Q_0-TREE: A DYNAMIC STRUCTURE FOR ACCESSING SPATIAL OBJECTS WITH ARBITRARY SHAPES

Ratko Orlandic
John L. Pfaltz

IPC-TR-91-010
December 6, 1991

Institute for Parallel Computation
School of Engineering and Applied Science
University of Virginia
Charlottesville, VA 2290l

This research was supported in part by
DOE Grant #DE-FG05-88ER25063 and
JPL Contract #957721

Abstract

The items in a spatial database have location, extent, and shape with respect to a spatial coordinate system. Simple approximations to these attributes, say by bounding rectangles, are storage efficient and easy to manipulate. But effective spatial retrieval (on either location, extent, or shape) require a more precise representation of these attributes. In this report, we describe a highly compressed quadtree representation, called a Q_0-tree, which supports spatial queries without false drops or unnecessary storage accesses. This access structure is dynamic. Moreover, because it is an exact representation of the spatial configuration, the spatial operators union, intersection, and difference can be coded with respect to the Q_0-tree itself without needing a separate representation of the configuration, and, in worst case, exhibit linear performance.

We discuss quadtrees, octtrees, grid files, R-trees, cell trees, and zkd B-trees; and provide a more detailed qualitative comparison between the latter two and Q_0-trees, contrasting both storage and processing overhead.

Index Terms:

Database, spatial search, image, quadtree, Q_0-representation.

1. Introduction

Spatial retrieval naturally occurs in many advanced computer applications, whenever it is beneficial to process objects according to their positions in a D-dimensional space. Traditionally, it has been associated with computer-aided design and geographic applications, but more recently the number of applications that require the spatial-search capability has grown substantially. They include robotics, computer vision, natural-resource management, environmental studies, medical imaging, etc. The types of "spatial queries" that are most useful for these applications require the ability to search efficiently for D-dimensional objects which: (1) contain a specified point in space (point query); (2) intersect a specified region in space (region intersection); (3) enclose a region in space (region enclosure); or (4) are enclosed by a region in space (region containment) [SeK88].

In this context, as in image processing, the actual representation of spatial objects plays a crucial role. A variety of different representations have been considered [ROG88]. The simplest representation uses the approximation of complex objects by its minimal enclosing rectangles whose sides are parallel to the axes of the data space. It has the advantage of low storage overhead; but at the expense of decreased precision. More complex vector representations of enclosing polygons with arbitrary sides have also been considered. On the other extreme, we have exhaustive pixel-by-pixel representations, which are precise but extremely inefficient in terms of their storage requirements. Quadtrees and octtrees [Sam84] reduce this overhead by decomposing objects into constituent squares or cubes, respectively, with variable size.

Many spatial retrieval algorithms assume the simplest spatial representation of objects using minimal enclosing rectangles. As noted in [SeK88], they can be organized into three groups: (1) transformation techniques handle spatial queries by mapping each object to a single point in a multi-dimensional space; (2) overlapping-region schemes use hierarchical organization of overlapping rectangles, where each higher-level rectangle encloses several low-level regions; and (3), clipping methods perform decomposition of rectangles along vertical and horizontal lines (or hyperplanes of a higher-dimensional space). Each class of methods appears to have its advantages and disadvantages. But, the common problem of all spatial-search schemes based on enclosing-rectangle approximations is the increased number of *false drops*, or access of blocks not containing relevant data. We next examine selected methods within each of these three groups using minimal enclosing rectangles.

Grid files [NHS84] are a transformation method which maps each rectangle to a point in a higher-dimensional space. The structure requires a D-dimensional array of pointers to disk blocks kept on secondary storage, which is called a grid directory. Additionally, it uses a set of linear scales for each dimension of space, which are assumed to be memory resident. Spatial queries, containing values or ranges of values, are converted into interval boundaries by scanning the linear scales. The resultant boundaries provide the direct access to the elements of the grid directory, and subsequently to the data blocks which have to be searched. In addition to its sensitivity to input distributions, this method has an apparent memory-management problem, arising from its need to keep the linear scales in memory. A new method, called hB-trees [LoS90], which is a combination of transformation and clipping schemes, provides better resolution of these problems.

R-trees [Gut84] are a representative example of an overlapping-region scheme. Like B-trees, it is a hierarchical collection of nodes, yielding efficient main memory management. Leaf nodes contain vectors describing the bounding rectangles of D-dimensional objects in space. An upper-level entry, pointing to a lower-level node, holds dimensions of the D-dimensional box enclosing all rectangles of the indicated node. The search procedure recursively descends the tree to find all objects whose bounding boxes satisfy the search query, e.g. intersect the given search rectangle. Unfortunately, the R-trees cannot guarantee a good worst-case behavior when there is an excessive overlap of rectangles in higher levels of the tree. One may end up searching a large portion of the file for a single query.

Clipping methods are usually extensions of transformation, overlapping-region, or exact-match retrieval methods. For example, R^+-trees [SRF87] are a variant of R-trees and the k-D-B-tree clipping method [Rob81]; while multi-level grid files [SiW88] represent a clipping variant of grid files. Clipping schemes based on multi-dimensional hashing often use known hashing methods as their paradigms, such as PLOP-hashing [SeK88] which is based on linear hashing [Lit80]. The essence of these methods is that they partition a D-dimensional space into D-dimensional slices by a set of hyperplanes parallel to the axes. The bounding rectangle of an object can intersect several hyperplanes and its parts may lie in different slices. Then the search procedure is restricted to those pages whose corresponding slices intersect the given search rectangle. Splitting of overfilled pages is consistent with the partition of the corresponding slices by newly added hyperplanes.

All of these methods base their search on a form of bounding rectangle. But, to minimize the number of false drops and hopefully increase the processing efficiency, different representations of spatial extent should be considered. One promising method is the cell tree [GuB91], which allows objects with arbitrary shapes, but requires their decomposition into convex cells both for storage and for search purposes. Convex pieces of a single object must be inserted in the structure individually. Leaf nodes of a cell tree contain complete geometric specifications of individual cells, while upper-level nodes incorporate larger convex cells enclosing the ones in the indicated lower-level blocks. The indexing structure is dynamic, adjustable to gradual changes of content; but splitting of pages is not guaranteed, so overflow buckets can be required. The search procedure begins by decomposing the search object into its convex cells and performing the same procedure for each of these cells. As in R-trees, a query may need to examine multiple branches of the tree hierarchy for a single cell of the search region. This can lead to a recursive type of search which is generally less preferred than a range search.

Orenstein and Merret [OrM84] proposed a high-precision spatial access method, called zkd B-trees, based on the so-called z order of points in the D-dimensional space. Each point is associated with a string, called its *z value*, obtained by interleaving bits of the coordinates of the point given in their binary form. Points whose z values have a common prefix form a rectangular region which can be identified by the prefix, serving as the z value of the region. An object is decomposed into a set of elementary regions, such that each element is uniquely identified by a single z value. Then the individual (*z_value*, *object_id*) pairs are inserted into a B-tree [Com79]. The ordered sequence of all (*z_value*, *object_id*) pairs at the lowest-level of the B-tree is termed GF-sequence. Spatial search proceeds by a similar decomposition of the search region and by performing a prefix-matching search for each z value obtained. A point query must examine each entry in the structure whose z value is a prefix of the point's z value, and take the union of the respective object identifiers. This can involve a fairly large portion of the file if many objects have spatial overlap. In addition, spatial search relies on costly bit manipulations.

The potential cost of false drops remains an issue for many contemporary spatial access methods, including those above. False drops can be introduced either as a result of bounding approximations of objects (as in methods based on bounding rectangles) or the approximations in upper layers of the structure (as in cell trees), or both (as in R-trees). In zkd B-trees several entries may keep object identifiers which belong to the result set of a simple point query and no guarantees on the number of disk accesses can be given even in this simplest case. This is true for most spatial indexing structures. Frequently, entries falling outside the search region must be examined in the course of a spatial query. Further, few methods support other operations that frequently occur in spatial databases, such as set operations (unions, intersections and differences) or similarity assessment for common overlap between different images. Most of them do not contain accurate description of the spatial image. As a consequence, huge raster images must be stored separately and manipulated directly.

In this report we introduce a dynamic index structure for spatial retrieval, called a Q_0-tree, which has none of these disadvantages. It provides a fast point search along a single path in the structure, involving only 2 or 3 disk accesses in most realistic situations. It supports arbitrarily shaped objects and search regions, and incurs no false drops, neither with respect to the objects themselves, nor with respect to higher-level blocks within the structure. No entry representing a region that does not overlap the search region will ever be examined to answer a spatial query. If a block of the structure has been accessed, than it contains at least one entry whose corresponding region overlaps the search object. In addition, the spatial structure itself contains a complete representation of the original image in a much more compact form, and supports various operations on images accurately and efficiently. In that sense, Q_0-trees are both the spatial structures and the isomorphisms of their corresponding raster images, which eliminate the need to explicitly store the images.

The structure is based on a new representation of region quadtrees, called a Q_0-representation, given in the next section of the report. In section 3 we partially support the claim that the structure is a true isomorphism of the original image, by presenting the algorithms for set operations and the point search. In section 4 we introduce hierarchical Q_0-trees and give simple algorithms for implementing the spatial queries; while section 5 contains the description of a practical update procedure for dynamic environments. In section 6 the comparative performance of Q_0-trees is discussed. Section 7 concludes the report by reviewing the properties of the structure.

2. Image Representation

In this section we discuss the use of region quadtrees to represent spatial objects and introduce a new image representation for which the concept of the quadtree serves as an abstract model. We begin by stating some useful definitions.

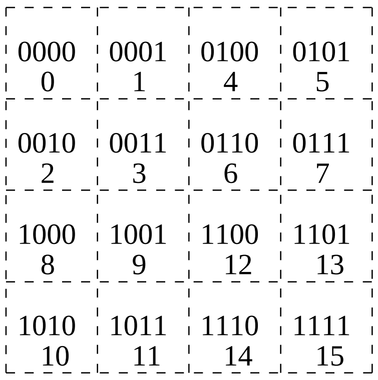
A **pixel space** is a square D-dimensional matrix whose dimensions are powers of 2, i.e. 2^K ($K \geq 1$). The number K is called the **degree of resolution**. The total number of pixels in the space is 2^{DK}, where D is the number of dimensions. The assignment to each pixel of a gray level, of a color, or more generally of a pointer to a list of spatial objects containing that pixel, we will call an **image**. A region of the image is **homogeneous** if all its pixels have the same color, or in our case, belong to the same set of spatial objects.

If a D-dimensional image is not homogeneous, it can be decomposed into 2^D equal D-dimensional squares, or quadrants, whose dimensions are 2^{K-1}. The decomposition can be applied recursively to each quadrant until the image is subdivided into homogeneous squares. Each quadrant obtained in the process can be uniquely identified by a string, called a **locational code** (called its z *value* by Orenstein), that describes its place in the decomposition process, or alternatively its position in the D-dimensional space. For example, the 4 subquadrants of a 2-dimensional square can be denoted by Nw, Ne, Sw and Se, or by 00, 01, 10 and 11, respectively.[1] Figure 2-1(a) shows a 2-dimensional pixel space with resolution $K = 2$. The locational codes of each pixel are indicated. Those of the four upper-leftmost squares in Figure 2-1(a) are NwNw, NwNe, NwSw and NwSe, or 0000, 0001, 0010 and 0011, respectively. The quadrant enclosing them can be identified by Nw or 00. Binary locational codes have maximal length $M = D \cdot K$. Observe that the locational code of a quadrant Q is a proper prefix of the locational codes of all squares enclosed by Q. The 16 pixels of the image space have also been numbered in what is called Morton order (or Morton matrix) [Sam84]. The Morton order is obtained by simply interpreting each locational code as a binary integer.[2]

Regions in the image space which can be uniquely identified by a single locational code need not necessarily be square. For example, the rectangle composed of pixels 8 and 9 can be designated by the 3 digit locational code 100, which is the prefix of both 1000 and 1001, or 100* where * means *don't care*. Similarly the 2 digit locational code 00, or 00**, denotes pixels 0 through 3. We call these **simple regions**. Some D-dimensional squares and rectangles are sim-

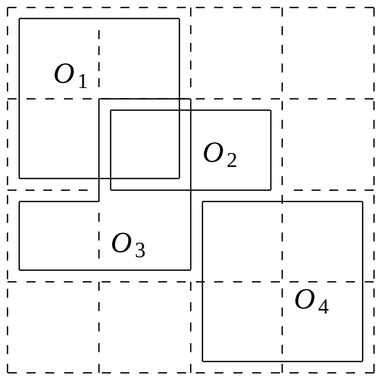
[1] Throughout the report we will, for expository purposes, consider only 2-dimensional images; but the discussion applies to arbitrary D-dimensional spaces.

[2] The z order of Orenstein can be obtained from the Morton order by rotating the image counter-clockwise 90°.



Pixel Space

(a)



O_1 = { 00 }

O_2 = { 0011, 0110 }

O_3 = { 0011, 100 }

O_4 = { 11 }

A four object image

(b)

Objects in a 2-dimensional pixel space.
Figure 2-1.

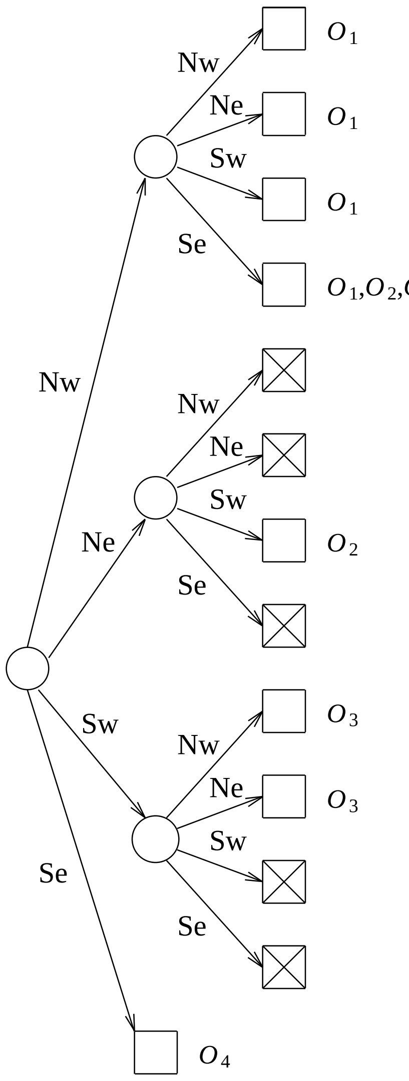
ple regions; others are not. For example, neither the rectangular region comprised of pixels 0 and 2 nor the square comprised of pixels 2, 3, 8, and 9, can have a single locational code. Instead these **complex regions**, i.e. non-simple regions, must be decomposed into a set of simple regions whose locational codes identify the location, extent, and shape of the complex region. Figure 2-1(b) shows an image space with four sample objects in it. Observe that the objects O_1 and O_4 are simple regions, while O_2 and O_3 are complex. Region O_3 is a concave object composed of pixels 0011, 1000 and 1001. However, pixels 1000 and 1001 comprise the simple rectangle denoted by the locational code 100.

2.1. Region Quadtrees

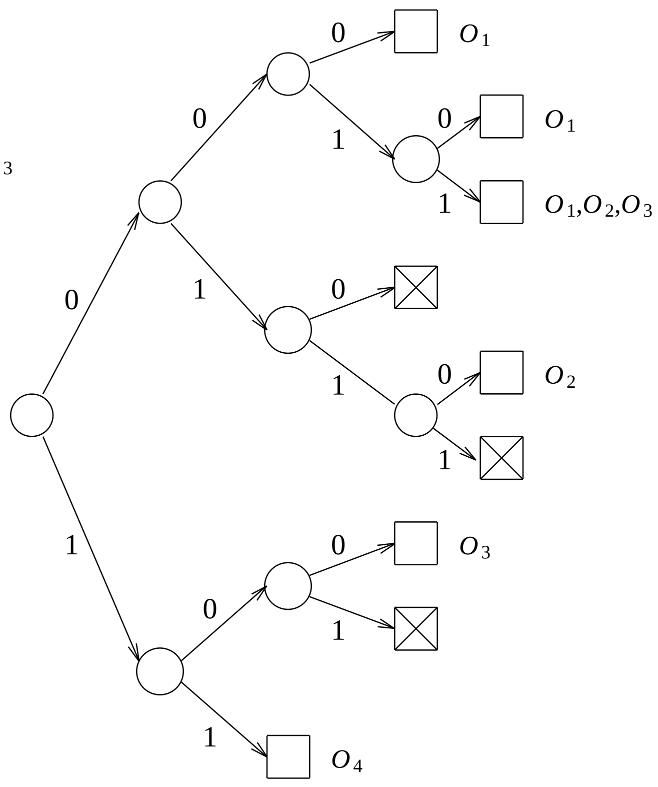
The recursive D-dimensional decomposition of an image can be expressed as a tree (or more precisely, a *trie* [Fre60]) with the degree 2^D, i.e. with each interior node possessing exactly 2^D sons. The root of the tree corresponds to the entire image, and each leaf stands for a homogeneous quadrant contained in a single object, in an overlap area of a set of objects, or in the empty portion of the pixel space. This structure is called **region quadtree**. An intensive discussion of 2D-region quadtrees, including other forms of quadtrees (e.g. PR quadtree, edge quadtree, etc.), is given in [Sam84]. By region quadtrees, however, we will mean not only 2-dimensional quadtrees, but 3-dimensional octtrees, etc., as well. Figure 2-2(a) shows the region quadtree of the image of Figure 2-1(b). Empty leaves have been marked with an X; after each nonempty leaf the objects enclosing the corresponding quadrant have been enumerated. Note that the quadrant Nw must be decomposed into its 4 subsquares because it is not uniform: 3 of its quadrants comprise only O_1, while the fourth one belongs to two more objects as well.

A clear advantage of quadtrees over the original raster representation is its potential for greatly reduced storage overhead. In addition, it supports set operations (union, difference and intersection), searching, scaling based on the power of 2, windowing, rotation, and other transformations [Sam84], whose computational cost is proportional to the number of nodes in the tree rather than the number of pixels in the image. Moreover, it preserves locations of arbitrarily shaped objects in space needed for spatial search.

Region quadtrees need not be represented as tries of degree 2^D. An alternative organization is that of a **bintree** [Tam84], obtained by inserting binary locational codes of the simple homogeneous rectangles of the image into a binary trie. As in binary tries in general, every arc in a bintree is implicitly labeled by either a 0 or a 1. A node (interior or leaf) will be called a **0-node** if its entering arc is labeled 0. One can similarly define the **1-nodes** in the bintree. The **path** to a node is the binary string obtained by concatenating edge labels from the root down to the node. The unique path of a node n will be denoted either by $path(n)$ or by $lcode(n)$, since a path to a node in the bintree spells the binary locational code of the region corresponding to the node. The length of $path(n)$ is the **depth** of the node n. As in other kinds of trees, the nodes of a bintree can be ordered by a traversal. Throughout the report we adopt $depth-first$ traversal (also called $pre-order$ traversal) of nodes in a bintree, which we will be referring to simply as **0-order** because of the way we draw binary tries. Observe, once the bintree has been constructed, the



Region quadtree
(a)



Bintree
(b)

Region quadtree and the corresponding bintree.
Figure 2-2.

distinction between the 1-dimensional, 2-dimensional and higher-dimensional spaces becomes irrelevant.

Bintrees are always *complete* in the sense that every interor node has precisely two sons. Because two adjacent homogeneous squares (which might require two separate leaves in the region quadtree) may form a simple homogeneous rectangle which can be represented by a single leaf in the bintree, the number of leaves in a bintree is always less than or equal the number of leaves in the corresponding region quadtree of degree 2^D. The bintree of Figure 2-2(b) has four fewer leaves than the equivalent region quadtree of Figure 2-2(a). On the other hand, bintrees require two interior nodes to record each level, so that region quadtrees will tend to have smaller total number of nodes.

In general, trie structures have been well analyzed and the problems of their efficient implementation and compression have received considerable attention. Quadtrees and bintrees are no exception. As pointed out by Samet [Sam84], there are two principal ways to represent quadtrees in a linear, pointerless fashion. The first approach treats the trie as a collection of leaf nodes. Let us disregard for a while the lists of objects associated with the leaves of the quadtree. The simplest leaf-encoding method for representing quadtrees and bintrees is to use an ordered sequence of pairs $e_i = (lcode_i, length_i)$ where $lcode_i$ denotes a locational code, while $length_i$ is its length. In this representation only the locational codes for leaves corresponding to the non-empty regions need to be retained. (This representation is identical to the GF-sequence of Orenstein when objects do not overlap in space.) Some variations on this scheme make the *length* field implicit [Gar82]. The spatial search is accomplished by traversing the structure as achieved by a modular arithmetic or by manipulating letters (bits) in the locational codes. This, of course, entails a substantial computational overhead.

The second approach to represent a quadtrie is in terms of a traversal of its nodes, usually depth-first [KaE80]. Let us treat the leaves as being "W-leaves" (if they correspond to empty, i.e. white regions) or "B-leaves" (otherwise). An interior node can be denoted with an "I". Then the depth-first traversal of the quadtree of Figure 2-2(a) produces: IIBBBBIWWBWIBBWWB. Equivalently, the 0-order traversal of the bintree yields: IIIBIBBIWIBWIIBWB. Although very compact, either encoding significantly complicates the operations on the representation, including the spatial search which tends to be very expensive [KEM83].

2.2. The Q_0-Representation

We now introduce a new, efficient linear representation of quadtrees (more precisely, bintrees), called Q_0-representation (Q standing for quadtree and subscript 0 for 0-order traversal), which can be viewed as a combination of both the leaf-encoding and the traversal representation schemes. Consider the simplest leaf-encoding scheme of ordered tuples (*lcode*, *length*), where each pair corresponds to a leaf (empty or not) in a bintree. We begin our development of the Q_0-representation by showing that it is sufficient to retain just the *length* fields.

Let $leaves = <L_1, L_2, \ldots, L_{n-1}, L_n>$ be the sequence of leaves in a bintree as they appear in the 0-order. (The reader may note that the 0-order of leaves in tries coincides with their top-down placement in our figures.) Let $lengths = <l_1, l_2, \ldots, l_{n-1}, l_n>$ be the sequence of their corresponding depths. Let $lcodes = <lcode_1, lcode_2, \ldots, lcode_{n-1}, lcode_n>$ be the sequence of binary locational codes such that $lcode_i = lcode(L_i)$, i = 1...n, or alternatively $lcode_i = path(L_i)$. Let $icodes = <icode_1, icode_2, \ldots, icode_{n-1}, icode_n>$ be the sequence of "decompressed" binary locational codes, such that each $icode_i$ is a fixed length bit sequence of $M = D \cdot K$ bits, whose prefix of length l_i is $lcode_i$ and whose remaining tail contains all zeros. Observe that each $icode_i$ is unique within the sequence *icodes*, and that by inserting them into a binary trie we obtain the original bintree. Then a simple proof by induction establishes the following recurrence relation:

$$icode_i = \begin{cases} 0 & \text{if } i = 1 \\ icode_{i-1} + 2^{M-l_{i-1}} & \text{if } 1 < i \le n \end{cases} \tag{2.1}$$

where each $icode_i$ is interpreted as a binary integer of length $M = D \cdot K$. The recurrence (2.1) demonstrates that from just the sequence *lengths* we could reconstruct the original bintree, first by using it to obtain the sequence *icodes* and then by inserting each individual *icode* into the binary trie.

Perhaps the major innovation of Q_0-trees is that we do not, in fact, use the length of the lcode (or equivalently its depth in the bintree) to represent the bintree. Instead we record only the depths of the 1-nodes of the bintree sorted by the 0-order traversal of nodes in the trie![3] If n is the number of leaves in the bintree, let *depths* $= <d_1, d_2, \ldots , d_{n-1}, 0>$, be the depths of 1-nodes $< N_1, N_2, \ldots , N_{n-1} >$ encountered in 0-order sequence (and augmented with a terminating 0). Since the number of 1-nodes in a bintree is always one less than the number of leaves, we adopt the convention of appending 0 to the sequence of 1-node depths in order to have as many entries as there are leaves in the tree. Given only this sequence *depths* of depths of 1-nodes, as they appear in the 0-order traversal of the tree, it is still possible to reconstruct the depths of leaves, and subsequently the entire bintree. First, observe that every leaf L_i is immediately followed by the 1-node N_i in the 0-order. We will often refer to N_i as the *successor* of the leaf L_i. Now, L_i and N_i appear at the same level if and only if L_i is a 0-leaf. However, if L_i is a 1-leaf then $L_i = N_{i-1}$. In that case, the node N_i appears at a level closer to the root of the tree than N_{i-1} (i.e. L_i). The actual rule is the following:

(a) L_1 is always a 0-leaf and $l_i = d_i$;
(b) If $d_i > d_{i-1}$ then L_i is a 0-leaf and $l_i = d_i$, for $1 < i \le n$; (2.2)
(c) If $d_i < d_{i-1}$ then L_i is a 1-leaf and $l_i = d_{i-1}$, for $1 < i \le n$.

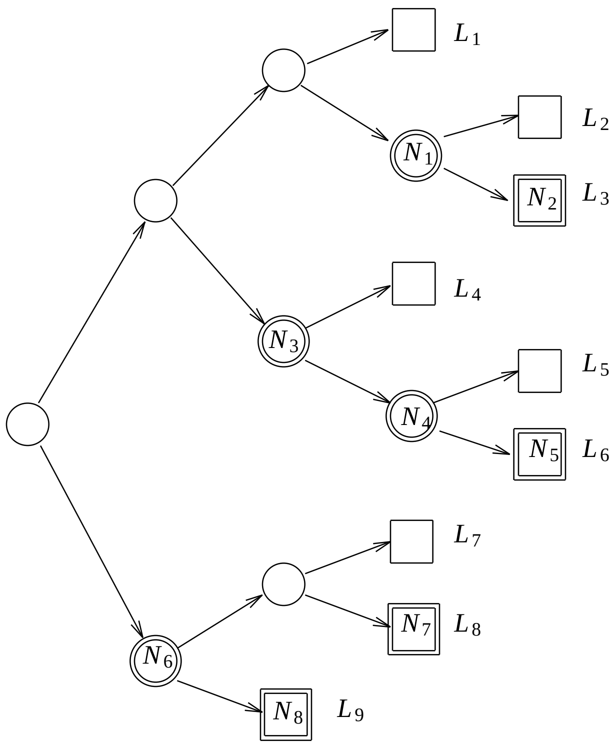
The case when $d_i = d_{i-1}$ need not be considered because no two consecutive 1-nodes can appear at the same level in a bintree. A formal proof of the rule (2.2) is given in [Orl89].

The actual Q_0-representation of a bintree is now an ordered collection of entries $e_i = (d_i, o_i)$, $1 \le i \le n$, where d_i is the depth of the i^{th} 1-node in the 0-order traversal of the tree (or 0 if $i = n$), and o_i is the *object list* of identifiers of all objects containing the region corresponding to the leaf L_i. Individual objects, O_i, in the list will normally record additional information, such as the color of the region corresponding to the entry, etc. However, the pairs we described are sufficient to perform spatial search. We will call d_i the **depth value** of the entry e_i, and o_i will

[3] Recording the depths of 1-nodes relative to a 0-order traversal can also be used to compress ordinary B-trees [OrP88].

be referred to as its **object list**.[4] Since, only $\lceil \log_2(M+1) \rceil$ bits are needed to represent a depth value, for 2-dimensional images and the resolution degree of up to $K = 31$ a 6-bit depth field would suffice. An 8-bit depth field covers most realistic resolutions and dimensions.[5]

Figure 2-3 illustrates the conversion of the bintree in Figure 2-2(b) into its compact Q_0-tree representation. In 2-3(a) we have replicated the bintree in Figure 2-2(b), and in addition



(a)

icode	leaf depth	Q_0-rep.
0000	3	3:O_1
0010	4	4:O_1
0011	4	2:O_1, O_2, O_3
0100	3	3:
0110	4	4:O_2
0111	4	1:
1000	3	3:O_3
1010	3	2:
1100	2	0:O_4

(b) (c) (d)

Bintree and its transformation into the Q_0-representation.
Figure 2-3.

[4] As in cell trees, the actual list of object id's can be included in the entries themselves rather than just putting a pointer to the list in the entry. This should yield somewhat faster spatial search, at the cost of having variable length entries in the Q_0-representations.

[5] The bintrees in this report are all *complete*, in that every non-leaf has exactly two children. Further compression of the Q_0-representation can be achieved by eliminating the empty 1-leaves from the bintree, thereby creating a 0-*complete* binary tree [OrP88, Orl89]. This will not affect the processing algorithms on the representation. Nevertheless, we will omit any more discussion of this in order to retain an intuitive notion of bintrees which is likely to be clearer to the reader. Thus, for purposes of this report, complete bintrees will serve as the abstract model for Q_0-representations.

emphasized the 1-nodes by embolding their outline and labeling them N_i to emphasize their correspondence with leaves labeled L_i. Note that there is no 1-node, N_9 corresponding to L_9. Figure 2-3(b) lists the sequence *icodes* (i.e. the "decompressed" locational codes); 2-3(c) gives the ordered sequence of *leaf* depths; while 2-3(d) using 1–*node* depths illustrates final Q_0-representation of the bintree of Figure 2-2(a). Each entry e_i records the depth of N_i and the list of objects assigned to L_i.

From Figure 2-3 alone one could conclude that the conversions from the original image to the Q_0-representation, and back, are complex and expensive processes requiring several stages. This, however, is not the case. Such algorithms for the direct conversion, which rely on the Morton ordering of pixels in the image space and whose execution times are proportional to the number of homogeneous regions in the image, are not discussed in this report.

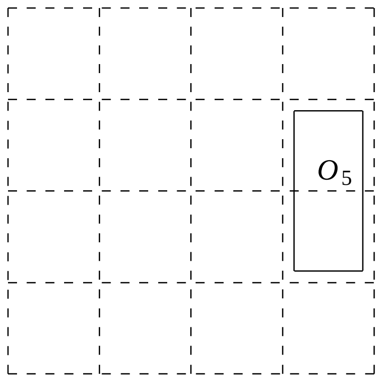
3. Image Operations

The Q_0-representation is a very compact, linear representation of quadtrees. However, if one is concerned solely with storage overhead, there exist still more compact representations. The real advantage of the Q_0-representation is that it supports relatively simple, linear procedure to perform important image operations. In this section we justify this claim by examining the set operations (union, difference and intersection) and the search for point location.

Q_0-representations have six important properties which can be exploited in developing image operators. First, it is a lossless, location-preserving representation so that any operation on a quadtree can be performed on the compact representation itself. Second, it is a leaf-encoding scheme which eliminates the computational overhead incurred by processing interior nodes during an image traversal. Third, leaves in the Q_0-representation are enumerated in increasing lexicographic order of their binary locational codes. This will be effectively used to develop a simple point-location search algorithm which relies on integer comparisons instead of more costlier bit-extraction and bit-manipulation operations. Fourth, as we will see later in the report, it is possible to construct a hierarchical Q_0-representation, in which upper levels of the hierarchy effectively represent the same image with smaller degree of resolution. Fifth, each subtree of the bintree is represented by a continuous portion of its Q_0-sequence, which reduces the search for a subtree to a range search within the representation. Sixth, and possibly most important, by storing the 1-node depths, which implicitly denote the *end* of a homogeneous region, and by letting the current position in an index denote the beginning of a homogeneous region, we can write code which linearly scans the representation (follows a 0-traversal of the conceptual quadtree) and needs search no more to process the region. This is the key to our implementation of the set operators in the following section.

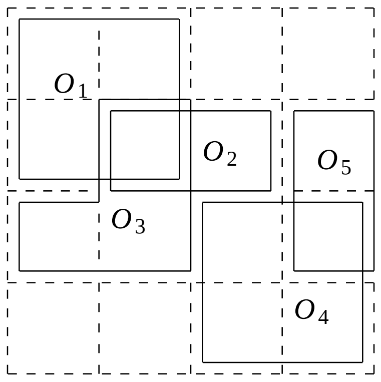
3.1. Set Operations — Union, Intersection, Difference

In Figure 3-1(a) we show a new image containing the spatial object O_5 with the set of locational codes {0111, 1101}. Figure 3-1(b) is the image resulting from the union of images 2-1(b) and 3-1(a). Figures 3-2(a-b) show the bintrees for the images 3-1(a-b), respectively. That is, 3-2(b) represents the bintree result of unioning the bintrees 2-2(b) and 3-2(a). (In the figures, empty



A one object image

(a)

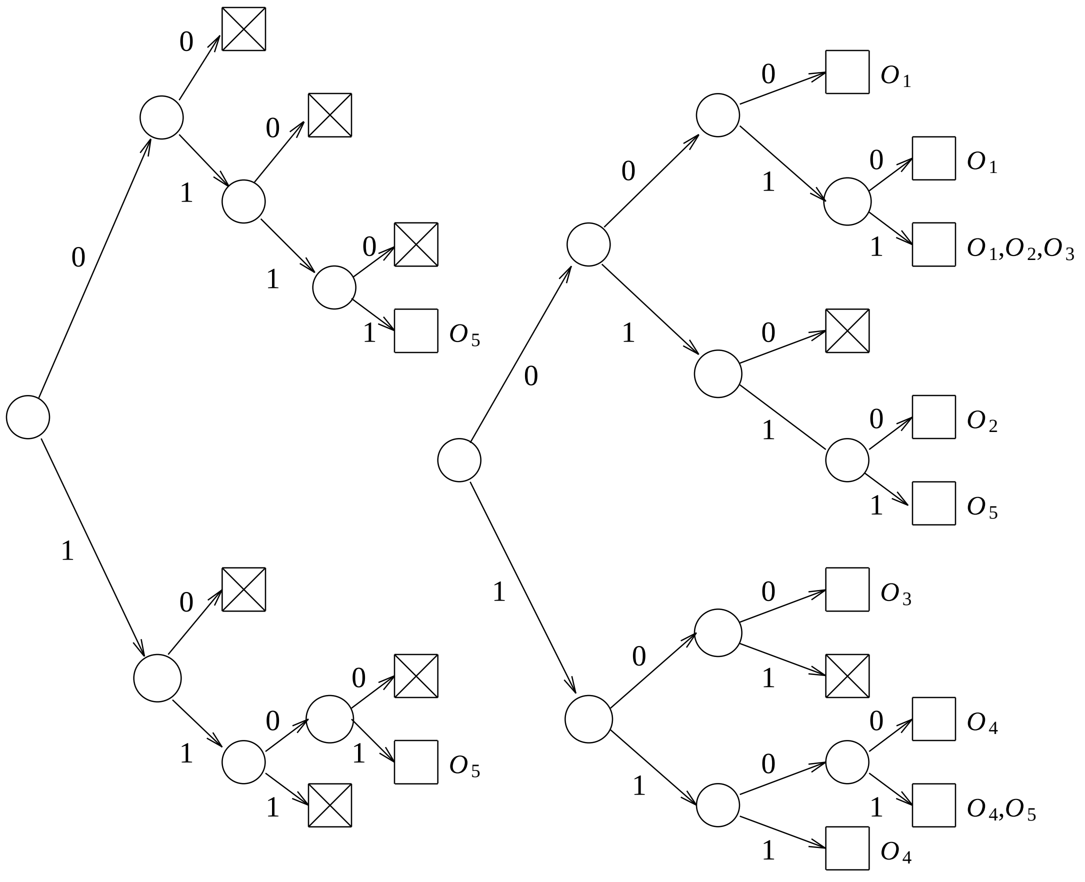


Its union with the image 2-1(b)

(b)

The union of images
Figure 3-1.

leaves in the bintree have been emphasized by X's.) Alternatively, bintree of Figure 2-1(b) could be derived by taking difference of bintrees in 3-2(b) and 3-2(a) (i.e. "subtracting" 3-2(a) from 3-2(b)). The intersection (or "spatial join" as it is sometimes called) of the bintrees 3-2(a) and 3-2(b) just produces the bintree 3-2(a) again; while an intersection performed on bintrees of Figures 2-1(b) and 3-2(a) yields a tree with a single empty leaf as its root. Figure 3-3(a-b) gives the corresponding Q_0-representations of the bintrees 3-2(a-b) respectively (i.e. of images 3-1(a-b)). The representations of Figures 2-3(d) and 3-3(a-b) can similarly be viewed as the results of unions, differences and intersections, in a manner described for their respective bintrees.



Bintree of the new object
(a)

Bintree of the union image
(b)

Bintrees corresponding to the images of Figure 3-1.
Figure 3-2.

2:	3:O_1
3:	4:O_1
4:	2:O_1,O_2,O_3
1:O_5	3:
2:	4:O_2
4:	1:O_5
3:O_5	3:O_3
0:	2:
	4:O_4
	3:O_4,O_5
	0:O_4
Representation of the new object	Representation of the union image
(a)	(b)

Q_0-representations of the images of figure 3-1.
Figure 3-3.

The set operations on two bintrees, representing images with arbitrarily shaped spatial objects, require the processing time proportional to the combined number of nodes in both trees and the inconvenience of doing the actual traversals of nodes in both trees simultaneously while performing the operations. The set operations on two Q_0-sequences take $O(n + m)$ time, where n and m denote the number of leaves in the corresponding bintrees. They involve sequential scans of the representations with no backtracking.

Figure 3-4 gives two general-purpose procedures to implement set operations on Q_0-representations. In our algorithms Q_0-representations are assumed to be global structures, R_1, R_2, with resultant R_3. They could equally well be parameters.

All three set operations use the same, operation-invariant *merge* procedure to which a parameter is passed denoting the actual operation to be performed. It scans the two given Q_0-representations in the following manner. Let i_1 indicate the current entry of R_1 and i_2 the current entry of R_2. The depth fields of the two entries are compared to determine which of them is less.

```
global   R1, R2, R3: <Q_0-representation>

procedure  merge (op: <op_code>);
       | Merge the Q_0-representations R1 and R2 to create
       | a representation R3 of the image I3 resulting from
       |      I3 ← I1 <op> I2
       var   i1, i2, i3: integer;
begin
i1 ← 1;  i2 ← 1;  i3 ← 0;
while  R1[i1].depth > 0  or  R2[i2].depth > 0  do
       case  R1[i1].depth <comp> R2[i2].depth  do
         '<':  update (i1, i2, i3, R2[i2].depth, op);
               i2 ← i2 + 1;
         '>':  update (i1, i2, i3, R1[i1].depth, op);
               i1 ← i1 + 1;
         '=':  update (i1, i2, i3, R1[i1].depth, op);
               i1 ← i1 + 1;  i2 ← i2 + 1;
       esac;
update (i1, i2, i3, R1[i1].depth, op);
end

procedure  update (i1, i2, i3, depth: integer; op: <op_code>)
       | Perform the <op> on the corresponding object lists of
       | R1 and R2 to create an object list for a new entry in R3.
       | In the cases of intersection and difference, this may
       | require deletion of previous entries made in R3.
       var   objects: <object_list>;
begin
case  op  of
  'union':      objects ← R1[i1].o_list ∪ R2[i2].o_list;
  'intersect':  objects ← R1[i1].o_list ∩ R2[i2].o_list;
  'difference': objects ← R1[i1].o_list – R2[i2].o_list;
esac
if  op ≠ 'union'  then
       while i3 ≥ 1 and depth < R3[i3].depth and objects = R3[i3].o_list do
              begin
              erase  R3[i3].entry;  i3 ← i3 - 1;
              end
i3 ← i3 + 1;  R3[i3].depth ← depth;  R3[i3].o_list ← objects;
end
```

Set operations.
Figure 3-4.

Suppose it is $R_1[i_1].depth$. Then the subscript i_1 stays unchanged (indicating the entry $R_1[i_1]$), while the index i_2 is advanced in R_2 until an entry $R_2[i_2]$, is reached with the same depth value as $R_1[i_1]$. In that case, both subscripts i_1 and i_2 are advanced one entry further. The process is repeated until both representations are exhausted. It is a classic linear "merge" process.

With each pair of entries $R_1[i_1]$ and $R_2[i_2]$, regardless of their comparative depths, the procedure *update*, which modifies the resulting representation R_3 in an incremental fashion, is invoked. If a union operation is being performed, the procedure just adds an entry $R_3[i3]$ to the resulting representation. Its depth value, $R_3[i3].depth$ is the greater of $R_1[i_1].depth$ and $R_2[i_2].depth$, and its list of objects is obtained by taking the union of object lists denoted by $R_1[i_1].o\_list$ and $R_2[i_2].o\_list$. For the difference and intersection operations, the procedure *update* constructs a new entry in a similar manner. However, before the entry is output, a *while* loop is executed which simulates the combination of sibling homogeneous leaves into a single leaf of the conceptual bintree.

The procedures *merge* and *update* can be further optimized, e.g. by deferring some updates to the resulting representation. But, even the ones we presented run in $O(n+m)$ time. The reader can verify their correctness by performing individual operations on Q_0-representations of Figures 2-3(d) and 3-3(a-b). In particular, one might verify that taking the intersection of representations 2-3(d) and 3-3(a) yields a representation consisting of only one entry with depth value 0 and empty list of objects, as claimed above. Using the equivalence between the bintrees and their Q_0-representations, it is possible to give formal proofs of correctness of these algorithms, similar to the ones given in [Orl89]. This is a major reason for developing both *bintrees* and Q_0*–representations* in tandem.

To add a new spatial object to an existing Q_0-representation one can simply construct the Q_0-sequence of its image and take the union of the two representations. Similarly, to delete a given spatial object one might take the difference of its representation and the one corresponding to the current image. This will suffice in fairly static environments, such as map generation and processing, and many other geographic applications. Other "dynamic" update procedures will be given later in this report.

3.2. Pixel Search

Since any pixel can be denoted by an M-bit binary locational code, the search of the Q_0-representation for the given point could be accomplished using just the recurrences (2.1) and (2.2). With the sequence of 1-node depths of entries in the Q_0-representation we can reconstruct the ordered sequence of "decompressed" locational codes (i.e. *icodes*) of the simple, homogeneous regions using these recurrences, and perform repeated comparisons until the first *icode* greater than the given locational code is found. The entry *before* the one at which the match occurs is the desired one. But any implementation of the reconstruction of *icodes* must involve multiple comparisons and additions per entry, which is at best slow.

A much simpler search algorithm, although still sequential in nature, is based on the realization that the comparison of the pixel *lcode* and a reconstructed *icode* is just a comparison of their 1-bits. As noted above, each depth value in a Q_0-representation indicates the position of the 1-bit to be set in the *icode* of the immediately following entry. Thus the positions of 1-bits of *icodes* are implicitly given. It is easier, and much more efficient to simply transform the search code into a sequence $B = <b_1, b_2, ..., b_l>$, denoting the positions of 1-bits in the pixel *lcode*, and then match them against the recorded depth values in the representation. The sequence B is appended with a value $M+1$ to ensure successful termination of the search algorithm. A complete description of the algorithm, borrowed from [OrP88] where its correctness proof also appears, is given in figure 3-5. We call the search procedure *search_block* for the reasons which will become clear when we discuss hierarchical Q_0-representations. This also applies to the use of the parameter *first*. For now we assume that *first* is always equal to 1.

To convince himself that the algorithm actually works, the reader might try to search the Q_0-sequences of Figures 2-3(d) and 3-3(b) for pixel locations 0011 and 1101. Since in the examples of this report M=4, the corresponding 1-bit sequences of the locational codes are B = <3,4,5> and B = <1,2,4,5>. With the first locational code the search will return i = 3 in both representations. In the second case, it returns i = 9 indicating the last entry of the Q_0-representation of Figure 2-3(d), while for the representation of 3-3(b) it returns i = 10. In all cases the search stops at the correct place.

```
procedure search_block (Q : <Q_0-representation>,
                    B : <vector_of_1-bit_positions>; first : integer): integer;
       |  Search the representation 'Q' for the leaf corresponding to the
       |  pixel location denoted by B.
       |  Return the subscript i of the entry which must denote
       |  this pixel, if it exists in the image at all.
       var  i, k : integer;
begin
i ← 1;  k ← first;
while B[k] ≤ Q[i].depth do
       begin
       if B[k] = Q[i].depth then k ← k+1;
       i ← i+1
       end;
first ← k;                       | Only used in iterated calls
return  i
end;
```

Algorithm for pixel search in a Q_0-representation.
Figure 3-5.

The discussion on the search for a point location serves as a prelude to the next section which contains the main results of this report.

4. Hierarchical Representation and General Spatial Search

Using the simple Q_0-representations described in the preceding sections is not realistic for very large images for two reasons. First, large images are most likely to reside in secondary storage. Creating a single large array of entries, which may span many blocks of secondary storage, introduces significant memory management problems. It is much more effective to subdivide the representation into fixed-size blocks corresponding to the natural size of blocks in the I/O environment. Second, since pixel search has a linear search time proportional to the number of leaves in the conceptual bintree, it can be relatively slow. A tree structured pixel search will yield logarithmic behavior. To achieve this, we form a hierarchy of Q_0-representations, in which each element of a higher level denotes only a small portion of a lower-level representation. It is advantageous to think of that portion of the representation denoted by an upper-level entry in terms of a *node* which can be assigned a single block of storage. The highest level of the hierarchy can be viewed as a single node, called *root*, and the whole hierarchy becomes a typical tree structure. All nodes of the same layer in the hierarchy appear at the same level in the tree.

One way to obtain a hierarchical Q_0-representation, called Q_0*–tree*, is to scan the linear sequence assigning separate portions of it (i.e. nodes) to fixed-size blocks. We adopt the following rule which will dictate the partition of the linear representation into individual nodes. Let the linear Q_0-representation be interpreted as a sequence $<e_1, \ldots, e_n>$ of entries of the form e_i = $(depth_i, ptr_i)$, $i=1 \ldots n$, where if e_i corresponds to a leaf of the bintree, we use o_list_i instead of ptr_i to denote the list of objects associated with that homogeneous region, if any.[6] A node, or block, will contain a portion of the linear Q_0-representation from an entry e_{first} to an entry e_{last} (inclusively) such that:

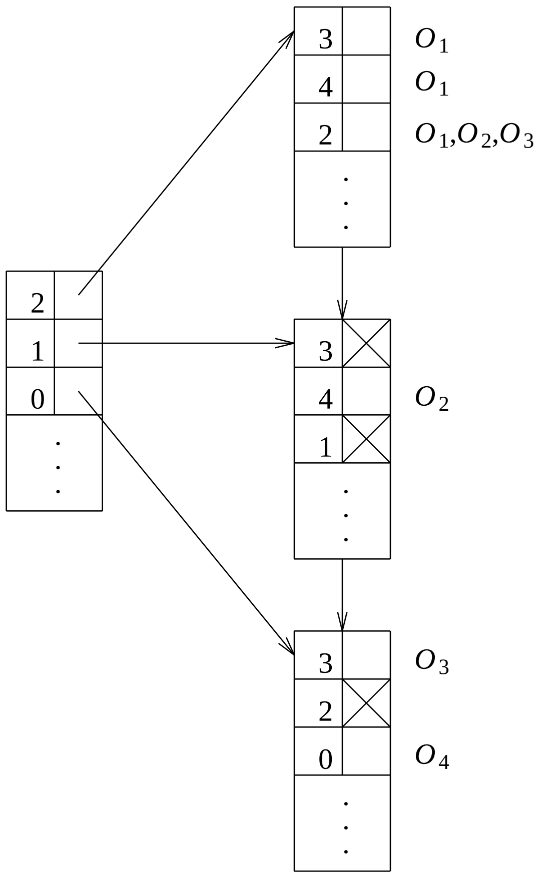
(a) e_{first} is the first element following the last entry of the previous node, or e_1 if it is the first node;
(b) the portion of entries e_{first} through e_{last} can fit on a block of storage; and (4.1)
(c) e_{last} is the entry *with the largest index e* such that its depth value is less than the depth value of each entry e_j, $first \leq j < last$.

Due to the part (c) of the rule above, blocks of storage will rarely be completely filled. Moreover, in extremely pathological cases it can happen that only one entry is assigned to a block, or node.[7]

Once the linear representation has been partitioned into blocks, the higher layer is obtained as a sequence of fixed-length elements $e_i = (depth_i, ptr_i)$, where ptr_i denotes to the i^{th} node of the lower layer, and $depth_i$ equals the *depth of the last entry in the node*, that is $e_{last}.depth$. If the higher level of the hierarchy does not fit in a block, than the rule (4.1) can be applied to it in order to obtain an additional level of hierarchy, etc. The **consistency criterion** for upper layers of a Q_0-tree is that the value $depth_i$ of each higher-level entry e_i is equal to the depth value of the last entry in the block indicated by ptr_i.

[6] In this report we assume all entries in a block at level > 1 are of the form $(depth, ptr)$, and all entries in a block at level = 0 have the form $(depth, o\_list)$. The member o_list may be interpreted as either a pointer to an object list represented elsewhere, or as the actual object_list itself. In either case o_list denotes the object list. Actually storing object lists within the leaf block itself is more efficient, in that it eliminates the need for a pointer and *an additional disk access*. The complication is that variable length entries must be accommodated.

[7] In the rest of the report we will use the terms node and block interchangeably.



Hierarchical Q_0-tree of the representation of Figure 2-3(d).
Figure 4-1.

Figure 4-1 shows a 2-level Q_0-tree obtained from the linear Q_0-representation of Figure 2-3(d). Here, the lowest-level leaf blocks are chained to facilitate easier traversal. Observe, the second layer of representation describes a bintree in which certain subtrees are replaced with single leaves. A "higher-level" entry corresponds to a block in the Q_0-tree, and the list of objects conceptually assigned to it is the set of all object id's appearing in leaf blocks of its sub-tree. We can similarly describe the third and even higher layers of the hierarchy. In that sense, every layer of the hierarchy corresponds to the same image, but with different resolution.

The rather unusual way of partitioning a linear Q_0-representation based on rule (4.1), in order to obtain the hierarchical Q_0-tree, is necessary to ensure that the search for a pixel will be confined to a single path in the tree. Thus, the number of blocks accessed to perform a pixel search will be equal to the number of layers in the Q_0-tree. If the Q_0-tree resides in secondary storage this translates to a fixed number of disk accesses. Each block accessed in the process is searched by the *search_block* procedure of Figure 3-5 which employed the transformation of the locational code of the specified pixel into the array B of its 1-bit positions (augmented with $M+1$). The root block is searched with $first = 1$, while the search of lower-level blocks begins with $first = k$, where k is set in the preceding invocation of the *search_block* procedure. The reader can verify that the search of the Q_0-tree of Figure 4-1 for the pixels 0011 and 1101 stops at the entries denoting the object lists <O_1,O_2,O_3> and <O_4>, respectively, as it should.

The above paragraph describes the simple way to search for all objects containing a specified pixel in the image space. Our next step is to show how one can find objects overlapping a *simple region* (given by a single locational code) of the image space, which is a natural generalization of the single pixel search problem. In essence, *simple region* search is different from pixel search in only one important respect. Each pixel of a specific image is enclosed by a simple homogeneous region (the region can contain only that particular pixel or be larger than the single pixel). Consequently, there will be a single leaf in the bintree of the image corresponding to the enclosing homogeneous simple region, and all that is required is to access the leaf. On the other hand, if the simple search region is larger than a pixel, then it need not be homogeneous on the actual image. In the corresponding bintree of the image it will be represented either by a single leaf (if the search region is homogeneous on the image), or by a subtree rooted at an interior node whose path is equal to the locational code of the search region (if it is not homogeneous on the actual image). Thus in general case, to respond to a simple region search one will have to access all leaves L_i whose $path(L_i)$: (a) *is a prefix* of the locational code of the simple search region; or (b) *has a prefix* equal to the locational code of the specified simple region. Observe, if case (a) applies then only one leaf has to be accessed, whereas in case (b) a whole subtree needs to be traversed.

Following this discussion, we can view the problem of finding all objects overlapping a simple region as a prefix-match retrieval problem. Accordingly, in the Q_0-representation of the image we would have to locate all entries whose corresponding "decompressed" codes (*icodes*), obtained by the recurrences (2.1) and (2.2), have prefixes equal to the locational code of the given simple region. Since such entries appear as a continuous portion of the Q_0-representation, one way to implement the prefix search is to locate the first entry satisfying the criterion, and then to access subsequent entries up to the last one whose corresponding *icode* has the given prefix. In a Q_0-tree the first entry in the order satisfying the query can be located by a pixel search based on the "decompressed" locational code of the given rectangle, i.e. the locational code padded with 0's up to the maximal length M. (Notice, appending $M+1$ to the vector B of 1-bit positions of the locational code is equivalent to constructing the corresponding *icode*.)

As in *search_block* procedure, we do not need to reconstruct the actual *icodes* to perform the prefix search. In order to explain this, the following observation is required. Closer inspection of a bintree (e.g. those of figures 2-2(b) or 3-2(a-b)), reveals that the 1-node following the last leaf in a 0-order traversal of a subtree is less than the depth of any 1-node within the subtree. It must be at a level of the bintree not exceeding the level of the root of the subtree (i.e. not exceeding the length l of the given locational code). On the other hand, all 1-nodes within the subtree itself must apparently be at a level greater than l. Consequently, once the first entry has been reached by the pixel search with the given decompressed locational code, all we have to do is to access the subsequent entries up to the one having the depth value less than l (inclusively).

The actual procedure *prefix_match* implementing the search in Q_0-trees for all objects overlapping a simple search rectangle is given in figure 4-2. Note that it is a generalized pixel search procedure in that it can be given the locational code of a pixel of length M and only the leaf corresponding to the homogeneous region enclosing that pixel will be accessed (the inner *while* loop of the procedure will not execute since no depth value $Q[i].depth$ can be greater than M). In general, the sequential scan of the targeted entries of the lowest level of a Q_0-tree can access

```
procedure prefix_match (Q_tree_root : <Q$_0$-block>,
                        B : <vector_of_1-bit_positions>,
                        lcode_length : integer ) : <object_list>;
        |  Given a lcode denoted by B of 'lcode_length' bits
        |  return a list of objects which overlap the specified lcode.
        var     i, first, level: integer;
                Q : <Q$_0$-block>;
                objects : <object_list>;
begin
first ← 1;
level ← Q_tree_depth;
Q     ← Q_tree_root;
while ( level >= 0) do
        begin
        i ← search_block (Q, B, first);
        if ( level > 0 )
           then  Q ← Q[i].ptr;
        level ← level - 1;
        end
objects ← Q[i].o_list;
while lcode_length < Q[i].depth do
        begin
        i ← i+1;
        if i > Q.nbr_entries
           then begin  Q ← next 0-level block; i ← 1 end
        objects ← objects ∪ Q[i].ptr;
        end;
return  objects
end;
```

Search for objects overlapping a simple region.
Figure 4-2.

several blocks of storage if the size of the targeted portion of the linear Q_0-representation spans over several blocks, but not a single false drop to a block in the tree occurs in the process. This is a rather remarkable property of Q_0-trees when compared to many other spatial structures, e.g. R-trees and cell trees.

With the *prefix_match* procedure, it becomes easy to describe the algorithms for all spatial queries, given an arbitrary shaped search region, which need not be connected. The simplest versions of these algorithms are given next.

Pixel Query ("Find all objects containing the given pixel in the image space"):

1. Construct the binary locational code *lcode*(p) for the given pixel p;
2. From *lcode*(p) construct B and record *lcode_len*;
3. Return prefix_match(*Q_root*, B, *lcode_len*);

Region Intersection ("Find all objects intersecting the given region"):

```
1.    Decompose the search region into a set $S = \{s_1, ... , s_k\}$ of simple rectangles and
          construct their locational codes $lcodes = \{lcode(s_1), ... , lcode(s_k)\}$;
2.    For each $lcode(s_i)$, construct $B_i$ and record $lcode\_len_i$;
3.    For each $i$, $1 \le i \le k$, do:  $O_i$ := prefix_match($Q\_root$, $B_i$, $lcode\_len_i$);
4.    Return  $O_1 \cup O_2 \cup ... \cup O_k$.
```

Region Enclosure ("Find all objects enclosing the given region"):

```
1.    Perform steps 1 and 2 as in Region Intersection;
2.    Return  $O_1 \cap O_2 \cap ... \cap O_k$.
```

Region Containment ("Find all objects enclosed by the given region"):

```
1.    Perform Region Intersection;
2.    For each object in the resulting set check whether it is completely contained
          by the search region, and if so include it in the return set $O$.
```

For the region containment queries the geometries of the stored objects (e.g. sets of locational codes of their constituent simple rectangles) must be stored somewhere separately. Other types of spatial queries do not require this. Notice that, although the pixel query implementation uses the *prefix_match* procedure, only one entry of the lowest-level representation will be accessed.

As in cell trees and zkd B-trees, the algorithms for region intersection, enclosure and containment queries involve a decomposition of the search region into its constituent parts (in our case those are *simple rectangles* instead of convex cells). The simplest way to perform the decomposition is to scan the image of the search object in the Morton ordering, grouping "black" pixels (i.e. those that are covered by the search region) into larger simple "black" regions. If a simple region cannot be expanded further its locational code is constructed. Then the rest of the pixels are examined analogously. The execution time of this algorithm is proportional to the number of pixels in the image. Alternative decomposition is by means of clipping.

There are many optimizations of the spatial-search algorithms presented above that avoid needless pixel searches down through the Q_0-tree for each locational code of the search region individually. We leave them to the reader as an exercise.

5. Image Updates

In highly dynamic situations it is desirable that the search structure be able to gracefully adjust itself to the changing environment through incremental growth and contraction. A single step in the incremental expansion or contraction of the structure is called insertion or deletion, respectively. In the growing process the structure will gradually acquire new blocks of storage, while during the contraction it may be useful to release blocks. In most multiway-tree search structures this is accomplished by splitting the overfilled nodes, or conversely by merging the underfilled ones, because it requires that only a small portion of the structure be updated during an insertion or deletion, respectively. In this section, we will be primarily concerned with the growth of Q_0-trees. Contraction is essentially an inverse process.

While we have not yet shown how a node of a Q_0-tree can overfill, we first demonstrate the splitting of an overfilled node into two blocks by means of the following rule:

(a) A node Q is split immediately after the entry e_i closest to the middle of the block, such that its depth d_i is less than the depth of each preceding entry in the block; (5.1)

(b) Insert a new entry into the upper layer above, such that the *consistency criterion* of upper layers of the Q_0-tree is preserved (see previous section).

If a higher-level block overfills it is split according to the same rule. The splitting can propagate up to the root node (in which case a new root is allocated), but it is not hard to see that only blocks on a single path in the tree are involved in the process. The rule (5.1) does not guarantee that the blocks will be split into equal parts, but it is a close approximation of even splitting of blocks which yields reasonably good storage utilization [Orl89].

As in zkd B-trees and cell trees, the insertion of a new spatial object in a Q_0-tree must be preceded by its decomposition into constituent parts. In the case of Q_0-trees, this will produce an ordered sequence, *lcodes*, of the locational codes of simple regions comprising the object. Then the object's identifier is inserted at each place in the structure which is "covered" by an element of *lcodes*. A place is "covered" by a locational code $lcode_i$ if it would be accessed by the *prefix_match* procedure of Figure 4-2 and no search with an $lcode_j$, which is not a prefix of $lcode_i$, will access it. In essence, the process can be viewed as one of "inserting" pairs ($lcode_i$, *object_id*) for each $lcode_i$ in the sequence *lcodes* for the given object. The effect is to introduce redundancy, similar to the one found in cell trees and zkd B-trees [Ore89], in order to achieve higher search performance.

The actual algorithm for "inserting" the pair (*lcode*, *object_id*) is given in Figure 5-1. To understand how the process works the reader should consider the analogue process on the concep-

```
procedure insert(Q_tree_root : <Q0-block>,
                B : <vector_of_1-bit_positions>,
                lcode_length : integer,  O_id : <object> );
       |  Insert the object 'O_id' whose location code, of length
       |  'lcode_length', is represented by 'B', into an image
       |  represented by a Q0-tree.
       var    d, i, leaf_depth  : integer;
              Q : <Q0-block>;
begin
Perform pixel_search with B, Q_tree_root;
Let Q[i] be the entry so accessed;
Compute the actual depth leaf_depth of Q[i] using rule (2.2);
if lcode_length ≤ leaf_depth
   then begin
       Q[i].o_list ← Q[i].o_list ∪ O_id;
       while lcode_length < Q[i].depth do
              begin
              i ← i+1;
              Q[i].o_list ← Q[i].o_list ∪ O_id;
              end
       end
   else
       begin
       d ← leaf_depth + 1;
       while d ≤ lcode_length do
              begin
              insert_new_entry (d, Q[i].o_list) immediately before Q[i];
                              | (d, Q[i].o_list) becomes the i^th entry of
                              | Q block, Q[i] becomes the i+1^st entry.
              if d ∈ B        | bit_d of lcode is 1
                 then  i ← i+1;
              d ← d+1
              end;
       Q[i].o_list ← Q[i].o_list ∪ O_id;
       end;
Perform necessary splits of overfilled blocks using rule (5.1);
end;
```

Insertion algorithm.
Figure 5-1.

tual bintree. First, perform the digital search in the bintree (e.g. the one of Figure 2-2(b)) with the given *lcode* (e.g. 0111 or 1101) to reach a node X. If the $path(L)$ fully spells the *lcode* than the *object_id* must be placed in each leaf of the subtree rooted at X. This is equivalent to performing a prefix search using *lcode* and then entering the *object_id* in every accessed leaf. The first branch of the *insert* procedure actually performs this task. On the other hand, if there is no path in the bintree that completely spells out *lcode*, then the accessed node X will be a leaf L with $path(L)$ being the proper prefix of *lcode*. In order to ensure that *lcode* has a node in the tree covered by it, we must replace L with a minimal subtree in which there is a leaf L' whose $path(L') = lcode$. All leaves in the newly obtained subtree (including L') inherit the content of the old leaf L, but only L' is assigned the new *object_id* as well. The second branch of the procedure *insert* emulates this process.

Similar reasoning will explain the procedure *delete*, given in Figure 5-2, which deletes a pair (*lcode*, *object_id*). As with insertion, deletion of an object also begins by constructing its set *lcodes* and performing the *delete* procedure for each *lcode* within. The algorithm is somewhat simplified by not taking into account the boundaries of the blocks in the structure.

6. Performance Evaluation

The problem of analyzing and comparing spatial access methods is in many ways the most troublesome one. There is a marked lack of formal performance measures to study and compare the behavior of different spatial methods. Therefore, choosing an appropriate standard and determining the comparison strategy would be a significant contribution to the research area. But, definition of an acceptable set of performance measures or a comparison strategy is fraught with difficulty. Alternative access methods differ significantly with respect to their objectives and underlying mechanisms. One group of methods may favor certain conditions under which some other methods may exhibit worst-case behavior. The assumption of uniform key distribution which is widely accepted for simplifying exact-match retrieval analysis, cannot be as readily

```
procedure delete(Q_tree_root : <Q_0-block>,
                 B : <vector_of_1-bit_positions>,
                 lcode_length : integer,  O_id : <object> );
        |  Delete the object 'O_id' whose location code, of length
        |  'lcode_length', is represented by 'B', into an image
        |  represented by a Q_0-tree.
        var    d, i, true_depth  : integer;
               Q : <Q_0-block>;
begin
Perform pixel search with B;
Let Q[i] be the entry accessed in the level 0 block Q;
Q[i].o_list ← Q[i].o_list - O_id;
coalesce_entries(Q, i);
while lcode_length < Q[i].depth do
        begin
        i ← i+1;
        if i > Q.nbr_entries
           then begin Q ← next 0 level block; i ← 1 end;
        Q[i].o_list ← Q[i].o_list - O_id;
        coalesce_entries(Q, i);
        end;
If necessary merge underfilled adjacent blocks maintaining
the consistency criterion of the upper layers of the tree;
end;


procedure coalesce_entries(Q : <Q_0-block>, i : integer);
begin
while i > 1 and Q[i-1].depth > Q[i].depth and Q[i-1].o_list = Q[i].o_list do
        begin
        Q[i-1].depth ← Q[i].depth;
        erase Q[i];
        i ← i-1
        end;
while Q[i].depth > 0 and Q[i].depth > Q[i+1].depth and Q[i].o_list = Q[i+1].o_list do
        begin
        Q[i].depth ← Q[i+1].depth;
        erase Q[i+1];
        end
end;
```

Deletion Algorithm.
Figure 5-2.

applied in the analysis of spatial methods, because it leads to simplifications that may not reflect reality. What is a uniform distribution of objects to a pixel space? For instance, if a uniformity assumption implies that pixels of the image form a "chessboard" pattern of square objects, then the methods based on bounding rectangles are clearly favored over those that stress precision in describing the objects. Compounding the problem is the fact that a formal analysis of any spatial method must take into account not only the distribution of objects on the image plane, but also their shape, texture, overlap and volume, as well as the shape and volume of the search region.

6.1. Basic Parameters

In the light of the preceding paragraph any formal performance evaluation of a spatial method, including the one for Q_0-trees, must inevitably be incomplete. Still, since Q_0-trees are an encoding of complete, or 0-complete, binary tries, several definitive conclusions regarding storage overhead and access performance can be made provided the number of leaves n in the corresponding bintree is known. However, it is precisely n that is the hardest to estimate, because it depends not only on the number of objects, but also on their shape and location.

The storage utilization of blocks in a Q_0-tree and its total storage overhead can be estimated by viewing the blocks of Q_0-trees as leaves in paginated binary tries (i.e. binary tries whose leaves are blocks of data) with n items. Then from [Fae79, OrP89] it directly follows that the expected storage utilization of blocks in a Q_0-tree asymptotically converges to $\ln 2 \approx 0.693$ as in B-trees [Yao78]. Consequently, the total size of the lowest level of the Q_0-tree will be approximately $(1/ln\ 2)\cdot len \cdot n \approx 1.443 \cdot len \cdot n$, where len is the entry length[8] and n is the number of leaves in the corresponding bintree (i.e. the number of entries in the index blocks). Upper layers of a Q_0-tree are rather small and typically constitute no more than 1 or 2 percent of the total size of the tree. Thus, the formula:

$$S(n) \approx 1.47 \cdot len \cdot n,$$

gives a good estimate of the total Q_0-tree storage requirements.

As in hB-trees, but unlike cell trees and zkd B-trees, the number of disk accesses per pixel search in a Q_0-tree is always equal to the number of layers in the tree. For a balanced multiway tree, such as B-trees and Q_0-trees, the depth of the tree structure is a function of n which exhibits a stepwise growth. It is not hard to establish that the expected maximal number of entries[9],

[8] For purposes of this report we assume that len = 6, with 2 bytes for the depth and 4 bytes for the pointer. In some applications, a 1 byte depth and 3 byte pointers may be sufficient. If to avoid an extra disk access, as suggested in section 4, the entire o_list is represented in the Q block entry instead of just a pointer to it, then len must denote the *average* entry length.

$Max(k)$, in a k-level Q_0-tree can be given by:

$$Max(k) \approx (C \cdot \ln 2)^k,$$

where C is the absolute maximal number of entries per block, or *capacity* of a block. With a physical block size of 1K bytes, $C = 170$. Thus, even for $n = 1{,}635{,}000$ the depth of a Q_0-tree will still be only three levels and this will be the case in most realistic situations.

6.2. Qualitative Comparisons

To estimate the expectation for the parameter n, i.e. the number of entries in the lowest level of the Q_0-tree (or alternatively, the number of simple, homogeneous regions on the image), one would have to consider many different situations. It does not depend solely upon the number of spatial objects on the image, but also upon their shape and location. If objects have "non-regular" shape they would have to be decomposed into simpler pieces, each of which is stored separately. This occurs in cell trees and zkd B-trees as well. However, the difference is that the individual pieces in cell trees are arbitrary convex cells, while both Q_0-trees and zkd B-trees are much more restrictive in that they require each piece to be a simple region which can be uniquely identified by a single locational code. Thus, even some squares and rectangles within the objects may have to be decomposed, depending upon their location on the image.

In addition to this, Q_0-trees introduce somewhat more redundancy by storing in each entry the identifiers of all objects overlapping the simple homogeneous region denoted by the leaf. This was done in order to accommodate faster and simpler spatial search. Finally, in contrast to zkd B-trees, Q_0-trees keep some entries corresponding to the empty regions on the image. Because of this, Q_0-trees do not lend themselves well for handling single pixel objects, since in that case corresponding bintrees tend to have many more empty leaves than the full ones (c.f. Fig-

[9] By *expected maximal* we mean the expected number of entries in the lowest level blocks, i.e. leaf entries, just before an insertion will force splitting the root block and creation of another level in the tree.

ure 3-2(a)).[10]

Cell trees, zkd B-trees and Q_0-trees each achieve precision by introducing redundancy in the accessing structure. The least redundancy occurs in cell trees; it is solely a result of the shape and texture of objects. In zkd B-trees redundancy occurs as a result of shape, texture and the locations of objects in the image space. In Q_0-trees, however, redundancy is introduced not only because of the shape, texture and locations of objects, but also because of the overlap of different objects. Moreover, Q_0-trees must keep some entries representing the empty pixels of the image. The situation is reversed when considering the lengths of individual entries. Q_0-trees need at most 6 bytes per entry, zkd B-trees usually 8 bytes, while for cell trees a typical entry length of 20 bytes was reported [GuB91].

Considering all this we can contrast anticipated behavior of Q_0-trees and zkd B-trees in different situations. For heavily populated images, with not much overlap between the spatial objects (usually geographic maps are such), Q_0-trees can be expected to perform better, both in terms of storage overhead and in terms of execution performance. For moderately sparse objects similar storage overhead will most probably be reported, but the performance of spatial queries should generally be in favor of Q_0-trees. For very sparse objects, or heavily overlapping objects, it is likely that zkd B-trees will have lower storage overhead, but not necessarily better execution performance. Notice, overlap is handled nicely in zkd B-trees in terms of storage overhead, but it has an adverse effect on execution time. With Q_0-trees the situation is just the opposite. For single pixel objects we would expect zkd B-trees to perform much better.

Things are not that simple when trying to contrast cell trees to either Q_0-trees or zkd B-trees. As we noted earlier, their entries seem to be extremely long, significantly longer than those

---

[10] However, all empty leaves of bintrees need not be represented in the actual Q_0-tree. As noted in section 2, some empty 0-leaves can safely be removed to create a 0-complete bintree. This will have little affect on processing algorithms; only the update procedures must be modified slightly (in fact simplified). Further details can be found in [OrP88, Orl89].

of zkd B-trees and Q_0-trees. Their processing algorithms are also more complex. But generally, one would expect fewer entries in cell trees than in either of the other two methods. In consequence, it is unclear when cell trees will be inferior, superior, or comparable to Q_0-trees in terms of storage and processing cost. With cell trees much depends upon the decomposition algorithm used, which is non-trivial and computationally expensive if maximal convex cells are to be extracted from objects.

6.3. Controlling Representational Redundancy

Since, the critical parameter, which governs the performance of the structure, is the number of leaves in the conceptual bintree, it would be desirable to have some control over it. This boils down to the question on how much redundancy should be included in the spatial structure.

As noted by Orenstein [Ore89], methods based on z ordering, i.e. Morton ordering (and a Q_0-tree is one of them), are unique in the sense that they can effectively control redundancy. The simplest way of doing this is to constrain the degree of resolution, i.e. to constrain the process of decomposition of the spatial objects into smaller pieces, thus trading accuracy for space efficiency. All of the methods for controlling redundancy proposed in [Ore89, Ore90] are applicable for Q_0-trees as well, but their effects might be somewhat different from those observed for zkd B-trees. Real experimental studies should be conducted to determine which strategy to redundancy in Q_0-trees works best.

Reduction of redundancy in Q_0-trees, however, has several adverse effects. It reduces accuracy of representation so that the structure becomes a "filter" which does not select precisely those objects that do satisfy the spatial query, but rather selects out those that cannot possibly belong to the targeted set. Consequently, false drops are introduced and additional checks are needed to answer a spatial query. With a full Q_0-tree, the false drop of a image object falling completely outside the given search region will never occur for any spatial search query, except possibly region containment queries. Only leaf entries, falling within the search region or intersecting it, are ever accessed. This is a unique property of Q_0-trees, not found in cell trees or zkd B-trees, and definitely not in schemes based on bounding approximations.

The main advantage of keeping the original degree of resolution is that it preserves the isomorphism between the original image and its representation. This enables many useful operations (e.g. set operations, image display and transformations, area and moment computations, and especially similarity assessment) to be directly performed on the Q_0-representation, rather than on a separate "original image".

By maintaining the original degree of resolution the lowest level of a Q_0-tree is a one-to-one match of the original image. However, the storage used to keep a Q_0-tree is not proportional to the number of pixels on the image, but to the number of simple, homogeneous regions comprising it. This is normally more than sufficient to compensate for the storage expansion factor of about $\log_2 e \approx 1.443$ and the explicit maintenance of small depth values within the entries of Q_0-trees. Binary operations on Q_0-trees (e.g. set operations and similarity assessment for common overlap between two images) take essentially $O(N_1+N_2)$ disk accesses, where N_1 and N_2 denote the numbers of blocks in the lowest layers of the two structures involved. Unary operations (e.g. spatial search, image display, scaling by the power of 2, rotation by 90^0, etc.) require $O(logN)$, $O(N)$ or $O(N\log N)$ disk accesses, where $O(logN)$ is essentially a constant, that is 2 or 3 in most realistic situations. Even if an operation is more easily performed on the original image, it can always be restored using $O(N)$ disk accesses.

7. Summary and Discussion

In this report we introduced a tree structure which can serve both as a method for encoding images, and as a region search mechanism in large spatial databases. It performs both tasks exceptionally well. As an encoding scheme, it is a compact, lossless representation of images, which can be viewed as a variant of region quadtrees. It fully preserves the functionality of quadtrees, but as illustrated in section 3, operations on Q_0-representations are simpler and usually faster than on the equivalent quadtrees. Consequently, this method also enhances the functionality of quadtrees. For region retrieval in large spatial databases, we have extended the basic Q_0-representation to a hierarchical, balanced, multiway Q_0-tree structure, which has an efficient memory-management mechanism. Lowest level leaf blocks of a Q_0-tree encode a complete description of the image, while upper layers represent the same image with lower resolution.

Q_0-trees have a number of attractive properties, some of which are unique among the methods for spatial retrieval. (1) They are a dynamic structure. They can be constructed directly from the image or grown incrementally, through insertions and deletions. (2) Retrieval incurs no false drops. Entries denoting objects lying outside the search region, will never be examined. This has been achieved, even though objects and search regions can be of arbitrary shape and texture. (3) Pixel query will always require exactly only d disk accesses, where d (normally ≤ 3) is the number of layers in the Q_0-tree. (4) Pixel, region intersection, and region enclosure queries require no additional storage access, other than those made while processing the spatial structure. Only region containment queries require separate to object geometries.

The properties of Q_0-trees listed above have been achieved by introducing a measure of redundancy in the structure. An object identifier may appear at many places within the structure. As indicated in the previous section, it is possible to reduce the redundancy to achieve higher storage efficiency. All of the methods for reducing redundancy in zkd B-trees can be applied to Q_0-trees as well. However, this also sacrifices a number of the advantages of Q_0-trees, i.e. accuracy, no false drops, no additional checks, and especially the isomorphism with the original image.

Storage and execution performance is not the only aspect on which evaluation of a spatial access method should be based. Aside from spatial search there are many other activities occurring in large spatial databases. Huge spatial images may have to be merged, "joined" (i.e. intersected), assessed for their similarity, transformed in one way or the other, etc. Because a Q_0-tree preserves an isomorphic copy of the original image, and because these operators can be performed directly on the Q_0-representation, *no additional space is needed to store the image itself, and no functionality is lost*. This is the real storage savings offered by the scheme.

Except perhaps for zkd B-trees, other spatial access methods cannot provide the functionality of Q_0-trees. Methods based on bounding rectangles are imprecise and their accessing structures never correspond to the original image. Cell trees are not suited for performing intersections and similarity assessments between different spatial images. Even if zkd B-trees are used as the isomorphisms of their respective images, the operations on them will not nearly be as simple and efficient as on Q_0-trees. They would involve repeated bit extractions and bit manipulations, whereas on Q_0-trees simple integer comparisons and arithmetic will suffice.

In summary, Q_0-trees are a novel, efficient access method for large spatial databases, capable of performing tasks that most other methods are unable to do. Future work on Q_0-trees includes their implementation, extensive experimental studies, cost-benefit assessment of different redundancy control methods, and experimental (and to some extent analytical) comparison with various different spatial access methods.

References

[Com79] D. Comer, The Ubiquitous B-Tree, *Computing Surveys 11*,2 (June 1979), 121-137.

[Fae79] R. Fagin and et.al., Extendible Hashing---A Fast Access Method for Dynamic Files, *Trans. Database Systems 4*,3 (Sep. 1979), 315-344.

[Fre60] E. Fredkin, Many-way Information Retrieval, *Comm. of the ACM 3*(1960), 490-500.

[Gar82] I. Gargantini, An Effective Way to Represent Quadtrees, *Comm. of the ACM 25*,12 (Dec. 1982), 905-910.

[GuB91] O. Gunther and J. Bilmes, Tree-Based Access Methods for Spatial Databases: Implementation and Performance Evaluation, *IEEE Trans. Knowledge and Data Engineering 3*,3 (Sep. 1991).

[Gut84] A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, *Proc. ACM SIGMOD Conf. on Management of Data*, Boston, MA, 1984, 47-57.

[KaE80] E. Kawaguchi and T. Endo, On a Method of Binary Picture Representation and Its Application to Data Compression, *IEEE Trans. Pattern Anal. Mach. Intell. 2*,1 (Jan. 1980), 27-35.

[KEM83] E. Kawaguchi, T. Endo and J. Matsunaga, Depth-First Expression Viewed from Digital Picture Processing, *IEEE Trans. Pattern Anal. Mach. Intell. 5*,4 (July 1983), 373-384.

[Lit80] W. Litwin, Linear Virtual Hashing: A New Tool For Files and Tables Implementation, *Proc. 6th Conf. on VLDB*, Montreal, Canada, Oct. 1980, 212-223.

[LoS90] D. Lomet and B. Salzberg, The hB-Tree: A Multi-Attribute Access Method with Good Guaranteed Performance, *Trans. Database Systems 15*,4 (Dec. 1990), 625-658.

[NHS84] J. Nievergelt, H. Hinterberger and K. C. Sevcik, The Grid File: An Adaptable, Symmetric Multikey File Structure, *Trans. Database Systems 9*,1 (Mar. 1984), 38-71.

[OrM84] J. A. Orenstein and T. Merret, A Class of Data Structures for Associative Searching, *Proc. ACM SIGACT News-SIGMOD Conf. Principles Database Sys.*, Waterloo, Canada, 1984, 181-190.

[Ore89] J. A. Orenstein, Redundancy in Spatial Databases, *Proc. ACM SIGMOD Conf. Management of Data*, Portland, OR, 1989, 295-305.

[Ore90] J. A. Orenstein, A Comparison of Spatial Query Processing Techniques for Native and Parameter Spaces, *Proc. ACM SIGMOD Conf. Management of Data*, Atlantic City, NJ, 1990, 343-352.

[OrP88] R. Orlandic and J. L. Pfaltz, Compact 0-Complete Trees, *Proc. 14th VLDB Conf.*, Long Beach, CA, Aug. 1988, 372-381.

[OrP89] R. Orlandic and J. L. Pfaltz, Analysis of Compact 0-Complete Trees: A New Access Method to Large Databases, in *Proc. 7th FCT Conf., Szeged, Hungary*, Springer-Verlag, Berlin-Heidelberg-New York, Aug. 1989, 362-371.

[Orl89] R. Orlandic, *Design, Analysis and Applications of Compact 0-Complete Trees*, PhD Dissertation, Univ. of Virginia, May 1989.

[ROG88] D. Rhind, S. Openshaw and N. Green, The Analysis of Geographical Data: Data Rich, Technology Adequate, Theory Poor, *Proc. 4th Conf. SSDBM, in Lecture Notes in Computer Science 339*(June 1988), 427-454, Springer Verlag.

[Rob81] J. T. Robinson, The k-D-B-Tree: A Search Structure for Large Multidimansional Dynamic Indexes, *Proc. ACM SIGMOD Conf. on Management of Data*, Ann Arbor, MI, 1981, 10-18.

[Sam84] H. Samet, The Quadtree and Related Hierarchical Data Structures, *Computing Surveys 16*,2 (June 1984), 187-260.

[SeK88] B. Seeger and H. Kriegel, Techniques for Design and Implementation of Efficient Spatial Access Methods, *Proc. 14th VLDB Conf.*, Long Beach, CA, Aug. 1988, 360-371.

[SRF87] T. Sellis, N. Roussopoulos and C. Faloutsos, The R+-Tree: A Dynamic Index for Multi-Dimensional Objects, *Proc. 13th Conf. on VLDB*, Brighton, England, 1987, 507-518.

[SiW88] H. W. Six and P. Widmayer, Spatial Searching in Geometric Databases, *Proc. Conf. on Data Engineering*, 1988.

[Tam84] M. Tamminen, Comment on Quad- and Oct-trees, *Comm. of the ACM 27*,3 (Mar. 1984), 248-249.

[Wan91] F. Wang, Relational-Linear Quadtree Approach for Two-Dimensional Spatial Representation and Manipulation, *IEEE Trans. Knowledge and Data Engineering 3*,1 (Mar. 1991), 118-122.

[Yao78] A. C. Yao, Random 3-2 Trees, *Acta Inf. 2*,9 (1978), 159-170.

Table of Contents