A Paging Scheme for Pointer-Based Quadtrees

Clifford A. Shaffer

Department of Computer Science
Virginia Tech
Blacksburg, VA 24061
shaffer@vtopus.cs.vt.edu

Patrick R. Brown

IBM Corporation
Neighborhood Road
Kingston, NY 12401
pbrown@vnet.ibm.com

Abstract: Hierarchical data structures, most notably the quadtree, have become increasingly popular for the indexing of the large databases required by GIS since they support the efficient computation of traditional GIS spatial analysis algorithms. While the quadtree has many representations, the *linear quadtree* has become the standard. Initially thought to be more space efficient than traditional *pointer-based* quadtree representations, its main operational virtue has been its ability to minimize data transfers between main memory and disk. Recent results show that a good pointer-based representation is more space efficient than the linear quadtree. This paper presents a pointer-based representation for quadtrees called the *paged-pointer quadtree*, which partitions the nodes of a pointer-based quadtree into pages and manages the pages using B-tree techniques. A paged-pointer quadtree always requires less space than the corresponding linear quadtree. Our initial implementation provides better performance than a highly optimized system based on linear quadtrees.

1. Introduction

This paper describes the *paged-pointer quadtree*, a disk-based implementation for pointer-based quadtrees. The *quadtree* [Sam90a, Sam90b] is a family of data structures that represent spatial data using recursive subdivision. The quadtree subdivides the unit square into four equal quadrants, which themselves may be further subdivided. Subdivision ceases when some decomposition criteria are met. For example, the *region quadtree* subdivides a $2^n \times 2^n$ array of pixels until all the pixels in each quadrant are of the same color.

The *pointer-based quadtree* is the direct representation of the hierarchy obtained when a quadtree is built (see Figure 1). Each block created by the subdivision corresponds to a node in a 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 meeting the decomposition criteria (e.g., a uniform block) has no children and is referred to as a *leaf node*. A traditional pointer-based quadtree explicitly represents both internal nodes and leaf nodes, with pointers in internal nodes explicitly determining the tree structure.

The *linear quadtree* [Gar82] provides a way to represent the tree structure of the quadtree without the space overhead involved with pointers. A linear quadtree consists of a sorted list of records corresponding to the leaf nodes of the quadtree. Stored with each leaf node is a *locational code*. Locational codes represent the location of a leaf within the image space and are usually based on the Morton code [Mor66]. For a given integer coordinate pair, the Morton code is created by bit interleaving the x and y values. In particular, the *FD location code* [Sam90a] stores a fixed length Morton code for some specified pixel within the node (such as the upper left corner) combined with an indication of the size for the node. The linear quadtree representation is simply a sorted list of records consisting of the locational code and the attribute values for each leaf.

The implementation of linear quadtrees on disk is fairly straightforward. The linear quadtree is simply a sorted list of leaves, with the locational code serving as the sort key. This list can be managed easily by a B^+-tree, as described in [Abe84].

Nodes in pointer-based quadtrees have historically been implemented as two distinct types. An internal node stores four pointers, one to each of its children. A leaf node stores the values of its attributes. To identify each node as one of these two types, a tag bit is needed. This can be done in the node itself, but can also be done by reducing the size of each pointer by one bit and using that bit to identify the type of node the pointer points to.

This traditional representation has a subtle space inefficiency. In the parent of each leaf node, a pointer locates the node, while the node stores only the values of its attributes. The *leafless quadtree* (for example, see [DT81, OHS90]) overcomes this problem. A leafless quadtree stores only internal nodes, each of which contains four *child fields*. For each internal node, the corresponding child field in its parent holds a pointer to that node. For each leaf, the corresponding child field in its parent holds the values of the leaf's attributes. Thus, each child

field in an internal node contains either a *pointer* or a *leaf*. Not only does this representation save space, but the lack of separate leaf nodes reduces the number of nodes in a leafless quadtree by a factor of four. This allows pointers in a leafless quadtree to be two bits smaller than those of traditional pointer-based quadtrees. In a system with fixed-size pointers, this means images with more nodes can be stored using leafless quadtrees than with traditional representations.

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 locate 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. While parent pointers are not strictly necessary, the implementation described below supports them despite the space cost. Leafless quadtrees, even with parent pointers, still require less space than linear quadtrees.

2. A Comparison of Quadtree Representations

Because the linear quadtree avoids the overhead involved in representing pointers and internal nodes, it has generally (but incorrectly, as demonstrated in [SW89]) been thought to require less storage space than the pointer-based quadtree. Linear quadtrees are also easy to represent on disk because they reduce the tree structure to a simple sorted list. Several prototype geographic information systems (e.g., [SPMA87, SSN90]) are based on the linear quadtree. QUILT [SSN90] is used later for comparison with our pointer-based implementation.

The pointer-based quadtree representation offers several significant advantages over the linear quadtree. For example, the data stored in a pointer-based quadtree can be accessed in any traversal order by simply following pointers. Linear quadtrees are arranged to allow quick preorder traversals, i.e., visits in a predetermined order such as NW, NE, SW, SE, but require logarithmic-time searches to find the next node for any other order. Back-to-front display of data represented by an octree (the three dimensional analog of the quadtree) is an example in which traversal order is dependent on the viewing position. Such methods can be used to display 3D views of topographic data. Quadtrees are also useful when implementing *progressive refinement* [BFGS86], in which a rough approximation of an image is built quickly and then the image is later refined to its final form. One way of supporting progressive refinement in quadtrees is to store attribute values in internal nodes and traverse the tree only to a certain depth. These techniques are crucial to quadtree-based hierarchical data generalization algorithms. 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 a new field to existing internal nodes, introducing far less overhead than in linear quadtrees.

The DF-expression [KE80] is an alternative quadtree representation that is extremely space efficient. The DF-expression is simply a preorder listing of all nodes of the quadtree (leaf and internal nodes) without pointers. The DF-expression requires less overhead than either linear or pointer-based representations, but does not allow random access to nodes. The S^+-tree [JSS91] is a proposed paging scheme for DF-expressions. The S^+-tree appears promising and may be more efficient than the paged implementation for pointer-based quadtrees described here. However, to our knowledge S^+-trees have not been implemented, and there are several reasons to suspect that they may not be efficient. The S^+-tree breaks the DF-expression into pages such that each page contains a complete, self-contained tree structure by adding "dummy" nodes to each page. These require additional overhead, and additional empty space within a page results from the restriction that the nodes on a page form an acceptable tree structure. The S^+-tree may also have poor performance under dynamic updates to the tree if they require major adjustments on overflow and underflow of pages. Finally, like the linear quadtree, S^+-trees are biased in favor of one predetermined traversal order. We hope to see empirical work on S^+-trees in the future that addresses these issues. S^+-trees will not be considered further in this paper.

3. Analysis of Space Requirements

In this section we briefly compare space requirements of FD linear quadtrees and leafless pointer-based quadtrees. This is a special case of the analysis of space requirements given in [SW89], where Samet and Webber have shown that contrary to conventional opinion, the pointer-based quadtree often requires less space than the linear quadtree.

For any particular image, the subdivision process is the same for either quadtree representation. Let L and I denote the number of leaves and internal nodes, respectively, in the quadtree. In 2 dimensions, L is slightly less than $3I$. For both representations, each leaf has a set of attribute values to represent, requiring a total of Lb bits to store attribute values. Attribute compaction schemes apply equally to both representations, so are not considered for this analysis. Additional space used to index the leaves is referred to as *overhead*. Let h represent the *height* of the image; if there are 2^h pixels in each dimension, the resulting quadtree has a height of at most h.

Each leaf in an FD linear quadtree consists of two components: its attribute values and its FD locational code. The number of bits required for an FD locational code for a 2D image with h levels is $2h + \lceil \log(h+1) \rceil$ (two bits for each level of resolution plus representation for the height). Since there are L

leaves, the total number of bits to store an entire FD linear quadtree is simply:

$$S_{lq} = L\,(b + 2h + \lceil \log(h+1) \rceil)$$

Leafless quadtrees 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 implementation discussed in the next section satisfies these assumptions, although we do require that each node begin on a byte boundary.

A pointer must be large enough to distinguish among the internal nodes of the quadtree. A single tag bit is also required to indicate that the field holds a pointer rather than a data value. 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 $I\lceil 1 + \log I \rceil$. 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:

$$S_p = I\lceil 1 + \log I \rceil + L(1 + b).$$

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 simplifies implementation. In a leafless quadtree with parent pointers, the number of bits required by the entire quadtree is:

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

By suitable manipulation of these three equations, it is easy to show that $S_p < S_{pp} < S_{lq}$.

In practice (e.g., both the University of Maryland's QUILT system [SSN90] and our implementation), the size of both pointers and locational codes is fixed prior to use. Because it simplifies programming when node structures fall on word boundaries, both pointers and locational codes are typically 32 bits long. With fixed size fields, a comparison of space requirements is simpler and more faithfully reflects implementation results. With pointer size fixed to 32 bits, and since there is one pointer to each internal node, a leafless pointer-based quadtree requires $L\,(1 + b + 32/3)$ bits, while a linear quadtree requires $L(32+b)$ bits. Including parent pointers raises the cost for the pointer-based quadtree to $L\,(1 + b + 64/3)$ bits. 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 1/3 that of a linear quadtree (2/3 with parent pointers). For 3D data, the leafless octree requires 1/7 the overhead of a linear octree (2/7 with parent pointers).

The primary 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 a resolution of $2^{14} \times 2^{14}$ in two dimensions, and $2^9 \times 2^9 \times 2^9$ in three dimensions. For higher resolution, larger locational codes are necessary. On the other hand, leafless pointer-based quadtrees with 32-bit pointers can represent a resolution of $2^{15} \times 2^{15}$ in two dimensions and $2^{11} \times 2^{11} \times 2^{11}$ in three dimensions. Moreover, substantially higher resolution databases can often be represented as pointer-based quadtrees since the number of nodes required is significantly smaller than in a worst-case image (i.e., where every leaf is one pixel). This advantage can be important, particularly for data sets with large areas of low-resolution data but with pockets of high-resolution data. For such data, linear quadtrees require locational codes large enough to encode leaves at the finest resolution while pointer-based quadtrees only need pointers large enough to distinguish between the actual number of nodes found in the tree [SW89]. Therefore, the restriction of image resolution due to fixed field sizes is far more serious for linear quadtrees than pointer-based quadtrees. When field sizes are fixed, the leafless quadtree is both more compact and can represent substantially larger images than the linear quadtree.

4. The Paging Scheme

We now describe how the paged-pointer quadtree maps quadtree nodes to disk pages. Nodes are shifted between pages as necessary following B-tree split and merge rules. For additional implementation details, as well as extensive comparisons of our page swapping algorithms versus the operating system's virtual memory handler, see [Brown92].

The paged-pointer quadtree is represented on disk as a file broken into a collection of pages. The first page of this file holds global information on the image represented by the quadtree, such as image type, coordinates, scale, orientation, and storage space required for attribute values. Subsequent pages contain a collection of nodes. A pointer in a paged-pointer quadtree has two parts: *page number* and *offset*. The page number identifies the page of the quadtree file containing the node pointed to; the offset identifies the location of the node within that page.

Our implementation uses a buffer pool that caches several pages from the disk-based paged-pointer quadtree, replacing pages as chosen by an algorithm (described below) whose behavior approximates least recently used page replacement.

Child fields may naturally be implemented in one of two ways. The simplest is to make the size of the field be independent of its contents. The advantage is that when child fields are of constant size, both nodes within a page and fields within a node can be found quickly. However, when attribute values and pointers require different amounts of space, fixed-size child fields require more space than necessary. Important examples include binary images (where the data value is

much smaller than a pointer value) and lists of line segments or objects (where the data value is much larger than a pointer value). Variable-size fields eliminate such wasted space by representing pointers and leaves in the minimum amount of space needed for each node. The primary drawback of variable-sized nodes is that individual nodes within a page and individual fields within a node cannot be accessed directly.

Our implementation of the paged-pointer quadtree combines these two approaches, using fixed-size child fields in main memory and variable-size fields on disk. Using variable-size fields on disk allows a greater number of nodes to be placed on each page on disk and therefore decreases the number of disk pages required to store a quadtree (in turn reducing the number of page faults). The internal buffers holding pages using fixed-size fields allow direct computation of the location of a node within its page.

This combined approach makes it necessary to convert page representations when I/O is performed. When the disk is accessed, the cost of I/O far exceeds the cost of the computation needed to convert between representations. Our 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. For example, 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 used in the remainder of the field.

This combined approach requires buffers in memory larger than the disk page size. Because of the compression, each node will generally require less space on disk than in memory. The amount of free space on a page is defined in terms of the amount of available space when the page resides on disk. Each page stores a count of the amount of available space on the page, and updates this count as the nodes of the page change. An insertion or deletion that 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 between pages using traditional B*-tree rules.

One of the most serious limitations of traditional pointer-based quadtrees is that there is no natural mechanism for keeping neighboring nodes nearby in memory under dynamic updates of the tree. Virtual memory systems exploit the fact that memory accesses patterns (e.g., the fetch-execute cycle for instructions) are often sequential in nature so many consecutive accesses all involve a small number of pages. Access patterns in traditional pointer-based quadtrees need not have this sequential nature, even when processing is done in preorder, since trees that have been modified are not likely to remain in preorder within memory. When a pointer-based quadtree is stored on disk or in virtual memory, the resulting lack of locality of node references can cause an unacceptable number of page faults.

Like leaves in linear quadtrees, the nodes in our paged-pointer quadtree

are arranged according to a preorder traversal. Storing nodes in this order has several advantages. First, this order is not too difficult to maintain using B-tree techniques. Second, quadtree processing is frequently done in preorder. (Note that the pointer-based quadtree pointer structure still allows direct access to nodes in any traversal order.) Third, the nodes of subtrees deep in the quadtree are grouped together and likely to be on the same page. Thus, random access to specified nodes will require fewer page faults than the depth of the tree, and neighboring nodes are likely to be on the same page. This is significant when performing neighbor-finding operations [Sam90a]. Other approaches to node ordering may lead to different and perhaps more efficient clusterings. However, optimal clusterings methods often require substantial computation whenever the structure changes.

A traditional B-tree stores records in sorted order on a page so that a binary search can be used within the page. When a new record is added to a page, other records on the page are shifted to make room for the new record (this accounts for a significant fraction of total processing time in the QUILT system). 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.

There is an important difference between the index structures of the paged-pointer quadtree and the B^+-tree storing the linear quadtree. The internal nodes of a B^+- or B^*-tree serve as an index for a sorted list stored at the leaf nodes of the B-tree. In the paged-pointer quadtree, this index for the sorted list is replaced by the pointer-based quadtree's own pointer structure. The concept of internal and leaf pages is lost. In this sense, the paged-pointer quadtree is more like a B-tree than a B^+-tree. Most significantly, there is no need to perform binary search on nodes within a page since the pointer structure of the quadtree explicitly stores the locations of nodes, allowing the location of a child or parent to be found directly. As a consequence,whenever a node is moved (for example, when rebalancing pages), all pointers to that node must also be updated. While the close relatives of many of these nodes are on the same page, updating off-page relatives introduces the possibility of page faults.

As a result, while pages of the paged-pointer quadtree are organized in preorder, the nodes within a page need not and should not be. This reduces the amount of updating required when inserting or deleting nodes. See Figure 1 for an example of a paged-pointer quadtree broken into pages. Each page contains a contiguous section from a preorder traversal of the tree, but the nodes within the page may not be in preorder. For details on the page updating algorithms that reorder and reallocate the nodes onto pages when overflow or underflow occurs, see [Brown92].

The page replacement algorithm in a virtual memory system (whether implemented in hardware or software) significantly affects the performance of the system. Performance suffers if pages are accessed in a manner not matching the page replacement strategy. General-purpose virtual memory systems typically seek to replace the least recently used (LRU) page and are often very effective. Unfortunately, general-purpose LRU algorithms cannot determine which

nodes will be revisited during a tree traversal and which will not be used again. However, this information can be maintained by a traversal algorithm. When a traversal visits a node N and continues with N's descendants, N will certainly be revisited later. Our buffer pool implementation provides a locking feature not supported by traditional virtual memory systems, so as to exploit the special characteristics of a tree traversal. Since a traversal returns to each ancestor of the current node in the traversal, the pages containing these nodes are locked into main memory when first visited. Locking indicates to our page replacement algorithm that a 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 should be large enough to hold a number of pages other than those that are locked; otherwise, the presence of locked pages severely decreases the effective size of the buffer pool. In practice, twice the depth of the tree should be sufficient.

Although our implementation is slightly more complicated than that of the B-tree used to organize linear quadtrees, all the necessary code can be provided in a library defining a quadtree abstract data type. The application programmer need not be concerned with the intricacies of implementation or the fact that these quadtrees are actually stored on disk. In all respects except one, the application code can be written as though a traditional pointer-based quadtree were being used. The one exception relates to explicit pointers to quadtree nodes. It is possible that the position of a node may be changed by the quadtree page management algorithms, invalidating the previous pointer values. Library functions that change the shape of the tree return a pointer to the modified node. Library users must be careful not to rely on the old values of pointers after the tree structure is modified. In practice, this has not been an issue in the implementations of three standard quadtree algorithms described in the next section.

5. Time and Space Performance

The paged-pointer quadtree presented in Section 4 provides a compact disk-based representation for quadtrees. We now compare 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 [SSN90]. Three representative application functions are used in our tests: raster to quadtree conversion (a good test of dynamic quadtree updating), quadtree to raster conversion (a good test of random access to a static quadtree) and unaligned intersections (a good test of both search and update in quadtrees).

Our paged-pointer quadtree implementation is written in the programming language C, as was QUILT. Both systems have been implemented on several computers running the Unix operating system, including a DECStation 3100 and a 25 Mhz Commodore Amiga 3000UX.

Our first test algorithm converts a raster image to a region quadtree. 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.

Shaffer and Samet [SS87] present algorithms for building linear and pointer-based quadtrees with node accesses proportional to the number of nodes in the final quadtree. The linear quadtree algorithm uses a small buffer holding leaves in which one but not all of its pixels have been visited in the original image. The pointer-based algorithm begins with an "uncolored" quadtree and modifies it as pixels are read. It traverses the tree by ***neighbor-finding*** [Sam90a], an average-case constant-time operation, to locate the node adjacent to the current one.

QUILT implements the linear quadtree building algorithm of [SS87]. Our paged-pointer quadtree building algorithm is the pointer-based building algorithm based on neighbor-finding given in [SS87], slightly modified to process odd rows from left to right and even rows from right to left. This significantly decreases the number of times that high-level subdivisions of the quadtree being built are crossed during neighbor finding.

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. They consist of 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. They contain 5,206, 28,549 and 25,012 leaves, respectively. 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 by their quadtrees and contain 19,426, 112,819 and 98,491 leaves, respectively.

The build algorithms were tested on the three 400×450 pixel data sets on the Amiga, and the three 800×900 data sets on the DECStation. The resulting pointer-based quadtrees are 38-42% 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. QUILT, whose B-tree splits three pages for four [SSN90], leaves approximately 20% of each page free. Our paged-pointer quadtrees, which splits two pages for three, leave about 25% free. Both implementations provide a compaction operation that eliminates this free space. Our packed pointer-based quadtrees are 44-46% smaller than QUILT's linear quadtrees. Using analysis similar to that of Section 3, 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 used 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, the paged-pointer implementation is also storing 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.

Table 1. Quadtree build times (in seconds).

Image Name	Pointer-Based		QUILT	
	DEC	Amiga	DEC	Amiga
Floodplain	5.6	5.4	7.3	4.5
Landuse	54.9	21.0	104.9	48.8
Topography	47.4	19.4	85.0	34.0
Total (real)	121.5	45.8	222.2	87.3

The average times required to build the quadtrees on our two machines are given in Table 1. On each system, builds are nearly twice a fast using our pointer-based representation.

To display the spatial data represented by a region quadtree, the quadtree is often converted to raster form. This operation has been implemented both by QUILT and our pointer-based system, and performance was measured by reconverting the six test quadtrees. The linear quadtree algorithm implemented by QUILT maintains an array holding those leaf nodes currently part but not completely processed. When a new pixel is processed, the linear quadtree is searched only if the pixel is not contained by a leaf in this array. 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 quadtree building algorithm were found to be significantly slower than the linear quadtree algorithm. Our new 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 and using neighbor-finding to update the heap when a new row is processed. As Table 2 indicates, this algorithm using the paged-pointer quadtree is significantly faster than that of the linear quadtree.

The region quadtree efficiently performs set operations (e.g., area intersection, union) on spatial data sets, since large homogeneous parts of the two images are compared all at once. The algorithms used to perform set operations on quadtrees depends on whether the images represented by the quadtree are *aligned* (same origin and orientation) or *unaligned*.

Set operations on aligned quadtrees [HS79] simply requires traversing the input quadtrees in parallel and performing the proper set operation on pairs of leaves. Because the input quadtrees are traversed in preorder, aligned set operations execute efficiently using both linear quadtrees and paged-pointer quadtrees.

Table 2. Time required for quadtree-to-raster conversion.

Image Name	Pointer-Based		QUILT	
	DEC	Amiga	DEC	Amiga
Floodplain	4.1	5.9	4.5	6.1
Landuse	12.6	7.0	22.2	10.6
Topography	11.8	6.3	20.6	9.7
Total	28.5	19.2	47.3	26.4

Set operations on unaligned quadtrees are considerably more complex. An efficient algorithm for set operations on unaligned linear quadtrees is presented in [SS90].

Table 3. Time required (in seconds) for unaligned set operations - Amiga.

Image & Offset	Pointer-Based			QUILT		
	0	1	100	0	1	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. Time required (in seconds) for unaligned set operations - DEC.

Image & Offset	Pointer-Based			QUILT		
	0	1	100	0	1	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

Set operations on unaligned quadtrees require searches in the unaligned quadtree. The need for searching makes unaligned set operations a good test of random access to the pages of the quadtree. Tables 3 and 4 compare the times required to perform unaligned set operations using the nine test cases presented in [SS90]. In each test, a portion of "floodplain" 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 outperforms QUILT on the Amiga, while execution times are nearly identical on the DECStation. Both systems implement the unaligned set

operation of [SS90], which was designed for linear quadtrees. It may be possible to implement a more efficient pointer-based algorithm taking advantage of the internal structure of the quadtree.

To help understand the why the paged-pointer implementation is faster than QUILT, we studied the number of page faults generated by both systems and also profiled various critical sections of the code. On each machine, the linear quadtree system spends substantially more time performing I/O, while our pointer-based quadtrees incur greater computational overhead involved in translating each pointer into an address in the buffer pool. Profiling our code indicates that our address translation code consumes between 20% and 30% of total execution time. At the same time, the paged-pointer implementation generates between 1/2 and 3/4 the number of page faults generated by QUILT (with significant variation between images). However, there is only modest correlation between time improvement and page fault improvement. For more timing details and information on page faults, see [Brown92].

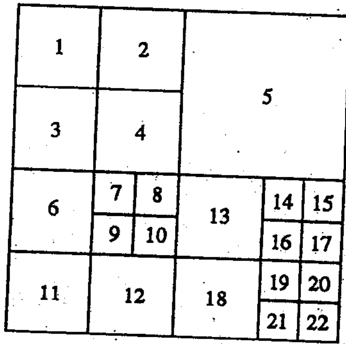
6. Conclusions

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 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. We have described 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.

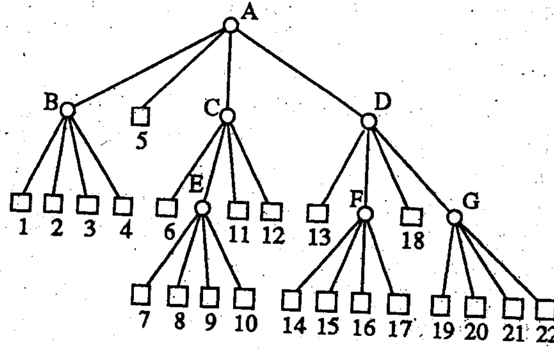
Because of its perceived efficiency, the linear quadtree has historically been used to represent large quadtrees on disk, despite the many other advantages of pointer-based quadtrees. The primary motivation for our work is to produce a disk-based representation that exploits the many advantages of pointer-based quadtrees while providing adequate performance. The tests reported in Section 6 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.

7. Acknowledgements

During the course of this work, we have benefited from discussions with Robert E. Webber. We also wish to thank the reviewers for their comments, which helped us to improve our presentation.



(a) Block Structure



(b) Tree Structure

Page 1	Parent	NW Child	NE Child	SW Child	SE Child
A	➢ NULL	➢ B	5	➢ C	➢ ➢ E
C	➢ A	6	➢ ➢ D	11	12
B	➢ A	1	2	3	4

Page 2	Parent	NW Child	NE Child	SW Child	SE Child
E	➢ ➢ C	13	➢ ➢ F	18	➢ ➢ G
D	➢ ➢ A	7	8	9	10

Page 3	Parent	NW Child	NE Child	SW Child	SE Child
F	➢ ➢ D	14	15	16	17
G	➢ ➢ D	19	20	21	22

(c) Paged Pointer Quadtree Representation. Assumptions: Maximum number of nodes per page is 4; Minimum number of nodes per page is 2; a single arrow is a pointer to a node on this page; a double arrow is a pointer to a node on another page.

Figure 1. Three representations for an example quadtree.

8. References

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

[BFGS86] L. Bergman, H. Fuchs, E. Grant and S. Spach. Image rendering by adaptive refinement, *Computer Graphics*, 20(4):29-37, August 1986.

[Brown92] P.R. Brown. A paging scheme for pointer-based quadtrees. Masters Thesis, Virginia Tech, Blacksburg VA, May 1992.

[DT81] L.J. Doctor and J.G. Torborg. Display techniques for octree-encoded objects. *IEEE Computer Graphics & Applications*, 1(3): 29–38, July 1981.

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

[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, July 1979.

[JSS91] W. de Jonge, P. Scheuermann and A. Schijf. Encoding and manipulating pictorial data with S^+-trees, in *Advances in Spatial Databases: Proceedings of SSD'91*, Lecture Notes in Computer Science 525, O. Günther and H-J. Schek, eds., Springer Verlag, Berlin, 401-419, 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*, 2(1):27–35, January 1980.

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

[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, November 1990.

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

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

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

[SS90] C.A. Shaffer and H. Samet. Set operations for unaligned linear quadtrees. *Computer Vision, Graphics, and Image Processing*, 50(1):29-49, April 1990.

[SSN90] C.A. Shaffer, H. Samet, and R.C. Nelson. Quilt: A geographic information system based on quadtrees. *International Journal of Geographic Information Systems*, 4(2):103-131, August 1990.

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

[SPMA87] T.R. Smith, D.J. Peuquet, S. Menon and P. Agarwal. KBGIS-II: A knowledge-based geographical information system. *International Journal of Geographical Information Systems*, 1(2):149–172, April 1987.