Efficient Processing of Window Queries in The Pyramid Data Structure[*]

Walid G. Aref
Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
The University of Maryland
College Park, Maryland 20742

Abstract

Window operations serve as the basis of a number of queries that can be posed in a spatial database. Examples of these window-based queries include the exist query (i.e., determining whether or not a spatial feature exists inside a window) and the report query, (i.e., reporting the identity of all the features that exist inside a window). Algorithms are described for answering window queries in $O(n \log\log T)$ time for a window of size $n \times n$ in a feature space (e.g., an image) of size $T \times T$ (e.g., pixel elements). The significance of this result is that even though the window contains n^2 pixel elements, the worst-case time complexity of the algorithms is almost linearly proportional (and not quadratic) to the window diameter, and does not depend on other factors. The above complexity bounds are achieved via the introduction of the incomplete pyramid data structure (a variant of the pyramid data structure) as the underlying representation to store spatial features and to answer queries on them.

1 Introduction

Today, many applications benefit from having a spatial processor in conjunction with a database management system in the same environment. The function of the spatial processor is to maintain and operate on spatial features efficiently. Maintenance includes storing, adding, or deleting features. Operations include Boolean set operations (e.g., intersection, union, difference, negation, etc.), windowing, arithmetic operations (e.g., area, perimeter, centroid, etc.), and proximity operations (e.g., left, north, nearest, farthest, within a given zone, etc.). Having the spatial processor perform these functions efficiently (while utilizing powerful spatial architectures) enriches the query power of the database environment for application programs and end users.

One key operation that we consider here is the window operation. A window is the region specified by the cross-product of two closed intervals over the integers. For example,

$$w = \{(x,y) \mid (x,y) \in [x_1 : x_2] \times [y_1 : y_2] \text{ where } x,\ y,\ x_1,\ x_2,\ y_1, \text{ and } y_2 \text{ are integers}\}$$

is a window that represents all the internal points inside the horizontal interval $[x_1 : x_2]$ and the vertical interval $[y_1 : y_2]$.

Window operations serve as the building block for a number of queries and as a composite operation to many other queries. Usually spatial features span over a wide feature space. However, users are more interested in viewing or querying only portions of the feature space instead of the whole space. Extracting parts of the space to work with in the next operations is done through windowing. Given a window w, some examples of window-based queries are: report all features inside w, intersect feature f with feature b only inside w, does feature f exist in w, etc.

In this paper, we concentrate on the efficient implementation of window operations. We describe a bottom-up algorithm for handling these operations. The main idea behind this algorithm is to decompose the operation over the whole window into sub-operations over smaller window partitions. Within this algorithm, spatial features are organized in a pyramid-like data struc-

[*]The support of the National Science Foundation under Grant IRI-88-02457 is gratefully acknowledged.

ture. We demonstrate that our algorithm is better than several other approaches. Using this algorithm as a building block, we show how to perform window-based queries efficiently.

The paper proceeds as follows. In Section 2, we review the pyramid data structure and its main characteristics. In Section 3, we present all the techniques we apply in order to perform the window operation efficiently. Section 3.1 demonstrates how spatial features are encoded into the pyramid data structure. In Section 3.2, we show how we can access pyramid nodes directly. This enables us to extract the features stored in a given pyramid node in constant time. The feature retrieval algorithm is given in the same section. Section 3.3 contains a new mechanism for decomposing windows. In Section 4, we describe several window-based queries and algorithms to answer them efficiently utilizing the window operation and the window decomposition mechanism. Section 5 analyzes the worst-case complexity of these algorithms in terms of the complexity of the decomposition process, along with a brief comparison with related work. Section 6 contains concluding remarks.

2 The Complete Pyramid

The pyramid [10] is a multiresolution data structure. Given a $2^d \times 2^d$ image array, say $A(d)$, a pyramid is a sequence (or stack) of arrays $A(i)$ such that $A(i-1)$ is a version of $A(i)$ at half the scale of $A(i)$. $A(0)$ is a single cell. It is called the root of the pyramid. We say that array $A(i)$ lies at *level i* in the pyramid, and that the entire feature database is stored at the bottom level. For a $T \times T$ image, where $T = 2^d$, the pyramid has resolution d which is the maximum depth or the number of levels in the pyramid.

In this context, we view the pyramid as the space reserved for a complete recursive decomposition of the image space into quadrants down to the pixel level. This representation has an overhead of one third extra storage because of the need to store intermediate nodes. In other words, for a $T \times T$ image where $T = 2^d$, the entire pyramid requires $4 \times (T \times T - 1)/3$ nodes. This is the same amount of storage necessary for a complete quadtree [7] where every node of the quadtree has been expanded down to the pixel level. This structure is called a *complete pyramid* in contrast to the incomplete pyramid that we define later.

In this paper, we do not use the term pyramid in its classic sense as a multiresolution stack of arrays. Instead, we make use of a variant of the pyramid that enables us to represent a collection of different features of interest in the underlying image. The same approach was adopted by Shaffer and Samet [9]. Note that features can overlap, e.g., rivers, bridges, roads, and counties.

Although the pyramid is implemented by an array structure as is shown below, it is more convenient to define a pyramid in the form of a tree. Again, assuming a $2^d \times 2^d$ image, a recursive decomposition into quadrants is performed just as in quadtree construction, except that we keep subdividing until we reach the individual pixels. The leaf nodes of the resulting tree represent the pixels while the nodes immediately above the leaf nodes correspond to the array $A(d-1)$, which is of size $2^{d-1} \times 2^{d-1}$. The non-leaf nodes are assigned a value that is a function of the nodes immediately below them (i.e., their four sons) such as the average gray level. For our purposes, this function is the Boolean OR set operation. In other words, we say that a certain feature is present in the parent node when this feature is available in any of its four sons.

The pyramid is implemented by a linear array in a heap-like fashion [12]. Each pyramid node is represented by an array entry. Given a parent node p with relative index j at level i, we can access the kth son of p (where $k = 0, 1, 2, \text{or } 3$) by the following formula:

$$son(i, j, k) = 4^i + 4 \times j + k$$

The number of array elements at levels 0 (the root level) through $i-1$ is $(4^i - 1)/3$.

Also, there is a correspondence between the pyramid nodes and the spatial database blocks. Assume a coordinate system such the origin is at the upper-left corner and x increases to the right and y increases downwards. A block of size $2^i \times 2^i$ $(0 \leq i \leq d)$, having (r, c) as the coordinates of its upper-left corner $(0 \leq r, c \leq T - 1)$ such that $r \bmod 2^i = c \bmod 2^i = 0$ corresponds to the pyramid node at level $d - i$ whose relative index within the array $A(d-i)$ (also at level $d - i$) is equal to $(bit\_interleave(r, c)) \gg (2 \times i)$, where $\gg$ is the shift-right operation. Thus $y \gg x$ means that y is shifted to the right by x bit positions.

Pyramid nodes have the following internal structure: each node has a fixed number m of feature bit slots, and summarizing variables that summarize information about the descendents of the node. For now, we are interested in the feature bits. Each bit slot corresponds to a feature stored in the pyramid.

3 Techniques

In this section, we define the necessary data structures and techniques to perform the window operation efficiently. First, we describe how spatial features are stored in the pyramid. Then, we show how we can directly access any pyramid node, whether external or internal, in constant time. Next, we demonstrate an efficient method to decompose the window into smaller units, each of which serves as the basis of a separate query to the pyramid. The discussion of how these structures and techniques are combined to compose answers to window-based queries is deferred to the next section.

3.1 Feature Encoding

Each feature is encoded into its bit slot through the whole pyramid independent of the features in the other bit slots. Below we describe how one feature is encoded using this coding technique. The same technique is applied to the remaining features. In this discussion, we use the term *node* to denote one bit slot. Features are stored in the pyramid using the GL method devised by Shaffer and Samet [9]. It is a quadtree-like encoding of regions for the pyramid data structure.

Nodes in a region quadtree are either leaf or gray. Leaf nodes correspond to those blocks that are either entirely inside the feature region in the image (labeled *black*) or are entirely outside (labeled *white*). Each nonleaf node is said to be *gray* (i.e., its corresponding block is neither entirely inside nor entirely outside the feature region). The parent of a leaf node is a gray node (i.e., leaf nodes are maximal).

The GL method works as follows. If an intermediate node is gray, then its bit slot value is 1. If an intermediate node is a leaf node, then its bit slot value is 0. Since it is necessary to determine the actual value of a leaf node (i.e., black or white), this value is stored in all of the descendents of the leaf node down to the pixel level (i.e., 1 and 0, respectively). If a bottom-level node is a leaf node, then its value is stored in the node itself since it has no descendents. Clearly, a gray node cannot appear in the bottom level.

3.2 Pyramidal Direct Access

The physical location of any node in the pyramid can be directly accessed in constant time [11]. The pyramid is implemented as an array with the root at location 0, nodes at level 1 in locations 1 – 4, followed by the nodes at level 2 through d (the maximum level). Given x and y, the values of the coordinates of the upper-left corner, and the size $2^i \times 2^i$, of a block in the image, we can compute the pyramid address of the 'corresponding' node, say Q, as follows:

$$\begin{aligned} pyr\_address(i,x,y) &= (4^i - 1)/3 + \\ &\quad (bit\_interleave(x,y) \gg 2i) \end{aligned}$$

Procedure *features*, given below, shows how features stored in Q can be retrieved from the pyramid structure in constant time. Q can be at an intermediate or bottom level in the pyramid.

```
set_of_features procedure features(i, x, y);
begin
  value integer i, x, y;
  set_of_features s;
  global integer d;
  if(i = d) then /* bottom level*/
    s ← contents(pyr_address(d, x, y))
  else/* features in the node or in one of its sons */
    s ← contents(pyr_address(i, x, y))
        ∪ contents(pyr_address(i+1, x, y));
  return (b);
end;
```

3.3 Decomposing Windows

A window is stored implicitly in a region quadtree [5, 7]. The nodes inside the window are colored black while those outside are colored white. Note that we do not need to store the window explicitly in a quadtree structure since we can generate all the maximal quadtree blocks inside the window without building the quadtree. The generation process only requires that the window be specified. It works for an arbitrary rectangular window (i.e., it need not be square). We refer to it as *window_gen*.

Achieving a low execution time bound for *window-gen* is non-trivial. In particular, we want to visit each maximal block only once. Otherwise, the execution time could be as bad as $O(n^2)$ for an $n \times n$ window. If this were not the case, then, for example, block A in Figures 1a and 1b would be visited once for each row and column, respectively, in the image.

The idea behind the window block-generator procedure is to scan the window row-by-row (in the block domain rather than in the pixel domain). During this process maximal blocks are generated and the columns of the row that lie within the block are skipped. This is illustrated in Figure 2, where 6 scans are needed to cover the 6×6 window with maximal blocks. The blocks are labeled with numbers that correspond to the order in which they have been visited. The first scan visits blocks 1, 2 and 3; the second scan visits blocks 4, 5, 6, and 7; the remaining scans visit blocks 8 and 9; 10

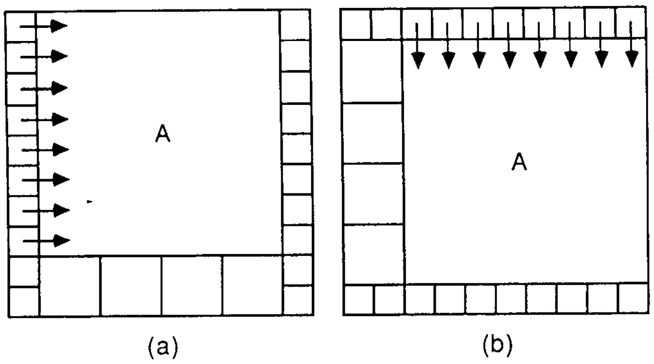

Figure 1: In (a) and (b), maximal block A is accessed more than once by the smaller blocks to its left and top, respectively.

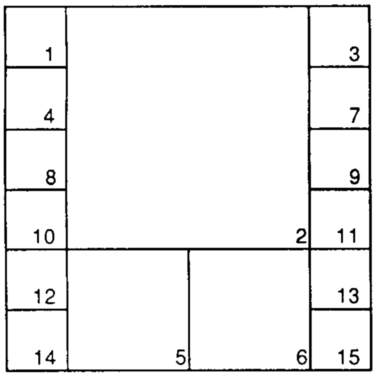

Figure 2: The decomposition of a 6 × 6 window into maximal blocks.

and 11; 12 and 13; and 14 and 15. Notice that once blocks 5 and 6 have been visited their columns (i.e., 2-5) have been completely processed. For each block that is generated, say B, *window_gen* also generates B's southern neighboring blocks for processing on the next scan. These blocks are stored in a linked list.

Observe that we are always generating maximal neighboring blocks, or at least a bounded number of non-maximal neighboring blocks. An example of this situation is illustrated by the 10 × 10 window in Figure 3 when processing blocks A-J in the first row. Each of blocks B, C, D, F, G, H, and J can generate at most one non-maximal neighboring block. Even though these non-maximal blocks are generated, they are skipped by the next scan since they are subsumed (i.e., contained) in the previously processed maximal block in the scan. For example, when scanning block K, blocks L, M, and N are skipped since they are contained in it. This is easy to detect because for each block we know the coordinate values of its upper-left corner and its size.

Generating a maximal block is straight-forward. Given a window, say w, and the values of the row and column coordinates of a point, say (row, col), inside w,

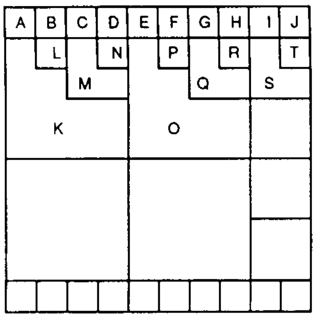

Figure 3: The neighboring blocks to the south of blocks A-J in a 10 × 10 window. Blocks L, M, N, P, Q, R, and T are non-maximal while blocks K, O, and S are maximal.

we want the largest block, say B, inside w for which (row, col) is the upper-leftmost pixel. B is of size $S = 2^s$ where s is the maximum value of i $(0 \leq i \leq \log T)$ such that $row \bmod 2^i = col \bmod 2^i = 0$ and the point $(row + 2^i, col + 2^i)$ lies inside w.

Generating the southern neighbors of a block, say B, is done as follows. Check whether or not the southern boundary of B (denoted by $sb(B)$) is tangent to the southern boundary of window w. Notice that $sb(B)$ is either tangent to $sb(w)$ or is inside w. It cannot be outside w since by definition B is a maximal block inside w. If $sb(B)$ is tangent to $sb(w)$, then no more blocks to the south of B need to be generated (e.g., after processing blocks 5 and 6 in the second scan of the window in Figure 2). If $sb(B)$ is inside w, then maximal blocks to the south of B are generated. Notice that all the southern neighbors that are generated are blocks that lie entirely inside the window (e.g., blocks 5 and 6 after processing block 2 in the first scan of the window in Figure 2).

The algorithm terminates whenever during a scan no new southern blocks are added to the list. At this time all the maximal blocks inside the window have been generated.

4 Window-based Queries

We consider the following window-based queries:

- The Exist Query: Determine whether or not a certain feature f exists inside (i.e., overlaps) window w.
- The Report Query: Report the identity of all the features that exist inside (i.e., overlaps) window w.
- The Select Query: Determine the locations of all occurrences of feature f inside window w.

In the rest of this section we show how to answer these queries using the window decomposition algorithm presented in Section 3.3. Our approach to answering these queries can be classified as bottom-up, in contrast to top-down algorithms that start at the pyramid's root and proceed downwards.

4.1 The Exist Query

The main idea behind the algorithm to answer this query is to decompose the query over a window into a sequence of smaller queries. In particular, we use procedure *window_gen* of Section 3.3 to decompose the window into the maximal quadtree blocks inside it. Each of these square blocks will serve as a query unit. Note that coordinate values of the upper-left corner and size of every block generated are known (from the generation procedure itself). Section 3.2 also shows how to extract features from a pyramid node given the specification of the node's corresponding block. Combining these two techniques yields the following algorithm. Generate the maximal quadtree blocks inside the queried window. For each one of these blocks, query the pyramid to retrieve the features inside this block (i.e., hence inside the window). If the feature we are looking for exists in any of these blocks, then the algorithm halts and outputs YES. If the algorithm does not find the feature in any internal block after generating all of them and querying the pyramid for each one, then the algorithm halts and outputs NO.

4.2 The Report Query

The report query algorithm is implemented in an identical manner as the exist query algorithm, except that features are extracted from each block and accumulated in a set, say S. After all blocks are examined, the entire set S is returned as the set of features that overlap the window.

4.3 The Select Query

The output of the select query is a list of all blocks inside the query window that feature f covers entirely. This list may be thought of as a linear (pointer-less) quadtree [6] with the white nodes being omitted.

One crucial problem with the GL coding technique described earlier is that if a pyramid node is accessed directly and found to have a value of 1 for a particular bit slot, then there is no easy way (i.e., $O(1)$) of determining whether this node is gray (non-leaf) or black (leaf). The problem is that if its four sons are coded with a 1,

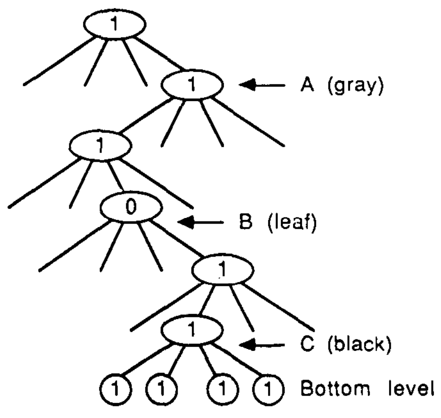

Figure 4: Example of GL coding.

then the node can be both gray or black, and we have to check its grandsons again. This process may possibly not terminate until we reach the nodes at the bottom level of the tree. As an example, consider Figure 4. Node A is a gray node while node C is in the subtree of leaf node B. Directly accessing node A, we cannot say whether it is gray or black (in constant time). The same applies for node C. The bottom-up select algorithm that we use (described below) depends on the efficiency of the process of making such a distinction between gray and black nodes. Thus the GL coding technique for the pyramid has to be modified slightly in order to support efficient select window queries. This leads us to use an incomplete pyramid as defined below. These modifications do not adversely affect the complexity of the other window queries.

4.3.1 The Incomplete Pyramid

The difference between the normal GL coding scheme and the modified scheme is that in the modified scheme when a leaf node (a 0 value) appears at an intermediate level, then all its descendents have a 0 value except for those nodes at the bottom level which are assigned the values 1 or 0 depending on whether they are black or white, respectively. If a leaf node appears at the bottom level then it is assigned the value 1 or 0 depending on whether it is black or white, respectively. Notice that using this coding scheme, we are able to overcome the deficiency that arises with normal GL coding. In particular, when nodes are accessed directly, we can distinguish between gray and leaf nodes. As an example, consider Figure 5. When node A is accessed directly, we know it is a gray node since its value is 1, while node B is a leaf node since its value is 0. To determine if

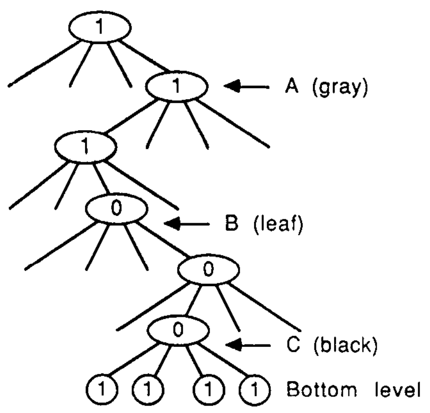

Figure 5: Example of modified GL coding.

node B is black or white, we access node C directly at the bottom level of the pyramid which tells us that B's value is black (since C has a value of 1).

Using the modified GL coding, procedure *features* is modified slightly as described below. In particular, when a leaf node is encountered at an intermediate level (i.e., a value 0 was encountered), the corresponding features stored in this leaf node are retrieved from the bottom level of the pyramid instead of from the node's sons as in the normal GL coding scheme. Procedure *features* still runs in constant time.

```
set_of_features procedure features(i, x, y);
begin
  value integer i, x, y;
  set_of_features s;
  global integer d;
  if(i = d) then /* bottom level*/
    s ← contents(pyr_address(d, x, y))
  else   /* features in the node or in one of its sons */
    s ← contents(pyr_address(i, x, y))
        ∪ contents(pyr_address(d, x, y));
  return (s);
end;
```

4.3.2 Window Queries

The procedures for answering the exist and report queries are unchanged for the modified GL coding. Of course, they make use of the modified features procedure routine described above. The select query algorithm proceeds in the following manner. First, generate the maximal black blocks that comprise the window. Next, for each block, test the corresponding pyramid node. If it is white, then do nothing. If it is black, then report this block in the output. If it is gray, then traverse the pyramid subtree corresponding to this node and report in the output all the black blocks descending from it.

5 Complexity Analysis

In this section we first derive the worst-case execution time complexity of the window decomposition algorithm. Next, we use this result to compute the worst-case execution time for the window-based query answering algorithms. Our analysis assumes that the window is square.

5.1 Analysis of the Window Decomposition Algorithm

Dyer [3] and Shaffer [8] prove that in the worst-case, the number of maximal quadtree blocks inside a square window of size $n \times n$ is $N = 3(2n - \log n) - 5$. What remains to be done is to compute the cost of determining the maximal blocks comprising the window. This consists of the work, say T_m, to generate a maximal block, say B, and the work that is wasted, say T_w in generating southern neighboring blocks of B that are non-maximal. Therefore, the total execution time of the window decomposition algorithm is $N \cdot (T_m + T_w)$.

Given a point (x,y) in a $T \times T$ space, there can be at most $\log T + 1$ different blocks of size 2^i $(0 \leq i \leq \log T)$ with (x,y) as their upper-left corner. We use binary search through this set of blocks to determine the maximal block inside the window. Thus T_m is $O(\log\log T)$.

It can be shown [2] that each block can generate at most one non-maximal neighboring southern block (recall the discussion of Figure 3 in Section 3.3). Thus T_w is $O(\log\log T)$. Combining T_m and T_w implies that the worst-case execution time for the window decomposition algorithm is $O(n \log\log T)$.

5.2 The Exist Query

This query can be answered in $O(n \log\log T)$ time. The exist query algorithm described in Section 4.1 queries the pyramid for each maximal window block generated. It stops running when the feature exists in any of the blocks or when it exhausts all the blocks without finding the feature in any of them. As we already know, each maximal block requires at most two pyramid direct accesses to be answered, and each such access takes constant time as shown earlier. The number of maximal blocks inside the window is $O(n)$, and each takes

$O(\log\log T)$ to generate. Direct pyramid access takes $O(1)$ time. This leads to an $O(n \log\log T)$ cost for the whole procedure.

5.3 The Report Query

The analysis for the exist query also applies to the report query algorithm described in Section 4.2. The only difference is that it always runs in the same amount of time since it has to examine all the maximal blocks before reporting the feature. Thus, its cost is $O(n \log\log T)$.

5.4 The Select Query

The worst-case time complexity of the select query procedure described in Section 4.3 depends on the distribution of the feature in space. In that sense, a feature forming a checkerboard in the pyramid bottom level will have the worst time complexity since all nodes are gray down to the pixel level. Such an image does not take advantage of the compression provided by the GL coding technique. In such a case, the running time is proportional to $4n^2/3 + \log n + 5/3$.

This dependence on n^2 and $\log n$ is derived as follows. We make use of the fact that the number of maximal blocks inside a square window of width n is bounded by $(2n-1) + \sum_{i=0}^{m-1} n/2^{i-1} - 3$ where $m = \log n$ [3]. Moreover, a $2^i \times 2^i$ block in the pyramid has $(4^{i+1} - 1)/3$ descendants. The select query causes the pyramid to be accessed once for each maximal block, say B, in the window. In the worst case, the pyramid node corresponding to B is gray and all of its descendents (to the pixel level) are also gray. In such a case, our implementation will visit $4n^2/3 + \log n + 5/3$ nodes.

Notice that on the average many of the maximal blocks inside the window correspond to leaf nodes in the pyramid. Also, not all the descendents of gray nodes in the pyramid are gray. Thus from a practical point of view, we can usually get better performance in the average case (since more compaction takes place).

At this point, we sketch how to reduce the worst-case performance to $O(n \log\log T)$, regardless of the feature distribution in space, by introducing *slot-* or *feature-dependencies*. For the purpose of this discussion, suppose that spatial feature f is stored in bit slot b_1, and we want to select the part of f that lies inside window w. Instead of producing a list of all the blocks in w that contain f, we allocate a new bit slot, say b_2, in the pyramid which will be set to 1 for all maximal blocks in w that contain it. If these maximal blocks are gray, then the b_2 slots of their sons are not marked. This is a form of lazy evaluation [4] in the sense that we do not copy the subtrees of gray nodes inside w that store f. Instead, we use an auxiliary table to record that bit slot b_2 is dependent on b_1. Thus what we are saving here is the time to copy the information rooted at a gray node. Subsequently, when the result of the select query is desired, we simply retrieve the information from bit slot b_2 and if the node is a gray node, then we retrieve the rest of the information from bit slot b_1.

Implementing feature-dependencies is straightforward. First, generate the maximal blocks inside w, an $O(n \log\log T)$ process. For each such maximal block, say B, directly access B's corresponding pyramid node, say p, and copy the contents of its b_1 slot into its b_2 slot (but do not copy the information in B's descendents). This takes $O(1)$ time per block. Once all the maximal blocks have been processed, we store the feature dependency in the table. Next, we propagate the presence of feature f in the b_2 bit slot to all nodes above p up to the pyramid root. In order to avoid repeated accessing of the same nodes we use one of the feature loading techniques described in [1]. The cost of the propagation step over the entire window is $O(n)$ [1].

Thus we see that the use of feature dependencies reduces the cost of the select query to $O(n \log\log T)$ at the cost of an extra bit slot per node in the pyramid. This extra bit slot can be reused once we are done with the result of the select query.

Tables 1 and 2 summarize the execution times and space requirements for window operations using our approach (i.e., the bottom-up pyramid) and compare them with other data structures. Q_f is the number of quadtree nodes for feature f. We assume an $n \times n$ window in a $T \times T$ image.

Table 1: Window query time requirements

Data	Query		
Structure	Exist	Report	Select
Incomp. Pyr.	$O(n \log\log T)$	$O(n \log\log T)$	$O(n \log\log T)$
Comp. Pyr.	$O(n \log T)$	$O(n \log T)$	$O(n \log T)$
Quadtree	$O(Q_f)$	$O(Q_f)$	$O(Q_f)$
Pixel-Based	$O(n^2)$	$O(n^2)$	$O(n^2)$

Table 2: Window query space requirements (in bits)

Data	Query		
Structure	Exist	Report	Select
Incomp. Pyr.	$4fT^2/3$	$4fT^2/3$	$4(f+1)T^2/3$
Comp. Pyr.	$4fT^2/3$	$4fT^2/3$	$4fT^2/3$
Quadtree	$4fQ_f/3$	$4fQ_f/3$	$4fQ_f/3$
Pixel-Based	fT^2	fT^2	fT^2

6 Conclusions

This paper contains two new ideas. The first is the incomplete pyramid data structure, and the second is a window decomposition algorithm. Together they enable us to answer spatial queries efficiently. We have shown how to adapt the pyramid data structure to match our needs to store and retrieve spatial features. Although not mentioned in this paper, the algorithm to load features into the incomplete pyramid is much faster than that for the complete pyramid [1]. The speed-up arises from the fact that for the incomplete pyramid there is no need to fill all the intermediate nodes descending from a leaf node. The bottom-up window decomposition algorithm served as the underlying mechanism on top of which query answering algorithms were built. It had to run quickly and this requirement influenced our design heavily.

Directions for future work include the development of other spatial operations using the incomplete pyramid data structure. Combining spatial features with non-spatial data that describes them, while maintaining efficient access to both spatial and nonspatial data, is a goal of our research.

References

[1] W. G. Aref and H. Samet. Efficient techniques for the check-out operation in spatial databases. Submitted for publication, 1990.

[2] W. G. Aref and H. Samet. Window queries in spatial databases. In preparation, 1990.

[3] C. R. Dyer. The space efficiency of quadtrees. *Computer Graphics and Image Processing*, 19(4):335–348, August 1982.

[4] P. Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall, Englewood Cliffs, NJ, 1980.

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

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

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

[8] C. A. Shaffer. A formula for computing the number of quadtree node fragments created by a shift. *Pattern Recognition Letters*, 7(1):45–49, January 1988.

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

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

[11] L. Tucker. *Computer Vision Using Quadtree Refinement*. PhD thesis, Polytechnic Institute of New York, Brooklyn, May 1984.

[12] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, June 1964.