# A Window Retrieval Algorithm for Spatial Databases Using Quadtrees

Walid G. Aref
Matsushita Information Technology Laboratory
Panasonic Technologies, Inc.
2 Research Way,
Princeton, NJ 08540
Phone: 609-734-7349, Fax: 609-987-8827
E-mail: aref@mitl.research.panasonic.com

Hanan Samet
Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
The University of Maryland
College Park, Maryland 20742
Phone: 301-405-1755, Fax: 301-314-9115
E-mail: hjs@cs.umd.edu

**Abstract**

An algorithm is presented to answer window queries in a quadtree-based spatial database environment by retrieving the covering blocks in the underlying spatial database. It works by decomposing the window operation into sub-operations over smaller window partitions. These partitions are the quadtree blocks corresponding to the window. Although a block $b$ in the underlying spatial database may cover several of the smaller window partitions, $b$ is only retrieved once. As a result, the algorithm generates an optimal number of disk I/O requests to answer a window query (i.e., one request per covering block). The algorithm uses an auxiliary main-memory data structure, called the active border, which requires additional storage of $O(n)$, for a window query of size $n \times n$. An analysis of the algorithm's execution time and space requirements are given, as are some experimental results.

## 1 Introduction

Because of the large volume of spatial databases, spatial access methods are usually used to organize and speed up the retrieval of spatial objects. Several spatial access methods make use of a regular decomposition of space (such as that induced by a quadtree) in order to organize and store spatial data. We focus on a disjoint decomposition of space (i.e., features are not permitted to overlap). Some examples of spatial databases with disjoint features include crop coverage, road networks, topography, etc.

The large volume of spatial data imposes the need to store it in disk files. However, indexing techniques based on disjoint decomposition enables spatial features to be accessed quickly without having to search the entire database. On the other hand, as a result of decomposing the underlying space, a spatial feature gets partitioned into multiple smaller pieces, which needs special treatment both from the view points of the indexing method and the spatial operations. In this paper, our focus is on one of the very important spatial operations, namely the window retrieval operation.

Usually, spatial features span 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 subsequent operations is done by windowing, where the query window can be of any polygonal shape. Given a window $w$, some examples of window-based queries are: report all features inside $w$, intersect feature $f$ with feature $b$ only inside $w$, determine if feature $f$ exists in $w$, etc. In this paper, we present a new window retrieval algorithm that performs a minimal amount of disk reads to the underlying spatial database to answer the window query.

The rest of the paper is organized as follows. Section 2 covers some background material. In Sections 3 and 4 we present our new window algorithm and analyze its performance. Section 5 gives empirical results of the performance of our new algorithm in contrast to other window retrieval algorithms. Section 6 contains concluding remarks.

## 2 Background

### 2.1 Alternative Data Representations

There are many ways of representing and organizing spatial objects inside a spatial database. One way is to represent a spatial object by only one entity inside the data structure, e.g., by a point in higher dimensions as in the case of representing an $n$-dimensional polygon having $k$ boundary points by a point in $nk$-dimensions and then store it in a point data structure (e.g., the Grid File [26]), or by some conservative approximation of the object as in the case of representing the same polygon by its minimum enclosing rectangle (e.g., the R-tree [17]). An alternative way is to represent a spatial object by more than one entity inside the data structure, e.g., by partitioning the spatial object into a collection of convex polygons (the cell tree [15]), a collection of square blocks (the quadtree [21]), or a collection of rectangles (the R$^+$-tree [12]). In some of these data structures a spatial object is represented by its internal region, i.e., based on the spatial occupancy of the object. Examples of data structures that make use of this representation are the region quadtree [21, 32], the bintree [22], and the Z-Order [28]. In this paper, we focus on the latter group of data structures.

There are several advantages for using access methods that are based on spatial occupancy. Representations of spatial data that are based on this method are very suitable for a wide variety of data intensive applications (e.g., medical imagery and geographical information systems). Several researchers have investigated the usage of these structures inside a database environment (e.g., PROBE [27], SIRO-DBMS [1], and SAND [3]). As pointed out in [13], it is straightforward to implement these data structures in a database system because they require common facilities that are already present in almost all database systems (mainly any access method that provides both sequential and direct access, e.g., the B-Tree). In this paper, we use the *linear quadtree* [13, 32] as the underlying representation of the objects in the spatial database. For a comparison of the different data structures and the advantages and disadvantages of each one, see [32].

### 2.2 The Linear Quadtree

We assume that the underlying spatial database is stored in a disk-based linear quadtree. A quadtree is based on the principle of *recursive regular decomposition* of space into a maximal set of blocks whose sides are of size power of two and are placed at predetermined positions. The spatial objects are stored into the overlapping quadtree blocks. This way, the quadtree serves as a spatial index for the objects in the underlying spatial database.

The linear quadtree is constructed as follows. We start with a $T \times T$ object array of unit squares (termed *blocks*) where $T$ is a power of 2, and successively subdivide the array into quadrants. The subdivision process stops when blocks are obtained that consist of homogeneous data, i.e., blocks that are entirely contained in an object or entirely outside it. One way of representing this process is by a tree of degree 4 in which the root node represents the entire object array, non-leaf nodes correspond to non-homogeneous blocks (partly inside and partly outside an object), and the leaf nodes correspond to those blocks of the array for which no further subdivision is necessary. Leaf nodes are said to be black or white depending on whether their corresponding blocks are entirely within or outside of an object, respectively. All non-leaf nodes are said to be gray.

After decomposing the object into blocks using quadtree decomposition, instead of representing an object by a tree structure, an alternative way is to use the *linear quadtree*. In the linear quadtree, we store only the non-empty (black) leaf nodes. As a result, each object is represented by a set of squares (termed Morton blocks [24]) that collectively approximate (and cover) the object. A one-dimensional key value is assigned to each Morton block which maps the two-dimensional Morton blocks into the one-dimensional space. Then, one of the common one-dimensional indexing techniques, e.g., the B-tree, is used to index the blocks. Morton blocks are ordered by their key value which is a function of two parameters: the coordinate values of one of its corners and the size of the square region corresponding to the Morton block. In addition, each Morton block has an identifier that indicates the object to which it belongs.

One way of performing the mapping of the $n$-dimensional space into the one-dimensional space is by using space filling curves, e.g., the Peano curve [29], or the Hilbert curve [18]. A space filling curve acts like a thread that passes through every cell element (or pixel) in the $n$-dimensional space so that every cell is visited only once. Thus, a space filling curve imposes a linear order of the cells in the $n$-dimensional space. The linear quadtree is based on the two-dimensional Peano curve (also termed the *Z*- or *N*-order [28]). Many studies have been conducted to compare the performance of spatial operations for spatial databases adopting certain space filling orders. The reader is referred to [2, 10, 14, 20].

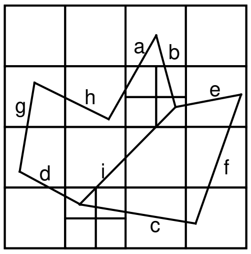



Figure 1: An example quadtree for storing line segments.

The linear quadtree can be used to store a variety of spatial object types. For example, a quadtree data structure for storing line segments [25] subdivides the feature space successively into four equal-sized quadrants. If the space contains more line segments than the capacity of a quadrant, then it is subdivided into quadrants, subquadrants, and so on, until blocks are obtained that overlap with at most a maximum number of line segments or that are entirely empty. A sample quadtree for storing line segments is given in Figure 1. For a comprehensive discussion of space filling curves and quadtrees variants, see [31, 32].

### 2.3 Spatial Operations

We overview three very important operations for spatial query processing, namely the nearest neighbor, the spatial join, and the window retrieval operations. We use example spatial databases for illustration. Figure 2 contains sample schemas for the spatial databases and Figure 3 gives a road-network database where roads are the database objects.

(a) road-network - name, type: char(30); left-zip-code,
right-zip-code: integer; coords: line-segment

(b) land-use - name, usage: char(30); zip-code: integer;
density: float; region: polygon

Figure 2: Sample schemas of the (a) road-network database, and (b) the land-use database.

The "nearest_to" operation can be considered as an aggregate function over the entire set of objects in the database (e.g., the spatial equivalent of the aggregate function "min()"). In fact, nearest_to can be expressed as min(distance($o, p$)), where distance is a scalar function that returns the distance between object $o$ and a query point

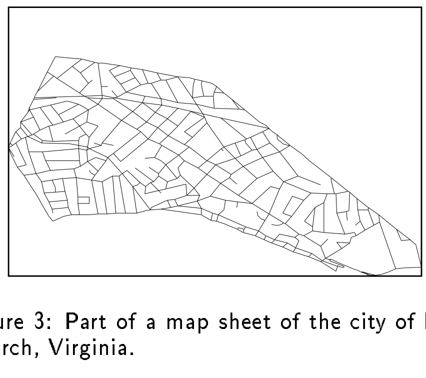
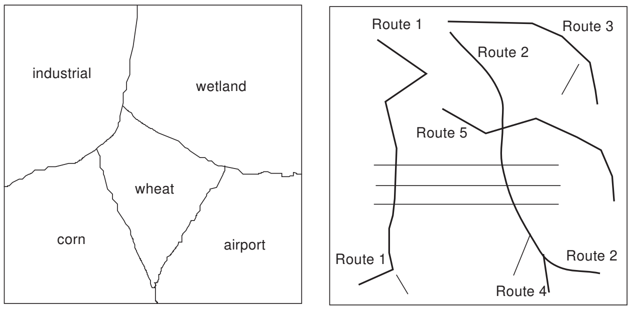


Figure 3: Part of a map sheet of the city of Falls Church, Virginia.




(a) (b)

| land.usage | road.name |
|---|---|
| industrial | Route 1 |
| corn | Route 1 |
| airport | Route 2 |
| airport | Route 4 |
| ⋮ | ⋮ |

(c)

Figure 4: (a) A simple land-use database, (b) a simple road-network database, (c) the land-use/road pairs that intersect each other.

$p.\ o$ ranges over all the objects of the underlying database. Efficient algorithms exist for evaluating the min(distance) function that avoid scanning the entire database [11, 19].

### 2.3.1 The Spatial Join Operation

Spatial join is a fundamental operation for answering queries that involve spatial predicates. It combines entities from two spatial databases into single entities whenever the combination satisfies the spatial join condition (e.g., if they overlap in space). For example, Figure 4c shows the result of joining a land-use database (Figure 4a) with a road-network database (Figure 4b). The join condition in this case is: `landuse.region intersects road.coords` (the schemas for the road-network and land-use databases are given in Figure 2).

As pointed out in [8], both the CPU and the disk read costs of the spatial join operation are very significant. As a result, extensive research has been conducted on alternative ways of processing the spatial join efficiently (e.g., see [6, 7, 8, 16, 27, 30]). Becker [7], and Becker, Hinrichs, and Finke [6] propose an algorithm for the efficient evaluation of spatial join for databases of point objects. Günther [16] presents a hierarchical spatial join algorithm that applies efficiently for a family of tree-based data structures, termed the generalization tree. Brinkhoff, Kriegel, and Seeger [8] apply a similar idea in the context of the R-tree [17]. In addition, they present several techniques that reduce both the disk read and CPU costs of the spatial join significantly. Rotem [30], and later, Lu and Han [23], suggest precomputing the spatial object pairs satisfying a certain spatial relationship and storing them in *spatial join indices* in order to speed up the spatial join at query runtime. Orenstein and Manola [27] present two algorithms for spatial join where the underlying representation of spatial data is the Z-Order [28].

### 2.3.2 The Window Overlap and Containment Operations

Window overlap and containment are central operations in spatial databases, and serve as building blocks for a number of queries. A window can be in the form of a rectangle or a polygon. For example, Figure 5 shows the roads in the city of Falls Church that are contained in window `w` (with corner values (100,100) and (300,300), respectively.) The roads that qualify can be fed to other operations for closer investigation.

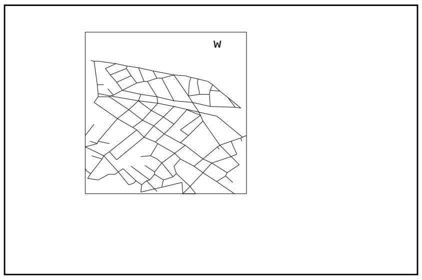



Figure 5: Roads of the city of Falls Church that are contained in window `w`. The origin of the space is at the lower-left corner.

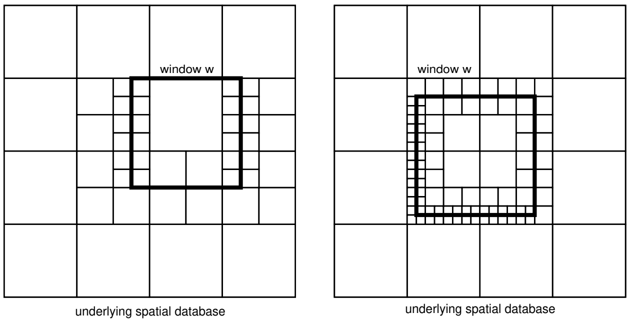



Figure 6: The decomposition of (a) a 12 $\times$ 12 window, and (b) a 13 $\times$ 13 window into maximal quadtree blocks.

Orenstein and Manola [27] give an algorithm for answering window queries where they treat a window query as a special form of a spatial filter operation. They treat the spatial database as the first input stream and the window as the second input stream and apply their spatial filter algorithm.

Consider a window query which seeks to determine the spatial objects that overlap window $w$ on the spatial database of line segments given in Figure 1. Our approach is to retrieve the set of blocks, say $S_w$, of the underlying spatial database that overlap the set of quadtree blocks, say $W_w$, that comprise the window. The rationale for using the quadtree blocks of the window is to match the quadtree decomposition of the underlying spatial database. This makes it more straightforward to answer the window query since there is a direct correspondence between each window block and some overlapping quadtree block(s) in the underlying spatial database. This approach is also the same as the one in [27]. The answer to the window query is the union of all the answers generated by querying the underlying spatial database with the maximal quadtree blocks comprising the window. We term this algorithm `Algorithm-1`. We briefly describe `Algorithm-1` below. A more detailed description is given in [4].

`Algorithm-1` uses a quadtree window decomposition mechanism. Figure 6 shows the quadtree decomposition of two windows. The decomposition can be achieved using a window decomposition algorithm given in [5]. It decomposes a

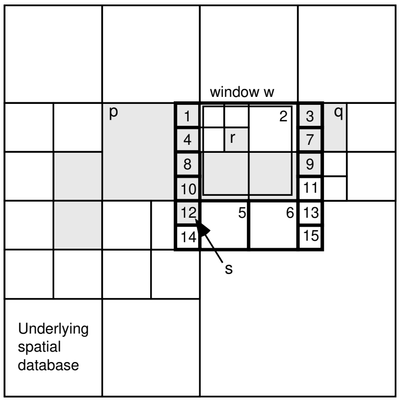



Figure 7: Examples where more than one window block retrieves the same block of the database.

two-dimensional window of size $n \times n$ in a feature space (e.g., an image) of size $T \times T$ into its maximal quadtree blocks in $O(n \log \log T)$ time. Once the set $W_w$ has been determined, `Algorithm-1` accesses the underlying spatial database for each window block $b$ in $W_w$, and retrieves the database blocks that intersect $b$.

The drawback of this algorithm is that many of the elements of $S_w$ may be retrieved more than once. For example, in Figure 7, the algorithm would retrieve block p of the underlying spatial database four times (once for each of the maximal window blocks 1, 4, 8, and 10). We assume that the underlying spatial database is disk-resident, and we often speak of the operation of retrieving a block of the underlying spatial database as a disk I/O request. This means that redundant disk I/O requests will result.[1]

The problem of `Algorithm-1` is that the process of generating the maximal blocks that comprise the window only depends on the query window and not on the decomposition of space induced by the underlying spatial database. We overcome this problem by generating and retrieving each covering block in the database just once. This is achieved by controlling the window decomposition procedure through the use of information about blocks of the underlying spatial database that have already been retrieved. We use an approach based on active borders [33], at the expense of some extra storage. A detailed description of our approach is given in the next section.

It is important to note that we retrieve blocks in the underlying spatial database by use of partial information about their relationship to other blocks (e.g., containment and overlap). We do not retrieve a block of the database by its identifier. Instead, we are given the spatial description of a window block, say $b$. The spatial description of $b$ is used to retrieve all the blocks of the database that are spatially related to $b$ (e.g., the blocks that contain, or are contained in, $b$). Blocks in the database can be retrieved more than once if they satisfy some spatial relationship with respect to different window blocks.

[1] This problem may be alleviated via appropriate use of buffering techniques. However, in this paper we show how to avoid the problem entirely by retrieving each block of the underlying spatial database just once without relying on buffering techniques.

## 3 The New Algorithm

In `Algorithm-1`, described in Section 2, when a block $q$ in the underlying spatial database covers more than one maximal quadtree block in the window, $q$ will be retrieved several times. This could be overcome by avoiding the invocation of the retrieval step for some of the maximal quadtree blocks. The issue then is how do we skip some of the maximal quadtree blocks in the window. In order to understand this issue, we briefly focus on the relation between the maximal quadtree blocks of the window decomposition and the quadtree blocks in the underlying spatial database.

Assume that $b$ is a maximal window block that is generated by the window decomposition algorithm. Due to the quadtree decomposition of both the window and the underlying spatial database, $b$ can either be contained in, or contain, one or more quadtree blocks of the underlying spatial database. In particular, there are three possible cases as illustrated by Figure 7. Case 1 is demonstrated in the figure by window block 2 which contains more than one quadtree block of the underlying spatial database. All of these blocks have to be retrieved (e.g., from the disk), and processed by the algorithm (e.g., the spatial objects associated with these blocks will be reported as intersecting the window). The second case is illustrated by window block 9 of Figure 7. Block 9 contains exactly one block of the underlying spatial database which will have to be retrieved (e.g., from the disk) as well. The third case is demonstrated by window blocks 1, 4, 8, and 10 of Figure 7 which all require retrieving (e.g., from the disk) the same quadtree block (i.e., block p of the underlying spatial database). Case 3 arises frequently in any typical window query, as shown by the experiments conducted in Section 5, thereby resulting in a large number of redundant disk I/O requests.

Our new algorithm is based on the following observation (its proof can be found in [4]):

> Assume that a block, say $b$, is a maximal block that lies inside the window $w$ and overlaps with a block of the underlying spatial database, say $q$. If $q$ is of greater size than $b$, then $q$ must intersect with at least one of the boundaries of the window $w$ (refer to Figure 8 for illustration).

In other words, there cannot be database blocks that are bigger than the the intersecting window blocks which are in the middle of the query window, These big database blocks have to intersect the boundary of the query window. Our window retrieval algorithm is based on this observation which we illustrate further later in this section.

The new algorithm consists of procedures `WINDOW_RETRIEVE`, `GEN_SOUTHERN_MAXIMAL`, and `MAX_BLOCK`. It works for an arbitrary rectangular window (i.e., it need not be square). We avoid generating non-maximal quadtree blocks in the window (or at least generate a bounded number of them) by using the same technique as in [5], which we outline below. Note that there are $O(n^2)$ non-maximal blocks inside an $n \times n$ window. Also, each maximal quadtree block in the window is processed only once (i.e., as a neighbor of another node) regardless of its size.

We make use of an *active border* data structure [33] which is a separator between the window regions that have already been processed and the rest of the window. The active border serves as the spatial analog to the hash-table, that we tailor to match the needs of this type of spatial retrieval. The active border can also be viewed as simulating the spatial equivalent of a sort-merge list of pages which is used


Figure 8: Possible overlaps between blocks of the database and the (a) northern, (b) western, (c) eastern, and (d) southern borders of the window.

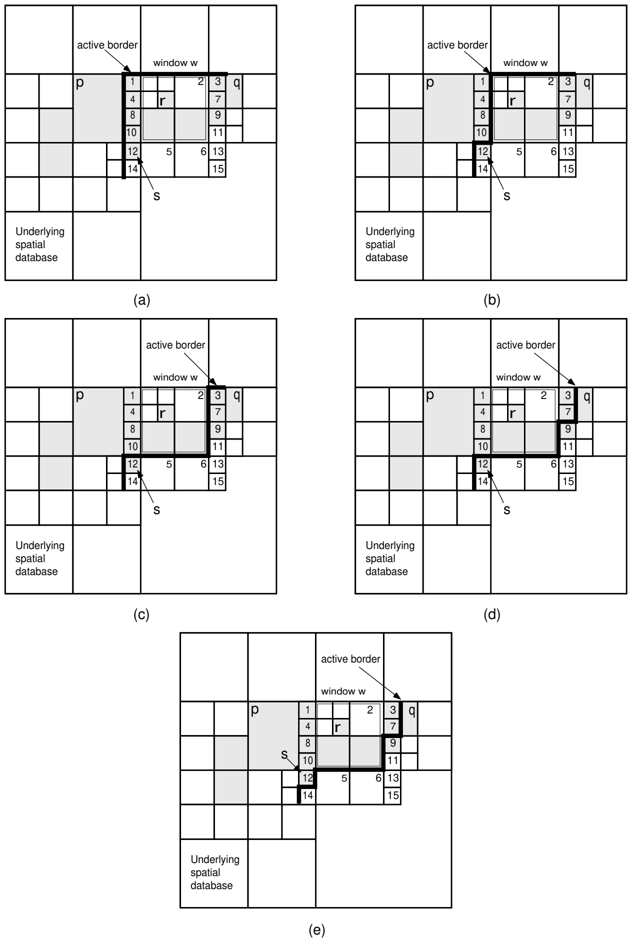



Figure 9: The active border at (a) the initial stage. The active border after processing (b) window block 1, (c) window block 2, (d) window block 3, (e) window block 12.

in database query processing when accessing data through secondary indexes [9].

Note that the active border in our case differs from that in [33] (which looks like a staircase) because of the nature of the block traversal process. In particular, we traverse the blocks in the window in a row-by-row manner rather than in quadrant order (i.e., `NW`, `NE`, `SW`, `SE`).

Figure 9 represents the first five steps of the execution of the algorithm for the query window $w$. The heavy lines in Figure 9a represents the active border for window $w$ at the initial stage of the algorithm. In generating a new block, the window decomposer has to consult the active border in order to avoid generating a disk I/O request for a window region that has already been processed by a block of the underlying spatial database that has already been retrieved.

The active border is maintained as follows. First, a window block, say $b$, is generated by the window decomposer and a disk I/O request is issued to access the region of the underlying spatial database corresponding to $b$. Assume that $b$ overlaps in space with block $u$ in the database. Therefore, $u$ is retrieved as a result of the disk I/O request corresponding to $b$. The spatial objects inside $u$ are processed and thus there is no need to retrieve $u$ again. As a result, the active border needs to be updated by block $b$ or $u$ depending on which one provides more coverage of the window region. Figure 9 illustrates the updating process of the active border. If $u$ has a larger overlap with the unprocessed portion of the window than $b$ (e.g. window block `1` and block `p` of the underlying spatial database in Figure 9a, as well as window block 3 and block `q` of the database), then the active border is expanded using $u$'s region (Figure 9b). If $u$ is contained in $b$ (e.g., window block `2` and block `r` of the database in Figure 9a), then all the other blocks in the database have to be retrieved as well, and the active border is expanded by $b$'s region (Figure 9c). If the sizes of $b$ and $u$ are the same (e.g., window block `12` and block `s` of the database in Figure 9a), then the active border is expanded by either one of them (Figure 9e). Notice that, if we were using `Algorithm-1`, window blocks `4`, `8`, `10`, and `7` would still be processed and hence would generate four redundant disk I/O requests to retrieve blocks `p` and `q`.

Up to this point, we have not mentioned how we generate the maximal quadtree blocks inside a given window. This process is controlled by procedure `WINDOW_RETRIEVE`. Procedure `WINDOW_RETRIEVE` scans the window row-by-row (in the block domain rather than in the pixel domain), and visits the blocks within it that have not been visited in previous scans[2]. For each visited window block, say $b$, the underlying spatial database is queried and a corresponding quadtree block, say $q$, is retrieved from the database. According to the three cases presented in earlier in this section, that relate the location and size of both $b$ and $q$ with respect to the query window, procedures `GEN_SOUTHERN_MAXIMAL` and `MAX_BLOCK` generate $b$'s or $q$'s maximal southern neighboring blocks (in fact, only the portion of $q$ that lies inside the window will be used). `WINDOW_RETRIEVE` also makes sure that any of the remaining columns of row $r$ that lie within $b$ or $q$ are skipped. For example, consider Figure 7, where five scans are needed to cover the $12 \times 12$ window with maximal blocks. The first scan visits blocks `1`, `2`, and `3`; the second scan visits blocks

[2] Observe that we could have chosen to scan the window in a column-by-column fashion instead of row-by-row. The result is unchanged as long as the data structures for keeping track of the active border are reoriented appropriately.

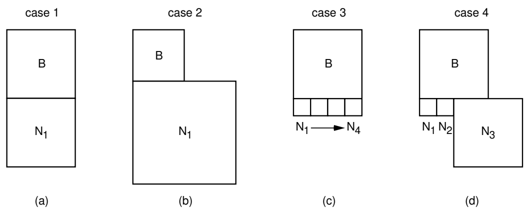



Figure 10: (a), (b), and (c) are examples of possible block/southern-neighbor pairs; (d) cannot occur in a quadtree decomposition.

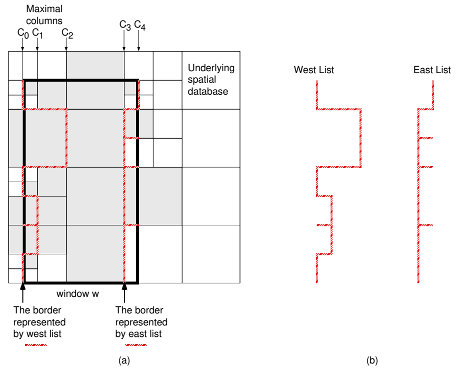



Figure 11: (a) Example of a window (heavy line) and the pockets (heavy lines) along the west and east boundary induced by the database, and (b) the representation of the `WestList` and `EastList` data structures corresponding to the active border.

`12`, `5`, `6`, and `9`; the remaining scans visit blocks `14` and `11`; `13`; and `15`. Notice that once blocks `5` and `6` have been visited, their columns (i.e., 2-5) have been completely processed. Also, observe that when block `1` is generated, block `p` of the underlying spatial database, which overlaps with block `1`, is retrieved. As a result, window blocks, `4`, `8`, and `10` are skipped. This way, the algorithm can avoid reaccessing `p` by skipping all the window blocks that overlap with `p`. As a consequence, the southern neighbors of `p` (and not those of block `1`) are generated by the algorithm.

Procedure `GEN_SOUTHERN_MAXIMAL` generates the southern neighbors (maximal blocks) $N_1$ through $N_m$ for each maximal block $B$ generated by `WINDOW_RETRIEVE` and that is not contained in another maximal block. There are a number of possible cases illustrated in Figure 10. If $m = 1$, then $N_1$ is greater than or equal to $B$. Otherwise, the total width of blocks $N_1$ through $N_m$ is equal to that of $B$. It is impossible for the total length to exceed that of $B$ unless there is only one neighbor (see Figure 10b). Procedure `MAX_BLOCK` takes as its input a window, say $w$, and the values of the $x$ and $y$ coordinates of a pixel, say ($col$,$row$), and returns the maximal block in $w$ with ($col$,$row$) as its upper-leftmost corner. The resulting block has width $2^s$, where $s$ is the maximum value of $i$ ($0 \leq i \leq \log T$, where $T \times T$ is the size of the image space) such that $row \bmod 2^i = col \bmod 2^i = 0$ and the point ($row + 2^i$, $col + 2^i$) lies inside $w$.

A detailed proof in [4] suggests that the active border does not contain any holes, and hence it is enough to describe it by its boundary. Moreover, since there are no holes in the active border, when a block of the underlying spatial database, say $q$, is retrieved and the algorithm checks its size against the corresponding window block, say $b$, if $q$'s size is larger than that of $b$, then the algorithm knows that $q$ has to intersect one of the window's boundaries. We make use of this property here. Figure 8 shows the four possible cases where the block retrieved from the underlying database intersects one of the window boundaries. Each of the four cases must be treated separately by the algorithm.

There is no need to maintain any data structures to explicitly store the northern portion of the active border since `WINDOW_RETRIEVE` can handle this portion directly. During the first row-by-row scan of the window by `WINDOW_RETRIEVE`, if a block of the underlying spatial database, say $q$, is retrieved that happens to intersect the northern boundary of the window (Figure 8a), then `WINDOW_RETRIEVE` skips the window blocks in the current row scan that overlap with $q$. The portion of the southern boundary of $q$ that lies inside the window is used to generate the southern neighboring blocks to be processed in the next scan.

When block $q$ of the underlying spatial database intersects only the southern boundary of the window (Figure 8d), then it also suffices for `WINDOW_RETRIEVE` to skip all the window blocks that are adjacent to the window block that initiated $q$'s retrieval. Although this seems intuitive, it is not straightforward to see that all of the processing of block $q$ by `WINDOW_RETRIEVE` is localized in one part of the algorithm. In particular, although true (for a proof, see [4]), it is not directly obvious that all the blocks that overlap with $q$ will be processed by `WINDOW_RETRIEVE` at the same time so that they can be skipped. Thus as a result of this localized processing, there is no need to maintain any explicit data structures in this case either. If $q$ intersects the western or eastern boundaries (Figures 8b and 8c), its overlap with the window creates a pocket-like region that needs to be stored in two separate lists, `WestList` or `EastList`, respectively. Each time a window block is generated, it is checked against the active border to make sure that the block is not covered by a previously retrieved block of the database. Below, we show how to perform this check in constant time.

To facilitate our presentation, we represent both `WestList` and `EastList` as two one-dimensional arrays, each of length equal to the height of the window: `WestList`$[r : r+n-1]$ and `EastList`$[r : r+n-1]$, where the window height is $n$ and $(r, c)$ corresponds to the $x$ and $y$ coordinate values of its upper-left corner. Figure 11b shows the border represented by each of the two arrays as a result of extracting an $8 \times 12$ window from the database in Figure 11a. Let $(r_q, c_q)$ be the location of the upper-left corner of $q$. If $q$ intersects the west boundary of the window, then `WestList`$[r_q]$ is set to the pair $< c_q + s_q, s_q >$ where the first component of the pair denotes the $x$ coordinate value of $q$'s east boundary while the second component (i.e., $s_q$) denotes the size of $q$. The pair $< c_q + s_q, s_q >$ represents the pocket-like region resulting from the intersection of $q$ with $w$. Similarly, if $q$ intersects the east boundary of the window, then `EastList`$[r_q]$ is set to the pair $< c_q, s_q >$. Each time a window block is generated it is checked against the active border to make sure that the block is not covered by a previously retrieved block of the database. Notice that updating the active border only requires one array access (either updating `WestList` or `EastList` depending on whether $q$ intersects the west or east boundaries of the window, respectively), while checking a window block against the active border takes only two array accesses (one access to each of `WestList` and `EastList`). Therefore, maintaining the active border, whether updating or checking, takes $O(1)$ time.

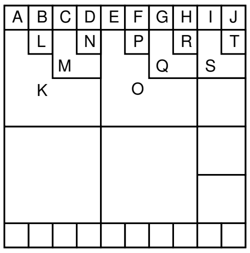



Figure 12: The neighboring blocks to the south of blocks A-J in a $10 \times 10$ window. Blocks L, M, N, P, Q, R, and T are non-maximal, while blocks K, O, and S are maximal.

Observe that WINDOW_RETRIEVE always generates maximal neighboring blocks, and a bounded number of non-maximal blocks. We refer to Figure 12 for illustration. When processing the first row of the window, 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, procedure WINDOW_RETRIEVE skips them in the next scan since they are 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 $x$ and $y$ coordinate values of its upper-left corner and its size. A pseudo-code listing and a proof of correctness of the algorithm can be found in [4].

## 4 Complexity Analysis

Based on the observation in Section 3, that relates the size of the query window blocks and their relation with the blocks in the underlying database, we are able to restrict the size of the active border to a worst-case space complexity of $O(n)$, instead of $O(n^2)$ for an $n \times n$ query window.

The CPU time complexity of the algorithm is affected by two factors: window decomposition, and the maintenance of the active border. In [5], we show that the worst-case CPU time for window decomposition is $O(n \log\log T)$. In Section 3, we have shown that by simultaneously traversing the active border while generating and traversing the window blocks inside the query window, we can perform all of the active border maintenance operations in $O(1)$ time, i.e., constant time. As a result, the overall CPU time for the window retrieval algorithm is $O(n \log\log T)$.

Assume that the number of output quadtree blocks is $M$ database blocks. As a result of the indexing technique (e.g., a B+-tree) used for organizing the linear quadtree that represents the spatial database, each access to retrieve one of the $M$ blocks results in a search into the B+-tree. If the spatial database is represented by $N$ linear quadtree blocks that are stored in the B+-tree, then the overall I/O execution time of the window retrieval algorithm is $O(M \log N)$.

## 5 Empirical Results

Figure 13 shows the results of experiments comparing the number of disk I/O requests (i.e., blocks retrieved) to answer a window query using Algorithm-1 (described in Section 2.3.2) with the number of disk I/O requests generated by WINDOW_RETRIEVE (the algorithm described in this paper). Our data consists of maps of the road network of the

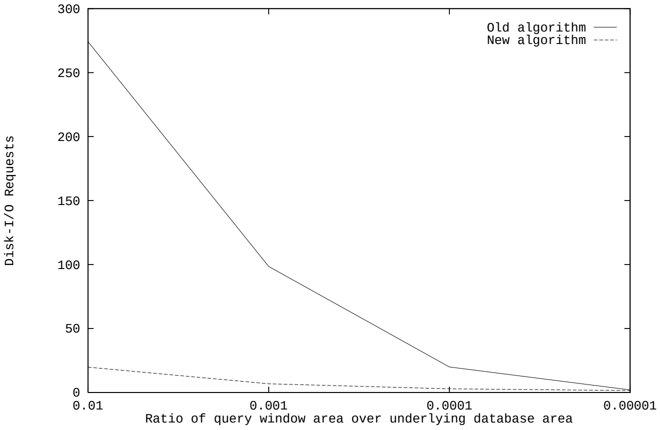



Figure 13: Results of an experiment to compare the disk I/O performance of the new and old window retrieval algorithms.

US provided by the Bureau of the Census. A sample map containing line segments is given in Figure 3. The maps are represented using the PMR quadtree [32], a variant of a quadtree for storing vector data.

The $x$-axis corresponds to the ratio between the window area and the area of the underlying spatial database. Experiments were run for the ratios .01, .001, .0001, and .00001. For each such ratio, a set of 500 randomly positioned rectangles were generated. A window query is processed for each rectangle using both algorithms. The $y$-axis corresponds to the log of the average of the disk I/O requests for each set of rectangles. The result of using WINDOW_RETRIEVE is a reduction in disk I/O requests varying from 25%-92%. Notice that the window decomposition part of the two algorithms have the same worst-case execution time complexity (i.e., $O(n \log\log T)$) as shown in Section 4.

## 6 Conclusions

An algorithm was presented for retrieving the blocks in a quadtree-based spatial database that overlap a given window. It is based on decomposing a window into its maximal quadtree blocks, and performing simpler sub-queries to the underlying spatial database. Each block in the database is only retrieved once. The algorithm is proven (analytically and experimentally) to have an improvement in disk I/O performance. The algorithm requires some extra space (on the order of the width of the window), to store the active border. It remains to consider how the algorithm can be adapted to handle databases with non-disjoint objects.

## 7 Acknowledgements

The second author gratefully acknowledges the support of the National Science Foundation under Grant IRI-92-16970.

## References

[1] D. J. Abel. SIRO-DBMS: A database tool-kit for geographical information systems. *Intl. J. of Geographical Information Systems*, 3(2):103–116, Apr.–Jun. 1989.

[2] D.J. Abel and D.M. Mark. A comparative analysis of some two-dimensional orderings. *Intl. J. of Geographical Information Systems*, 4(1):21–31, Jan.–Mar. 1990.

[3] W. G. Aref and H. Samet. Extending a DBMS with spatial operations. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases - 2nd Symposium, SSD'91. Lect. Notes in Computer Science 525*, pp. 299–318. Springer-Verlag, Berlin, 1991.

[4] W. G. Aref and H. Samet. An efficient window retrieval algorithm for spatial query processing. Tech. Report CS-2866, University of Maryland, College Park, MD, Mar. 1992.

[5] W. G. Aref and H. Samet. Decomposing a window into maximal quadtree blocks. *Acta Informatica*, 30:425–439, 1993.

[6] L. Becker, K. Hinrichs, and U. Finke. A new algorithm for computing joins with grid files. In *Proc. of the 9th Intl. Conf. on Data Engr.*, pp. 190–197, Vienna, Austria, Apr. 1993.

[7] L. A. Becker. *A New Algorithm and a Cost Model for Join Processing with Grid Files.* PhD thesis, University of Siegen, Jul. 1992.

[8] T. Brinkhoff, H. P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, pp. 237–246, Washington, DC, May 1993.

[9] R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems.* Benjamin/Cummings, Redwood City, CA, 1989.

[10] C. Faloutsos. Analytical results on the quadtree decomposition of arbitrary rectangles. *Pattern Recognition Letters*, 13(1):31–40, Jan. 1992.

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

[12] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, pp. 426–439, San Francisco, May 1987.

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

[14] M.F. Goodchild and A.W. Grandfield. Optimizing raster storage: An examination of four alternatives. In *Proc. of the 6th Intl. Symposium on Automated Cartography*, pp. 400–407, Ottawa, Canada, Oct. 1983.

[15] O. Günther. The design of the cell tree: an object-oriented index structure for geometric databases. In *Proc. of the 5th IEEE Intl. Conf. on Data Engr.*, pp. 598–605, Los Angeles, Feb. 1989.

[16] O. Günther. Efficient computation of spatial joins. In *Proc. of the 9th Intl. Conf. on Data Engr.*, pp. 50–59, Vienna, Austria, Apr. 1993.

[17] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Intl. Conf. on Mgmt. of Data*, pp. 47–57, Boston, Jun. 1984.

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

[19] E. G. Hoel and H. Samet. Efficient processing of spatial queries in line segment databases. In O. Günther and H. J. Schek, editors, *Advances in Spatial Databases - 2nd Symposium, SSD'91. Lect. Notes in Computer Science 525*, pp. 237–256. Springer-Verlag, Berlin, 1991.

[20] H. V. Jagadish. Spatial search with polyhedra. In *Proc. of the 6th IEEE Intl. Conf. on Data Engr.*, pp. 311–319, Los Angeles, Feb. 1990.

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

[22] K. Knowlton. Progressive transmission of grey-scale and binary pictures by simple, efficient, and lossless encoding schemes. In *Proc. of the IEEE 68*, number 7, pp. 885–896, Jul. 1980.

[23] W. Lu and J. Han. A new algorithm for computing joins with grid files. In *Proc. of the 8th Intl. Conf. on Data Engr.*, pp. 284–292, Tempe, Arizona, Feb. 1992.

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

[25] R. C. Nelson and H. Samet. A consistent hierarchical representation for vector data. *Computer Graphics*, 20(4):197–206, Aug. 1986. (also Proc. of the SIGGRAPH'86 Conference, Dallas, Aug. 1986).

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

[27] J. A. Orenstein and F. A. Manola. PROBE spatial data modeling and query processing in an image database application. *IEEE Trans. on Software Engr.*, 14(5):611–629, May 1988.

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

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

[30] Doron Rotem. Spatial join indices. In *Proc. of the 5th Intl. Conf. on Data Engr.*, pp. 500–509, Kobe, Japan, Apr. 1991.

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

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

[33] H. Samet and M. Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE Trans. on Pattern Analysis and Machine Intelligence*, 7(2):229–240, Mar. 1985.