A Paging Scheme for Pointer-Based Quadtrees

by

Patrick R. Brown

Thesis submitted to the faculty of the

Virginia Polytechnic Institute and State University

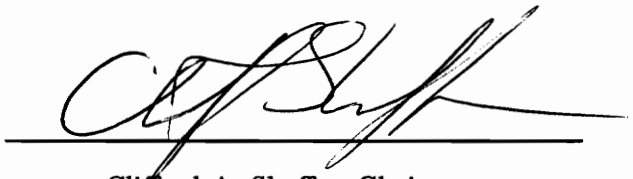
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

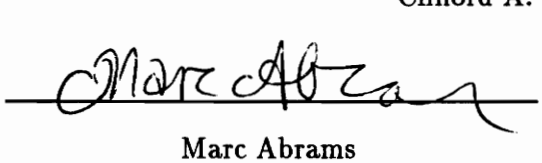
in

Computer Science and Applications

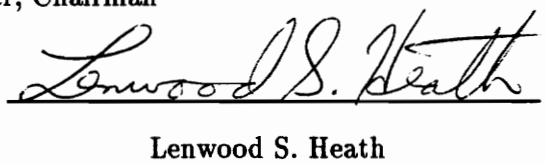
APPROVED:



Clifford A. Shaffer, Chairman



Marc Abrams



Lenwood S. Heath

May, 1992

Blacksburg, Virginia

A Paging Scheme for Pointer-Based Quadtrees

by

Patrick R. Brown

Committee Chairman: Clifford A. Shaffer

Computer Science

(ABSTRACT)

The quadtree is a family of data structures that organize spatial data using recursive subdivision. A *pointer-based quadtree* uses an explicit tree structure to represent the subdivision, while a *linear quadtree* holds a sorted list of records corresponding to the leaves of the tree structure. Small quadtrees are typically represented using pointers since this leads to simpler algorithms. However, linear quadtrees have been historically used to represent larger data sets. The primary reason is that linear quadtrees are easily organized on pages in disk files. In addition, linear quadtrees were thought to require less space than pointer-based quadtrees. Though pointer-based quadtrees have many other advantages, there has still been much interest in the linear quadtree.

This thesis presents a pointer-based representation for quadtrees called the *paged-pointer quadtree*. The paged-pointer quadtree overcomes both of the historical advantages of the linear quadtree. It partitions the nodes of a pointer-based quadtree into pages, stores these nodes in order, and manages pages using B-tree techniques. In addition, a paged-pointer quadtree always requires less space than a corresponding linear quadtree. Our representation overcomes the performance problems associated with representing traditional pointer-based quadtrees on disk. As a result, our implementation produces better performance than highly optimized systems based on linear quadtrees.

ACKNOWLEDGEMENTS

First, and most importantly, I would like to thank my wife, Cynthia, for her undying support of both this work and its author.

I would also like to thank those fellow students with whom I have shared both time and office space, including Tim Ryan, Mark Lattanzi, Mahesh Ursekar, Vincent Miranda, and Sheryl Kriss. Thanks also to my good friend Joe Lavinus.

I would like to acknowledge the technical support of my principal advisor, Dr. Clifford Shaffer, without whose knowledge and advice this work may not have succeeded. I would like to thank Dr. Robert Webber, whose insights provided a significant contribution to this work. Thanks also to the other members of my committee, Dr. Lenwood Heath and Dr. Marc Abrams, for their suggestions regarding this work.

Finally, I am grateful to the National Science Foundatation, which provided me with a Graduate Fellowship and without whose financial support this work would not have been possible.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

This thesis describes the design and implementation of the paged-pointer quadtree, an effective disk-based representation for pointer-based quadtrees. The *quadtree* [Sam90a, Sam90b] is a family of data structures that represent spatial data using recursive subdivision. The quadtree subdivides an image into quadrants, which themselves may be further subdivided until some criterion for stopping the subdivision is met. The subdivision criteria used depend on the type of data to be represented. For example, in an array of pixels, subdivision could stop when all the pixels in a quadrant are of the same color. Figure 1.1 shows this process applied to a simple black and white image.

For spatial data, the quadtree has several advantages over more obvious representations. It organizes an image spatially, so that the contents of a small portion of the image can be determined without examining the entire structure. More importantly, since the subdivision process stops when "homogeneous" subimages are encountered, the quadtree can represent these (possibly large) areas as units. This can save a significant amount of storage space in many cases, thus saving time due to less data being processed.

There are several well-known representations of quadtrees. The *pointer-based quadtree* represents an image as a collection of *nodes* corresponding to the subimages resulting from the subdivision process. Pointers are used to organize these nodes into a tree structure.

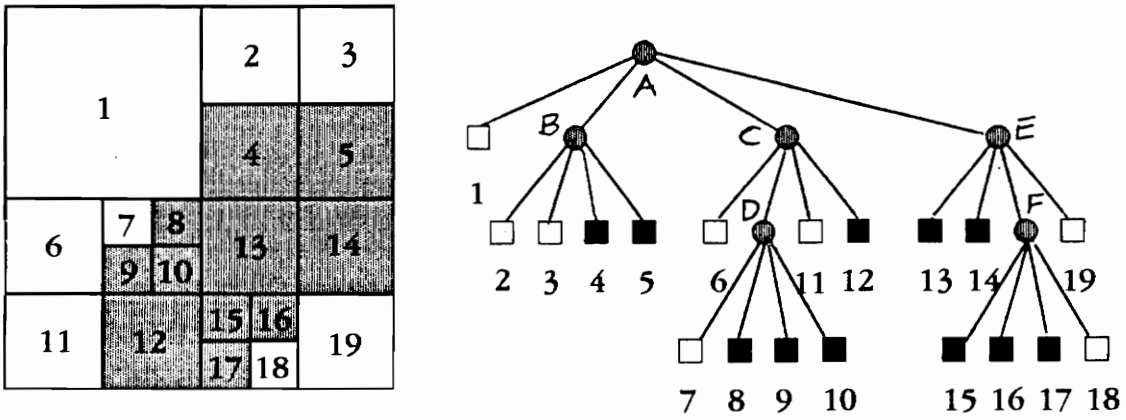


Figure 1.1: The Region Quadtree

On the other hand, the *linear quadtree* [Gar82] decomposes the image into a collection of records corresponding to the homogeneous parts of the image. These records are represented as a list sorted by location. Because the linear quadtree avoids the overhead involved in representing pointers and internal nodes, the linear quadtree has generally but incorrectly been thought to require less storage space than the pointer-based quadtree. In addition, linear quadtrees are easy to represent on disk because they reduce the tree structure to a simple sorted list. Because of these advantages, linear quadtrees have been used in several experimental geographic information systems [Abe84, SPM87, SSN87]. This thesis presents a pointer-based representation that is organized on disk and requires less storage space than a linear quadtree.

Chapter 1 introduces well-known concepts and methods used in database applications, considers the nature of spatial data, and explores techniques used to represent such data. Chapter 2 focuses on the quadtree as a representation method and describes several competing representations for quadtrees, including the pointer-based quadtree. It also compares the amounts of storage required by these representations. Chapter 3 proposes criteria for representing quadtrees and describes the design and implementation of a method of mapping pointer-based quadtrees to pages on a disk. Chapter 4 presents several quadtree algorithms

and compares the performance of our implementation of the paged-pointer quadtree to the University of Maryland's QUILT [SSN87], a geographic information system based on linear quadtrees. Chapter 5 presents the conclusions of this research and suggests future directions for this work.

1.1 Uses of Spatial Data

There are numerous applications that require the storage and manipulation of *spatial data*. Spatial data are characterized as a collection of data organized somehow in a space that contains it. The containing space is referred to as an *image space* or simply as an *image*. There are many application areas that require the processing of spatial data.

There is currently much interest in quadtrees and hierarchical data structures in the field of *geographic information systems* (GIS). Computer-based mapping provides several advantages over traditional paper-based methods. Since spatial data (e.g., cities, rivers) are recorded on-line, it is much easier to modify the data stored. Geographic information systems can quickly extract the information stored. Automatic processing allows extensive analysis of this data as well. An efficient GIS must be able store a large amount of data and extract it quickly. Hierarchical data structures such as the quadtree provide the organization necessary to achieve this goal [Bur87, SSN87].

Much computer graphics research has been involved with the realistic rendering of scenes. All such applications require a model of the objects existing in the space to be rendered. For example, a technique known as *ray tracing* follows rays of light backward, starting at the "eye" (i.e., viewpoint) and moving back to sources of light. Rays will often reflect off of or be absorbed by objects in the model. In order to achieve adequate performance, a ray tracing program must be able to quickly determine which objects each ray will hit. Spatial subdivision methods such as the quadtree have been successful in obtaining good performance in graphics applications [Gla84, HSA91].

In robotics, a model of the environment surrounding a robot is required in planning

its actions. For example, a robot must avoid collisions with the obstacles in its way. In addition, it must be able to find a path between places while avoiding these obstacles. To do this, extensive processing of the environment model is necessary. The representation of the model will have a major impact on the time required to do this processing. Quadtree-based approaches have also been successfully used in this field [SH92].

All of these applications share the common property that they work with data organized in space. The nature of the actual data may be very different—pixels make up a simple rectangular grid, while the objects in a robot's environment model may be much more complicated. Still, each application requires a collection of data items organized in space. Since the models used in many of these applications have a "permanent" nature, the data involved must be stored on a permanent medium. *Database systems* have been researched and developed for many years and this experience lends some insight into how to store spatial data.

The remainder of this chapter examines ways in which spatial data can be represented. The literature on databases is extensive, and studying these systems will give some insight on how to implement a spatial database system. Section 1.2 identifies many of the central concepts of database systems and how they apply to spatial data. Section 1.3 briefly explains the considerations involved in implementing a database package and describes the *B-tree* [Com79], from which our paging scheme borrows several concepts. Section 1.4 discusses the quadtree as a family of hierarchical representations for spatial data. Section 1.5 discusses possible representations for spatial data, concentrating on hierarchical methods such as the quadtree.

1.2 Database Concepts Applied to Spatial Data

Atre [Atr88] identifies seven key terms used to describe data and its representation.

- An *enterprise* is an organization requiring the data in question.
- An *entity* is a person, place, thing, or concept about which information is recorded.

- Each entity has one or more *attributes* that completely characterize the entity for the purposes of the enterprise. Attributes are sometimes also called *data elements*.
- A *key attribute* uniquely identifies an entity. Unique identifiers such as Social Security numbers often serve as key attributes.
- An *attribute value* is the actual data contained in an attribute. For example, "last name" may be an attribute for a person-entity, while "Brown" might be its attribute value.
- A *data record* is a collection of values taken by the attributes of a given entity. Each data record describes a single entity.
- A *data file* is simply the collection of data records used by the enterprise in question.

As an example, one class of enterprise concerned with spatial data are users of geographic information systems. Entities in a GIS might include rivers, cities, towns, and road links. The position and shape of an entity as well as other desired properties (e.g., city size, city name, road capacity) comprise the attributes of a spatial entity. While a unique identifier could serve as the key attribute, the location of the entity is often more meaningful and commonly serves as the key.

Any of the attributes can be used to locate entities in the database. However, entities are frequently found using only location within an image space. For an example, a restaurant chain looking to build a new restaurant might use a GIS to locate and evaluate several possible building sites within a city. A driver unfamiliar with an area might want to find the roads constituting the quickest path to his destination. While other attributes are often used in a query (e.g., locating a city by name), the location of an entity is used so often that it should serve as the key attribute. A spatial database system cannot ignore the need for efficient data access by location.

1.3 Implementation of Database Systems

The purpose of database systems, spatial or otherwise, is to store the data used by the enterprise. Extracting such data is typically done by *queries*. Knuth [Knu73] identifies three basic types of queries:

1. A *simple query*, which gives a specific value of a specific attribute.
2. A *range query*, which gives a specific range of values for a specific attribute.
3. A *Boolean query*, which consists of the previous types of queries combined by the Boolean operations **AND**, **OR**, and **NOT**.

For example, a simple query might be "find all customers in the database named Brown". An example range query would be "find all customers with reported incomes between $30,000 and $60,000." "Find all female customers in Virginia with incomes over $45,000" would be an example of a Boolean query. One of the primary objectives of database systems is to minimize query response time in each of these cases. The data structure used to represent the data is an important factor in query response time.

A list of records sorted by the value of the key attribute allows a database system to quickly respond to simple and range queries involving the key attribute. *Inverted files* (e.g., in [Knu73]) are useful in speeding up simple and range queries in which non-key attributes are used. However, Boolean queries involve several different attributes in combination. While searching inverted files and sorted lists can often eliminate many records from consideration, all remaining records must be exhaustively processed. Even the simplest range queries involving location reduce to Boolean queries. For example, finding all large cities within 400 miles of Omaha, Nebraska require that both the latitude and longitude of a city be in a given range.

Of course, query response time is not the only measure of the performance of a database system. The data stored in a database will often change. Customers will move; the population of a city will change; new entities will appear. The time required to make these changes is another important performance metric. Most modern database systems are based on the *B-tree* [Com79]. The B-tree has been found to provide excellent performance in each of the four basic operations identified by [Com79]:

- *Insert* adds a new data record to a file, also checking to ensure that the record's key is unique.
- *Delete* removes the data record with a specified key attribute value.

- *Find* retrieves the data record with a specified key attribute value.
- *Next* retrieves the record whose key value immediately succeeds that of the current record.

The B-Tree

The central motivation for data structures such as the B-tree is the fundamental difference between accessing main memory and permanent storage (e.g., magnetic disks). In main memory, units as small as a byte are often individually addressable. On the other hand, the physical space on a disk is typically divided into units known as *sectors*. Disk files are commonly divided into units called *pages*, which map directly to sectors. Page sizes vary from machine to machine, but are at least 512 bytes in modern systems. To access *any* data item stored on a particular page, the corresponding sector must be retrieved from disk. However, if more than one item on the same page is used, retrieving the first item also retrieves all other items on that page. To exploit this fact, a page can be treated as a unit, rather than a collection of data items considered individually.

Another major difference between main memory and permanent storage is the time required to access data. Accessing main memory requires the CPU to retrieve the data electronically from the proper memory bank. To retrieve a sector from disk, mechanical heads must first move to its location on the disk and then read it. Because of this, the time required to access a disk sector is very high. In a database system, this cost often exceeds that of all other computations the system performs. Acceptable performance is achieved by minimizing the number of disk accesses.

The B-tree is a multi-way search tree designed to be used on disk. In a multi-way search tree, each node holds $n - 1$ keys used in comparisons and pointers to n children. The binary search tree is a special case of a multi-way search tree (where $n = 2$). As in a binary tree, a record in a multi-way search tree can be located by comparing the value of its key attribute with the keys stored in the tree. Within a node, determining the subtree holding a particular record requires $O(\log n)$ comparisons. These comparisons require more

time than following links between nodes in main memory. However, the time required for comparisons is small compared to the time required to fetch a new node from disk. In a B-tree, the value of n is typically chosen so that the size of each node of the tree is equal to the page size.

One significant barrier to using binary search trees or their variants on permanent storage is problem of *fragmentation*. Fragmentation refers the unused space resulting from the addition and deletion of records. *Compaction* is a process that eliminates holes by moving data to consecutive locations in memory. This is the same idea used in many garbage collection algorithms working in main memory. Unfortunately, compaction can be quite slow, since it involves traversing the entire data set. The B-tree uses another approach, reorganizing its pages whenever a record to be deleted causes the amount of data stored on a page to drop below a preset threshold. When this happens, "neighboring" pages are examined and records are moved between pages to reduce the amount of fragmentation. When a record is added to a full page, records are moved to make way for the new record. In either case, only local changes to the affected area of the tree are required. The standard B-tree guarantees that each page used will be at least half full. A variant known as the B*-tree guarantees that each page will be at least two-thirds full.

While B-trees are excellent in organizing sorted lists, not all of its techniques are applicable to pointer-based quadtrees. For instance, the internal structure of the B-tree adds an index to organize its collection of records. On the other hand, the decomposition of an image defines the structure of the corresponding pointer-based quadtree. In a B-tree, records can be moved between pages with only local changes to the structure. However, each node moved in a pointer-based quadtree requires updating pointers to reflect the node's new location. Despite the differences, the problem of fragmentation handled nicely by the B-tree is also found in pointer-based quadtrees. Whenever there is significant fragmentation on a page, our approach relocates nodes in a manner similar to that used in the B-tree.

1.4 Hierarchical Data Structures and the Quadtree

Samet [Sam90a] uses the term *quadtree* to describe a family of hierarchical subdivision methods:

> The term *quadtree* is used to describe a class of hierarchical data structures whose common property is that they are based on the principle of recursive decomposition of space. They can be differentiated on the following bases:
>
> 1. The type of data they are used to represent.
> 2. The principle guiding the decomposition process.
> 3. The resolution (variable or not).

Note that the generic term quadtree does not necessarily imply that each subdivided region is split into four equal-sized quadrants. In three dimensions, a subdivision into eight identical parts is typically used, and this representation is called an *octree*. In addition, the subdivision may be done along alternating axes, resulting in a binary tree-like structure. Such a subdivision is referred to as a *bintree*.

The type of hierarchical data structure called for depends on the type of the data represented. For example, data in an image represented as an array of pixels is distributed uniformly throughout the image space. For such uniform distributions, dividing an image into several equal-size parts will create a hierarchy that can be searched quickly. On the other hand, point data (e.g., a collection of cities) may not be uniform. For example, most large cities in the United States are found in the eastern half of the country. Dividing the U.S. into parts of equal area will produce a lopsided distribution, one in which some regions contain very few points while others contain a large number.

The decomposition processes used in hierarchical representations come in two distinct flavors. These approaches are called *image-space hierarchies* and *object-space hierarchies*. This classification is discussed in [Sam90a] and introduced in a different context in [SSS74]. In image-space hierarchies, an image is typically decomposed into parts of equal size. The foremost example of an image-space hierarchy in non-spatial applications is the *trie* (e.g., [Knu73]). A trie used with a collection of words would divide them into groups based on

the first letter. The words with the same first letter are then divided using subsequent letters of the words. In object-space hierarchies, the process of decomposition is governed by the input. The most common example of an object-space hierarchy is the binary search tree. In this kind of tree, the first record inserted into the tree serves as the root. The subsequent records are inserted into one of two subtrees based on a key comparison with the root element.

There are also different hierarchical representations for fixed or variable resolution images. For fixed-resolution images, the decomposition process can be specified in advance, which is not always possible with variable-resolution images. However, representations for variable-resolution images have the advantage of being more general. A pointer-based representation supports variable-resolution images, since subdivision will proceed and the tree will deepen as far as necessary. For variable-resolution images, there is no conceptual difference between dividing the whole image into quadrants and further dividing one of these quadrants. For fixed-resolution images, though, subdivision must stop when the limits of resolution are reached.

1.5 Representation of Spatial Data

This section examines several representations for *pixel* (picture element) and point data. Pixel data in two dimensions or *voxel* data in three dimensions consist of a grid of fixed-size rectangular elements. Such images are typically a fixed-resolution discrete approximations of continuous images and are encountered quite often in computer graphics and image processing. These data have the property that for every pixel in the image, there is a corresponding collection of attribute values. Because of the uniform distribution of data, image-space hierarchies effectively represent pixel data. Object-space hierarchies are not as useful, since there is no inherent advantage in dividing an image relative to any particular point.

While pixel data is uniformly distributed and of fixed resolution, point data has neither

of these properties. It is simply a collection of data records corresponding to discrete points in the image space. For example, large American cities might be represented as point data by reducing each city to a single point. These points may not be uniformly distributed across the image. Some parts of the image may contain few points while others might have many. In addition, when points are very close to one another, image-space hierarchies will require many levels but store little actual data. On the other hand, object-space hierarchies can split the set of points into regions of unequal area but with a roughly equal number of points. A good object-space hierarchy for point data would generally have fewer levels than an image-space hierarchy. Still, since image-space representations are simpler to implement and require less computational overhead to maintain, spatial database systems have often used such hierarchies for point data as well [SSN87].

Raster Storage of Pixels

The most straightforward representation for an array of pixels is a standard array. This approach is called *raster-scan order*, since pixels are stored individually in the order a raster graphics device would encounter them. Like standard arrays, raster data are stored either in *row-major* or *column-major* form. In a row-major representation, neighboring elements on the same row are stored in adjacent locations. Neighboring elements in the same column but in different rows, however, would not be adjacent. In a column-major representation, these properties are reversed. In either method, the adjacency of pixels along one axis is preserved, but adjacent pixels along the other axis can be located far apart in the file. We would like a representation keeping neighboring pixels as near as possible. For neighbors in one direction, raster scan order is optimal. In the other direction, however, neighboring pixels are much farther apart.

Simple representations such as raster storage have several advantages. First, a raster representation is simple—the location of a pixel within the array can be computed quickly and its attribute values can then be found directly. Second, it requires none of the overhead of pointers or other special codes found in the representations presented later. For images

without large homogeneous areas, the overhead involved with these later methods exceeds any potential space savings. Third, a raster image can be compressed using standard compression techniques such as run-length encoding. Although compression can reduce the space required by a raster-scan representation, operations on the compressed data first require decompression. Fourth, graphics hardware is often tailored for very fast drawing of raster images. Such hardware will take a stream of pixel attributes (i.e., colors) and place them directly on the screen, row by row. Because of this hardware support, a two-dimensional raster image can be rendered quickly. For other representations, software would have to convert the data represented into a raster-scan ordering; drawing such images may be significantly slower.

Morton Ordering

Despite the advantages of raster-scan ordering, there are other promising ways to order pixels within a data file. Morton [Mor66] identifies a recursive ordering, now commonly referred to as Morton order. It subdivides a large square image into quadrants, and then orders the pixels within each quadrant in turn. In addition to the recursive process of dividing an image into quadrants, Morton order must also specify an order for quadrants. Samet [Sam90a] identifies two commonly-used orderings for quadrants, called N and Z order due to their shapes.

Figure 1.2 illustrates the use of Morton order on an 8×8 square grid, using Z order. Note that the pixels of the upper left 4×4 submatrix are numbered 0 through 15. The pixels of the upper right, lower left, and lower right submatrices then follow in turn. Within the first 4×4 submatrix, the pixels of the upper left 2×2 submatrix are numbered 0 through 3. Again, the pixels of the other 2×2 submatrices are ordered using Z order. This process continues until the limits of resolution are reached. The location of a pixel within this ordering is referred to as its *Morton code*.

Though this subdivision process may seem complicated, the location of a pixel within Morton order can be computed fairly quickly by *bit interleaving*. Representing each of a

0	1	4	5	16	17	20	21
2	3	6	7	18	19	22	23
8	9	12	13	24	25	28	29
10	11	14	15	26	27	30	31
32	33	36	37	48	49	52	53
34	35	38	39	50	51	54	55
40	41	44	45	56	57	60	61
42	43	46	47	58	59	62	63

Figure 1.2: Morton order for an 8×8 Square Grid, using Z order

pixel's coordinates as a string of bits, bit interleaving is done by alternately concatenating single bits from the coordinate strings. If the x-coordinate of a pixel is $(x_{n-1} \ldots x_2 x_1 x_0)_2$ and the y-coordinate is $(y_{n-1} \ldots y_2 y_1 y_0)_2$, the Morton code of the pixel in Z order would be $(y_{n-1} x_{n-1} \ldots y_2 x_2 y_1 x_1 y_0 x_0)_2$. Although computing coordinates by bit interleaving is not nearly as fast as the simple arithmetic used to compute raster-scan position, it can still be done quickly.

Morton order has most of the advantages of raster-scan ordering, but has several drawbacks as well. First, as in raster-scan order, each pixel must be represented explicitly. If there are large homogeneous blocks of pixels in the image, other representations can represent these blocks as units rather than storing each pixel separately. Second, the image must be square and its dimensions must be powers of two in order to subdivide the image to the lowest level of resolution. If an image's actual dimensions are not powers of two, it must be transformed into a larger one whose dimensions are. In the worst case, this will cause the size of a square image to increase by a factor of almost four. For oblong images, the amount of additional storage required could be even higher. Still, the Morton order is directly related to the subdivision process found in the quadtree and Morton codes are used directly in the linear quadtree.

The Region Quadtree

Pixel representations based on either raster-scan or Morton order representations yield data files holding the attributes of each pixel as a record. In both representations, each pixel is explicitly represented. For large images, large amounts of storage are required. For example, representing each pixel of a 4096×4096 grid would involve storing over 16 million records. Compression techniques might save a lot of this space, but at the cost of disallowing direct access to individual pixels.

One property of many images that can be exploited is that attributes often assume the same values in neighboring pixels. Examples of this include the continuity of countries as well as the existence of large, uniform ocean areas. If one pixel is found in a particular country or ocean area, more likely than not, so are its neighbors. This fact suggests that explicitly representing each pixel may not be necessary. It is possible to combine identical neighboring pixels into larger regions and store attribute values only for these regions. In this way, large uniform areas can be represented as one data record instead of as many. To exploit this uniformity, some form of decomposition of the original image into uniform regions is required.

One representation that would be very compact if feasible would divide an image into a minimal collection of rectangles, within which attribute values are constant. Unfortunately, determining an optimal partition is computationally expensive. A simpler but related problem, decomposing a region into a minimum number of rectangles, is known to be NP-complete if the region is permitted to contain holes [Sam90a].

The *region quadtree* [Sam90a] is a well-known, multi-level decomposition process based on Morton order. A region quadtree begins by considering the node at the top of the tree, which represents the entire (square) image. As discussed earlier, the image is repeatedly subdivided until it is decomposed into uniform regions. For each uniform region, a single record can be used to store the attributes of all the pixels of the region. For pixel data, this subdivision process can be represented by a tree of degree four. For voxel data, a similar

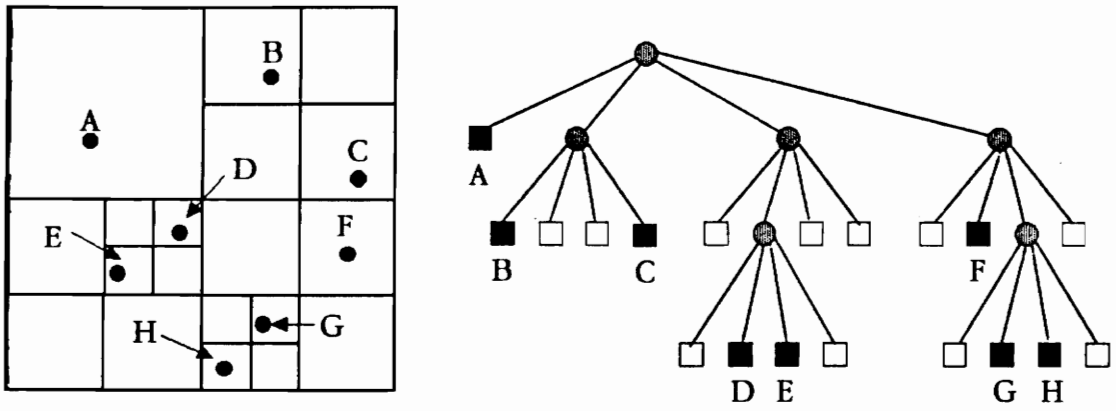


Figure 1.3: The PR Quadtree

tree is of degree eight, and called an *octree*. Figure 1.1 shows this process applied to a simple black and white image and the tree structure obtained.

The PR Quadtree

Although image-space hierarchies have several drawbacks when representing point data, they are still easier to implement and require less computational overhead than object-space hierarchies. The simplest approach to representing point data with an image-space hierarchy is the *MX quadtree* (MX for matrix). This approach reduces points to single pixels and then represents the image as a region quadtree. However, this technique requires a discrete and finite domain for the set of points. A more general representation is the *PR quadtree* (PR for point/region) [Sam90a]. In a PR quadtree, the image is subdivided in the same manner as in a region quadtree, except a different subdivision criterion is used. A node in the tree is subdivided if and only if more than one point is contained in the corresponding region. When the subdivision stops, the attribute values of the point contained in the region (if any) are stored. Figure 1.3 illustrates the decomposition process of the PR quadtree.

In a PR quadtree, large empty areas need not be subdivided, and queries involving such areas will immediately be able to identify them as empty. If two points are very close (e.g.,

the cities of Minneapolis and St. Paul), many subdivisions may be required to place the points in separate nodes of the tree. This makes for a deep tree, and also introduces a large number of empty nodes along the way. Queries involving areas where the subdivision runs deep will be rather slow. To limit the impact of this problem, *bucketing* can be used. Instead of holding only a single point, each node in the tree is capable of holding several points. A node of the tree is split only if the number of points in the region exceeds a bucket's limit. When small numbers of points are found close together, bucketing can substantially reduce the depth of a PR quadtree.

Summary

There are many ways to represent spatial data. This section has presented several frequently used image-space hierarchies. This includes the region quadtree for representing pixel data and the PR quadtree for representing point data. Other common hierarchical representations include the *PMR quadtree* [NS86], one of a variety of quadtrees used to represent a collection of lines. The concepts of each of these image-space hierarchies are similar—the primary differences are the representation of specific entities and the rule used to govern the subdivision process. The issues involved in the implementation of each of these structures are quite similar, and the techniques used to implement a quadtree are applicable to each of these representations. The region quadtree is the simplest of these image-space hierarchies and serves as the basis for all subsequent discussion and implementation. However, the implementation of a region quadtree can be extended in a straightforward manner to apply to quadtree representations for other types of spatial data as well.

Chapter 2

Representation of Quadtrees

This chapter discusses several different approaches used to represent quadtrees. While region quadtrees serve as the standard example for these representations, the approaches used for region quadtrees are applicable to other members of the quadtree family as well. Section 2.1 describes linear quadtrees, which consist of a sorted list of records corresponding to the homogeneous parts of the image. Section 2.2 discusses pointer-based quadtrees, which consist of a collection of nodes corresponding to all subimages resulting from the decomposition process. It also introduces the *leafless quadtree*, which does not represent the homogeneous subimages as separate nodes. Section 2.3 introduces the *DF-expression*, which consists of a list of quadtree nodes in the order they are found in a preorder traversal.

Both the linear quadtree and the DF-expression can be thought of as implementation-driven optimizations applied to the pointer-based quadtree. These optimizations are intended to require less space and allow faster processing of large data sets. Section 2.4 analyzes the amounts of space required by each representation and demonstrates that our leafless quadtree always requires less storage space than a corresponding linear quadtree.

2.1 Linear Quadtrees

The *linear quadtree* [Gar82] was introduced as a way to represent quadtrees without the space overhead involved with pointers. A linear quadtree consists of a sorted list of records corresponding to the homogeneous areas of the image. These records are referred to as *leaves*, since they correspond to the leaf nodes of a pointer-based quadtree. The linear quadtree, as a list of leaves ordered by location, can easily be stored on disk using a standard B-tree. In addition, linear quadtrees save the space required to store pointers. Even though quadtree algorithms are more straightforward using a pointer-based representation, these implementation advantages have generated much interest in linear quadtrees.

While linear quadtrees eliminate the need for pointers, they still must represent the decomposition process. The pointers stored by a pointer-based quadtree provide this kind of organization. In a linear quadtree, each leaf stores not only its attribute values but also a *locational code*. Locational codes represent the location of a leaf within the image space and are based on the Morton code. An entire linear quadtree, then, is simply a sorted list of records consisting of the attribute values and the locational code for each leaf.

Location Coding Schemes

Gargantini [Gar82] presents a technique for obtaining locational codes similar to the Morton code. The code for each leaf in a $2^h \times 2^h$ image is an h-digit base 5 number $(x_{h-1} \ldots x_1 x_0)$. For each digit x_i, $(0 \leq i < h)$, x_i is 0, 1, 2, or 3 depending on the position of the leaf within the block of size 2^{i+1} containing it. If the this block is contained entirely within the leaf, x_i is 4. This scheme produces a fixed-length locational code requiring $\log_2 5^n$ bits. Samet [Sam90a] refers to this scheme as the *FL locational code*. In an FL code, when a digit of 4 is found, all subsequent digits are also 4. Since these added digits contribute no information, a variable-length approach uses the code of 4 as an end marker. This approach is referred to as the *VL locational code* (VL for variable-length). The major drawback of both of these schemes is that they require base 5 arithmetic. Computer hardware does not

perform multiplication and division by 5 as fast as when powers of two are involved. For that reason alone, a base 4 representation is preferable.

The most commonly used locational code is the *FD locational code*, a slight modification of the Morton code. An FD locational code code consists of two components. The first component gives the Morton code of the leaf. For leaves larger than a pixel, a specific pixel within the leaf (e.g., upper left) is chosen and used to obtain this code. In a $2^h \times 2^h$ image, a Morton code requires $2h$ bits. The second component indicates the size of the leaf. Since there are only $h + 1$ different leaf sizes in a $2^h \times 2^h$ image, this component requires is $\lceil \log_2(h + 1) \rceil$ bits. An FD code can be implemented either as a fixed-length code or a variable-length code, but implementation is much simpler when a fixed-length code is used. One commonly implemented code length is 32 bits [SSN87], with which it is possible to encode the leaves of images as large as $2^{14} \times 2^{14}$, using 28 bits for location and 4 bits for depth. In three dimensions, however, no images larger than $2^9 \times 2^9 \times 2^9$ can be represented. A linear quadtree using this locational code is referred to as an *FD linear quadtree*.

Implementation of Linear Quadtrees

Given a method of obtaining locational codes, the implementation of a spatial database system based on linear quadtrees is fairly straightforward. The linear quadtree is simply a sorted list of leaves, with the locational code serving as the sort key. Each leaf is characterized by a locational code and the attribute values of the corresponding region. The entire linear quadtree can then be represented as a sorted list of nodes with a one-dimensional key. This list can be managed easily by a B^+-tree, as described in [Abe84]. Several prototype geographic information systems (e.g., [SPM87, SSN87]) are based on the linear quadtree. QUILT [SSN87], developed at the University of Maryland, is used later for comparison with our pointer-based implementation.

Other Considerations

As [Gar82] points out, in data sets where pixel attributes require only one bit (e.g., black and white images), the linear quadtree is particularly useful. In such images, one color (e.g., black) is considered foreground and the other is considered background. Not only does the linear quadtree avoid representing the internal structure of the corresponding pointer-based quadtree, but it also does not store the leaves of the background color. If a query is performed in a "white" region, no black leaf with a matching location code will be found. When this is the case, the absence of a leaf from the data set completely identifies it, since it is known not to be the foreground color. In a pointer-based hierarchy, nodes of both colors must be stored.

However, in multicolor images, the savings from storing only "foreground" colors is not as great unless there is still a predominant "background" color. In any case, a linear quadtree must still store all non-background leaves. Because the space savings is much lower for multicolor images and search algorithms are more complex when background leaves are left out, systems such as QUILT represent all leaves, foreground or background.

2.2 Pointer-Based Quadtrees

The *pointer-based quadtree* is the direct representation of the hierarchy obtained when a standard quadtree is built. Each block processed by the subdivision corresponds to a node in a standard pointer-based quadtree. A node corresponding to a block that is subdivided has four children (its quadrants) and is referred to as an *internal node*. A node representing a uniform block has no children and is referred to as a *leaf node*. A traditional pointer-based quadtree consists of both internal nodes and leaf nodes. Figure 1.1 illustrates the tree structure derived from the decomposition of a sample image.

Representation of Quadtree Nodes

There are two distinct node types in a pointer-based quadtree. An internal node stores four pointers, one to each of its children. Some implementations also include a pointer to the parent of the node. A leaf node only stores the values of its attributes, but might again also store a pointer to its parent. The nodes of a pointer-based quadtree have historically been represented in one of three ways:

1. A representation based on an *undiscriminated union* stores each node of the quadtree as the same type. An internal node stores pointers to its children, but leaves its attribute field blank. A leaf node, on the other hand, stores attribute values but fills its pointer fields with NULL pointers.

2. A representation based on a *discriminated union* represents the nodes as different types. An internal node stores only pointers, while a leaf node stores only attributes. To discriminate between the node types, and additional bit of information is needed to identify the type of a node.

3. A *leafless quadtree* [OHS90] stores only internal nodes. An internal node consists of pointer fields for its four children. If a particular child is another internal node, the corresponding pointer field holds a pointer to the child. If a child is a leaf node, the values of the leaf's attributes are stored in the pointer field. Thus, each pointer field in an internal node contains either a *pointer* or a *leaf*. The paged-pointer quadtree adopts this representation because of its space-efficiency.

Algorithms traversing a quadtree with nodes represented as an undiscriminated union repeatedly follow pointers, using the NULL pointer fields to identify leaf nodes. While this representation is easy to implement, it wastes a substantial amount of space. Since three-fourths of all nodes in a quadtree are leaf nodes, the cost of storing NULL pointers is very high. A discriminated union eliminates this waste of space, at the cost of one extra bit per node. The tag field could be added to the record of each node, but this new field might increase node size by a byte or a word due to alignment considerations. An equivalent approach reduces the size of pointers by one bit and uses the removed bit to specify the type of that node.

While the discriminated node type is more space-efficient than the undiscriminated node type, it has a more subtle inefficiency. In the parent of a leaf node, one pointer locates the

Node ID	Parent	NW child	NE child	SW child	SE child
A	→ NULL	white	→ B	→ C	→ E
B	→ A	white	white	black	black
C	→ A	white	→ D	white	black
D	→ C	white	black	black	black
E	→ A	black	black	→ F	white
F	→ E	black	black	black	white

Figure 2.1: Representation of the leafless quadtree

leaf node, which itself separately stores the values of its attributes. Instead of storing both a separate leaf node and a pointer to that node, the leafless quadtree saves space by encoding the attribute values of the leaf into a pointer field. The lack of separate leaf nodes reduces the number of nodes in a leafless quadtree by a factor of four. Since pointers need be only large enough to distinguish among the nodes of the quadtree, pointers in a leafless quadtree can be two bits smaller than those of traditional pointer-based quadtrees. A system with fixed-size pointers can store larger images using leafless quadtrees than with traditional representations. Figure 2.1 shows this approach applied to the simple quadtree of Figure 1.1.

It is often useful to have a pointer to the parent of a node in addition to the children. Access to parent pointers allows an algorithm to directly located the immediate ancestor of a node. The ancestors of a node can be found without parent pointers by maintaining a stack holding the path from the root of the tree to the node. When parent pointers are used, a pointer to a node is sufficient to access both the ancestors and descendants of that node. Without parent pointers, both a pointer to the node and the stack of ancestors are necessary. Despite this advantage, an implementation with parent pointers increases the amount of storage required for each internal node by 25% in pointer-based quadtrees and 12.5% in octrees. While parent pointers are not strictly necessary, our implementation supports them despite the cost. Leafless quadtrees, even with parent pointers, still require less space than linear quadtrees.

Comparison of Pointer-Based and Linear Quadtrees

Pointer-based quadtrees offer several significant advantages over linear quadtrees. For example, the data stored in a pointer-based quadtree can be accessed in many different ways. Given an internal node, finding the node corresponding to a particular child involves simply decoding the pointer and following it. Linear quadtrees require searching for a matching node in the entire list of nodes. Though a search can be done in logarithmic time, the time to locate a node increases with the size of the quadtree. Though linear quadtrees are ordered to allow quick preorder traversals (i.e., visits in the order NW, NE, SW, SE), traversals in other orders are often necessary. For example, back-to-front display of the nodes of an octree requires a traversal order dependent on the viewing position [Rya92].

Progressive refinement of images is desired in many application areas, and quadtrees are useful in accomplishing this [Rya92]. Using progressive refinement, a rough approximation of an image is built quickly and then the image is refined to its final form. For example, when a complicated image is rotated in a CAD package, a rough approximation of the image gives the user a quick idea of the orientation of the image without spending the time required to generate a totally faithful reconstruction. Progressive refinement can be supported in a pointer-based representation by storing attribute values in internal nodes. While such attribute values can not completely describe the corresponding region, they may provide a good approximation. Displaying an image represented by a quadtree using progressive refinement would initially traverse the tree only to a certain depth. Later refinements of the image go deeper into the tree. While linear quadtrees can be extended to store internal nodes as well as leaves, each internal node must store both attribute values and a locational code. Pointer-based quadtrees simply add another field to an existing internal node, so this extension adds substantially more overhead in linear quadtrees.

One further advantage of the pointer-based quadtree is the ability to hold more data in implementations that require the size of pointers or locational codes to be fixed. For a $2^h \times 2^h$ image, the absolute maximum number of leaves found in any quadtree is 2^{2h}. Locational

codes in linear quadtrees must distinguish among all *possible* leaves of a quadtree. Because of this, they require somewhat more than $2h$ bits each (see Section 2.1). For example, linear quadtrees with 32-bit locational codes can not represent any image larger than $2^{14} \times 2^{14}$ in two dimensions or $2^9 \times 2^9 \times 2^9$ in three dimensions.

On the other hand, in a pointer-based representation, pointers need only distinguish between those nodes that are actually found in the tree. Pointers in a leafless quadtree only need to distinguish among the internal nodes of the tree. Since there are only one third as many internal nodes as leaves and the number of leaves is bounded by 2^{2h}, pointers (which also store a tag bit) require at most $\lceil 1 + \log \frac{2^{2h}}{3} \rceil$ bits. When pointers are 32-bit quantities, they are sufficiently large to store even worst-case images of size $2^{15} \times 2^{15}$ or $2^{11} \times 2^{11} \times 2^{11}$. Typical images have substantially fewer than 2^{2k} leaves. Thus, pointer-based quadtrees are capable of storing substantially larger images than linear quadtrees with locational codes of the same size.

This seemingly small advantage of pointer-based quadtrees can be rather important. For example, the underwater mapping system described in [OHS90] stores depth information for a large number of points. There are several sources from which this information is obtained. Pre-stored maps give depth information at a coarse resolution. On-ship sonar systems can determine depth around a vessel at a much finer resolution. In such images, some regions have very refined data, while data elsewhere is much coarser. Using linear quadtrees to implement this kind of mapping system requires locational codes large enough to encode leaves at the finest resolution. Pointer-based quadtrees only need pointers large enough to distinguish between the actual number of nodes found in the tree. Because large portions of the image hold only coarse-grain data, pointers can be substantially smaller than locational codes. If pointers and locational codes are of fixed size, pointer-based quadtrees can easily handle many such multi-resolution data sets, while linear quadtrees may not.

One of the major shortcomings of traditional pointer-based quadtrees is the overhead required to maintain pointers. Another is the fact that these quadtrees store each internal node, even though such nodes typically hold no data. A more serious drawback of pointer-

based quadtrees involves images too large to be represented in main memory. In this case, a standard pointer-based quadtree uses virtual memory, if available. When this is the case, those nodes that can not be held in the main memory are stored on a permanent storage and brought into main memory when required. Memory management techniques that exploit the ways pointer-based quadtrees are accessed can outperform generic virtual-memory approaches. Chapter 3 presents the paged-pointer quadtree, which represents large pointer-based quadtrees on disk with performance comparable to that of linear quadtrees. No competing technique has previously been reported.

2.3 DF-expressions

While linear quadtrees are often more compact than traditional pointer-based quadtrees, and leafless quadtrees are yet more compact, these are not the most compact representations of quadtrees. One representation that requires less space than either the pointer-based or linear quadtree representation is known as the *DF-expression* [KEY80]. A DF-expression can be encoded as a string of symbols representing the unique preorder traversal of the corresponding quadtree. Internal nodes of the tree are represented by a symbol such as I (for internal node). Leaves are represented by the values of their attributes (e.g., B for black, W for white). For the sample quadtree in Figure 1.1, the corresponding DF-expression is:

(IW(IWWBB)(IW(IWBBB)WB)(IBB(IBBBW)W)).

The parentheses above are not represented and are shown only to show the decomposition to the reader. Other more space-efficient encoding methods have also been reported; see [KE80, Tam84].

Although the DF-expression requires very little space, its most significant drawback is that random access within an expression is not possible. This can lead to excessively long query response times, even for simple queries. For example, determining the value of the attributes of a pixel in the lower-right quadrant of an image will require traversing almost the entire expression. Finding the second (or subsequent) child of a particular node cannot

be done without processing the portions of the DF-expression representing prior children and their descendants. In comparison, finding any child in a pointer-based quadtree is trivial. For applications requiring random access, DF-expressions are clearly unacceptable.

A representation known as the S^+-tree [dJSS91] is based on the standard DF-expression but allows some random access to nodes as well. The S^+-tree has very strong similarities to the B-tree. A standard DF-expression is broken into pages, much like in the B-tree. At the beginning of each page, a code called the *linear prefix* is stored. A linear prefix represents a traversal of the tree including only those nodes found in prior pages. Using linear prefixes as keys, the internal structure of an S^+-tree is much like that of a B-tree. When a particular area of the tree is needed, its location can be compared to the prefix codes, and the page containing the region can be found quickly. To find actual attributes, only the pages containing parts of the region must be searched exhaustively. Despite these improvements, no implementation of the S^+-tree has been reported, and the amount of overhead it introduces is largely unknown.

Even though random access is not possible in DF-expressions, they are still very useful. After all, they are easily the most compact of the three representations. Because DF-expressions take up less space on a disk, time is saved because fewer pages have to be read in. For many image-processing applications, it is necessary to process the entire image. For such applications, the entire tree must be traversed anyway. But when random access is required, the DF-expression representation is not worth considering.

2.4 Analysis of Space Requirements

This section analyzes the space requirements of FD linear quadtrees, leafless pointer-based quadtrees and DF-expressions. For any particular image, the results of the subdivision process are identical for each representation. Let L and I denote the number of leaves and internal nodes, respectively, in the quadtree representing the decomposed image. The

values of L and I are closely related [Sam90a]:

$$L - 1 = I(2^d - 1).$$

For the purposes of this analysis, we approximate I as $L/(2^d - 1)$.

In a pixel-by-pixel representation, each pixel has a set of attribute values. In a quadtree decomposition, each leaf has a set of attribute values. Given that the image is represented by a quadtree, the attribute values of each leaf must be stored. These values are assumed to belong to a finite set; let b denote the number of bits required to represent this set. Any quadtree representation requires at least Lb bits to store attribute values. Additional space used to organize a representation is referred to as *overhead*.

In addition, let d represent the dimension of the image. For pixel data, $d = 2$; for voxel data, $d = 3$. Let h represent the *height* of the image; there are 2^h pixels along each axis. For most images of this size, h is also the height of the corresponding quadtree.

FD Linear Quadtrees

Each leaf in an FD linear quadtree consists of two components: its attribute values and its FD locational code. Attribute values require b bits per leaf; no tag bit is necessary since no pointers are represented. Recall that the number of bits required for an FD locational code in a d-dimensional image with h levels is $dh + \lceil \log(h+1) \rceil$. Since there are L leaves, the total number of bits to store an entire FD linear quadtree is simply:

$$S_{lq} = L\,(b + dh + \lceil \log(h+1) \rceil) \tag{2.1}$$

Leafless Pointer-Based Quadtrees

Leafless quadtrees [OHS90] store only internal nodes; the attribute values of each leaf are encoded in a pointer field of its parent. This analysis assumes that this encoding wastes no space. For example, if leaves only require 8 bits (including the tag field) while pointers require 32 bits, the leaves are actually encoded in a field using only 8 bits. The paged-

pointer quadtree satisfies these assumptions, although our implementation requires that each node begin on a byte boundary.

There are I internal nodes in our leafless quadtree. A pointer must be large enough to distinguish among all the nodes of the quadtree. A single tag bit is also required to indicate that the its field holds a pointer. Therefore, each pointer requires $\lceil 1 + \log I \rceil$ bits.

Each internal node in a leafless quadtree without parent pointers (except the root) is pointed to exactly once. Thus, the number of bits used to store pointers in a leafless quadtree is $I\lceil 1 + \log I \rceil$. In a leafless quadtree, the attribute values of each of the L leaves are also stored exactly once. Since each leaf requires b bits for its attribute values and also a tag bit, the number of bits used to store leaves in a leafless quadtree is $L(1 + b)$. Therefore, the total number of bits required to store a leafless quadtree is:

$$S_p = I\lceil 1 + \log I \rceil + L(1 + b) \tag{2.2}$$

The addition of parent pointers requires an additional pointer in each of the I internal nodes. Although parent pointers do not require a tag bit (since leaves are never stored in this field), the tag bit may be stored for simplicity of implementation. In a leafless quadtree with parent pointers, the number of bits required by the entire quadtree is:

$$S_{pp} = 2I\lceil 1 + \log I \rceil + L(1 + b) \tag{2.3}$$

DF-Expressions

The primary advantage of DF-expressions is space efficiency, so it is not surprising that the DF-expression is the most compact representation for quadtrees. Since it does not allow random access, it is unsatisfactory except for applications that process entire images. This analysis assumes the use of the compact variable-length encoding method presented in [KE80, Tam84]. In this scheme, each internal node uses only one bit. Each leaf node stores a tag bit and b additional bits for its attribute values. The number of bits required by a DF-expression encoded in this manner is:

$$S_{df} = I + L(1 + b) \tag{2.4}$$

Theorem 2.1 *In any image represented by a region quadtree,*

$$S_{df} < S_p < S_{pp} < S_{lq}.$$

Therefore, a DF-expression requires less space than a leafless quadtree, which requires less space than a linear quadtree.

PROOF: From (2.2), (2.3), and (2.4), it is trivial to show that:

$$S_{df} < S_p < S_{pp}.$$

The number of internal nodes in a quadtree, I, is slightly less than $L/(2^d - 1)$. Therefore, from (2.3),

$$S_{pp} < L\left(1 + b + \frac{2}{2^d - 1}\lceil 1 + \log\frac{L}{2^d - 1}\rceil\right).$$

In a quadtree representing a d-dimensional image with h levels, the number of leaves may not exceed 2^{dh}. From (2.1) and (2.3),

$$\begin{aligned} S_{pp}/L &< 1 + b + \frac{2}{2^d - 1}\lceil 1 + \log\frac{L}{2^d - 1}\rceil \\ &< 1 + b + \frac{2}{2^d - 1}\left(2 + \log L - \log(2^d - 1)\right) \\ &< 1 + b + \frac{2}{2^d - 1}(dh - d + 3) \\ &< b + dh + \lceil \log(h + 1)\rceil \\ &= S_{lq}/L. \end{aligned}$$

Therefore, $S_{pp} < S_{lq}$. □

Samet and Webber [SW89] similarly compare the amounts of space required by linear quadtrees and traditional pointer-based quadtrees (based on the discriminated union node type). They determine that which representation is more compact depends on the number of nodes found in the tree in relation to the size of the image. Since the number of bits required by pointers in a pointer-based quadtree depends on the number of nodes, it is not surprising that when there are fewer nodes, the pointer-based quadtree can be more

compact than the linear quadtree. As the number of leaves relative to the resolution of the data increases, pointer-based quadtrees become less efficient. Samet and Webber [SW89] determine a fairly low "cutoff" number of nodes beyond which the linear quadtree is more efficient than the pointer based quadtree. However, Theorem 2.1 shows that the leafless quadtree is more compact than the linear quadtree in all cases.

Space Requirements with Fixed-Size Fields

As pointed out in [SW89], analysis using the number of "raw bits" required is not terribly useful. Typical implementations align data structures along byte boundaries, and it may be desirable to have each node or record begin on a byte boundary. Moreover, the analysis here and that of [SW89] base the size of each pointer on the current number of nodes in a quadtree. Limiting the pointer size in this way may later require modifying the size of all pointers when new nodes are inserted.

In practice, the size of both pointers and locational codes is fixed prior to use. This is true in the University of Maryland's QUILT system [SSN87] and is also true in our implementation. Because it is desirable to align node structures on word boundaries, both pointers and locational codes are typically 32 bits long. With fixed size fields, the comparison of space requirements is simpler and more faithfully reflects implementation results. With pointer size fixed, the number of bits required by a leafless pointer-based quadtree is $L\left(1 + b + 32/(2^d - 1)\right)$, while the number of bits needed by a linear quadtree is $L(32 + b)$. Table 2.1 shows the amount of space per leaf required by these representations for several different values of b. Table 2.2 presents the amount of space required per leaf in three dimensions. In two dimensions, the amount of overhead (i.e., space required beyond b bits per leaf) in a leafless quadtree is a little more than $\frac{1}{3}$ that of a linear quadtree. In three dimensions, this ratio is slightly greater than $\frac{1}{7}$. In any event, the space savings resulting from a properly implemented pointer-based quadtree or octree is quite significant.

The drawback of fixing the size of pointers and location codes is the strict limit placed on image size. With 32-bit locational codes, a linear quadtree can represent an image of

b	Pointer-based Quadtree	Linear Quadtree	% space saved
1	12.67	33	61.6
3	14.67	35	58.1
7	18.67	39	52.1
15	26.67	47	43.3
31	42.67	63	32.3
63	74.67	95	21.4

Table 2.1: Bits per leaf node in quadtrees with 32-bit fields

b	Pointer-based Octree	Linear Octree	% space saved
1	6.57	33	80.1
3	8.57	35	75.5
7	12.57	39	67.8
15	20.57	47	56.2
31	36.57	63	42.0
63	68.57	95	27.8

Table 2.2: Bits per leaf node in octrees with 32-bit fields

size $2^{14} \times 2^{14}$ in two dimensions, and $2^9 \times 2^9 \times 2^9$ in three dimensions. For larger images, larger locational codes are necessary. On the other hand, leafless pointer-based quadtrees with 32-bit pointers can represent even worst-case images of $2^{15} \times 2^{15}$ in two dimensions or $2^{11} \times 2^{11} \times 2^{11}$ in three dimensions. Moreover, substantially larger images can often be represented as pointer-based quadtrees since the number of nodes in such images is significantly smaller than in a worst-case image. In the data sets used in Chapter 4, the number of leaves in each quadtree is less than one-sixth the number of pixels in the image. Therefore, the restriction of image resolution due to fixed field sizes is far more serious for linear quadtrees than pointer-based quadtrees.

Chapter 3

The Paging Scheme

Despite the advantages of pointer-based quadtrees enumerated in the previous chapter, alternative quadtree representations have attracted a great deal of attention for several reasons. The most important is the ease by which pointerless representations can be divided into pages on permanent storage. This chapter describes the *paged-pointer quadtree*, a representation organizing leafless pointer-based quadtrees into pages on disk. The paged-pointer quadtree maps each node in a leafless quadtree to a location on disk, arranges the nodes on pages according to a preorder traversal, and organizes the pages in a manner similar to the B-tree. Section 3.1 presents criteria that spatial database systems based on quadtrees should satisfy and evaluates the representations discussed in Chapter 2 with respect to these criteria. Section 3.2 discusses the architecture of the paged-pointer quadtree in general terms, while Section 3.3 describes specific considerations encountered during implementation.

3.1 Requirements for Quadtree Implementations

We are interested in an efficient and flexible representation for quadtrees. Efficient representations reduce the costs involved with representing quadtrees, both in terms of processing time and storage space required. Flexible representations are capable of representing a wide

variety of data. Traditional pointer-based quadtrees are very efficient in representing small data sets. However, the flexibility needed to represent larger data sets is generally obtained by sacrificing efficiency. Any representation for quadtrees should satisfy several flexibility and efficiency criteria in order to be considered effective:

- The representation should reside on permanent storage.
- The space overhead introduced by the representation should be minimized. Representations with low overhead reduce the number of disk pages accessed by quadtree algorithms.
- The representation should be organized so that every node in the tree can be accessed quickly. This minimizes query response time.
- The representation should be organized to produce a localized reference pattern, which minimizes page faults.

Permanent Storage

Since the spatial data represented a quadtree (e.g., maps) often has a permanent nature, it must be kept on permanent storage. In addition to storing attribute values, the structure of the quadtree must be represented as well; the links in a pointer-based quadtree, the locational codes in a linear quadtree, and the code bits of the DF-expression all represent the decomposition of the image space. More importantly, many data sets represented by a spatial database system are too large to be held completely in main memory. Virtual memory facilities allow a large pointer-based quadtree to be accessed as if all of it were in main memory. However, as [SSN87] points out:

> ... it is necessary to store the image on disk and bring in portions of the data as needed. A [pointer-based] tree structure is inconvenient for this purpose, since there may be little relationship between the proximity of nodes in the tree and their proximity on disk. This can lead to an intolerable number of disk accesses when manipulating the tree.

Memory management routines such as ours that exploit the specific access patterns and keep nodes nearby in storage produce better performance than more general routines typically

provided by operating systems. This observation has been confirmed empirically during the construction of QUILT and by experiments discussed in Section 4.1. Even if large quadtrees could be held in a virtual memory system with acceptable performance, they still must reside on permanent storage. With many virtual-memory systems, it is necessary to copy an entire quadtree into main memory before it can be used. This may require more disk accesses than a disk-based representation using buffers in main memory for operations that process only part of the quadtree.

The key advantage of linear quadtrees is that they can easily be stored on disk using a B-tree [Abe84, SSN87]. The B-tree is a well-known representation at the heart of most general-purpose database systems. On the other hand, pointer-based quadtrees have historically been used only on data sets small enough to be held entirely in main memory. Virtual memory allows in-memory representation of larger data sets, but at the cost of decreased performance. While our paged-pointer quadtree is more complicated than the B-tree used to organize linear quadtrees, it allows a quadtree held on permanent storage to exploit the many advantages of a pointer-based representation. This includes more convenient algorithmic expression—the added complication of the paging scheme is hidden from the algorithm designer.

Compact Representation

The primary advantage of quadtrees over non-hierarchical representations (e.g., raster-scan ordering of pixels) of the same data is space savings. Since large homogeneous areas are represented as single leaves, quadtrees usually require less storage space than raster representations. However, the spatial organization provided by the quadtree is not without cost; significant amounts of overhead are required to maintain the structure. In pointer-based quadtrees, pointers representing the structure of the tree require added space in the representation; in linear quadtrees, a locational code must be stored with each leaf node, again adding overhead. Even the compact DF-expression introduces small amounts of overhead. While the overhead introduced in any of these representations is usually less than the space

saved, it should be kept to a minimum.

A compact representation is important for more than simply saving space on permanent storage. In many areas of computer science, there are well-known space-time tradeoffs, where the availability of more storage space allows faster algorithms. However, when disk memory is heavily used, such tradeoffs are far less common. The dominant cost in algorithms that access permanent storage is almost always disk access time. The added complexity of the B-tree is well worth the difficulty because it generally reduces the number of disk accesses required. As Knuth [Knu73] (p. 554) points out, "space optimization is closely related to time optimization in a disk memory." For disk-based quadtrees, a more compact representation reduces the number of disk accesses for algorithms operating on the trees. This suggests that if only preorder traversals are necessary, the DF-expression should be used. However, when random access is required, the results of Section 2.4 clearly recommend the leafless quadtree over the linear quadtree.

Query Response Time

If the data stored in a quadtree can not be quickly extracted from the representation, even the simplest of queries may require unacceptable amounts of time. Access to individual nodes in a pointer-based quadtree is straightforward; links are followed from the root to the node in question. While linear quadtrees do not represent such a structure, leaves can still be found by generating the locational code of the area in question and searching in the list of leaf nodes. In a pointer-based quadtree, the number of links followed to locate a node is proportional to the depth of the tree. This number is typically logarithmic in the number of pixels in the original image. In a linear quadtree, the list of leaf nodes can be searched in logarithmic time as well. With either representation, nodes in the tree can be accessed fairly quickly. As mentioned previously, fast access of individual nodes in a standard DF-expression is not possible. While the S^+-tree [dJSS91] allows block-level random access, its costs have not yet been adequately studied.

Exploitation of Locality

Locality refers to the property of computer programs where most memory references are concentrated in small areas of the memory space. For example, memory caches store small "regions" of a main memory for fast access. Because of locality, even small caches can hold the actively used areas of memory and provide fast access much of the time. Virtual memory techniques also exploit locality of reference; only small areas of the memory are accessed at any one time. If access patterns are truly random, virtual memory routines would require excessive swapping to disk and be impractical. For this reason, exploitation of locality is important in quadtrees represented on disk. Even for small quadtrees in main memory, locality of reference allows quadtree algorithms to exploit memory caches, if available.

Linear quadtrees exploit locality by ordering the leaves of the quadtree in preorder. In this way, most neighboring areas of the quadtree are found near each other on storage. Standard pointer-based quadtrees are not nearly as effective in exploiting locality. For example, quadtrees built from raster data, with nodes allocated in the order they are incorporated into the quadtree, provide no guarantee that neighboring nodes in the tree are likely to be neighbors in the memory. Because of this, visiting even a small number of nodes may access several different pages and produce an unacceptable number of page faults. The paged-pointer quadtree, like the linear quadtree and DF-expression, orders its nodes according to a preorder traversal. As the tree changes, B-tree-like page management maintain this order. When pointer-based quadtrees are stored and maintained in this manner, the same expectation of locality in linear quadtrees applies equally to pointer-based quadtrees.

3.2 Architecture of the Paging Scheme

In the previous section, we present a number of criteria for evaluating quadtree implementations. Traditional pointer-based quadtrees are found to be lacking with respect to several of these criteria. First, the quadtree should reside on permanent storage. For quadtrees larger than available main memory, portions must be held on disk when the quadtree is used.

Second, traditional pointer-based quadtrees often require more space than linear quadtrees. Our analysis in Section 2.4 shows that leafless quadtrees have great potential for space savings, but their implementation must satisfy the assumptions of the analysis. Third, traditional pointer-based quadtrees, unlike linear quadtrees, do not provide much locality of reference. Poor locality can lead to many page faults and unacceptable performance. This section describes how the paged-pointer quadtree overcomes each of these difficulties.

3.2.1 Mapping Pointer-Based Quadtrees to Disk

The lack of a disk-based representation for the nodes of traditional pointer-based quadtrees is a serious shortcoming and a major motivation for linear quadtrees. In a linear quadtree, leaves can be mapped to disk by a B-tree structure [Abe84, SSN87] using the ordering imposed by Morton codes. For quick access, such systems provide buffers in main memory to hold a small number of the pages of the linear quadtree. When a required page is found in the buffer pool, it can be accessed directly. When it is not, the page is read from the disk and replaces the least recently used page in the buffer pool. A small number of buffers generally provides acceptable performance [SSN87].

The paged-pointer quadtree is represented on disk as a file, which is broken up into a collection of pages. The first page of this file holds information on the image represented by the quadtree, including image type, location, orientation, and storage space required for attribute values. Subsequent pages hold the nodes of the quadtree. These pages consist of two components: control information used for page management and a collection of nodes. The control information includes pointers to "neighboring" pages (such as found in B^+-tree) and the location and amount of free space available within the page. A pointer in a paged-pointer quadtree consists of two components: *page number* and *offset* within that page. The page number identifies the page of the quadtree file containing the node; the offset identifies the location of the node within that page.

Our implementation uses a buffer pool similar to that of the disk-based linear quadtree system. Each buffer in the pool is capable of holding one page of the quadtree file. When

a node specified by a pointer is needed, its page number component is used to determine whether the page containing the node is held in the buffer pool. If so, the offset of the node is used to locate the node within the buffer holding its page. If the page is not currently in the buffer pool, the page is read in from the quadtree file. If the buffer pool is full, our new page replaces the page in the buffer pool that was referenced least recently. That page is written back to disk if it was modified while in memory.

Our buffer pool approach is effectively identical to that used in standard virtual memory (VM) systems. Each address in a virtual memory space also consists of two components: page number and offset within the page. The main memory of the computer serves the same purpose as a buffer pool; each page in a virtual memory system is found either in main memory or on disk. The portion of a disk used to hold those pages not fitting in main memory is referred to as the *swap space*. If a page not in main memory is accessed, it is located in the swap space and brought in. Another page previously stored in main memory is moved to the swap space to make way for the new page. This page is chosen by an algorithm whose behavior approximates least recently used page replacement.

The traditional virtual memory systems used for our implementation are capable of mapping nodes to permanent storage. However, there are several reasons why our explicitly disk-based approach is preferable. First, our quadtree file serves two purposes. Like a virtual memory swap space, it holds the parts of the tree not fitting in the buffers. In addition, it is a permanent location to store the quadtree when it is not being used. The swap space of a traditional VM system requires additional disk space and holds its data only temporarily. Second, the entire quadtree would have to be copied into main memory before a traditional VM system could use it. Third, since the page structure is hidden from the programmer in a virtual memory system, nodes may straddle a page boundary and require two disk accesses to retrieve. The biggest disadvantage of our buffer pool approach is in decreased performance for a page access; address translation in a virtual memory system is much faster than anything we can do in software.

Several recent operating systems, including Mach [RTY88], have implemented *memory*

mapped files. Memory mapped files map the pages of a file into the virtual memory space of the machine. Such files are read by simple memory accesses. When a page that has not been read from disk is accessed, the memory access results in a page fault and the operating system fetches that page from the file. A memory-mapped file also serves as the swap space for its contents; the VM system performs the necessary reads and writes in the file, rather than in a separate swap space. Memory-mapped files also eliminate the need to copy the entire quadtree into memory before it is accessed. Application-specific virtual memory features (e.g., page fault handlers) can be used to implement our paged-pointer quadtree on such systems. A virtual memory system with memory-mapped files eliminates the need for software-based address translation and would significantly improve the performance of our implementation.

3.2.2 Representation of Quadtree Nodes

The node structure of the paged-pointer quadtree is based on the leafless quadtree, in which only internal nodes are explicitly represented. Each pointer field in a leafless quadtree serves as either a *pointer* or a *leaf*. A tag bit stored in each field identifies the contents of the field. Three different representations of quadtree nodes have been considered:

1. The size of a pointer field in a *fixed-size node* is independent of the type of the field; pointers and leaves consume the same amount of space.
2. The size of a pointer field in a *variable-size node* depends on the type of the field; pointers and leaves may consume different amounts of space.
3. A combined approach uses variable-size nodes on disk and fixed-size nodes in main memory.

The implementation of the leafless quadtree reported in [OHS90] uses fixed-size nodes. With fixed-size nodes, each field is found at a pre-determined offset within the node. Moreover, because node sizes are constant, a page can be represented as an array of nodes, within which individual nodes can be found quickly. However, when leaves and pointers require different amounts of space, fixed-size nodes require more space than necessary. For example,

the leaves of a black-and-white image would be stored in a large pointer field, even though only one bit is needed to represent the attribute values of a leaf. Since $\frac{3}{4}$ of all pointer fields hold leaves, fixed-size nodes potentially waste a great deal of space.

Variable-size nodes eliminate the cause of such wasted space. With variable-size fields, leaves requiring small amounts of space consume only the space required. On the other hand, attributes in quadtrees such as the PR (point) quadtree and the PMR (line) quadtree generally require *more* space than normal pointers. In this case, the size of pointer fields would have to increase with fixed-size nodes, even though the number of nodes in the quadtree does not justify such large pointers. Since variable-size nodes require less space, the number of pages needed to store the quadtree would be reduced. However, the primary drawback of variable-sized nodes is that individual nodes within a page and individual fields within a node cannot be accessed directly, as with fixed-size nodes.

The paged-pointer quadtree combines these approaches, using fixed-size nodes in main memory and variable-size nodes on disk. The variable-size nodes on disk allows a greater number of nodes to be fit on each page on disk and therefore decreases the number of pages required to store a quadtree. The buffers holding pages use fixed-size nodes to allow direct computation of the location of a node within its page. This combined approach makes it necessary to convert nodes between the two representations. We refer to this conversion process as *compression* and *decompression*.

Conversion is necessary only when a page is read from or written to disk. When the disk is accessed, the cost of the read or write far exceeds the cost of the computation needed to perform the compression and decompression. The compression algorithm generates a stream of bits to be written to the page on disk. Each node is added to the stream in order. Within a node, the tag bit of each field is written, followed by the contents of that field. If pointers require 31 bits and leaves require one bit (both plus the tag), we add 32 bits to the stream for each pointer and two bits for each leaf. Decompression reads from the stream of bits previously written to a page in the same manner. For each field, the first bit read (i.e., the tag bit) indicates the field type and thus the number of bits to read for the remainder

of the field.

This combined approach uses buffers in memory larger than the page size. Because of the compression, each node will generally require less space on disk than in memory. A collection of nodes filling a disk page completely would require more space, and hence a larger buffer, in memory. The amount of free space on a page, whether in memory or on disk, refers to the amount of available space when the page resides on disk. Each page holds the amount of available space on the page, and updates this count as the nodes of the page change. An insertion or deletion would cause the amount of free space to go below zero or above one-third of the page size would leave a page either overfull or underfull. In this case, nodes are moved using the B*-tree rules discussed in Section 3.3.2.

This approach has several advantages over the fixed-size method. The primary one is that the compression scheme reduces the amount of disk space used by a pointer-based quadtree. By saving disk space, compression also reduces the number of disk accesses required when traversing the structure. Second, because the compression is performed only when a page fault occurs, its cost is negligible compared to the cost of the disk access. Third, because a greater number of nodes are stored on any given page, the probability that neighboring nodes will be located on the same page increases. Finally, this representation can easily be extended to represent quadtrees where the number of bits required by leaves exceeds the size of the tree's pointers. A system based with fixed-size nodes can do this only by increasing the size of the pointers.

On the other hand, compressing nodes in this manner is not without its drawbacks. First, performing the compression and decompression requires extra computation not needed with fixed-size nodes. Even though this computation time is small compared to the time required for disk access, it is not completely negligible. Second, since the compression scheme holds nodes in uncompressed form when in main memory, the amount of buffer space needed to hold each page is greater than the disk page size. This means that for a given amount of buffer space, more pages can be held in main memory when compression is not used. The total number of nodes that can be held in the buffer pool should be the same for either

method. In any event, since very little buffer space is generally required, this factor is not crucial. It should also be noted that compression is not mandatory, and if omitted leaves a disk-based implementation with nodes identical to those of the fixed-size approach of [OHS90]. The paged-pointer quadtree was originally implemented using fixed-size nodes but now supports compression because of the space savings.

3.2.3 Ordering of Nodes

One of the most serious limitations of traditional pointer-based quadtrees is that the representation offers no guarantee of locality. Virtual memory systems exploit the fact that memory accesses patterns (e.g., fetch-execute cycle) are often sequential in nature. This means that when a page is accessed, that access and many subsequent ones will all involve this single page. Access patterns in traditional pointer-based quadtrees have no such sequential nature. If a quadtree is built in preorder, neighboring nodes can be found near each other in the same way as in linear quadtrees. However, this is unlikely in quadtrees that have been frequently modified. When a quadtree is stored on disk, this lack of locality can cause an unacceptable number of page faults.

The nodes of a paged-pointer quadtree are arranged according to a preorder traversal to provide better locality. The records of both linear quadtrees and DF-expressions are also arranged in preorder, either explicitly or via locational codes. Storing the nodes of a pointer-based quadtree in this order has several advantages. First, it is relatively easy to keep the nodes in this order. Second, when the entire data set must be processed, traversals of the quadtree are generally done in preorder. Storing nodes in preorder produces a sequential reference pattern allowing the entire data set to be processed while using each page only once. Third, when a quadtree is stored in preorder, the nodes of subtrees deep in the quadtree will be grouped together, suggesting that they are likely to be on the same page. When searches within a quadtree reach these lower levels, further page faults are unlikely. Because of this, the number of page faults actually occurring in a search is somewhat less than the depth of the tree. Other orderings may lead to different and perhaps more efficient

clusterings; Schkolnick [Sch77] presents algorithms to order existing hierarchical structures to minimize page faults. However, such optimal clusterings must be recomputed whenever the structure changes.

While ordering quadtrees in preorder has several advantages, it is not completely trivial to accomplish. As nodes are added and removed from a pointer-based quadtree, there is potential for fragmentation. While the page-management routines of B-trees limit fragmentation with only local changes to the structure, corresponding changes in a pointer-based quadtree are not necessarily local. Each pointer to a node moved to reduce fragmentation must also be updated to reflect the new location of the node. While the close relatives (i.e., parent, children) of a node are often found on the same or neighboring pages, updating relatives on different pages may cause page faults. Since moving nodes can be so costly, the order cannot be maintained simply by always keeping nodes physically in order. Section 3.3.2 explains in detail the problems in managing a frequently changing tree and how the page-management methods of the B-tree are extended successfully to the paged-pointer quadtree.

3.3 Implementation of the Paging Scheme

This section describes the more complicated issues encountered in implementing the paged-pointer quadtree. Section 3.3.1 describes how our implementation manages the buffer pool. Section 3.3.2 describes how the pages of the paged-pointer quadtree are managed as the structure of the quadtree changes. Section 3.3.3 analyzes the way pointer-based quadtrees are accessed and shows the shortcomings of standard virtual memory techniques when applied to pointer-based quadtrees.

3.3.1 Design of the Buffer Pool

The most serious drawback of implementing application-specific memory management routines rather than using standard virtual memory techniques is the lack of hardware support.

When an address is referenced somewhere in a virtual memory space, support hardware can quickly locate the needed page. If the page is somewhere in main memory, then the address is translated in hardware to the physical location of the page in main memory. If the page is not in main memory, the operating system first fetches the page in from the swap space. When virtual memory facilities are supported only in software, address translation is a much slower process.

The traditional virtual memory facilities found on the machines used in our implementation have several major shortcomings (see p. 39). Therefore, our implementation of the paged-pointer quadtree provides the necessary VM facilities (i.e., buffer pool) in software. Since every node access requires the use of this software, it must be able to quickly find a node in the memory space. In our buffer pool, a page table holds the page number and address of the buffer holding that page. This is implemented as a hash table (using page number as a key) to allow constant average-time access to page table entries. We employ a simple modular hash function using the page number. A quick lookup in the hash table can determine if a page is in the buffer pool, and if so, where it is located. To facilitate easy replacement of the least recently used page, the records in the hash table also have fields to implement a doubly-linked list ordered by reference time. Whenever a page is referenced, this list is updated to reflect the new reference order.

Because address translation must be performed on every node accessed, the address translation code consumes a significant amount of CPU time. In our implementation, converting this code from a normal function to an inline function reduced the time needed to build a quadtree from raster data by 30%. Other optimizations are also employed to speed the translation process. For instance, since the nodes of a paged-pointer quadtree are stored in preorder, many neighboring nodes in a traversal of the quadtree are stored on the same page. The buffer pool stores the number and address of the most recently referenced page and quickly checks if the desired node is on that page. We found that consecutive accesses involve the same page between 70% and 80% of the time when building a quadtree from raster data. While the address translation process in our buffer pool is not

nearly as fast as hardware-supported virtual memory facilities, it still provides adequate performance. Operating systems supporting memory-mapped files overcome the shortcomings of traditional VM systems and could satisfactorally replace our buffer pool code. Using memory-mapped files would significantly improve the speed of address translation, and thus our entire implementation.

3.3.2 Dynamic Management of Disk-Based Quadtrees

As discussed in Section 3.2.3, the nodes of a paged-pointer quadtree are stored in preorder. The primary motivations for this approach are to facilitate efficient preorder traversals and provide good locality of reference. As nodes are inserted into and deleted from a paged-pointer quadtree, nodes may have to be relocated to preserve this order.

In a B-tree, the records in a page are kept strictly in sorted order, so that a binary search can be used to determine the part of the tree in which a desired record belongs. When a new record is added to a page, other records on the page are shifted to make room for the new record. This process requires no disk accesses and keeps the records of any page in order so that they can be searched quickly. The same basic approach is used when overflow or underflow occurs on a page, except records are shifted among a small collection of pages, all in memory.

However, this approach is not as appropriate in a pointer-based quadtree. When a new record is added to a page in a B-tree, on average, half of the records on the page must be moved. Although nodes in a paged-pointer quadtree can similarly be moved in memory, every pointer in the quadtree pointing to one of these nodes must also be updated. While the close relatives of many nodes moved are on the same page, updating off-page relatives requires fetching these other pages and introduces the possibility of page faults. If nodes within a page are maintained strictly in preorder, then any insertion may be costly.

While our implementation of the paged-pointer quadtree conceptually stores nodes in preorder, it is too costly to order them physically. Our approach still guarantees that if one node appears before another in a preorder traversal, it is found on the same or a prior page.

Despite the cost of shifting nodes, there are times when shifting must take place. When a node is to be added to a full page, or when a node to be removed from a page would cause the page to drop below minimum capacity, nodes are moved to allow the insertion or deletion to take place. Our implementation manages the nodes of a paged-pointer quadtree in a way similar to the B*-tree, guaranteeing that each page is at least two thirds full.

The reorganization process in the paged-pointer quadtree resulting from overflow or underflow consists of four steps:

1. Identifying the pages involved in the reorganization, referred to as the *input pages*.
2. Deciding the specific action to take: *splitting* two pages into three, *merging* three pages into two, or *shifting* nodes among three pages. The action taken identifies the set of *output pages*, where the nodes involved in the reorganization end up.
3. Extracting the order of the nodes involved in the reorganization from the structure of the quadtree.
4. Assigning each of these nodes to an output page.

Identifying Input and Output Pages

Our implementation performs the first two steps in exactly the same manner as a B*-tree. When a node is added to a full page, both neighboring pages are examined. If sufficient free space is available on these pages, nodes are shifted from the full page to the neighbors. Similarly, when a record is deleted from a page at minimum capacity, neighboring pages are examined. If either neighbor is above minimum capacity, nodes are shifted to the underfull page. In either case, the page management routines perform the *shift* operation. Nodes may be moved among the affected page and its two neighbors; these three pages are both input pages and output pages.

If a node is added to a full page and no space is available on neighboring pages, a *split* is performed. In a split, a new page is added and some nodes from each page are moved to this new page. The current page and one of its neighbors are the input pages. These two pages and the new page are the output pages. If a node is deleted from an underfull page

and both neighbors are also near minimum capacity, a *merge* is performed. In a merge, the nodes of the current page are moved to the neighboring pages, and the current page is freed for reuse. The current page and its neighbors serve as the input pages; the two neighbors are the output pages.

Extracting Node Order

In this step, we determine the order of the nodes in the input pages. This is trivial in a B-tree, since records are ordered physically. In the paged-pointer quadtree, the cost of physically ordering nodes within a page is prohibitive. However, since the paged-pointer quadtree explicitly represents the quadtree structure, the order of the nodes can be extracted from this structure. To facilitate extraction, each page in our implementation stores a pointer to its first node in preorder. When pages are reorganized, these pointers are updated to reflect the new "first" node of each page. We order the nodes in the input pages by performing a preorder traversal of the quadtree starting with the first node in the input pages. Our preorder traversal visits the nodes of the input pages until it reaches a node not in the input pages or the last node of the quadtree.

The extraction is performed by the procedures EXTRACT and TRAVERSE in Figure 3.1. The page numbers of the input pages are stored in the list PAGES, and our ordered list of nodes is returned in NODES. The function TRAVERSE is used to traverse the subtree rooted at the node *root*. The nodes of this subtree are visited in preorder and appended to NODES. If TRAVERSE reaches a node not in the input pages, it returns FALSE. If it traverses the entire subtree without finding such a node, TRAVERSE returns TRUE.

EXTRACT begins its traversal with the first node in the input pages being the *current* node. It calls TRAVERSE to traverse the subtree rooted at *current*. If TRAVERSE returns false, all nodes in the input pages have been appended to NODES. If not, EXTRACT then visits the node's parent, which we call *parent*. Since *parent* precedes *current* in preorder, it is not added to NODES. When *current* is not the last child of *parent*, the subsequent

children and their descendants immediately follow in preorder. TRAVERSE appends the nodes of these subtrees to NODES, returning FALSE if the nodes of the input pages are exhausted. If all the descendants of *parent* are visited without reaching a node not in the input pages, EXTRACT repeats this process with *parent* as the new current node.

The algorithm EXTRACT produces an ordered list of the nodes in the input pages. However, when EXTRACT visits the parent of the current node, the page holding the parent may not be in the buffer pool, causing a page fault. However, it is likely that most of these nodes are ancestors of the node causing the need for reorganization. If the insertion or deletion process causing the reorganization begins at the root of the tree, such ancestors were visited just prior to the reorganization. Because of LRU replacement, these pages will probably remain in memory. Still, this may not be true of all ancestors visited. However, the number of page faults due to this algorithm is expected to be low.

Assigning Nodes to Pages

After we order the nodes of the input pages, each node is assigned to an output page. Since the ordering of the nodes among the output pages must be preserved, this assignment is performed by splitting the list obtained by EXTRACT. Two factors are used in determining where to split the list. First, as in a B-tree, it is desirable to balance the nodes among the output pages. An even split will allow room for several more insertions or deletions before another (possibly costly) reorganization is required. Second, the splitting algorithm seeks to avoid splitting nodes in the middle of a subtree. This approach often keeps entire subtrees on the same page and thereby reduces the number of page faults for searches extending deep into the quadtree. Prospective split points are evaluted using the heuristic function SPLIT, which incorporates both factors:

$$\text{SPLIT} = 100 \cdot \mathit{level} - (\mathit{size} - \mathit{ideal})^2.$$

In this function, *level* gives the height of the smallest subtree broken up by the prospective split point, *size* gives the number of bytes used in the page ending at the split point, and

```
var
    PAGES : page list;
    NODES : node list;

function TRAVERSE (node pointer root) : boolean
begin
  var child : node pointer;

  if root not in PAGES then
    return FALSE;

  append root to NODES;
  for each child of root do
    if child is a pointer then
      if TRAVERSE (child) = FALSE then
        return FALSE;
  return TRUE;
end


procedure EXTRACT (node pointer first)
begin
  var current, parent, child : node pointer;

  current := first;
  if TRAVERSE (current) = TRUE then
    while (current) is not the root of the quadtree do begin
      parent := parent of current;
      for each child of parent following current in preorder do
        if TRAVERSE (child) = FALSE then
          return;
      current := parent;
    end
end
```

Figure 3.1: Extracting node order from a paged quadtree

ideal gives the number of bytes used on that page resulting from a completely even split. The power of two gives preference to our first factor when a prospective split is not close to even. When a split is more even, the large constant gives preference to our second factor.

Though ordering of the input pages and assigning each node to an output page cause few page faults, moving nodes involves the possibly costly process of updating each pointer to a moved node. If a node in the input pages is already on its destination page, it is not moved. Otherwise, the node must be moved and may cause page faults as pointers to this node are updated. While it is difficult to determine a priori the number of page faults obtained, it is likely that the parents and children of any single node will be found on the same page. Still, the number of nodes moved gives a good indication of the number of page faults occuring. If the capacity of a page is k nodes, the worst case for page splits and merges involves moving $\frac{2}{3}k$ nodes. For balancing due to either overflow or underflow, at most $\frac{2}{9}k$ nodes will be moved.

3.3.3 Access Patterns in Pointer-Based Quadtrees

The page replacement algorithm in a virtual memory system (in either hardware or software) significantly affects the performance of the system. If the replacement algorithm closely matches the pattern in which memory is accessed, few page faults result. However, performance suffers if pages are accessed in a manner not matching the replacement algorithm. General-purpose virtual memory systems typically seek to replace the least recently used (LRU) page in memory. Because the most-recently referenced pages are often more likely to be used again in the near future, LRU replacement works well. When this is not the case, the number of page faults increases and performance suffers. Unfortunately, this is often the case when pointer-based quadtrees are traversed.

Standard virtual memory techniques succeed by exploiting locality in computer programs. For example, in the fetch-execute cycle, the next instruction to be fetched usually follows the current instruction in memory. Accesses to arrays and similar data structures often occur in a sequential manner. Loops within a program often cause the same area of

memory to be accessed over and over again. In each example, the memory access patterns are such that if a page has been recently used, it is likely to be used again. Denning [Den68] refers to the set of pages accessed in a given period of time as the *working set*. When the working set of a program is larger than the amount of available memory, *thrashing* occurs since pages are must be repeatedly fetched from the swap space.

One of the most common patterns used to access quadtrees is the preorder traversal. Since linear quadtrees and DF-expressions are arranged in preorder, the access pattern of a preorder traversal is strictly sequential and results in a minimal number of page faults. This is not necessarily the case with pointer-based quadtrees, even if they are stored in preorder. The traversal begins at the root of the tree and visits the great number of nodes found in the lower levels of the quadtree. As these pages holding these nodes are processed, accesses of the pages holding distant ancestors become less and less recent. This will eventually cause the pages containing these ancestors to be swapped out. When the traversal returns to such an ancestor, it is no longer in memory and its page be fetched again. Repeatedly fetching such pages will significantly slow a preorder traversal of a pointer-based quadtree.

When the quadtree to be traversed is not modified, maintaining a stack holding the ancestors of the current node will overcome this problem. When the traversal later returns to the ancestor, the algorithm accesses the copy of the ancestor kept in the stack and prevents the possibility of a page faults due to revisiting that node. Using the stack thereby minimizes the number of page faults resulting from a preorder traversal. However, when a quadtree is modified, the page management routines may relocate nodes. After a reorganization, the copy of an ancestor in the stack may be invalid, since it might point to a node which has since moved. When relocation occurs, the validity of each node in the stack is in doubt. Therefore, maintaining a stack is inappropriate when the quadtree is modified.

Even when a quadtree is modified, additional information exists that standard LRU algorithms do not exploit. General-purpose LRU algorithms cannot determine which nodes are to be revisited and which will not be referenced again. However, this information can be maintained by the traversal algorithm. When a traversal visits a node and continues

with its descendants of the node, the node will certainly be revisited later. Our buffer pool implementation provides a *locking* feature not supported by standard virtual memory system. Since a traversal returns to each ancestor of the current node, the pages containing these nodes are locked into main memory when first visited. Locking indicates to our page replacement algorithm that the page should not be swapped out since it will be accessed later. When the traversal returns to that node for the last time, the page is then unlocked. The number of pages locked at any one time is bounded by the depth of the quadtree. The buffer pool must be large enough to hold a good number of pages other than those that are locked; otherwise, the presence of locked pages severely decreases the effective size of the buffer pool.

3.4 Summary

There many issues involved in implementing paged-pointer quadtrees, the most crucial of which were discussed in Section 3.2 and Section 3.3. Although our implementation is more complicated than that of the B-tree used to organize linear quadtrees, all the necessary code is provided in a library. Our implementation provides a high-level interface to allow programmers to use paged-pointer quadtrees without being concerned about these minute details. Standard quadtree algorithms used with pointer-based quadtrees in memory are easily be adapted to our representation. The application programmer need not be concerned with the intricacies of implementation or the fact that these quadtrees are actually stored on disk. Several quadtree algorithms have been implemented using our kernel. The following chapter compares the performance of these programs to that of comparable programs based on linear quadtrees.

Chapter 4

Time and Space Performance

The primary motivation for using linear quadtrees is that they can be easily and efficiently represented on disk. The paged-pointer quadtree presented in Chapter 3 provides a compact representation for pointer-based quadtrees on disk allowing easy implementation of standard pointer-based quadtree algorithms. This chapter compares the efficiency of algorithms using our implementation of the paged-pointer quadtree to similar algorithms using QUILT, an experimental geographic information system based on linear quadtrees developed at the University of Maryland [SSN87].

Our implementation is written in the programming language C, using the GNU C compiler. The code for QUILT is also written in C. Both systems have been implemented on several computers running the Unix operating system, including:

- a DECStation 3100 with 32 MB of RAM and a 25 Mhz MIPS R3000 RISC processor, running Ultrix 4.2;
- a Commodore Amiga 3000UX with 8 MB of RAM and a 25 Mhz Motorola 68030 processor, running Amiga Unix 2.03;
- a Macintosh II with 8 MB of RAM and a 16 Mhz Motorola 68020 processor, running A/UX 2.0.

Our implementation was originally developed on a IBM-compatible 386 system, running ESIX System V, Release 3.2. The code for both systems could likely be ported to other

systems with little difficulty.

4.1 Building Quadtrees from Raster Data

The spatial data represented by a quadtree is rarely obtained in quadtree form. The most common sources of pixel data are binary, grayscale, or color images represented in raster-scan order. To use a quadtree to access and manipulate such data, a raster image must first be converted into quadtree form.

A simple approach, presented in [Sam81], begins with a quadtree (having only one node) that represents an entirely "uncolored" image. As each pixel is visited in raster-scan order, the quadtree is modified to reflect the attribute values of that pixel. This quadtree validly represents the pixels that have been visited and assumes that all other pixels are "uncolored." When the algorithm visits a pixel that breaks up a homogeneous region, the corresponding node is split. When the final pixel of a node is visited, the algorithm merges the children of the corresponding node if they are identically-valued leaves. Though the intermediate stages of this quadtree typically require little more space than the final version, the need to merge like-valued leaves requires additional computation. If the representation used in this algorithm is disk-based, merging may also cause additional disk accesses as nodes are inserted and later deleted.

Shaffer and Samet [SS87] present an improved version of this approach that builds a linear quadtree with node accesses proportional to the number of nodes in the final quadtree. In the algorithm of [Sam81], nodes are created and then later deleted, and the total number of nodes inserted and deleted is proportional to the number of pixels in the original image. The algorithm of [SS87], on the other hand, inserts a node only when it is known not to merge with its siblings. As rows of pixels are read, a leaf in the output quadtree is *active* if at least one but not all of its pixels have been visited. The number of active nodes at any point in the building process is less than 2^h for a $2^h \times 2^h$ image. Because of this, the active nodes at any point in the process can be kept in a suitably small buffer. While this

algorithm, which must process each pixel, still requires $\Theta(2^{2h})$ time, the cost of inserting and deleting nodes in a disk-based representation may substantially exceed that of processing the 2^{2h} pixels. In this sense, the extra insertions required by other approaches make this algorithm a vast improvement.

Shaffer and Samet [SS87] present a similar approach suitable for pointer-based quadtrees. After visiting the first pixel of an image, the quadtree is represented as a single leaf with the same color as that pixel. When a pixel of a different color is visited, the current leaf is split, and its four children assume the leaf's original color. The child containing this pixel then becomes the current node and the process is repeated until this pixel is found in the upper-left corner of the current leaf. At which time, the current leaf assumes the color of the pixel. When the process reaches a pixel not found in the current leaf, *neighbor-finding* is performed to locate the node adjacent to the current leaf that contains the pixel. Finding the neighbor of a node in a pointer-based quadtree involves locating the nearest ancestor containing the current node and the desired neighbor and then descending the tree to the neighbor. Neighbor-finding is a common operation in pointer-based quadtrees and visits a constant number of nodes in the average case (fewer than $\frac{7}{2}$) [SS85]. This construction algorithm does not insert any unnecessary nodes and thus runs with node accesses proportional to the number of nodes in the final quadtree.

QUILT implements the linear quadtree building algorithm of [SS87]. Three building algorithms were implemented for pointer-based quadtrees. The first (*rootbuild*) is a direct translation of the linear quadtree build algorithm, where leaves are inserted into the tree by a traversal beginning at the root. The second (*nfbuild*) is the pointer-based building algorithm based on neighbor-finding given in [SS87]. The third and most efficient algorithm (*build*) is a slight modification of *nfbuild*, processing one row from left to right and the next from right to left. This significantly decreases the number of times that high-level subdivisions of the output quadtree are crossed during neighbor finding. Though each of these algorithms is built on top of our paged-pointer representation, the order in which nodes are inserted depends on the algorithm used.

Test images for building operations—"floodplain," "landuse," and "topography."

Figure 4.1: Test data for building operation

The performance of the pointer-based and linear quadtree build algorithms was tested on six raster images. The first three images were generated as test data during the design of QUILT and are shown in Figure 4.1. They of consist maps portraying the flood plain, land usage, and topographical data in a river valley in northern California and are referred to as "floodplain," "landuse," and "topography." Each of these images is 400 pixels wide and 450 pixels high and are represented as 512×512 pixel images by the quadtrees. The other three images, called "big floodplain," "big landuse," and "big topography," are 800×900 and consist of four copies of the corresponding original image. They are represented as 1024×1024 pixel images as quadtrees.

The build algorithms were tested on the three 400×450 pixel data sets on each of the three target machines. The three 800×900 data sets were also built on the faster DECStation. As Table 4.1 indicates, the resulting pointer-based quadtrees are typically about 40% smaller than the corresponding linear quadtrees built by QUILT. When these quadtrees are built, nodes are not inserted in order and the B-tree algorithms leave free space within pages. Because nodes are inserted in different orders in our three pointer-based

Image Name	Leaf Count	Pointer-Based Quadtrees			QUILT build	% Savings for *build*
		build	*nfbuild*	*rootbuild*		
Floodplain	5,206	36,864	36,864	38,912	59,392	37.9
Landuse	28,549	186,368	188,416	178,176	302,080	38.3
Topography	25,012	161,792	161,792	159,744	268,288	39.7
Big floodplain	19,426	126,976	129,024	133,120	216,064	41.2
Big landuse	112,819	696,320	692,224	686,080	1,197,056	41.8
Big topography	98,491	608,256	618,496	614,400	1,045,504	41.8

Table 4.1: Bytes used to represent built quadtrees

algorithms, these page management routines may leave quadtrees of different sizes. QUILT, whose B-tree splits three pages for four [SSN87], leaves approximately 20% of each page free. Our paged-pointer quadtrees, which splits two pages for three, leave about 25% free.

Both representations provide a compaction operation that eliminates this free space. The space used by the compacted versions of these quadtrees is given in Table 4.2; our packed pointer-based quadtrees are about 45% smaller than QUILT's linear quadtrees. Using analysis similar to that of Section 2.4, we expect our paged-pointer quadtrees to use just over 37 bits per leaf and the linear quadtrees of QUILT to use 64 bits per leaf. This results in an expected space savings of 42%, which does not include the added cost of representing the internal structure of the B-tree needed by QUILT. A significant part of this advantage is due to the fact that our paged-pointer quadtree encodes leaves in 16-bit fields, while QUILT uses a fixed-size attribute value field of 32 bits. If we also used 32 bits, we would expect a space savings of only 17%. However, most of this is offset by the extra storage consumed by parent pointers, which are not strictly necessary. With 32-bit leaves but no parent pointers, our paged-pointer quadtrees would be just over 33% smaller.

The average times required to build the quadtrees on our three machines are given in Table 4.3, Table 4.4, and Table 4.5. Real time refers to the amount of time spent to execute the entire command, user time refers to the amount of that time spent executing the user's code, and system time refers to the amount of time spent in the operating system kernel. On

Image Name	Pointer-Based Quadtree	Linear Quadtree	Percent Savings
Floodplain	26,624	48,128	44.7
Landuse	137,216	251,904	45.5
Topography	120,832	221,184	45.4
Big floodplain	94,208	173,056	45.6
Big landuse	534,528	983,040	45.6
Big topography	466,944	858,112	45.6

Table 4.2: Bytes needed to represent packed quadtrees

Image Name	Pointer-Based (*build*)			QUILT		
	real	user	system	real	user	system
Floodplain	10.6	7.6	1.8	12.6	8.8	1.9
Landuse	35.2	29.7	2.7	39.4	27.1	9.7
Topography	37.0	27.9	3.4	34.3	24.0	8.2
Total	82.8	65.2	7.9	86.3	59.9	19.8

Table 4.3: Build times (in seconds) for first three data sets—Macintosh

each system, builds are faster using our pointer-based representation. On both the Amiga and the DECStation, our implementation is nearly twice as fast. On each machine, there is a pronounced difference in the system times of the algorithms; the linear quadtree system spends substantially more time performing I/O. However, user times are generally higher for our pointer-based quadtrees, due to the overhead involved in translating each pointer into a address in the buffer pool.

Because a great deal of computation time in our implementation involves translating addresses, these translations should be performed quickly. Profiling our code indicates that our highly optimized address translation code still consumes between 20% and 30% of total execution time. One of our optimizations is a quick test to determine if a node is on the same page as the previously visited one. When this is the case, neither table lookup nor page table maintenance is necessary. As Table 4.6 indicates, consecutively referenced nodes

Image Name	Pointer-Based (*build*)			QUILT		
	real	user	system	real	user	system
Floodplain	5.4	4.5	0.6	4.5	3.5	0.9
Landuse	21.0	19.2	1.4	48.8	11.4	9.7
Topography	19.4	17.1	1.3	34.0	11.2	7.8
Total	45.8	40.8	3.3	87.3	26.1	18.4

Table 4.4: Build times (in seconds) for first three data sets—Amiga

Image Name	Pointer-Based (*build*)			QUILT		
	real	user	system	real	user	system
Floodplain	1.3	0.7	0.2	1.9	0.8	0.8
Landuse	6.7	3.3	0.5	13.2	2.6	2.6
Topography	5.6	3.0	0.4	9.9	2.4	2.2
Big floodplain	5.6	2.9	1.1	7.3	3.3	1.7
Big landuse	54.9	34.0	7.3	104.9	11.4	22.1
Big topography	47.4	27.7	6.3	85.0	10.2	17.5
Total (real)	121.5	71.6	15.8	222.2	30.7	46.9

Table 4.5: Build times (in seconds) for all size data sets—DECStation

in the output quadtree are on the same page about 80% of the time with both *build* and *nfbuild* and 70% of the time with the algorithm *rootbuild*. While our neighbor-finding build algorithms sometimes reference more nodes overall, the increased frequency of same-page references translates into slightly faster execution for these algorithms.

In addition quickly translating a pointer into an address in the buffer pool, we wish to minimize the number of page faults encountered. While there is little difference in the number of page faults among the three algorithms on the small data sets of Table 4.6, *build* produces about half as many page faults with the two largest data sets (i.e., "big landuse" and "big topography"). During a build, nodes belonging to the left half of a large image and those belonging the right half are in different quadrants and consequentially nowhere near each other on disk. In both *rootbuild* and *nfbuild*, the working set of the build algorithm shifts from the left side of the tree to the right side and back again as each row is processed. Each time this occurs, the working set changes and nodes from the current side of the tree may replace those on the other side in memory. Because *build* alternately processes from left to right and then right to left, the working set will change only once on each row rather than twice. This observation is compatible with the observed behavior of the algorithms found in Table 4.6.

To compare the performance of our implementation the paged-pointer quadtree to a similar approach using virtual memory, we implemented an algorithm identical to *build* (i.e., bidirectional, neighbor-finding) that builds a pointer-based quadtree in memory. This algorithm was then tested against *build* on the Amiga, where the amount of available main memory was insufficient to hold our largest data sets. We used four different versions of each image in Figure 4.1, consisting of one, four, nine and sixteen copies of the original image. Several of these were large enough to require the use of virtual memory. The times required to build the quadtrees are given in Table 4.7. For smaller images, the virtual memory approach is faster since no swapping is necessary (i.e., virtual memory is not actually used). For larger images, the better locality provided by the paged-pointer quadtree results in faster builds; on the images consisting of nine copies of "landuse" and "topography", builds

Image Name	Statistic	Algorithm		
		build	*nfbuild*	*rootbuild*
Floodplain	Nodes referenced	57,953	60,264	49,867
	% on same page	82.2	80.7	73.7
	Page faults	17	17	18
	Execution Time	1.3	1.5	1.8
Landuse	Nodes referenced	244,516	248,952	277,885
	% on same page	80.2	79.8	69.7
	Page faults	151	185	135
	Execution Time	6.7	6.0	6.1
Topography	Nodes referenced	209,029	211,709	241,747
	% on same page	79.9	79.7	70.0
	Page faults	153	197	176
	Execution Time	5.6	5.8	5.5
Big floodplain	Nodes referenced	217,328	222,716	191,286
	% on same page	80.7	79.9	71.0
	Page faults	66	66	74
	Execution Time	5.6	5.9	8.4
Big landuse	Nodes referenced	963,299	973,177	1,167,927
	% on same page	80.0	79.7	67.0
	Page faults	5,167	11,038	10,278
	Execution Time	54.9	102.6	104.1
Big topography	Nodes referenced	827,452	837,793	1,006,642
	% on same page	79.4	79.1	67.3
	Page faults	3,955	8,333	7,276
	Execution Time	47.4	82.6	81.9

Execution time in seconds on DECStation

Table 4.6: Paging statistics for pointer-based build algorithms

Image Name	Number of Copies	Paged-pointer quadtree	Virtual memory quadtree
Floodplain	1×1	5.4	3.4
	2×2	18.6	11.8
	3×3	57.6	66.6
	4×4	137.5	145.9
Landuse	1×1	21.0	10.4
	2×2	73.1	98.7
	3×3	219.0	338.9
	4×4	384.3	*
Topography	1×1	19.4	9.0
	2×2	64.1	66.8
	3×3	194.1	280.3
	4×4	337.2	*

* — VM build algorithm unable to allocate enough virtual memory

Table 4.7: Build times (in seconds) for paged-pointer quadtrees and virtual memory quadtrees—Amiga

were roughly 50% faster.

4.2 Converting Quadtrees to Raster Data

To display the spatial data represented by a region quadtree, the quadtree is first converted back to raster form. This operation has been implemented both by QUILT and our pointer-based system, and performance was measured by reconverting the six quadtrees built in Section 4.1. The linear quadtree algorithm implemented by QUILT maintains an array holding those leaf nodes not completely in either the processed or unprocessed portion of the image. When a new pixel is processed, this array is consulted. If this pixel is covered by a leaf in the array, its attribute value (i.e., color) is already known. If no leaf covers this pixel, the database holding the linear quadtree is searched and the leaf located is added to the array. The array holding the "active" leaves is of reasonable size, since no more than 2^h leaves can be active in a $2^h \times 2^h$ image. By maintaining this small array, no leaf in the

Image Name	Pointer-Based			QUILT		
	real	user	system	real	user	system
Floodplain	5.9	1.2	1.4	6.1	1.1	1.4
Landuse	7.0	3.3	1.7	10.6	5.3	4.3
Topography	6.3	2.8	1.7	9.7	4.5	4.3
Total	19.2	7.3	4.8	26.4	11.9	11.0

Table 4.8: Time required for quadtree-to-raster conversion—Amiga

Image Name	Pointer-Based			QUILT		
	real	user	system	real	user	system
Big floodplain	4.1	1.0	0.6	4.5	0.8	0.6
Big landuse	12.6	5.3	1.6	22.2	4.1	8.3
Big topography	11.8	5.5	1.4	20.6	3.6	6.7
Total	28.5	11.8	3.6	47.3	8.5	15.6

Table 4.9: Time required for quadtree-to-raster conversion—DECStation

database is visited more than once.

Several pointer-based algorithms were implemented. Both a direct adaptation of the linear quadtree algorithm and a neighbor-finding approach visiting pixels in the same order used by the algorithm *build* were found to be significantly slower than the linear quadtree algorithm. A third pointer-based algorithm avoids visiting nodes more than once by maintaining a heap holding the internal nodes of the quadtree covering the row being processed. Each record at the bottom of the heap holds information on the currently "active" leaves. Since at most two quadrants of any portion of the image cover the current row, the storage required to implement this heap is also $O(2^h)$. When a new row is processed, the information found in the heap may need to be updated. As Table 4.8 and Table 4.9 indicate, this algorithm is significantly faster than that of the linear quadtree.

4.3 Set Operations

Two common operations in geographic information systems are forming the union and the intersection of two maps. For example, a map consisting of all farmland at an altitude of 2,000 feet or higher can be produced by overlaying a land use map with an elevation map. Using non-hierarchical data structures, this would typically involve finding each area of farmland on the land use map and then consulting the elevation map to determine the elevation. The quadtree is of great use in performing such operations, since large homogeneous parts of the two images are compared all at once. The algorithm used to perform set operations on quadtrees depends on whether the images represented by the quadtree are *aligned* (i.e., same origin and orientation) or *unaligned*.

Performing set operations on aligned quadtrees [HS79] simply requires traversing the input quadtrees in parallel. When leaves are encountered in the traversal, the proper set operation is performed and new nodes are inserted into the output quadtree based on the results. Because the images represented by the input trees are aligned, the current node in each input quadtree corresponds to the same region in the image space. The amount of time required to perform aligned set operations is proportional to the total number of leaves in the input trees. Because the input quadtrees are traversed in preorder, aligned set operations execute efficiently using linear quadtrees, DF-expressions, and paged-pointer quadtrees.

Set operations on unaligned quadtrees are considerably more complex. Shaffer and Samet [SS88] present a linear-time algorithm for set operations on unaligned linear quadtrees. Of the two input quadtrees, one is designated as the aligned quadtree, with which the output tree will be aligned. The other is considered unaligned. This algorithm traverses the aligned quadtree in preorder while searching for those blocks in the unaligned tree that intersect the current node of the aligned tree. This algorithm uses the concept of *active nodes* in the output tree (as in the building algorithm of [SS87]) to minimize the number of insertions. In addition, it stores additional information about the unaligned tree to mini-

mize the number of searches performed. The result is an algorithm for either pointer-based or linear quadtrees that runs in time proportional to the number of nodes in the input trees plus the number of nodes in the output tree. Since searches are performed in the unaligned quadtree, unaligned set operations provide a good test for random access to the page structure of a disk-based quadtree.

4.3.1 Aligned Quadtrees

As a test for the performance of aligned set operations, [SSN87] presents a list of test cases. Most of these involve set operations on portions of the quadtrees built from the sample images of Figure 4.1. Tests 1 and 2 intersect portions of "landuse" and "floodplain," while tests 3–6 intersect portions of "landuse" and "topography." Test 7 returns the union of portions of "floodplain" and "landuse," while test 8 returns the union of portions of "floodplain" and the intersection of portions of "landuse" and "topography." Tests 9 and 10 return the union of portions of "floodplain" and "topography." Table 4.10 and Table 4.11 show the times required to extract the portions of the images and to perform the aligned set operations. The "subset" operation implemented by QUILT is much more general than ours, and as a consequence much slower. However, both the subset operation and aligned set operations are significantly faster using our pointer-based implementation.

4.3.2 Unaligned Quadtrees

Although set operations are often performed on aligned quadtrees, they are easy to implement efficiently. However, set operations on unaligned quadtrees are not so simple, since searches in the unaligned quadtree are necessary. The need for searching makes unaligned set operations a good test of random access to the pages of the quadtree. The unaligned set operation algorithm of Shaffer and Samet [SS88] was implemented directly by QUILT. We implemented a similar algorithm for pointer-based quadtrees.

Table 4.12 and Table 4.13 compare the times required to perform unaligned set operations using the nine test cases presented in [SS88]. In each test, a portion of "floodplain"

Test Number	Pointer-Based		QUILT	
	subset	set ops	subset	set ops
1	2.0	0.3	2.3	0.5
2	2.9	0.6	2.6	0.6
3	7.1	0.7	11.1	1.7
4	6.1	0.8	7.0	1.4
5	6.8	1.2	10.5	1.2
6	5.6	1.9	13.9	2.3
7	3.5	3.1	13.1	4.0
8	8.4	2.8	15.1	2.9
9	2.3	1.6	2.3	1.7
10	1.6	1.6	1.8	0.8
Total	46.3	14.6	79.7	17.1

Tests performed on "floodplain," "landuse," and "topography."

Table 4.10: Time (in seconds) required for aligned set operations—Amiga

Test Number	Pointer-Based		QUILT	
	subset	set ops	subset	set ops
1	9.5	0.5	15.7	1.7
2	8.3	1.9	14.4	2.1
3	23.8	2.3	44.5	5.2
4	21.8	2.2	39.5	5.4
5	17.5	3.0	43.4	5.7
6	18.6	3.4	53.5	7.9
7	16.6	8.6	61.1	9.5
8	29.1	10.2	71.2	11.4
9	11.3	2.9	12.8	3.7
10	5.9	1.6	9.4	0.9
Total	162.4	36.6	365.5	53.5

Tests performed on "big floodplain," "big landuse," and "big topography."

Table 4.11: Time (in seconds) required for aligned set operations—DECStation

Image & Offset	Pointer-Based			QUILT		
	(0,0)	(1,1)	(100,100)	(0,0)	(1,1)	(100,100)
Floodplain	2.3	5.3	2.4	2.1	5.4	2.6
Landuse	4.1	7.2	4.0	5.5	8.1	5.2
Topography	2.7	6.2	3.6	4.7	7.5	4.7
Total	9.1	15.7	10.0	12.3	21.0	12.5

Table 4.12: Time required (in seconds) for unaligned set operations—Amiga

Image & Offset	Pointer-Based			QUILT		
	(0,0)	(1,1)	(100,100)	(0,0)	(1,1)	(100,100)
Big floodplain	2.1	4.6	2.4	1.8	4.3	2.8
Big landuse	5.6	7.0	5.3	6.5	6.7	5.6
Big topography	3.2	6.2	5.6	5.2	5.9	5.5
Total	10.9	17.8	13.3	13.5	16.9	13.9

Table 4.13: Time required (in seconds) for unaligned set operations—DECStation

serves as the unaligned quadtree and is intersected at several different offsets with one of the original maps. Relative to each aligned quadtree, the origin of the unaligned quadtree was set at $(0,0)$ (i.e., aligned), $(1,1)$, and $(100,100)$. These tables show that our implementation performs these unaligned set operations significantly faster on the Amiga. The amounts of time required by the two implementations are nearly identical on the DECStation. The algorithm implemented by both systems is that of [SS88], which was designed for linear quadtrees. It may be possible to implement a more efficient pointer-based algorithm by taking advantage of the internal structure of the quadtree.

Chapter 5

Conclusions

There are many possible representations for spatial data. Quadtrees used to represent pixel, point, or line data have several strengths, including:

- compact representation of images with large homogeneous areas;
- a spatial organization of the image;
- low computational overhead necessary to preserve the structure;
- no inherent limit on the resolution of represented images.

Chapter 2 presents several approaches that have been used to represent quadtrees. The pointer-based quadtree represents the hierarchy with pointers, while the linear quadtree stores a sorted list of records corresponding to the leaves of the quadtree. The DF-expression stores a list of the nodes of a quadtree in the order they are found in a preorder traversal. Since random access is not possible in a DF-expression, it is of little use in many applications. Of the remaining representations, pointer-based quadtrees have several advantages over linear quadtrees:

- random-order traversals of the data are possible;
- attribute values can be stored in internal nodes with less overhead than in a linear quadtree;

- when fixed-size pointers and locational codes are used, pointer-based quadtrees can represent larger images;
- multi-resolution data is easily supported;
- leafless quadtrees are always more compact than corresponding linear quadtrees;
- algorithms operating on pointer-based quadtrees are more straightforward than corresponding ones for linear quadtrees.

The most significant drawback to traditional pointer-based quadtrees is that the structure is not easily stored in pages on disk. While this issue is not particularly important for quadtrees small enough to fit in main memory, it is important for larger data sets. Representing large quadtrees in memory requires the use of disk for swapping and can cause significant performance problems. On the other hand, linear quadtrees can be organized on disk by a B-tree, using the locational codes of the leaves to order the records. The present work presents the paged-pointer quadtree, a mapping of a leafless pointer-based quadtree to disk pages. The nodes of a paged-pointer quadtree are stored in preorder and its pages are managed by routines similar to those used in a B-tree. This mapping overcomes the problems associated with representing pointer-based quadtrees on disk.

Because of its perceived efficiency, the linear quadtree has historically been used to represent quadtrees on disk. On the other hand, traditional pointer-based quadtrees that require swapping to disk have produced unacceptable performance. Therefore, linear quadtrees have generally been used on large data sets, despite the many other advantages of pointer-based quadtrees. The primary motivation for this work is to produce a disk-based representation that exploits the many advantages of pointer-based quadtrees while providing adequate performance. The tests reported in Chapter 4 demonstrate that our implementation of the paged-pointer quadtree produces performance comparable to, and in most cases better than, a very efficient disk-based linear quadtree system.

Future Work

There are several obvious directions for extending the paged-pointer quadtree:

- The paged-pointer quadtree can be integrated into applications using quadtrees (e.g., geographic information systems) and would likely produce better performance than current systems based on the linear quadtree.

- The paged-pointer quadtree can be extended to exploit memory-mapped files provided by some newer operating systems. Representing a file in virtual memory would eliminate the need for the slow process of doing address translation in software.

- Our implementation of the paged-pointer quadtree can be extended to represent point and line data as well. This would require new subdivision rules and support for leaves larger than the size of a pointer.

- The paged-pointer quadtree can be used to test the performance of quadtree algorithms using a pointer-based representation. Current algorithms generally do not account for disk accesses needed when manipulating a large quadtree. In addition, little work has been done on some pointer-based algorithms (e.g., unaligned set operations). The availability of our implementation may motivate further investigation of pointer-based quadtree algorithms.

REFERENCES

[Abe84] D.J. Abel. A B+-tree structure for large quadtrees. *Computer Vision, Graphics, and Image Processing*, 27(1):19–31, 1984.

[Atr88] S. Atre. *Data Base: Structured Techniques for Design, Performance, and Management.* John Wiley and Sons, 2nd edition, New York, NY, 1988.

[Bur87] P.A. Burrough. *Principles of Geographical Information Systems for Land Resources Assessment.* Clarendon Press, Oxford, 1987.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[Den68] P.J. Denning. The working-set model for program behavior. *Communications of the ACM*, 11(5):323–333, 1968.

[dJSS91] W. de Jonge, P. Scheuermann, and A. Schijf. Encoding and manipulating pictoral data with S+-trees. In *Advances in Spatial Databases*, 401–419. Springer-Verlag, Berlin, 1991.

[Gar82] I. Gargantini. An effective way to represent quadtrees. *Communications of the ACM*, 25(12):905–910, 1982.

[Gla84] A.S. Glassner. Space subdivision for fast ray tracing. *IEEE Computer Graphics and Applications*, 17(11):7–25, 1984.

[HS79] G.M. Hunter and K. Steiglitz. Operations on images using quad trees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 1(2):145–153, 1979.

[HSA91] P. Hanrahan, D. Salzman, and L. Aupperle. A rapid hierarchical radiosity algorithm. *Computer Graphics*, 25(4):197–206, 1991.

[KE80] E. Kawaguchi and T. Endo. On a method of binary picture representation and its application to data compression. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 5(4):373–384, 1980.

REFERENCES

[KEY80] E. Kawaguchi, T. Endo, and M. Yokota. DF-expression of binary-valued picture and its relation to other pyramidal representations. In *Proceedings of the Fifth International Conference on Pattern Recognition*, volume 5, 373–384, Rome, November, 1980.

[Knu73] D.E. Knuth. *The Art of Computer Programming, Vol. 3: Sorting and Searching.* Addison-Wesley Publishing Company, Reading, MA, 1973.

[Mor66] G.M. Morton. A computer oriented geodetic data base and a new technique in file sequencing. Technical report, IBM, Ottowa, Canada, 1966.

[NS86] R.C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, 1986.

[OHS90] D.N. Oskard, T.H. Hong, and C.A. Shaffer. Real-time algorithms and data structures for underwater mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, 20(6):1469–1475, 1990.

[RTY88] R. Rashid, A. Tevanian, M. Young, D. Golub, R. Baron, D. Black, W. Bolosky, and J. Chew. Machine-independent virtual memory management for paged uniprocessor and multiprocessor architectures. *IEEE Transactions on Computers*, 37(8):896–908, August, 1988.

[Rya92] T.L. Ryan. Device-independent perspective volume rendering using octrees. Master's thesis, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, February 1992.

[Sam81] H. Samet. An algorithm for converting rasters to quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 3(1):93–95, 1981.

[Sam90a] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS.* Addison-Wesley, Reading, MA, 1990.

[Sam90b] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison-Wesley, Reading, MA, 1990.

[SS85] H. Samet and C.A. Shaffer. A model for the analysis of neighbor finding in pointer-based quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(6):717–720, 1985.

[SW89] H. Samet and R.E. Webber. A comparison of the space requirements of multidimensional quadtree-based file structures. *Visual Computer*, 5(6): 349–359, 1989.

[Sch77] M. Schkolnick. A clustering algorithm for hierarchical structures. *ACM Transactions on Database Systems*, 2(1):27–44, 1977.

REFERENCES

[SH92] C.A. Shaffer and G.M. Herb. A real-time robot arm collision detection system. To appear in *IEEE Robotics and Automation*, April, 1992.

[SJH89] C.A. Shaffer, R. Juvvadi, and L.S. Heath. A generalized comparison of quadtree and bintree storage requirements. Technical Report TR 89-23, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1989.

[SS87] C.A. Shaffer and H. Samet. Optimal quadtree construction algorithms. *Computer Vision, Graphics, and Image Processing*, 37(3):402–419, 1987.

[SS88] C.A. Shaffer and H. Samet. Set operations for unaligned linear quadtrees. Technical Report TR 88-31, Department of Computer Science, Virginia Polytechnic Institute and State University, Blacksburg, VA, 1988.

[SSN87] C.A. Shaffer, H. Samet, and R.C. Nelson. QUILT: A geographic information system based on quadtrees. Technical Report CS-TR-1885, Center for Automation Research and Computer Science Department, University of Maryland, College Park, MD, 1987.

[SPM87] T. Smith, D. Peuquet, S. Menon, and P. Agarwal. KBGIS-II: A knowledge-based geographical information system. *Geographical Information Systems*, 1(2):149–172.

[SSS74] I.E. Sutherland, R.F. Sproull, and R.A. Schumaker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[Tam84] M. Tamminen. Encoding pixel trees. *Computer Vision, Graphics, and Image Processing*, 28(1):44–57, 1984.

VITA

PATRICK RICHARD BROWN
5 Deer Run Road
Kingston, NY 12401

Education

Mr. Brown is an 1990 graduate of Virginia Tech, receiving Bachelor's degrees in Computer Science, Mathematics, and Economics. He graduated first in the University's graduating class of 3669 and was named as "Outstanding Graduating Senior" by the faculty of each of his three major departments. He received a National Science Foundation Graduate Fellowship in 1990, and returned to Virginia Tech to study Computer Science. He was awarded a Master's degree in Computer Science at Virginia Tech in May, 1992.

Honors and Activites

Mr. Brown is a member of Phi Beta Kappa, the Association for Computing Machinery, and Omicron Delta Kappa, a national leadership honor society. He is also a brother in Alpha Phi Omega, a national coeducational service fraternity. He was the captain of the Virginia Tech Programming Team in 1989-90, 1990-91, and 1991-92. Virginia Tech placed first in the Capital Region in 1989, 1990, and 1991. The team finished eighth in the ACM International Scholastic Programming Contest in 1990, third in 1991, and fourth in 1992.

Personal

Mr. Brown was born in Falls Church, Virginia, on August 5, 1968. He lived in Fairfax, Virginia from birth until his graduation from Chantilly H.S. in 1986. He was a resident of Blacksburg, Virginia from 1986 to 1992. He is currently employed by IBM and lives in Kingston, New York. He married the former Cynthia Pruitt on May 27, 1989.