuring process to *e* and its ancestors by merging and resplitting sibling nodes. In the dynamic case, the restructuring is applied with some fixed probability for each insertion thereby ensuring that the restructuring does not happen at each insertion. Although at a first glance, restructuring seems similar to forced reinsertion in the case of an R*-tree (see Section 5.4), they are quite different upon closer scrutiny. In particular, forced reinsertion takes individual node entries and reinserts them at the root. In contrast, restructuring operates on groups of node entries by repeatedly merging and resplitting them, as necessary, in order to obtain better query performance through greater storage utilization and less overlap among sibling nodes.

A shortcoming of all of the representations that are based on object hierarchies (i.e., including all of the R-tree variants) is that when the objects are not hyperrectangles, use of the bounding box approximation of the object eliminates only some objects from consideration when responding to queries. In other words, the actual execution of many queries requires knowledge of the exact representation of the object (e.g., the location query). In fact, the execution of the query may be quite complex using this exact representation. At times, these queries may be executed more efficiently by decomposing the object further into smaller pieces such as triangles, trapezoids, convex polygons, etc. (e.g., [29, 98]). For example, the TR$^*$-tree [29, 142] is such a representation where each object in an R$^*$-tree is decomposed into a collection of trapezoids. The DR-tree [100] is a related approach where the minimum bounding box is recursively decomposed into minimum bounding boxes until the volume of each box is less than predefined fraction of the volume of the initial bounding box. The result of the decomposition process is represented as a binary tree which is stored separately from the hierarchy that contains the minimum bounding boxes of the objects and can be processed in memory once it has been loaded.

# 6 Disjoint Object-based Hierarchical Interior-based Representations (k-D-B-trees, R$^+$-trees, and Cell Trees)

In our descriptions of the object pyramid and the object-tree pyramid in Section 5 we observed that we may have to examine all of the bounding boxes at all levels when attempting to determine the identity of the object *o* that contains location *a* (i.e., the location query). This was caused by the fact that the bounding boxes corresponding to different nodes may overlap. The fact that each object is associated with only one node while being contained in possibly many bounding boxes (e.g., in Figure 17, rectangle 1 is contained in its entirety in R1, R2, R3, and R5) means that the location query may often require several nonleaf nodes to be visited before determining the object that contains *a*. This problem also arises in the R-tree as seen in the following example.

Suppose that we wish to determine the identity of the rectangle object(s), in the collection of rectangles given in Figure 17 that contains point Q at coordinate values (22,24). We first determine that Q is in R0.

Next, we find that Q can be in both or either of R1 or R2, and thus we must search both of their subtrees. Searching R1 first, we find that Q could be contained only in R3. Searching R3 does not lead to the rectangle that contains Q even though Q is in a portion of rectangle D that is in R3. Thus, we must search R2 and we find that Q can be contained only in R5. Searching R5 results in locating D, the desired rectangle. The drawback of the R-tree as well as other representations that make use of an object pyramid is that unlike those based on the cell pyramid, they do not result in a disjoint decomposition of space. Recall that the problem is that an object is associated with only one bounding bounding box (e.g., rectangle D in Figure17 is associated with bounding box R5, yet it overlaps bounding boxes R1, R2, R3, and R5). In the worst case, this means that when we wish to respond to the location query (e.g., given a point, determining the containing rectangle in a rectangle database, or an intersecting line in a line segment database, etc. in the two-dimensional space from which the objects are drawn), we may have to search the entire database. Thus what we need is a hierarchy of disjoint bounding boxes.

An obvious way to overcome this drawback is to use one of the hierarchical image-based representations described in Section 4. Recall that these representations made use of a hierarchy of disjoint cells that completely spanned the underlying space. The hierarchy consists of a set of sets $\{C_j\}$ $(0 \leq j \leq n)$ where $C_n$ corresponds to the original collection of cells, and $C_0$ corresponds to one cell. The sets differed in the number and size of the constituent cells at the different depths, although each set was usually a containment hierarchy in the sense that a cell at depth $i$ usually contained all of the cells below it at depth $i+1$. The irregular grid pyramid is an example of such a hierarchy.

A simple way to adapt the irregular grid pyramid to our problem is to overlay the decomposition induced by $C_{n-1}$ (i.e., the next to the deepest level) on the bounding boxes $\{b_i\}$ of the objects $\{o_i\}$ thereby decomposing the bounding boxes and associate each part of the bounding box with the corresponding covering cell of the irregular grid pyramid. Note that we use the set at the next to the deepest level (i.e., $C_{n-1}$) rather than the set at the deepest level (i.e., $C_n$) as the deepest level contains the original collection of unit-sized cells $c_{nk}$ and thus does not correspond to any aggregation. The cells $c_{jk}$ at the remaining levels $j$ $(0 \leq j \leq n-2)$ are formed in the same way as in the irregular grid pyramid — that is, they contain the union of the objects corresponding to the portions of the bounding boxes associated with the cells comprising cell $c_{jk}$. Using our terminology, we term the result an *irregular grid bounding-box pyramid*. It should be clear that the depth of the irregular grid bounding-box pyramid is one less than that of the corresponding irregular grid pyramid.

The definition of the irregular grid pyramid as well as the other hierarchical image-based representations stipulates that each unit-sized cell is contained in its entirety in one or more objects. Equivalently, a cell cannot be partially in object $o_1$ and partially in object $o_2$. The same restriction also holds for block decompositions which are not hierarchical (see Section 3). In contrast, in the case of the irregular grid bounding-box pyramid, the fact that the bounding boxes are just approximations of the objects enables us to relax this restriction in the sense that we allow a cell (or a block in the case of the block decompositions of Section 3) to contain parts of the bounding boxes of several objects. In other words, cell (or block) $b$ can be partially occupied by part of the bounding box $b_1$ of object $o_1$, by part of the bounding box $b_2$ of object $o_2$, and may even be partially empty.

The irregular grid bounding-box pyramid is a hierarchy of grids, albeit that the grid sizes are permitted

to vary in an arbitrary manner between levels. This definition is still overly restrictive in the sense that we want to be able to aggregate a varying, but bounded, number of cells at each level (in contrast to a predefined number) that depends on the number of bounding boxes or objects that are associated with them so that we can have a height-balanced dynamic structure in the spirit of the B-tree. We also wish to use a hierarchy that makes use of a different block decomposition rule (e.g., a k-d tree, generalized k-d tree, point quadtree, bintree, region quadtree, etc. thereby forming a variant of the cell-tree pyramid) instead of a grid as in the case of the irregular grid pyramid.

Our solution is equivalent to a marriage of the bounding-box cell-tree pyramid hierarchy with one of the block decompositions described in Section 3. This is done by choosing a value $M$ for the maximum number of cells (actually blocks) that can be aggregated and a block decomposition rule (e.g., a generalized k-d tree). As we are propagating the identities of the objects associated with the bounding boxes up the hierarchy rather than the space occupied by them, we use an object-based variant of the block decomposition rule. This means that a block is decomposed whenever it contains the bounding boxes of more than $M$ objects rather than on the basis of the absence of homogeneity. Note that the occupied space is implicit to the block decomposition rule and thus need not be explicitly propagated up the hierarchy.

It should be clear that each object's bounding box can appear only once in each block as the objects are continuous. If more than $M$ of the bounding boxes overlap each other in block $b$ (i.e., they all have at least one point in common), then there is no point in attempting to decompose $b$ further as we will never be able to find subblocks $b_i$ of $b$ so that each of $b_i$ does not have at least one point in common with the overlapping bounding boxes. Observe also that although the block decompositions yield a partition of space into disjoint blocks, the bounding boxes at the lowest level of the hierarchy may not necessarily be disjoint. For example, consider a database of line segment objects and the situation of a vertex where five of the line segments meet. It is impossible for the bounding boxes of the line segments to be disjoint.

The object-based variants of the block decomposition rules are quite different from their image-based counterparts that were discussed in Section 3 which based the decomposition on whether the space spanned by the block was completely covered by an object. It is important to reiterate that the blocks corresponding to the leaf nodes do not represent hyperrectangular aggregates of identically-valued unit-sized cells as in the conventional pyramid. Instead, they represent hyperrectangular aggregates of bounding boxes of objects or pieces thereof.

Without loss of generality, assuming a generalized k-d tree block decomposition rule, the hierarchy of sets $\{H_j\}$ $(1 \leq j \leq n)$ is defined as follows. $H_0$ consists of one block. $H_1$ consists of a subset of the nodes of a generalized k-d tree decomposition $Z$ of the underlying space so that $Z$ has a maximum of $M$ elements whose corresponding blocks span the entire underlying space. $H_2$ is formed by removing from $Z$ all nodes corresponding to members of $H_1$ and their ancestors, and then applying the same rule that was used to form $H_1$ to each of the blocks in $H_1$ with respect to $Z$. In other words, $H_2$ consists of generalized k-d tree decompositions of the blocks $h_{1k}$ $(1 \leq k \leq M)$ that comprise $H_1$. Each element of $H_2$ contains no more than $M$ blocks for a a maximum of $M^2$ blocks. This process is repeated at each successive level down to the leaf level at depth $n-1$. The nodes at the leaf level contain the bounding boxes of the objects or parts of the bounding boxes of the objects. The pyramid means that the hierarchy must be height-balanced with all leaf

nodes at the same level, and that the cells at depth $j$ are disjoint and that they span the space covered by the cells at the immediately lower level at depth $j+1$.

We term the resulting data structure a *generalized k-d tree bounding-box cell-tree pyramid* on account of the use of the generalized k-d tree as the building block of the pyramid and the use of a tree access structure, although it is more commonly known as a *k-d-B-tree* [122] on account of the similarity of the node structure to that of a B-tree. If we would have used the point quadtree or the bintree as the building blocks of the hierarchy, then we would have termed the result a *point quadtree bounding-box cell-tree pyramid* or a *bintree bounding-box cell-tree pyramid*, respectively. It is interesting to note that the k-d-B-tree was originally developed for storing point-like objects although the extension to objects with extent is relatively straightforward as shown here.

Figure 21 is an example of one possible k-d-B-tree for the collection of 9 rectangle objects given in Figure 11. Broken lines denote the leaf nodes, and thin lines denote the space spanned by the subtrees rooted at the nonleaf nodes. Of course, other variations are possible since the k-d-tree is not unique. This particular tree is of order (2,3) (i.e., having a minimum and maximum of 2 and 3 entries, respectively) although in general it is not possible to always guarantee that all nodes will have a minimum of 2 entries, nor is the minimum a part of the definition of the k-d-B-tree. Notice that rectangle object D appears in three different nodes, while rectangle objects A, B, E, and G appear in two different nodes. Observe also that the example uses a partition scheme that cycles through the axes in the order $x$, $y$, $x$, $y$, etc. although, as we shall see below, this cycling is not guaranteed to hold once objects are inserted and deleted.

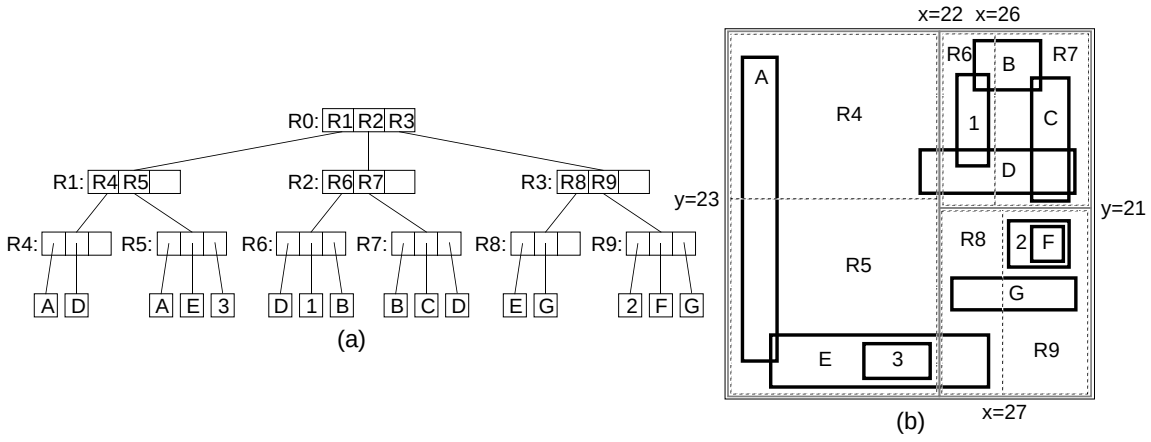


Figure 21: (a) k-d-B-tree for the same collection of rectangle objects given in Figure 11 with m=2 and M=3, and (b) the spatial extents of the nodes. Notice that only in the leaf nodes are the bounding boxes minimal.

Our definition of the structure was given in a top-down manner. In fact, the structure is often built in a bottom-up manner by inserting the objects one-at-a-time. Initially, the hierarchy contains just one node corresponding to the bounding box of the single object. As each additional object $o$ is processed, we insert $o$'s bounding box $b$ into all of the leaf nodes which overlap it. If any of these nodes become too full, then we split these nodes using an appropriate block decomposition rule and determine if the parent is not too full so that it can support the addition of a child. If not, then we recursively apply the same decomposition rule to

the parent. The process stops at the root in which case overflow will usually cause the hierarchy to grow by one level.

Variants of the bounding-box cell-tree pyramid such as the k-d-B-tree are good for answering both the feature and location queries. However, in the case of the location query, they act only as a partition of space. They do not distinguish between occupied and unoccupied space. Thus in order to determine if a particular location $a$ is occupied by one of the objects associated with cell $c$, we need to check each of the objects associated with $c$, which can be time-consuming especially if $M$ is large. We can speed this process by modifying the general definition of the bounding-box cell-tree pyramid so that a bounding box is stored in each node $r$ in the hierarchy, regardless of $r$'s depth, that covers the bounding boxes of the cells that comprise $r$. Thus associated with each node $r$ is the union of the objects associated with the cells comprising $r$ as well as a bounding box of the union of their bounding boxes. We term the result a *disjoint object pyramid*. Recall that the depth of any variant of the bounding-box pyramid is one less than that of the corresponding conventional pyramid, and the same is true for the disjoint object pyramid.

A key difference between the disjoint object pyramid and variants of the conventional pyramid, and to a lesser extent the bounding-box pyramid, is that the elements of the hierarchy of the disjoint object pyramid are also parts of the bounding boxes of the objects rather than just the cells that make up the objects which is the case for both variants of the bounding-box and conventional pyramids. The representation of the disjoint object pyramid, as well as variants of the bounding-box pyramid such as the k-d-B-tree, is also much simpler as they both just decompose the objects until a a criterion involving the number of objects that are present in the block is satisfied rather than one based on the homogeneity of the block. This results in avoiding some of the deeper levels of the hierarchy that are needed in variants of the conventional pyramid.

There are many variants of the disjoint object pyramid. They differ according to which of the block decomposition rules described in Section 3 is used. They are usually referred to by the general term $R^+$-*tree*[53, 145, 152] on account of the similarity to the R-tree since they both store a hierarchy of bounding boxes. However, the block decomposition rule is usually left unspecified although a generalized k-d tree block decomposition rule is often suggested. An alternative is not to use any decomposition rule in which case each node is just a collection of blocks as in Figure 4.

$R^+$-trees are built in the same incremental manner as any of the bounding-box cell-tree pyramids that we described (e.g., the k-d-B-tree, etc.). Again, as each additional object $o$ is processed, we insert $o$'s bounding box $b$ into all of the leaf nodes which overlap it. If any of these nodes become too full, then we split these nodes using the appropriate block decomposition rule and determine if the parent is not too full so that it can support the addition of a child. If not, then we recursively apply the same decomposition rule to the parent. The process stops at the root in which case the $R^+$-tree may grow by one level. The difference from the method used in the bounding-box cell-tree pyramids is that we also propagate the minimum bounding box information up the hierarchy. The entire process is analogous to that used in a B-tree upon overflow. The difference is that at times, as is also the case for the k-d-B-tree, the decomposition at a nonleaf node may result in the introduction of a new partition that may force the repartitioning of nodes at deeper levels in the $R^+$-tree.

Figure 22 is an example of one possible $R^+$-tree for the same collection of 9 rectangles given in Figure 11. Broken lines denote the bounding boxes corresponding to the leaf nodes, and thin lines denote the bounding boxes corresponding to the subtrees rooted at the nonleaf nodes. In this case, we simply took the k-d-B-tree of Figure 21 and added bounding boxes to the nonleaf nodes. This particular tree is of order (2,3) although in general it is not possible to always guarantee that all nodes will have a minimum of 2 entries. Notice that rectangle D appears in three different nodes, while rectangles A. B, E, and G appear in three different nodes. Of course, other variants are possible since the $R^+$-tree is not unique.

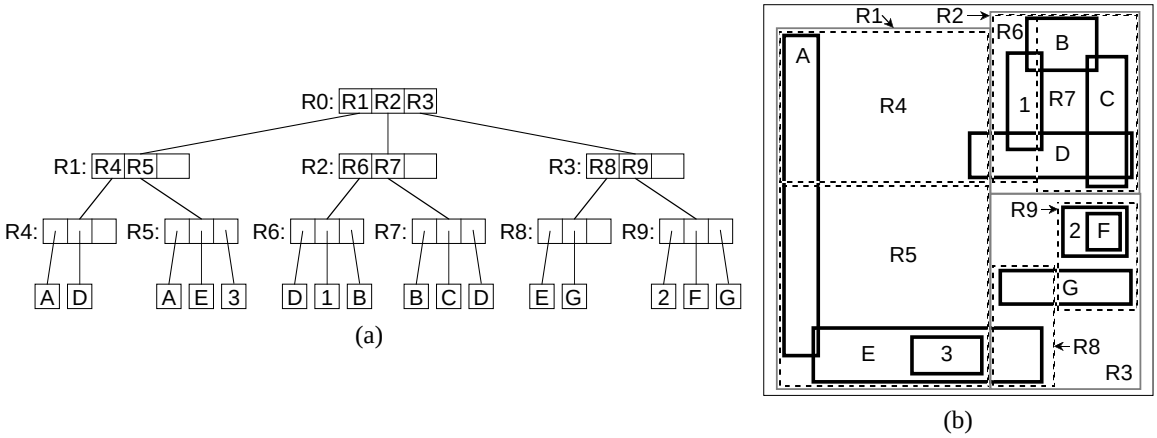


Figure 22: (a) $R^+$-tree for the same collection of rectangle objects given in Figure 11 with m=2 and M=3, and (b) the spatial extents of the bounding boxes. Notice that the leaf nodes in the index also store bounding boxes although this is shown only for the nonleaf nodes.

The *cell tree* of Günther [71, 72] is similar to the $R^+$-tree. The difference is that the nonleaf nodes of the cell tree are convex polyhedra instead of bounding rectangles. The children of each node, say *P*, form a binary space partition (BSP) [60] of *P*. The cell tree is designed to deal with polyhedral data of arbitrary dimension. As in the $R^+$-tree, the polyhedral data that is being represented may be stored in more than one node. When the decomposition causes an object to be split too many times, Günther and Noltemeier [73] store the object in what they term *oversized shelves* which are associated with nonleaf nodes in the structure. This is somewhat similar in spirit to the X-tree [22] which is a variant of an R-tree where a node is not split upon overflow if too much overlap would result among its children (in which case, the node is termed a *supernode*).

# 7 Concluding Remarks

We have reviewed a number of hierarchical image and object representations with a focus on hierarchical methods whose use enables us to answer the fundamental queries of what and where. As we have seen, there are two key classes of methods, and we distinguished between them on the basis of whether they were image-based or object-based. To get the most power in terms of the queries that can be handled, we can either use object hierarchies, which employ aggregates of bounding boxes, or space hierarchies, which employ a disjoint decomposition of the underlying space that is spanned by the objects. Neither method can be considered as

being the best. Each method has its advantages and disadvantages.

The drawback of object hierarchies (of which the R-tree is the most commonly used example) is that they do not yield a disjoint decomposition of the underlying space. This leads to the multiple coverage problem in the sense that the area containing a particular location *a* in object *o* may be spanned by several R-tree nodes while *o* is contained in just one R-tree node. Thus just because object *o* was not found in the search of one path in the tree whose bounding box contains *a*, does not mean that *o* would not be found in the search of another path in the tree. This makes search in an R-tree somewhat inefficient as in the worst case we may have to examine all of the bounding boxes at all levels of the hierarchy when attempting to determine the identity of the object *o* that contains location *a*.

The conventional alternative solution is to make use of a disjoint decomposition of the underlying space as provided by many structures such as the $R^+$-tree and variants of the quadtree/pyramid. These solutions do not suffer from the multiple coverage problem. However, their drawback is that an object *o* may need to be decomposed into several pieces and hence reported as satisfying the query several times as the area spanned by *o* may be contained in several blocks. For example, suppose that we want to retrieve all the objects that overlap a particular region (i.e., a window query) rather than a point. In this case, we could report the same object as many times as it has been decomposed into blocks. We can avoid reporting the object several times when using these methods by removing the duplicate objects before reporting the final answer. Removing the duplicate objects usually requires invocation of some variant of a sorting algorithm. Interestingly, there has been some work in developing algorithms for certain classes of objects and different data structures which are based on a disjoint decomposition that avoid reporting duplicate objects (e.g., [7, 9, 43]) without resorting to sorting.

The BV-tree [58] is an alternative solution that makes use of an object hierarchy similar to that of the R-tree and a more restricted form of a containment hierarchy where any pair of bounding boxes of two children *a* and *b* of node *r* must be either disjoint or one child is completely contained in the other child (i.e., *a* is in *b* or *b* is in *a*). At a first glance the BV-tree would appear to also be afflicted by the multiple coverage problem. However, the BV-tree overcomes the multiple coverage problem by making use of the concept of a guard which is carried along during the search process as the tree is descended thereby ensuring that only one path is followed in any search. The drawback of the BV-tree is that it is not balanced although the depth is bounded based on the maximum number of data points or objects. It is worth noting that the key idea in the BV-tree is the decoupling of the decomposition hierarchy from the directory hierarchy (i.e., the manner in which the various nodes are aggregated) [136]. The PK-tree [161] applies similar ideas to image hierarchies, with the same drawback of possibly being unbalanced.

We now point out a few more considerations which should be taken into account. Object-based methods such as the R-tree and the $R^+$-tree have the advantage of being able to distinguish between occupied and unoccupied space for a particular data set. However, they cannot correlate occupied space in two different data sets. In other words, the bounding boxes of the two data sets are not in registration which means that more intersection operations must be performed between the two sets when executing operations such as a spatial join if no preprocessing sorting step has been applied (e.g., [12, 97, 117]), although a number of good algorithms have been devised for spatial joins for object-based representations (e.g., [103, 104]). In contrast,

disjoint image-based methods that make use of a regular decomposition of the underlying space such as the region quadtree and the pyramid are good when operating on two different data sets as the occupied space in the two sets is correlated thereby simplifying the spatial join algorithms which makes them preferable to disjoint image-based methods that do not employ regular decomposition such as the $R^+$-tree (e.g., [80]). Nevertheless, there is the cost of dealing with duplicate answers (as mentioned above) which is incurred regardless of which disjoint method is used. Thus there is no one best or optimal representation. Ultimately, users make their decision on the basis of what is important to them, possibly making use of cost models (e.g., [8, 158]).

## Acknowledgements

I thank Houman Alborzi, Frantisek Brabec, and Gisli R. Hjaltason for generating the screenshots of the results of the applets and drawing the figures, and Edwin Jacox for help in the final preparation for publication. I would also like to thank the anonymous reviewers for their detailed comments which greatly sharpened the presentation of the ideas in this paper.

## References

[1] D. J. Abel and J. L. Smith. A data structure and algorithm based on a linear key for a rectangle retrieval problem. *Computer Vision, Graphics, and Image Processing*, 24(1):1–13, October 1983.

[2] C. Aggarwal, J. Wolf, P. Yu, and M. Epelman. The *S*-tree: an efficient index for multidimensional objects. In *Advances in Spatial Databases — 5th International Symposium, SSD'97*, M. Scholl and A. Voisard, eds., pages 350–373, Berlin, Germany, July 1997. Also Springer-Verlag Lecture Notes in Computer Science 1262.

[3] H. Alborzi and H. Samet. Execution time analysis of a top-down r-tree construction algorithm, 2004. Not published.

[4] C. H. Ang and T. C. Tan. New linear node splitting algorithm for R-trees. In *Advances in Spatial Databases — 5th International Symposium, SSD'97*, M. Scholl and A. Voisard, eds., pages 339–349, Berlin, Germany, July 1997. Also Springer-Verlag Lecture Notes in Computer Science 1262.

[5] W. G. Aref and I. F. Ilyas. Sp-gist: an extensible database index for supporting space partitioning trees. *Journal of Intelligent Information Systems*, 17(2–3):215–240, December 2001.

[6] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 265–272, Nashville, TN, April 1990. Also *Proceedings of the Fifth Brazilian Symposium on Databases*, pages 15–26, Rio de Janeiro, Brazil, April 1990.

[7] W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, pages 178–189, Charleston, SC, August 1992.

[8] W. G. Aref and H. Samet. A cost model for query optimization using r-trees. In *Proceedings of the 2nd ACM Workshop on Geographic Information Systems*, N. Pissinou and K. Makki, eds., pages 60–67, Gaithersburg, MD, December 1994.

[9] W. G. Aref and H. Samet. Hashing by proximity to process duplicates in spatial databases. In *Proceedings of the 3rd International Conference on Information and Knowledge Management (CIKM)*, pages 347–354, Gaithersburg, MD, December 1994.

[10] L. Arge. *Efficient external-memory data structures and applications*. PhD thesis, Computer Science Department, University of Aarhus, Aarhus, Denmark, 1996. Also BRICS Dissertation Series, DS-96-3.

[11] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic R-trees. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation (ALENEX'99)*, M. T. Goodrich and C. C. McGeoch, eds., pages 328–348, Baltimore, MD, January 1999. Also Springer-Verlag Lecture Notes in Computer Science 1619.

[12] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. S. Vitter. Scalable sweeping-based spatial join. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, A. Gupta, O. Shmueli, and J. Widom, eds., pages 570–581, New York, August 1998.

[13] R. Arman and J. K. Aggarwal. Model-based object recognition in dense range images — a review. *ACM Computing Surveys*, 25(1):5–43, March 1993.

[14] T. Asano, D. Ranjan, T. Roos, E. Welzl, and P. Widmayer. Space-filling curves and their use in the design of geometric data structures. *Theoretical Computer Science*, 181(1):3–15, July 1997.

[15] H. S. Baird. Design of a family of algorithms for large scale integrated circuit mask artwork analysis. Technical Report PRRL-76-TR-062, RCA Laboratories, Princeton, NJ, May 1976.

[16] D. H. Ballard. 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):213, March 1982.

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

[18] B. Becker, P. G. Franciosa, S. Gschwind, T. Ohler, G. Thiemt, and P. Widmayer. Enclosing many boxes by an optimal pair of boxes. In *Proceedings of the 9th Annual Symposium on Theoretical Aspects of Computer Science (STACS)*, A. Finkel and M. Jantzen, eds., pages 475–486, ENS Cachan, France, February 1992. Also Springer-Verlag Lecture Notes in Computer Science 577.

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

[20] S. B. M. Bell, B. M. Diaz, F. Holroyd, and M. J. Jackson. Spatially referenced methods of processing raster and vector data. *Image and Vision Computing*, 1(4):211–220, November 1983.

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

[22] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-tree: An index structure for high-dimensional data. In *Proceedings of the 22nd International Conference on Very Large Data Bases (VLDB)*, T. M. Vijayaraman, A. P. Buchmann, C. Mohan, and N. L. Sarda, eds., pages 28–39, Mumbai (Bombay), India, September 1996.

[23] J. van den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger. XXL - a library approach to supporting efficient implementations of advanced database queries. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB)*, P. M. G. Apers, P. Atzeni, S. Ceri, S. Paraboschi, K. Ramamohanarao, and R. T. Snodgrass, eds., pages 39–48, Roma, Italy, September 2001.

[24] J. van den Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds., pages 406–415, Athens, Greece, August 1997.

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

[26] C. Böhm, S. Berchtold, and D. A. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, September 2001.

[27] F. Brabec and H. Samet. Visualizing and animating R-trees and spatial operations in spatial databases on the worldwide web. In *Visual Database Systems (VDB4)*, Y. Ioannidis and W. Klas, eds., pages 123–140, Chapman and Hall, London, United Kingdom, May 1998. Also *Proceedings of the IFIP TC2//WG2.6 Working Conference on Visual Database Systems 4 (VDB4)*, L'Aquila, Italy, May 1998.

[28] F. Brabec, H. Samet, and C. Yilmaz. VASCO: visualizing and animating spatial constructs and operations. In *Proceedings of the 19th Annual Symposium on Computational Geometry*, pages 374–375, San Diego, CA, June 2003.

[29] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proceedings of the ACM SIGMOD Conference*, pages 197–208, Minneapolis, MN, June 1994.

[30] A. Brodsky, C. Lassez, J. Lassez, and M. J. Maher. Separability of polyhedra for optimal filtering of spatial and constraint data. In *Proceedings of the 14th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 54–65, San Jose, CA, May 1995.

[31] P. J. Burt. Tree and pyramid structures for coding hexagonally sampled binary images. *Computer Graphics and Image Processing*, 14(3):271–280, November 1980.

[32] F. W. Burton, J. G. Kollias, and N. A. Alexandridis. Implementation of the exponential pyramid data structure with application to determination of symmetries in pictures. *Computer Vision, Graphics, and Image Processing*, 25(2):218–225, February 1984.

[33] K. Chakrabarti and S. Mehrotra. High dimensional feature indexing using hybrid trees. Technical Report TR-MARS-98-14, Department of Information and Computer Science, University of California, Irvine, CA, July 1998.

[34] K. Chakrabarti and S. Mehrotra. The hybrid tree: an index structure for high dimensional feature spaces. In *Proceedings of the 15th IEEE International Conference on Data Engineering*, pages 440–447, Sydney, Australia, March 1999.

[35] S. Chaudhuri and U. Dayal. Data warehousing and olap technology. *SIGMOD RECORD*, 26(1):65–74, March 1997.

[36] L. Chen, R. Choubey, and E. A. Rundensteiner. Bulk-insertions into R-trees using the Small-Tree-Large-Tree approach. In *Proceedings of the 6th International Symposium on Advances in Geographic Information Systems*, R. Laurini, K. Makki, and N. Pissinou, eds., pages 161–162, Washington, DC, November 1998.

[37] R. Choubey, L. Chen, and E A. Rundensteiner. GBI: A generalized R-tree bulk-insertion strategy. In *Advances in Spatial Databases — 6th International Symposium, SSD'99*, R. H. Güting, D. Papadias, and F. H. Lochovsky, eds., pages 91–108, Hong Kong, July 1999. Also Springer-Verlag Lecture Notes in Computer Science 1651.

[38] R. A. DeMillo, S. C. Eisenstat, and R. J. Lipton. Preserving average proximity in arrays. *Communications of the ACM*, 21(3):228–231, March 1978.

[39] A. Dengel. Self-adapting structuring and representation of space. Technical Report RR-91-22, Deutsches Forschungszentrum für Künstliche Intelligenz, Kaiserslautern, Germany, September 1991.

[40] D. J. DeWitt, N. Kabra, J. Luo, J. M. Patel, and J. B. Yu. Client-server paradise. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, J. Bocca, M. Jarke, and C. Zaniolo, eds., pages 558–569, Santiago, Chile, September 1994.

[41] A. Di Pasquale, L. Forlizzi, C. S. Jensen, Y. Manolopoulos, E. Nardelli, D. Pfoser, G. Proietti, S. Saltenis, Y. Theodoridis, T. Tzouramanis, and M. Vassilakopoulos. Acces methods and query processing techniques. In *Spatiotemporal Databases: The CHOROCHRONOS Approach*, Manolis Koubarakis, Timos K. Sellis, Andrew U. Frank, Stéphane Grumbach, Ralf Hartmut Güting, Christian S. Jensen, Nikos A. Lorentzos, Yannis Manolopoulos, Enrico Nardelli, Barbara Pernici, Hans-Jörg Schek, Michel Scholl, Babis Theodoulidis, and Nectaria Tryfona, eds., chapter 6, pages 203–261. Berlin, Germany, 2003. Also Springer-Verlag Lecture Notes in Computer Science 2520.

[42] M. B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253–280, April 1992. Also Corrigendum, *Journal of the ACM*, 39(4):985, October 1992; University of Maryland Computer Science TR-2303.

[43] J.-P. Dittrich and B. Seeger. Data redundancy and duplicate detection in spatial join processing. In *Proceedings of the 16th IEEE International Conference on Data Engineering*, pages 535–546, San Diego, CA, February 2000.

[44] D. Dori and M. Ben-Bassat. Circumscribing a convex polygon by a polygon of fewer sides with minimal area addition. *Computer Vision, Graphics, and Image Processing*, 24(2):131–159, November 1983.

[45] D. H. Douglas. It makes me so CROSS. In *Introductory Readings in Geographic Information Systems*, D. J. Peuquet and D. F. Marble, eds., chapter 21, pages 303–307. Taylor & Francis, London, United Kingdom, 1990. Originally published as Internal Memorandum, Harvard University Laboratory for Computer Graphics and Spatial Analysis, 1974.

[46] D. H. Douglas and T. K. Peucker. Algorithms for the reduction of the number of points required to represent a digitized line or its caricature. *The Canadian Cartographer*, 10(2):112–122, December 1973.

[47] C. R. Dyer. A VLSI pyramid machine for hierarchical parallel image processing. In *Proceedings of the IEEE Conference on Pattern Recognition and Image Processing'81*, pages 381–386, Dallas, August 1981.

[48] C. R. Dyer. Pyramid algorithms and machines. *Multicomputers and Image Processing Algorithms and Programs*, pages 409–420, 1982.

[49] C. Esperanca and H. Samet. Representing orthogonal multidimensional objects by vertex lists. In *Aspects of Visual Form Processing*, C. Arcelli, L. P. Cordella, and G. Sanniti di Baja, eds., pages 209–220, World Scientific, Singapore, 1994.

[50] C. Esperança. *Orthogonal objects and their application in spatial databases*. PhD thesis, Computer Science Department, University of Maryland, College Park, MD, December 1995. Also Computer Science TR-3566.

[51] C. Esperança and H. Samet. Orthogonal polygons as bounding structures in filter-refine query processing strategies. In *Advances in Spatial Databases — 5th International Symposium, SSD'97*, M. Scholl and A. Voisard, eds., pages 197–220, Berlin, Germany, July 1997. Also Springer-Verlag Lecture Notes in Computer Science 1262.

[52] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the 8th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 247–252, Philadelphia, March 1989.

[53] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proceedings of the ACM SIGMOD Conference*, pages 426–439, San Francisco, May 1987.

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

[55] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley, Reading, MA, second edition, 1990.

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

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

[58] M. Freeston. A general solution of the n-dimensional B-tree problem. In *Proceedings of the ACM SIGMOD Conference*, pages 80–91, San Jose, CA, May 1995.

[59] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

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

[61] V. Gaede. Optimal redundancy in spatial database systems. In *Advances in Spatial Databases — 4th International Symposium, SSD'95*, M. J. Egenhofer and J. R. Herring, eds., pages 96–116, Portland, ME, August 1995. Also Springer-Verlag Lecture Notes in Computer Science 951.

[62] Y. J. García, M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading R-trees. In *Proceedings of the 6th International Symposium on Advances in Geographic Information Systems*, R. Laurini, K. Makki, and N. Pissinou, eds., pages 163–164, Washington, DC, November 1998. Also extended version University of Denver, Computer Science Department Technical Report 97-02.

[63] Y. J. García, M. A. López, and S. T. Leutenegger. On optimal node splitting for R-trees. In *Proceedings of the 24th International Conference on Very Large Data Bases (VLDB)*, A. Gupta, O. Shmueli, and J. Widom, eds., pages 334–344, New York, August 1998.

[64] Y. J. Garcia, M. A. Lopez, and S. T. Leutenegger. Post-optimization and incremental refinement of r-trees. In *Proceedings of the 7th International Symposium on Advances in Geographic Information Systems*, Claudia Bauzer Medeiros, ed., pages 91–96, Kansas City, MO, November 1999.

[65] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, December 1982.

[66] D. M. Gavrila. R-tree index optimization. In *Proceedings of the 6th International Symposium on Spatial Data Handling*, T. C. Waugh and R. G. Healey, eds., pages 771–791, Edinburgh, Scotland, September 1994. International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information. Also University of Maryland Computer Science TR-3292.

[67] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A hierarchical structure for rapid interference detection. In *Proceedings of the SIGGRAPH'96 Conference*, pages 171–180, New Orleans, August 1996.

[68] F. Gray. Pulse code communication. United States Patent Number 2632058, March 17, 1953.

[69] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings of the 5th IEEE International Conference on Data Engineering*, pages 606–615, Los Angeles, February 1989.

[70] L. J. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transactions on Graphics*, 4(2):74–123, April 1985. Also *Proceedings of the 15th Annual ACM Symposium on the Theory of Computing*, pages 221–234, Boston, April 1983.

[71] O. Günther. *Efficient structures for geometric data management*. PhD thesis, University of California at Berkeley, Berkeley, CA, 1987. Also Lecture Notes in Computer Science 337, Springer-Verlag, Berlin, West Germany, 1988; UCB/ERL M87/77.

[72] O. Günther and J. Bilmes. Tree-based access methods for spatial databases: implementation and performance evaluation. *IEEE Transactions on Knowledge and Data Engineering*, 3(3):342–356, September 1991. Also Department of Computer Science TRCS88-23, University of California at Santa Barbara, Santa Barbara, CA.

[73] O. Günther and H. Noltemeier. Spatial database indices for large extended objects. In *Proceedings of the 7th IEEE International Conference on Data Engineering*, pages 520–526, Kobe, Japan, April 1991.

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

[75] J. M. Hellerstein, J. F. Naughton, and A. Pfeffer. Generalized search trees for database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, U. Dayal, P. M. D. Gray, and S. Nishio, eds., pages 562–573, Zurich, Switzerland, September 1995. `http://gist.cs.berkeley.edu/`.

[76] A. Henrich. The $LSD^h$-tree: An access structure for feature vectors. In *Proceedings of the 14th IEEE International Conference on Data Engineering*, pages 362–369, Orlando, FL, February 1998.

[77] A. Henrich, H. W. Six, and P. Widmayer. The LSD tree: spatial access to multidimensional point and non-point data. In *Proceedings of the 15th International Conference on Very Large Databases (VLDB)*, P. M. G. Apers and G. Wiederhold, eds., pages 45–53, Amsterdam, The Netherlands, August 1989.

[78] D. Hilbert. Ueber stetige abbildung einer linie auf flächenstück. *Mathematische Annalen*, 38:459–460, 1891.

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

[80] E. G. Hoel and H. Samet. Benchmarking spatial join operations with spatial output. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, U. Dayal, P. M. D. Gray, and S. Nishio, eds., pages 606–618, Zurich, Switzerland, September 1995.

[81] P. M. Hubbard. Collision detection for interactive computer graphics applications. *IEEE Transactions on Visualization and Computer Graphics*, 1(3):218–230, September 1995.

[82] P. M. Hubbard. Approximating polyhedra with spheres for time-critical collision detection. *ACM Transactions on Graphics*, 15(3):179–210, July 1996.

[83] G. M. Hunter. *Efficient computation and data structures for graphics*. PhD thesis, Department of Electrical Engineering and Computer Science, Princeton University, Princeton, NJ, 1978.

[84] T. Ichikawa. A pyramidal representation of images and its feature extraction facility. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(3):257–264, May 1981.

[85] H. V. Jagadish. Linear clustering of objects with multiple attributes. In *Proceedings of the ACM SIGMOD Conference*, pages 332–342, Atlantic City, NJ, June 1990.

[86] H. V. Jagadish. Spatial search with polyhedra. In *Proceedings of the 6th IEEE International Conference on Data Engineering*, pages 311–319, Los Angeles, February 1990.

[87] K. I. Joy, J. Legakis, and R. MacCracken. Data structures for multiresolution representation of unstructured meshes. In *Hierarchical and Geometrical Methods in Scientific Visualization*, G. Farin, B. Hamman, and H. Hagen, eds., pages 143–170, Springer-Verlag, Berlin, 2003.

[88] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management (CIKM)*, pages 490–499, Washington, DC, November 1993.

[89] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th International Conference on Very Large Data Bases (VLDB)*, J. Bocca, M. Jarke, and C. Zaniolo, eds., pages 500–509, Santiago, Chile, September 1994.

[90] I. Kamel, M. Khalil, and V. Kouramajian. Bulk insertion in dynamic R-trees. In *Proceedings of the 7th International Symposium on Spatial Data Handling*, M. J. Kraak and M. Molenaar, eds., pages 3B.31–3B.42, Delft, The Netherlands, August 1996. International Geographical Union Commission on Geographic Information Systems, Association for Geographical Information.

[91] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, ed., pages 369–380, Tucson, AZ, May 1997.

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

[93] J. T. Klosowski, M. Held, J. S. B. Mitchell, H. Sowizral, and K. Zikan. Efficient collision detection using bounding volume hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, January 1998.

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

[95] D. E. Knuth. *The Art of Computer Programming: Fundamental Algorithms*, vol. 1. Addison-Wesley, Reading, MA, third edition, 1997.

[96] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, vol. 3. Addison-Wesley, Reading, MA, second edition, 1998.

[97] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, ed., pages 324–335, Tucson, AZ, May 1997.

[98] H.-P. Kriegel, H. Horn, and M. Schiwietz. The performance of object decomposition techniques for spatial query processing. In *Advances in Spatial Databases — 2nd Symposium, SSD'91*, O. Günther and H.-J. Schek, eds., pages 257–276, Zurich, Switzerland, August 1991. Also Springer-Verlag Lecture Notes in Computer Science 525.

[99] D. Lea. Digital and hilbert *k-d* trees. *Information Processing Letters*, 27(1):35–41, February 1988.

[100] Y.-J. Lee and C.-W. Chung. The DR-tree: a main memory data structure for complex multi-dimensional objects. *Geoinformatica*, 5(2):181–207, June 2001.

[101] S. T. Leutenegger, M. A. López, and J. Edgington. STR: A simple and efficient algorithm for R-tree packing. In *Proceedings of the 13th IEEE International Conference on Data Engineering*, A. Gray and P.-Å. Larson, eds., pages 497–506, Birmingham, United Kingdom, April 1997.

[102] X. Liu and G. F. Schrack. A new ordering strategy applied to spatial data processing. *International Journal Geographical Information Science*, 12(1):3–22, ?? 1998.

[103] M.-L. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the ACM SIGMOD Conference*, pages 209–220, Minneapolis, MN, June 1994.

[104] M.-L. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proceedings of the ACM SIGMOD Conference*, pages 247–258, Montréal, Canada, June 1996.

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

[106] R. Miller and Q. F. Stout. Pyramid computer algorithms for determining geometric properties of images. In *Proceedings of the Symphosium on Computational Geometry*, pages 263–269, Baltimore, MD, June 1985.

[107] A. Moitra. Spatio-temporal data management using R-trees. In *Proceedings of the ACM Workshop on Advances in Geographic Information Systems*, N. Pissinou, ed., pages 28–33, Arlington, VA, November 1993.

[108] G. M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. , IBM Ltd., Ottawa, Canada, 1966.

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

[110] S. M. Omohundro. Five balltree construction algorithms. Technical Report TR-89-063, International Computer Science Institute, Berkeley, CA, December 1989.

[111] P. van Oosterom. *Reactive data structures for geographic information systems*. PhD thesis, Department of Computer Science, Leiden University, Leiden, The Netherlands, December 1990.

[112] P. van Oosterom and E. Claassen. Orientation insensitive indexing methods for geometric objects. In *Proceedings of the 4th International Symposium on Spatial Data Handling*, vol. 2, pages 1016–1029, Zurich, Switzerland, July 1990.

[113] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the 3rd ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 181–190, Waterloo, Ontario, Canada, April 1984.

[114] J. O'Rourke and K. R. Sloan Jr. Dynamic quantization: two adaptive data structures for multidimensional spaces. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 6(3):266–280, May 1984.

[115] T. Ottmann and D. Wood. 1-2 brother trees or AVL trees revisited. *Computer Journal*, 23(3):248–255, August 1980.

[116] C. M. Park and A. Rosenfeld. Connectivity and genus in three dimensions. Computer Science TR-156, University of Maryland, College Park, MD, May 1971.

[117] J. M. Patel and D. J. DeWitt. Partition based spatial-merge join. In *Proceedings of the ACM SIGMOD Conference*, pages 259–270, Montréal, Canada, June 1996.

[118] G. Peano. Sur une courbe qui remplit toute une aire plaine. *Mathematische Annalen*, 36:157–160, 1890.

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

[120] D. R. Reddy and S. Rubin. Representation of three–dimensional objects. Computer Science Department CMU–CS–78–113, Carnegie–Mellon University, Pittsburgh, April 1978.

[121] P. Rigaux, M. Scholl, and A. Voisard. *Spatial Database Management Systems: Applications to GIS*. Morgan-Kaufmann, San Francisco, 2001.

[122] J. T. Robinson. The K-D-B-tree: a search structure for large multidimensional dynamic indexes. In *Proceedings of the ACM SIGMOD Conference*, pages 10–18, Ann Arbor, MI, April 1981.

[123] A. Rosenfeld. Quadtrees and pyramids. In *Proceedings of the 5th International Conference on Pattern Recognition*, pages 802–811, Miami Beach, December 1980.

[124] A. Rosenfeld. Some useful properties of pyramids. In *Multiresolution Image Processing and Analysis*, A. Rosenfeld, ed., pages 2–5. Springer-Verlag, Berlin, West Germany, 1984.

[125] A. Rosenfeld and J. L. Pfaltz. Sequential operations in digital picture processing. *Journal of the ACM*, 13(4):471–494, October 1966.

[126] K. A. Ross, I Sitzmann, and P. J. Stuckey. Cost-based unbalanced R-trees. In *Proceedings of the 13th International Conference on Scientific and Statistical Database Management*, ??, ed., pages 203–212, Fairfax, VA, July 2001.

[127] N. Roussopolos, Y. Kotidis, and M. Roussopolos. Cubetree: Organization of and bulk incremental updates on the data cube. In *Proceedings of the ACM SIGMOD Conference*, J. Peckham, ed., pages 89–111, Tucson, AZ, May 1997.

[128] N. Roussopoulos and D. Leifker. Direct spatial search on pictorial databases using packed R-trees. In *Proceedings of the ACM SIGMOD Conference*, pages 17–31, Austin, TX, May 1985.

[129] D. Rutovitz. Data structures for operations on digital images. In *Pictorial Pattern Recognition*, G. C. Cheng, R. S. Ledley, D. K. Pollock, and A. Rosenfeld, eds., pages 105–133. Thompson Book Co., Washington, DC, 1968.

[130] A. Saalfeld. It doesn't make me nearly as CROSS, some advantages of the point-vector representation of line segments in automated cartography. *International Journal of Geographical Information Science*, 1(4):379–386, ?? 1987.

[131] H. Sagan. *Space-Filling Curves*. Springer-Verlag, New York, 1994.

[132] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-tree: an index structure for high--dimensional spaces using relative approximation. In *Proceedings of the 26th International Conference on Very Large Data Bases (VLDB)*, A. El Abbadi, M. L. Brodie, S. Chakravarthy, U. Dayal, N. Kamel, G. Schlageter, and K.-Y. Whang, eds., pages 516–526, Cairo, Egypt, September 2000.

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

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

[135] H. Samet. Spatial data structures. In *Modern Database Systems, The Object Model, Interoperability and Beyond*, W. Kim, ed., pages 361–385. ACM Press and Addison-Wesley, New York, 1995.

[136] H. Samet. Decoupling partitioning and grouping: overcoming shortcomings of spatial indexing with bucketing. *ACM Transactions on Database Systems*, 29(4), December 2004. Also University of Maryland Computer Science TR-4526.

[137] H. Samet. *Foundations of Multidimensional and Metric Data Structures*. To appear, 2005.

[138] H. Samet and W. G. Aref. Spatial data models and query processing. In *Modern Database Systems, The Object Model, Interoperability and Beyond*, W. Kim, ed., pages 338–360. ACM Press and Addison-Wesley, New York, 1995.

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

[140] M. Schiwietz. *Organization and query processing of spatial objects*. PhD thesis, Ludwig-Maximilians Universität München, Munich, Germany, 1993. In German.

[141] M. Schiwietz and H.-P. Kriegel. Query processing of spatial objects: Complexity versus redundancy. In *Advances in Spatial Databases — 3rd International Symposium, SSD'93*, D. Abel and B. C. Ooi, eds., pages 377–396, Singapore, June 1993. Also Springer-Verlag Lecture Notes in Computer Science 692.

[142] R. Schneider and H.-P. Kriegel. The TR*-tree: a new representation of polygonal objects supporting spatial queries and operations. In *Computational Geometry – Methods, Algorithms and Applications*, H. Bieri and H. Noltemeier, eds., pages 249–264, Bern, Switzerland, March 1991. Also Springer-Verlag Lecture Notes in Computer Science 553.

[143] G. Schrack and X. Liu. The spatial u-order and some of its mathematical characteristics. In *Proceedings of the Pacific Rim Conference on Communications, Computers, and Signal Processing*, pages 416–419, Victoria, British Columbia, Canada, May 1995.

[144] B. Seeger. personal communication, 1990.

[145] T. Sellis, N. Roussopoulos, and C. Faloutsos. The $R^+$–tree: A dynamic index for multi-dimensional objects. In *Proceedings of the 13th International Conference on Very Large Databases (VLDB)*, P. M. Stocker and W. Kent, eds., pages 71–79, Brighton, United Kingdom, September 1987. Also Computer Science TR-1795, University of Maryland, College Park, MD.

[146] C. A. Shaffer and H. Samet. An in–core hierarchical data structure organization for a geographic database. Computer Science Department TR 1886, University of Maryland, College Park, MD, July 1987.

[147] M. I. Shamos and D. Hoey. Geometric intersection problems. In *Proceedings of the 17th IEEE Annual Symposium on Foundations of Computer Science*, pages 208–215, Houston, October 1976.

[148] S. Shekhar and S. Chawla. *Spatial Databases: A Tour*. Prentice-Hall, Englewood-Cliffs, NJ, 2003.

[149] K. R. Sloan Jr. Dynamically quantized pyramids. In *Proceedings of the 6th International Joint Conference on Artificial Intelligence*, pages 734–736, Vancouver, Canada, August 1981.

[150] N. Solntseff and D. Wood. Pyramids: A data type for matrix representation in PASCAL. *BIT*, 17(3):344–350, 1977.

[151] S. N. Srihari. Pyramid representation for solids. *Information Sciences*, 34:25–46, 1984.

[152] M. Stonebraker, T. Sellis, and E. Hanson. An analysis of rule indexing implementations in data base systems. In *Proceedings of the 1st International Conference on Expert Database Systems*, pages 353–364, Charleston, SC, April 1986.

[153] P. Suetens, P. Fua, and A. J. Hanson. Computational strategies for object recognition. *ACM Computing Surveys*, 24(1):5–61, March 1992.

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

[155] S. L. Tanimoto and T. Pavlidis. A hierarchical data structure for picture processing. *Computer Graphics and Image Processing*, 4(2):104–119, June 1975.

[156] Y. Theodoridis and T. Sellis. Optimization issues in R-tree construction. Knowledge and Database Systems Laboratory Report KDBSLAB-TR-93-08, National Technical University of Athens, Athens, Greece, October 1993.

[157] Y. Theodoridis and T. Sellis. Optimization issues in R-tree construction. In *IGIS'94: Geographic Information Systems, International Workshop on Advanced Research in Geographic Information Systems*, J. Nievergelt, T. Roos, H.-J. Schek, and P. Widmayer, eds., pages 270–273, Monte Verità, Ascona, Switzerland, March 1994.

[158] Y. Theodoridis, E. Stefanakis, and T. K. Sellis. Efficient cost models for spatial queries using R-trees. *IEEE Transactions on Knowledge and Data Engineering*, 12(1):19–32, January/February 2000.

[159] C. D. Tomlin. *Geographic information systems and cartographic modelling*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[160] H. Tropf and H. Herzog. Multidimensional range search in dynamically balanced trees. *Angewandte Informatik*, 23(2):71–77, February 1981.

[161] W. Wang, J. Yang, and R. Muntz. PK-tree: a spatial index structure for high dimensional point data. In *Proceedings of the 5th International Conference on Foundations of Data Organization and Algorithms (FODO)*, K. Tanaka and S. Ghandeharizadeh, eds., pages 27–36, Kobe, Japan, November 1998.

[162] W. Wang, J. Yang, and R. R. Muntz. STING: A statistical information grid approach to spatial data mining. In *Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB)*, M. Jarke, M. J. Carey, K. R. Dittrich, F. H. Lochovsky, P. Loucopoulos, and M. A. Jeusfeld, eds., pages 186–195, Athens, Greece, August 1997.

[163] D. A. White and R. Jain. Algorithms and strategies for similarity retrieval. Technical Report VCL-96-101, Visual Computing Laboratory, University of California, San Diego, CA, 1996. (see `http://vision.ucsd.edu/papers/simret`).

[164] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th IEEE International Conference on Data Engineering*, S. Y. W. Su, ed., pages 516–523, New Orleans, February 1996.

[165] M. White. N-trees: large ordered indexes for multi-dimensional space. Statistical research division, US Bureau of the Census, Washington, DC, 1982.