CAR-TR-611
CS-TR-2866

IRI-9017393
March 1992

AN EFFICIENT WINDOW RETRIEVAL ALGORITHM FOR SPATIAL QUERY PROCESSING

Walid G. Aref
Hanan Samet

Computer Science Department and
Center for Automation Research and
Institute for Advanced Computer Studies
University of Maryland
College Park, MD 20742-3275

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. The fact that b is only retrieved once means that when the underlying spatial database is stored in a disk-based quadtree data structure, 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 executes in $O(n \log\log T) + M$ time for an $n \times n$ window in a feature space (e.g., an image) of size $T \times T$ (e.g., pixels) where M is the number of quadtree blocks in the underlying spatial database that overlap the window. The algorithm avoids multiple retrieval of the same covering block in the underlying spatial database by using an active border. This requires additional storage of $O(n)$. A proof of correctness and an analysis of the algorithm's execution time and space requirements are given, as are some experimental results.

Keywords: databases, design of algorithms, data structures, spatial databases, range query, quadtree space decomposition, active border

The support of the National Science Foundation under Grant IRI-9017393 is gratefully acknowledged.

1 Introduction

Many 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). A disjoint decomposition enables spatial features to be accessed quickly without having to search the entire database. 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. In this paper, our focus is on the efficient retrieval from disk of the relevant parts of a spatial database for answering spatial window queries. Note that window queries are spatial analogs of range queries.

We consider a spatial database in which the spatial objects are organized using the quadtree data structure.[1] The 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. For example, a quadtree data structure for storing line segments [7] 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 quadtrees and other hierarchical spatial data structures, see [8, 9].

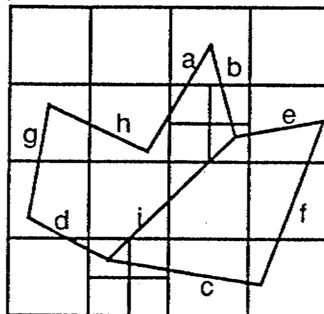


Figure 1: An example PMR quadtree for storing line segments.

For example, 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. 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.

Figure 2 shows the quadtree decomposition of two windows. The decomposition can be achieved using a window decomposition algorithm given in [2]. It decomposes a two-dimensional window of

[1] For spatial databases consisting of non-disjoint features, a pyramid is more appropriate. This is beyond the scope of this paper, but see [1].

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, we simply retrieve the elements of S_w that overlap each of its elements. The drawback of this algorithm is that many of the elements of S_w may be retrieved more than once. For example, in Figure 3, 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.[2] One solution is to keep track of all blocks that have already been retrieved. This is not easy without additional storage (see [3] for a discussion of the similar issue of uniquely reporting answers in a spatial database).

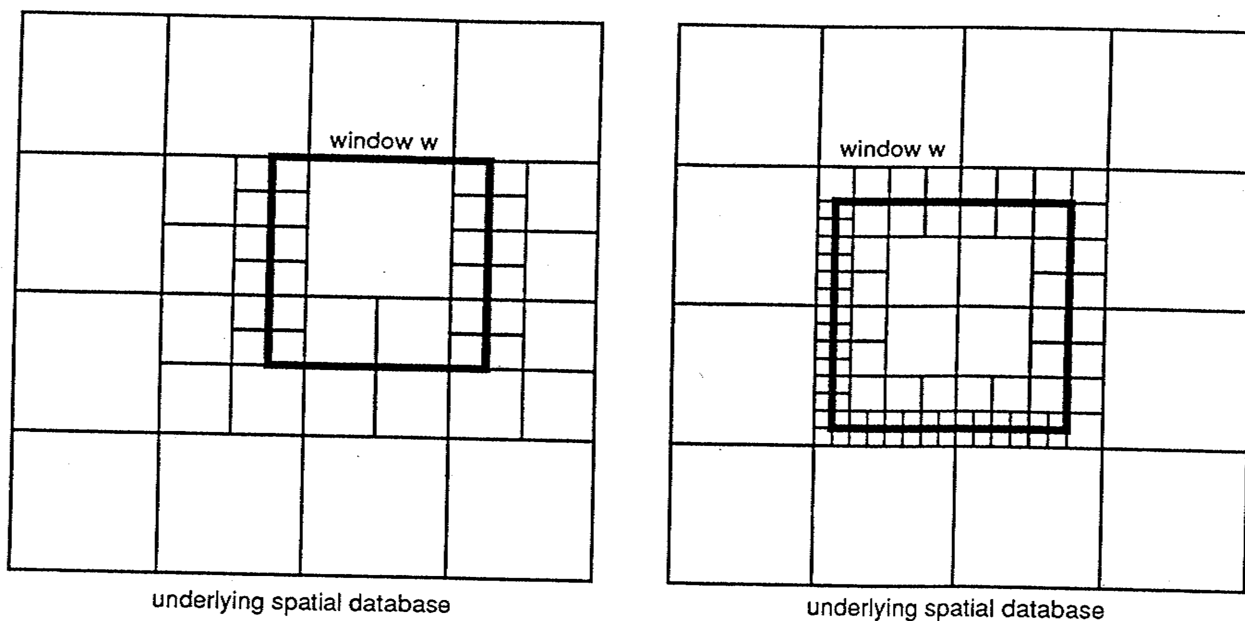


Figure 2: The decomposition of (a) a 12×12 window and (b) a 13×13 window into maximal quadtree blocks.

The problem with using the algorithm in [2] is that the process of generating the maximal blocks that comprise the window only depends on the query window and does not take into consideration the decomposition of space induced by the underlying spatial database. We overcome this problem by generating and retrieving each covering block in the underlying spatial 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 [10], at the expense of some extra storage. The algorithm that we present performs this task with the same worst-case execution time complexity as the one in [2] (i.e., $O(n \log \log T) + M$) where M is the number of quadtree blocks in the underlying spatial database that overlap the window, with an added storage cost of $O(n)$. The difference is that the additive term M is a constant rather than $\max(N, M)$, where N is the number of maximal quadtree blocks in the window, as in [2]. A general significance of both our algorithm and the one in [2] is that although the window contains n^2 pixel elements, the worst-case execution time complexity of

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

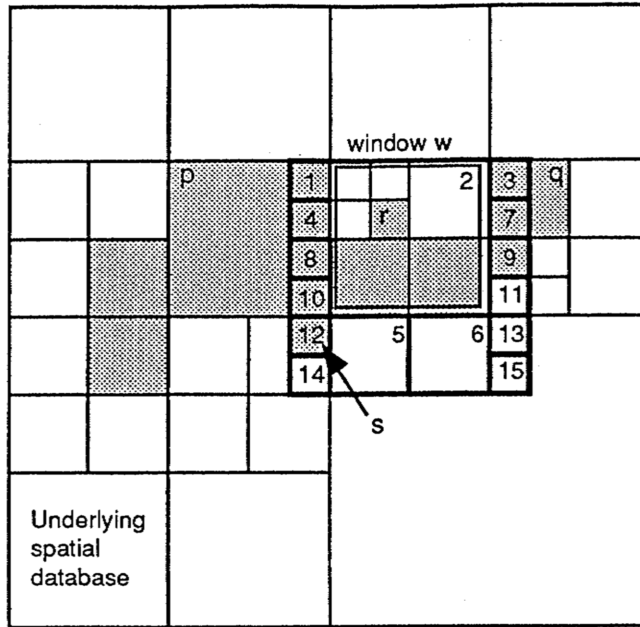


Figure 3: Examples where more than one window block retrieves the same block of the underlying spatial database.

the algorithms is almost linearly (and not quadratically) proportional to the window diameter, and is independent of other factors.

It is important to note that we retrieve blocks in the underlying spatial database by use of information (partial) about their relationships to other blocks (e.g., containment, overlap, subset, etc.). We do not retrieve a block of the underlying database by its identifier. If we could do this, then we could keep track of which blocks are retrieved via a hash table, for example, and avoid retrieving them again. 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 underlying spatial database that are spatially related to b (e.g., the blocks that contain, or are contained in, b). Blocks in the underlying spatial database can be retrieved more than once if they satisfy some spatial relationship with respect to different window blocks. In order to avoid retrieving the same block more than once when a different window block is processed, we maintain a spatial analog to the hash-table mechanism above. This is achieved through the usage of some spatial data structure, namely the active border, tailored 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 in database query processing when accessing data through secondary indexes [5].

The rest of this paper is organized as follows. Section 2 describes the algorithm and how to use it to answer window queries. It employs a different retrieval process from the one given in [2]. Section 3 contains an informal proof of correctness for the algorithm's block retrieval process, while an analysis of its worst-case execution time and space complexity is given in Section 4. Section 5 presents empirical results of the disk I/O behavior of the algorithm. Section 6 contains concluding remarks.

2 Algorithm

Answering a window query by first computing the maximal quadtree blocks comprising it and then retrieving the corresponding covering blocks in the underlying spatial database proceeds as follows. Assume a query window w, a spatial database S, a query function F that performs the appropriate variant of a window query test (e.g., a containment test) and a record of type answer_set that accumulates the answer to the window query.

```
answer_set procedure Algorithm-1(S,W,F);
  begin
    reference spatial_database S;
    value window W;
    value function F;
    block b;
    block set c;
    spatial_object set t;
    answer_set result;
    result := empty;
    decompose W into its maximal quadtree blocks;
    foreach block b in W do
      begin
        c := blocks in S that cover b;
        t := empty;
        foreach block q in c do t := F(W,q) U t;
          /* apply F to spatial objects associated with q */
        result := result U t;
      end;
    return(result);
  end;
```

By varying the function F and the data type answer_set, many window operations can be implemented using Algorithm-1. For example, to answer the report query (i.e., reporting the identity of all the features that exist inside a window), the function F simply identifies all the spatial objects inside the block of the underlying spatial database, and the data type answer_set is just a set of spatial object identifiers for the qualifying objects. To answer the exist query (i.e., determining if feature f exists in w), the function F tests whether or not f (or f's identifier) exists inside the block of the underlying spatial database, and the data type answer_set is the type Boolean while U is a *logical or* operation. To answer the select query (i.e., reporting the locations of all instances of feature f in the window), the function F simply tests whether or not f (or f's identifier) exists inside the block of the underlying spatial database, and the data type answer_set is a quadtree that stores in it the location of these blocks.

There is one principal issue in implementing this algorithm. This was discussed in Section 1. Recall that when 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 is how 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 3. 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 3. 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 3 which all require retrieving (e.g., from the disk) the same quadtree block (i.e., block p of the underlying spatial database)[3]). 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.

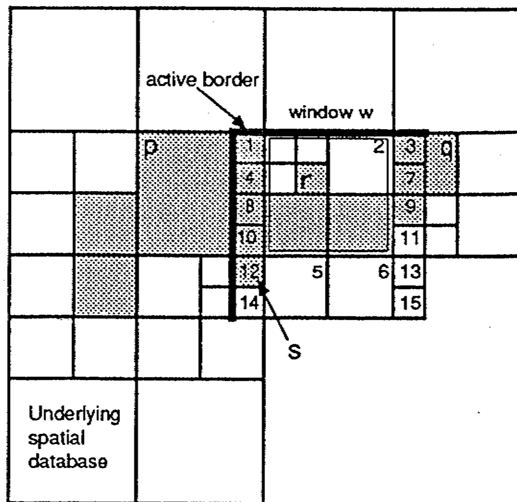
Our algorithm consists of procedures `WINDOW_RETRIEVE`, `GEN_SOUTHERN_MAXIMAL`, and `MAX_BLOCK`. They are outlined below. The code is given in the Appendix. It works for an arbitrary rectangular window (i.e., it need not be square). Each maximal quadtree block in the window is generated at most once. We avoid generating non-maximal quadtree blocks in the window (or at least generate only a bounded number of them) by using the same technique as in [2]. 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 [10] which is a separator between the window regions that have already been processed and the rest of the window. Note that the active border in our case will differ from the conventional one (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`).

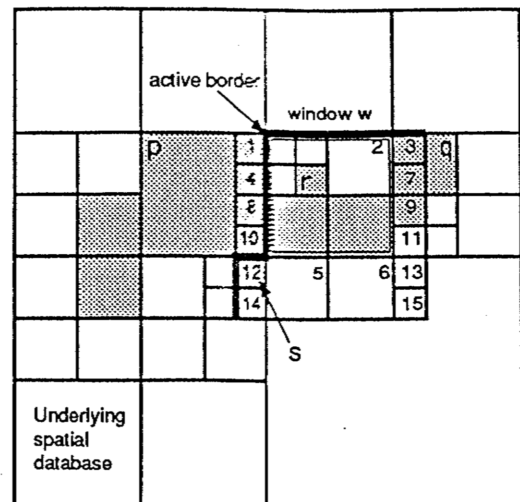
Figure 4 represents the first five steps of the execution of the algorithm for the query window w. The heavy lines in Figure 4a represent the active border for window w at the initial stage of the algorithm. In generating a new block, the window decomposer has to consult the active border in order to 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 underlying spatial 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 more than once. 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 4 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 4a, as well as window block 3 and block q of the underlying spatial database), then the active border is expanded using u's region (Figure 4b). If u is contained in b (e.g., window block 2 and block r of the underlying spatial database in Figure 4a), then all the other blocks in the underlying spatial database have to be retrieved as well, and the active border is expanded by b's region (Figure 4c). If the sizes of b and u are the same (e.g., window block 12 and block s of the underlying spatial database in Figure 4a), then the active border is expanded by either one of them (Figure 4e). Notice that, if we were using `Algorithm-1`,

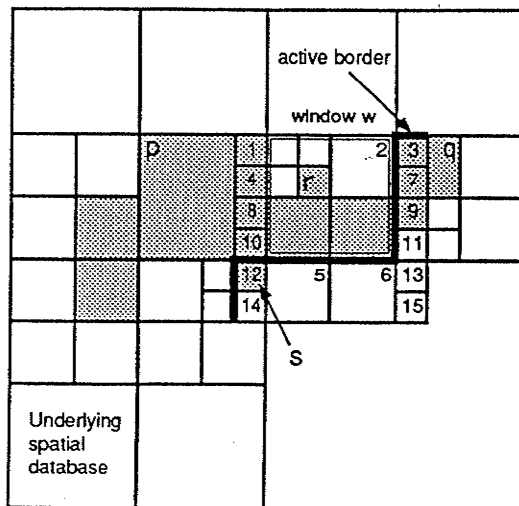
[3]Whether or not block p is in memory buffers is not of concern here.



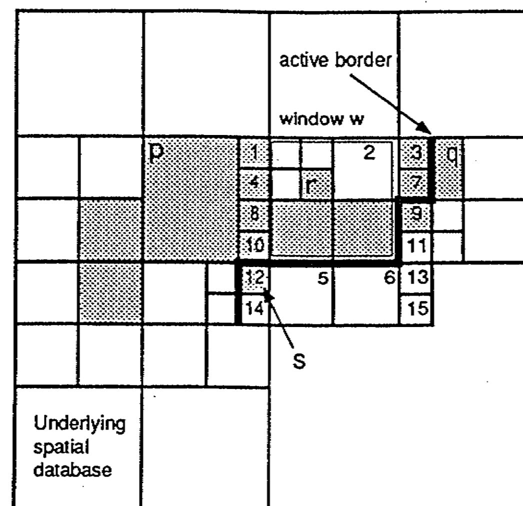
(a)



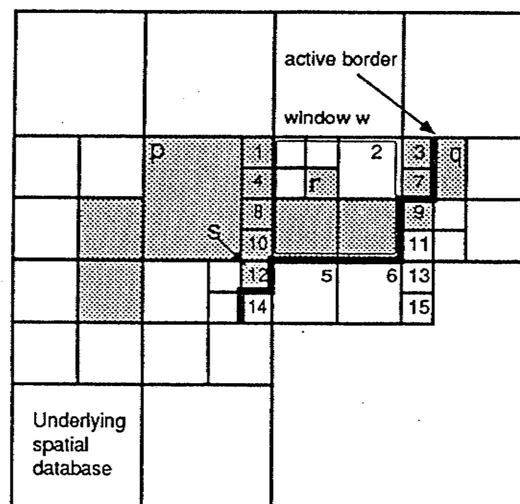
(b)



(c)



(d)



(e)

Figure 4: The active border at (a) the initial stage, (b) after processing window block 1, (c) after processing window block 2, (d) after processing window block 3, and (e) after processing window block 12.

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. It represents each window block by a record of type block containing the x and y coordinate values of its upper-left corner and the length of its side. 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.[4] 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 Section 2 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 3, where five scans are needed to cover the 12×12 window with maximal blocks. The first scan visits blocks 1, 2, and 3; the second scan visits blocks 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 accessing p more than once 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 5. 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 5b). 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.

Figure 6 gives the active border's most general form, i.e., the active border does not contain any holes (see Lemma 1 in Section 3). As a result, when a block of the underlying spatial database, say q, is retrieved, the algorithm checks its size against the corresponding window block, say b. If q's size is larger than that of b, then q has to intersect one of the window's boundaries (see Lemma 1 in Section 3). Figure 7 shows the four possible cases where the block retrieved from the underlying spatial database intersects with 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 7a), then WINDOW_RETRIEVE skips the window blocks in the current row scan that overlap with q. The

[4]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.

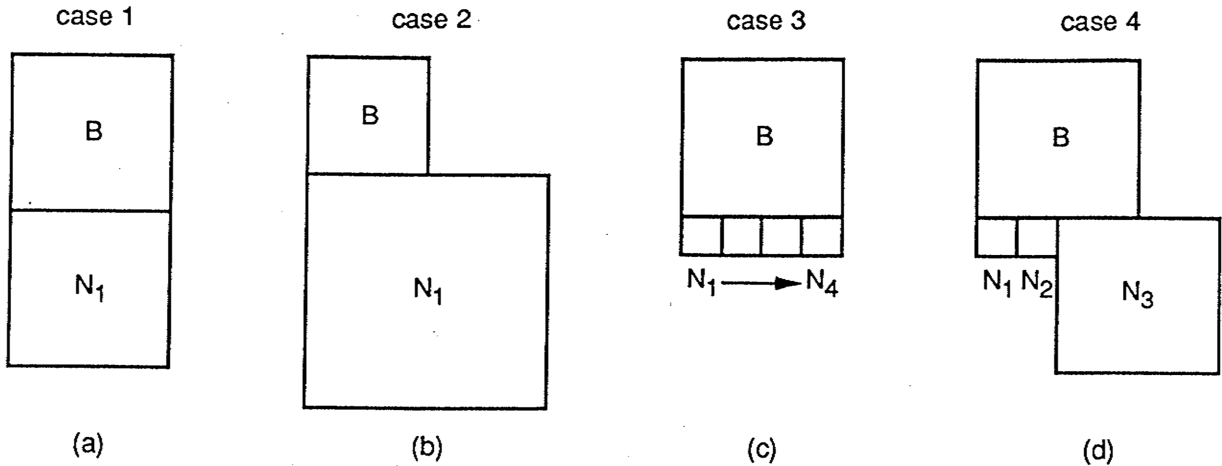


Figure 5: (a), (b), and (c) are examples of possible block/southern-neighbor pairs; (d) cannot occur in a quadtree decomposition.

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 7d), 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, 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 (see Lemma 6 in Section 3). 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 7b and 7c), its overlap with the window creates a pocket-like region that needs to be stored in one of two separate lists, WestList or EastList, respectively. Each time a window block is generated, it has to be checked against the active border in order to make sure that the block is not covered by a previously retrieved block of the underlying spatial 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 height of the window is n and (r, c) corresponds to the x and y coordinate values of its upper-left corner. Figure 8b shows the border represented by each of the two arrays as a result of extracting an 8×12 window from the underlying spatial database in Figure 8a. 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 has to be checked against the active border in order to make sure that the block is not covered by a previously retrieved block of the underlying spatial 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 boundary of

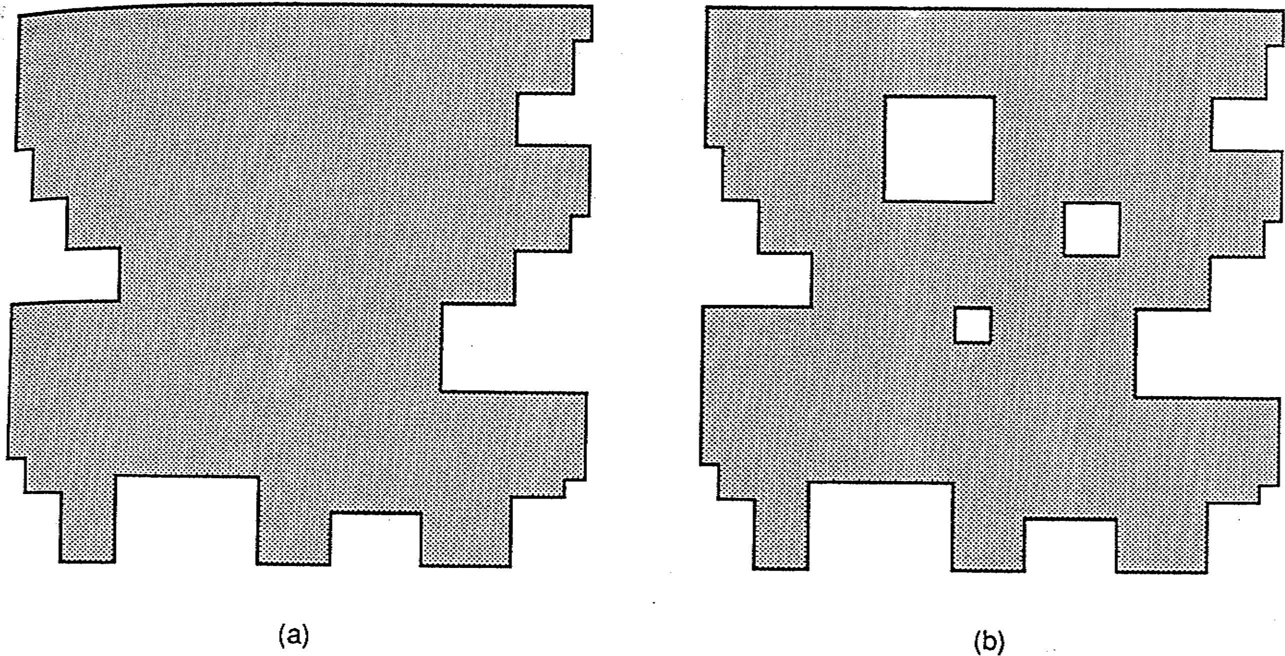


Figure 6: (a) The most general form of an active border. (b) An impossible active border as holes cannot occur.

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.

Observe that `WINDOW_RETRIEVE` always generates maximal neighboring blocks, or at least a bounded number of non-maximal neighboring blocks. An example of this situation arises when processing blocks `A`–`J` in the first row of the 10×10 window in Figure 9. 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` makes sure that 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` in Figure 9, 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.

3 Correctness

Proving that the algorithm is correct involves showing that every block of the underlying database that overlaps with the query window is retrieved and processed by the algorithm. In order to prove this, we can structure our algorithm in the following way. The algorithm consists of two mechanisms: one for generating maximal quadtree blocks inside the window (also termed the *window decomposition algorithm*), and the other for retrieving blocks from the underlying spatial database and maintaining the active border. The active border keeps track of the blocks on the boundary of the window that have already been retrieved. This guarantees that each block in the underlying spatial database is not retrieved more than once. Our strategy for proving that the algorithm is correct is to separate these two mechanisms, show that each one is correct, and then

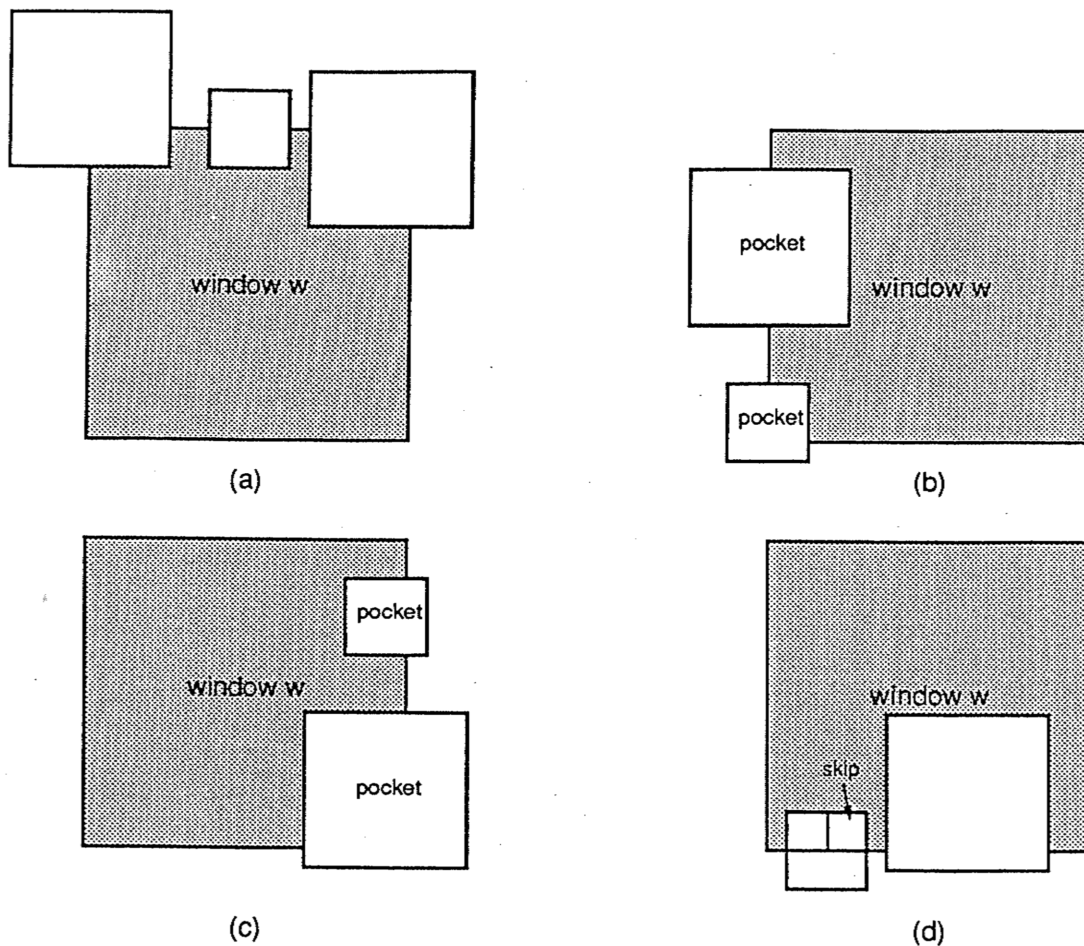


Figure 7: Possible overlaps between blocks of the underlying spatial database and the (a) northern, (b) western, (c) eastern, and (d) southern borders of the window.

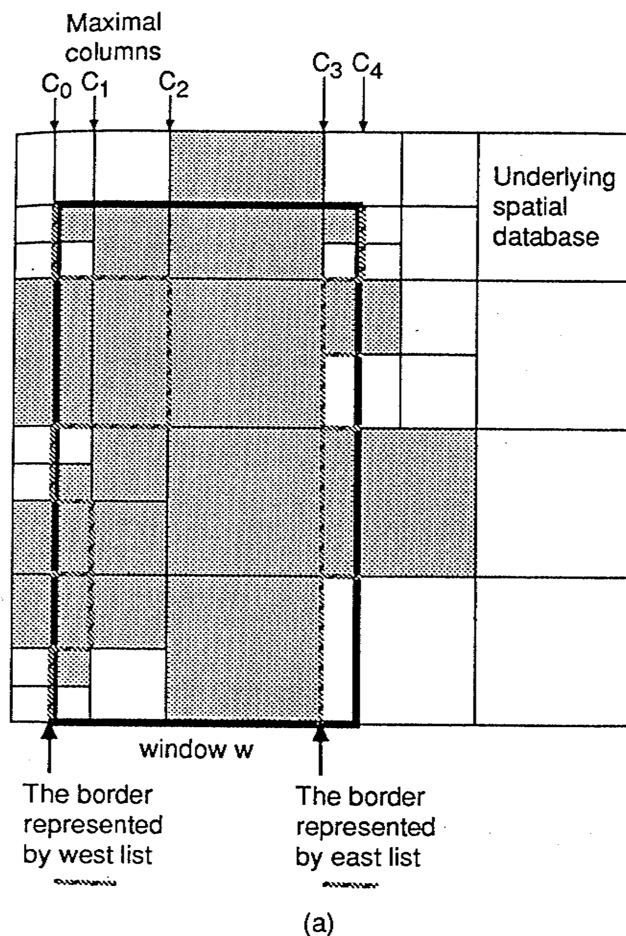
prove that they interact properly.

The algorithm has two cases. The first case arises when all the quadtree blocks of the underlying spatial database that overlap the window are smaller than or equal to the size of the smallest quadtree block in the window. The second case arises when this size criterion is not satisfied.

In the first case the window decomposition algorithm will have to generate all of the maximal quadtree blocks inside the window and none will be skipped—i.e., each one causes a block of the underlying spatial database to be retrieved. In other words, there are no pockets and thus the arrays `WestList` and `EastList` are never updated or accessed. This means that the algorithm reduces to the window decomposition algorithm given in [2].

The window decomposition algorithm is proved correct in [2] and thus we will not address it here. However, we only state that proving that the window decomposition algorithm is correct involves showing that the execution of the algorithm generates a list of maximal blocks that lie entirely inside the window and that cover each point inside the window. In other words, each point inside the window is covered by one maximal block that is generated through the execution of the algorithm. The following two theorems are proved in [2]:

Theorem 1: Each point inside a window is covered by one and only one maximal block generated by the algorithm.



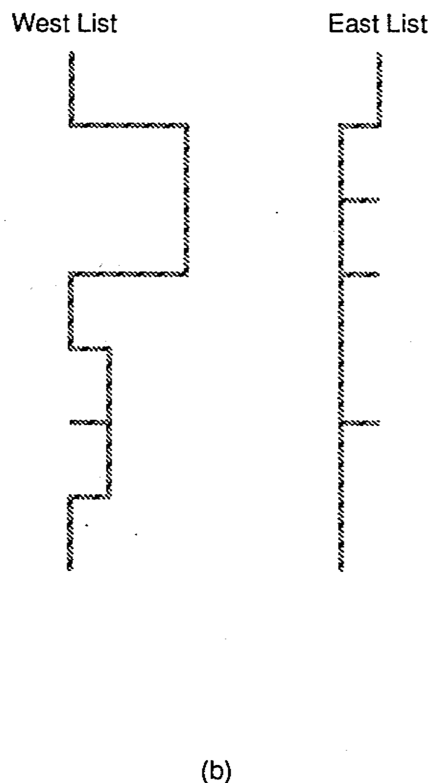


Figure 8: (a) Example of a window (heavy line) and the pockets (heavy lines) along the west and east boundary induced by the underlying spatial database, and (b) the spatial representation of the WestList and EastList data structures corresponding to the active border.

Theorem 2: The window decomposition algorithm generates all the maximal blocks inside the window and only maximal blocks, and hence is correct. □

We now address the second case where some of the blocks in the underlying spatial database are larger than the smallest block in the window—i.e., blocks of the underlying spatial database whose sizes are larger than the overlapping window blocks. We need to show that the interaction and maintenance of the active border with the window decomposition algorithm (1) guarantees that every block of the underlying spatial database that overlaps with the query window is retrieved and processed by the algorithm, and (2) does not interfere negatively with the window decomposition algorithm. In Section 4, we prove that every block of the underlying spatial database that overlaps with the window is retrieved only once.

First, we use the concept of a *maximal zone* [2] to facilitate the presentation of the proofs. Assume a window having (c, r) as the x and y coordinate values of its upper-left corner with height w_h (i.e., in the y direction) and width w_w (i.e., in the x direction). First, let us look at the x direction. Processing along the width w_w, we subdivide the window into p vertical strips with (c_i, r) $(0 \leq i \leq p)$ as coordinate values of their upper-left corners where $c_0 = c$, and $c_i = c_{i-1} + 2^j$

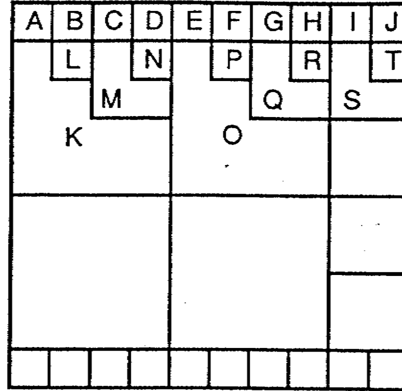


Figure 9: The neighboring blocks to the south of blocks A–J in a 10×10 window. Blocks L, M, N, P, Q, R, and T are non-maximal, while blocks K, O, and S are maximal.

such that $c_{i-1} \bmod 2^j = 0$ and $c_{i-1} \bmod 2^{j+1} \neq 0$ and $c_{i-1} + 2^j \leq c + w_w$. p is defined so that $c_p = c + w_w$. An example of such a decomposition into vertical strips is shown in Figure 10a. The vertical strips are termed *maximal columns*.

We now subdivide the window into horizontal strips in the same way. In particular, we have q horizontal strips with (c, r_i), $(0 \leq i \leq q)$ as the x and y coordinate values of their upper-left corner where $r_0 = r$ and $r_i = r_{i-1} + 2^j$ such that $r_{i-1} \bmod 2^j = 0$ and $r_{i-1} \bmod 2^{j+1} \neq 0$ and $r_{i-1} + 2^j \leq r + w_h$. q is defined so that $r_q = r + w_h$. An example of such a decomposition into horizontal strips is shown in Figure 10b. The horizontal strips are termed *maximal rows*.

Now we define the term *maximal zones* as follows. A maximal zone, say Z_{ij}, is the region between the vertical strips (i.e., maximal columns) having c_i and c_{i+1} as the x-coordinate values of their upper-left corner and the horizontal strips (i.e., maximal rows) having r_j and r_{j+1} as the y-coordinate values of their upper-left corner where $0 \leq i < p$ and $0 \leq j < q$. Figure 10c gives an example of decomposing a window into its maximal zones.

In the interest of brevity, we give below some properties of maximal zones without proving them. They are illustrated in Figure 10d.

Property 1: Each maximal block inside the window is entirely contained in one and only one maximal zone.

Property 2: All the maximal blocks inside a maximal zone are of the same size.

Property 3: A maximal zone contains either one maximal block, or one row of maximal blocks, or one column of maximal blocks.

Property 4: All the southern neighbors of a block lie in one maximal zone.

Property 5: There exists a maximal column, say c_k, inside the window such that
$\forall i : 0 < i < k, c_i - c_{i-1} < c_{i+1} - c_i \wedge \forall i : k < i < p, c_{i+1} - c_i < c_i - c_{i-1}$.

In other words, the sequence of distances between (width of) the maximal columns forms a monotonically increasing sequence followed by a monotonically decreasing sequence. An equivalent property exists for maximal rows.

A useful invariant that holds during the execution of the window decomposition algorithm that also relates to maximal columns is stated below.

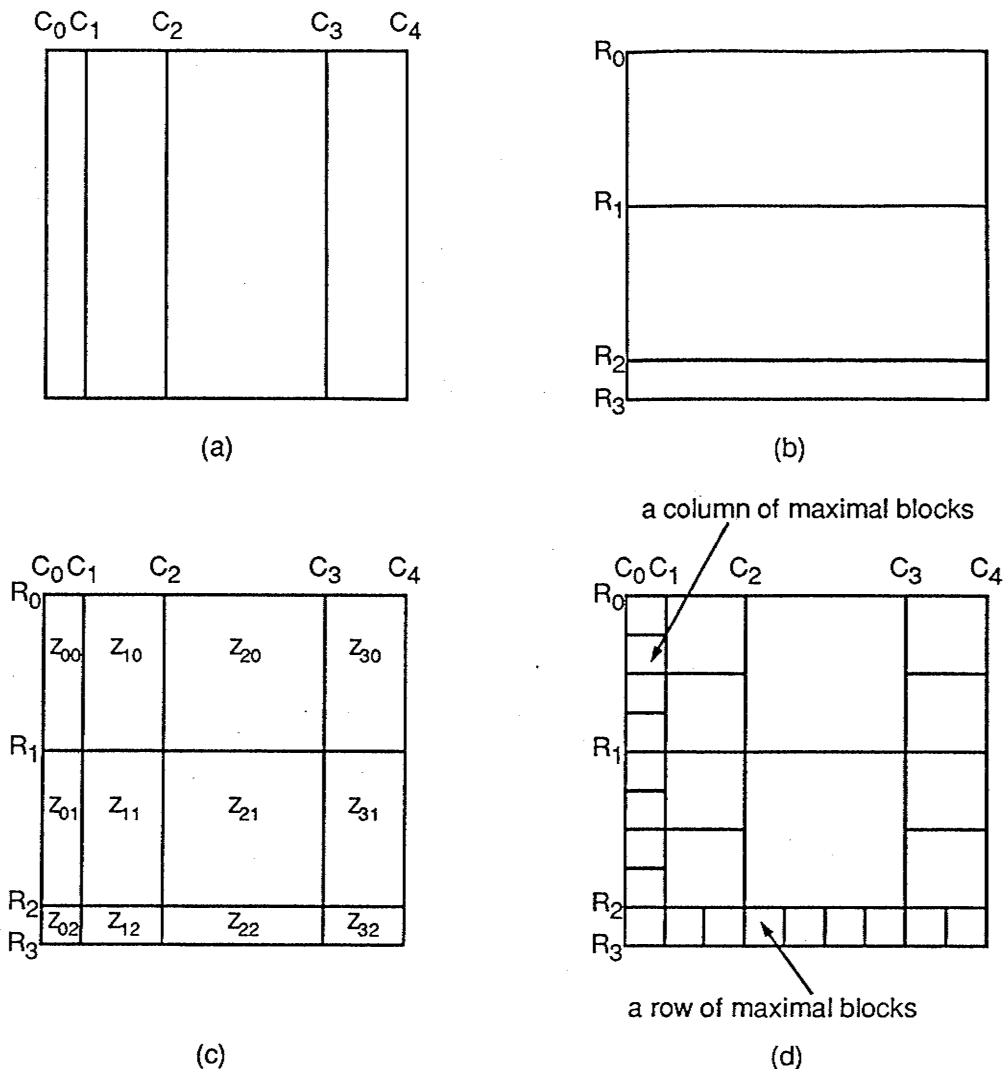


Figure 10: The subdivision of a window into (a) vertical strips, (b) horizontal strips, and (c) maximal zones. (d) The relationship between maximal blocks and maximal zones.

Invariant 1: Each maximal window block and its southern neighbor window blocks that are generated by the window decomposition algorithm always lie inside the same maximal column.

In other words, the blocks inside a maximal column are processed independently of the blocks in other maximal columns inside the window. Put differently, although the algorithm scans the window row-by-row (in the block domain) and generates the maximal neighboring blocks to the south of each block encountered, there is no interaction between blocks of different maximal columns. We make use of this invariant to prove the lemmas below.

Lemma 1: 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 (Figure 7).

Proof by Contradiction: Since b overlaps with q and b is smaller than q, then b is contained in q (by the definition of a quadtree decomposition of space). Assume to the contrary that the database block q lies entirely inside w. If q is of greater size than the window block b that overlaps

with it, then b is not a maximal block since we can use a window block b_1 that contains b and that coincides with q as our new maximal block, which leads to a contradiction. □

As a result, we deal with three categories of blocks of the underlying spatial database that intersect the window boundary: blocks that intersect the north boundary, blocks that intersect the east (west) boundary, and blocks that intersect the south boundary. Notice that the algorithm treats blocks that intersect both the west (east) and the south boundaries of the window as if they just intersect the west (east) boundary. On the other hand, it treats blocks that intersect both the north and west (east) boundaries of the window as if they just intersect the north boundary. Blocks intersecting the east or west boundary of the window receive the same type of processing and hence are considered as one group. We prove the correctness of the interaction of each category separately.

Lemma 1 means that the active border does not contain any holes (see Figure 6) since the query window is scanned row-by-row, and large-sized blocks of the underlying spatial database intersect only the window boundary. Therefore, storing only the outer boundary of the active border is enough.

Lemma 2a: If a block of the underlying spatial database, say q, intersects the west (east) window boundary, then the east (west) boundary of q that lies inside the window must coincide with a boundary of one of the maximal columns of the window.

Proof: We prove the lemma for the case when q intersects the west boundary of the window. The other case is similar. Assume the lemma does not hold—i.e., that q intersects the window boundary but that the eastern boundary of q that lies inside the window does not coincide with a maximal column of the window. Therefore, one of two possible cases must occur. These are illustrated in Figure 11. Both of the cases cannot happen since, by the definition of a quadtree decomposition, blocks cannot overlap in this manner. □

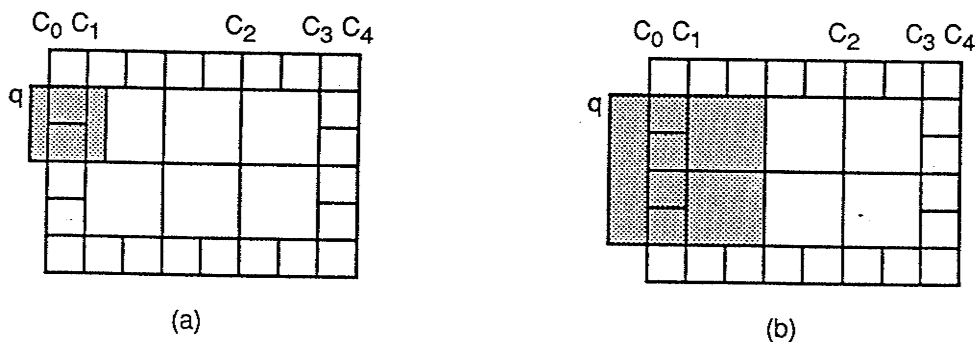


Figure 11: Examples of impossible block configurations in which the boundary of block q in the underlying spatial database does not coincide with a boundary of (a) a maximal block in the window, or (b) a maximal column.

An analogous lemma can be stated for blocks intersecting the north or south boundary of the window.

Lemma 2b: If a block of the underlying spatial database, say q, intersects the north (south) window boundary, then the south (north) boundary of q that lies inside the window must coincide with a boundary of one of the maximal rows of the window.

Lemma 3: If a block of the underlying spatial database, say q, intersects the west (east) window boundary, then the part, if any, of the south boundary of q, say s, that lies inside the window must

coincide with the north boundary of a maximal block inside the window.

Proof: Assume that q intersects the west boundary of the window. From Lemma 2a, q's east boundary coincides with a boundary of a maximal column of the window, say c. However, other maximal columns to the west of c may intersect q as well (for example, in Figure 8, maximal column C_1 intersects block p of the underlying spatial database). If q intersects with no maximal columns other than c, then only two cases are possible (as illustrated in Figures 12a and 12b). Figure 12a cannot occur in a quadtree decomposition, while Figure 12b satisfies the Lemma. If q intersects with one or more maximal columns other than c, then s must coincide with a maximal row inside the window (Figure 12c) as the other case cannot exist in a quadtree decomposition (Figure 12d). Since a maximal row coincides with the north boundary of maximal blocks across the whole window, this applies to q as well. □

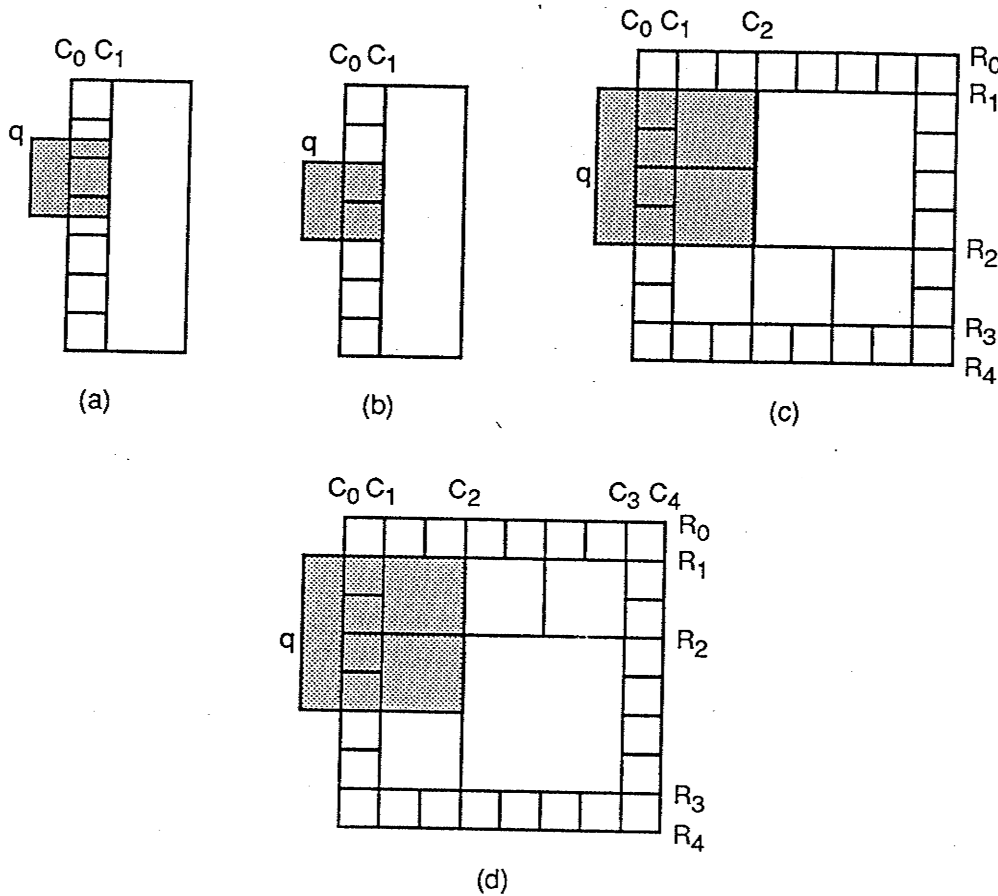


Figure 12: Examples of possible ((b) and (c)) and impossible ((a) and (d)) block configurations involving blocks from the underlying spatial database and the window.

Lemma 4: If a block of the underlying spatial database, say q, intersects the west (east) window boundary, then the window decomposition strategy will only skip the window blocks covered by q while maintaining normal processing otherwise. In other words, updating the active border with q does not adversely affect the mechanism used for window decomposition.

Proof: Assume that q intersects the west border of the window. By Lemma 2a, the east boundary of q coincides with a maximal column of the window. Therefore, the window decomposition mechanism will function properly to the east of q since, by Invariant 1, the block generation process works

independently inside each maximal column. The portion of the south boundary of q, say s, that lies inside the window, is used by the algorithm to generate the new window blocks to the south of q. However, from Lemma 3, all parts of s coincide with the north boundary of a maximal block inside the window. Therefore, by applying a maximal block computation at s, the algorithm would still generate maximal blocks of the window to the south of q after skipping the ones inside q (and hence avoid retrieving q more than once by the overlapping window blocks). If the south boundary of q lies outside the window, then the Lemma holds since no further processing to the south of q is needed. In addition, by Invariant 1, the window decomposition process to the east of q is not affected by q since the east boundary of q coincides with a maximal column. □

We now study the case where a block of the underlying spatial database intersects the south boundary of the window. We make use of the following lemma. Its proof is given in [2] (where it is Lemma 4).

Lemma 5: All the maximal blocks arranged in a row inside a maximal zone are processed in the same iteration of the main loop of procedure `WINDOW_RETRIEVE`. □

Lemma 6: If a block of the underlying spatial database, say q, intersects only the south boundary of the window, then q lies entirely inside one maximal column of the window.

Proof: By Property 5, if q overlaps with more than one maximal column of the window, then either the size of q is not a power of two (a contradiction) or q must intersect with the east or west boundary of the window (a contradiction). Therefore, q lies inside one maximal column. □

Combining Lemmas 5 and 6, we get the following result:

Lemma 7: If a block of the underlying spatial database, say q, intersects only the south boundary of the window, then the window decomposition strategy will only skip the window blocks covered by q, while maintaining normal processing otherwise.

Proof: By Lemma 6, q lies inside only one maximal column of the window. By Lemma 5, if one maximal window block, say b, results in retrieving q, then the rest of the window blocks in the maximal zone that lie in the same row as b, will exist in the same iteration of the main loop of procedure `WINDOW_RETRIEVE`. Therefore, all of them can be automatically skipped by the algorithm once q is retrieved, and hence no additional data structure is needed to record q's retrieval. Since the south boundary of q is already outside the window, no further processing is needed to the south of q. The effect of this is that it results in skipping all the window blocks that overlap with q and that lie to the south of b up to the south boundary of the window. Also, by Invariant 1, q lies inside only one maximal column and hence does not affect other portions of the window decomposition mechanism. □

Lemma 8: If a block of the underlying spatial database, say q, intersects the north boundary of the window, then the window decomposition strategy will only skip the window blocks covered by q while maintaining normal processing otherwise.

Proof: Since q intersects the north boundary of the window, q will be retrieved when the algorithm scans the first row in the window. In addition, q will be retrieved by the leftmost maximal window block, say b, that overlaps with q since scanning is from left to right. Therefore, all the window blocks to the right of b and that overlap with q are automatically skipped by the algorithm since all of them immediately follow b in `TopList`, the list of blocks to be processed. Processing of the algorithm resumes at the first window block to the right of q in the current row scan. By Lemma 2b, the part of q's south border, say s, that lies inside the window will coincide with a maximal row of the window. Since a maximal row coincides with the north boundary of maximal blocks across the whole window, this applies to s as well. Therefore, using s to generate maximal blocks to the south of q will resume regular processing of the decomposition algorithm as it results in generating

legitimate maximal blocks of the window after skipping the window blocks that overlap with q. Therefore, q is retrieved just once by the algorithm without affecting the normal processing of the algorithm.

Combining Theorems 1 and 2 and Lemmas 4, 7, and 8 we get the following theorem:

Theorem 3: Every block of the underlying spatial database that overlaps with the query window is retrieved by procedure WINDOW_RETRIEVE and hence the algorithm is correct.

Proof: By Theorem 1, maximal blocks of the window cover every point inside the window (without overlap). Therefore, if blocks of the underlying spatial database are smaller than the window blocks, then, by Theorem 2, the window decomposition algorithm will generate all the maximal blocks inside the window, and hence all the blocks of the underlying spatial database overlapping with the window blocks are retrieved. If some of the blocks of the underlying database, say D, are larger than the corresponding maximal window blocks, then by Lemma 1, each block, say q in D, has to intersect with some of the window boundaries. By Lemmas 4, 7, and 8, the algorithm will skip all but one of the maximal blocks of the window that overlap with q (this is because when one of the maximal blocks has to retrieve q, then the rest of the overlapping window blocks are skipped). Lemmas 4, 7, and 8 also show that the normal window decomposition mechanism is resumed after processing each block of the underlying spatial database that overlaps the window. □

4 Complexity Analysis

Analyzing the complexity of our algorithm is a bit complex as there are two processes going on, and hence two ways of measuring it. The first is in terms of the blocks of the underlying spatial database that are retrieved, while the second is in terms of the maximal blocks in the window (i.e., the window decomposition mechanism). The first is the most important as block retrieval is intimately tied to disk I/O operations.

The process of generating the maximal blocks in the window is a subset of the process used in the window decomposition algorithm reported in [2]. We characterize it as being a *subset* because all of the maximal blocks are only generated in the worst case (i.e., when none of the blocks in the underlying spatial database intersect the border of the window). Its complexity is obtained in the same way. It is known that in the worst case, the number of maximal quadtree blocks inside a square window of size $n \times n$ is $N = 3(2n - \log n) - 5$ [4, 6, 11]. Use of the active border requires additional $O(n)$ space.

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. As in [2], we use a binary search through this set of blocks to determine the maximal block inside the window. Thus T_m is $O(\log \log T)$.

To compute T_w, we need to show that each maximal block inside the window is generated once, and that only a limited number of non-maximal blocks are generated. We say that the work required to generate blocks that are not maximal with respect to a particular window is *wasted*. Such blocks are ignored (i.e., bypassed) in subsequent processing. For example, the work in generating the southern neighbors of blocks B, C, D, F, G, H, and J (i.e., L, M, N, P, Q, R, and T, respectively) in Figure 9 is wasted. This is formulated and proved in the following two Lemmas.

Lemma 9: Each maximal block inside window w is generated at most once.

Proof: This proof is the same as Lemma 7 in [2]. In Theorem 2, we proved that every maximal block inside window w is generated by the algorithm. To show that it is generated only once we observe that each window block processed by the algorithm generates only its southern neighbors. The facts that non-maximal window blocks are bypassed by the algorithm, and that maximal blocks do not overlap, mean that each maximal window block, say B, is generated as the southern neighbor of only one other maximal window block, say C. Note that this worst case only arises if `WINDOW_RETRIEVE` generates all of the maximal blocks (i.e., none are skipped). □

Lemma 10: Each window block visited by the algorithm can waste at most $O(\log\log T)$ work in generating intermediate non-maximal window blocks.

Proof: This proof is the same as Lemma 8 in [2]. Assume that window block B generates wasted work. We show that this work takes $O(\log\log T)$ time. B can generate neighboring southern maximal blocks that are either smaller or larger. When the size of the neighboring block is greater than or equal to the size of B, then the algorithm takes $O(\log\log T)$ time regardless of whether or not it is wasted and the Lemma holds. When more than one southern neighboring block is generated (this number can be of the same order as the size of B), we need to show that all the generated southern blocks are maximal, and cannot be bypassed, i.e, they are not wasted work. We shall prove this by contradiction. Assume that B generates more than one southern neighboring block and that all of them are bypassed (i.e., not visited) in subsequent processing. It should be clear that due to the nature of the quadtree decomposition of space, either all of them are visited, or all are bypassed. Our assumption means that there exists a block C whose width is greater than the total width of B's southern neighbors. Let (B_x,B_y) and (C_x,C_y) be the locations of the upper-leftmost pixels of blocks B and C, respectively. Also, let B_s and C_s be the widths of blocks B and C, respectively. It is easy to see that the fact that B and C are maximal blocks that are southern neighbors of other visited maximal blocks means that $C_y = B_y + B_s$. The fact that $C_s > B_s$ means that the lower-rightmost pixel of C is at $(C_x + C_s - 1, C_y + C_s - 1)$ which is in the window. Therefore, $(B_x + B_s - 1, B_y + B_s - 1)$ which is the lower-rightmost pixel of B's southern neighbor of equal size, say D, is also in the window. This means that D is B's neighboring southern maximal block. However, this contradicts the existence of more than one such block. Thus the assumption that all of the southern neighboring blocks of B are bypassed is invalid. Therefore, no work is wasted in generating B's southern neighbors in this case, and the Lemma holds. □

Combining the results for T_m and T_w, and Theorem 3 means that we have proven the following theorem.

Theorem 4: Given an $n \times n$ window in a $T \times T$ image, the worst-case execution time for the algorithm is $O(n \log\log T) + M$ where M is the number of quadtree blocks in the underlying spatial database that overlap the window. □

Theorem 5: Every block of the underlying spatial database that overlaps the query window is retrieved once, and only once, by `WINDOW_RETRIEVE`.

Proof: By Lemma 9, each maximal block is generated at most once. Let q be a block in the underlying spatial database and suppose that q overlaps the window. If q lies inside the window and is of equal or smaller size than the overlapping window block, say b, then q will be retrieved once by the algorithm when b is generated, and hence the theorem holds (notice that maximal blocks do not overlap). If q overlaps the window and if q contains more than one window block, then q will be retrieved by the first window block, say b, that encounters q. However, from that point onwards, all the window blocks that overlap q will be skipped and block q will not be retrieved again. By Lemma 1, q has to intersect one of the window boundaries. If q intersects the east or west boundaries of the window, then by Lemma 4 the active border (i.e., `WestList` and `EastList`) prevents block q

from being retrieved again by the remaining window blocks that overlap q. Otherwise, if q intersects the north or south boundaries of the window, then by Lemmas 7 and 8 the algorithm skips the remaining window blocks that overlap q. Therefore q will be retrieved once, and only once. Hence the theorem also holds when q is of larger size than the overlapping window blocks. □

Note that there is an onto relation between the set of blocks of the underlying spatial database that are retrieved by the algorithm and the set of maximal window blocks generated by the algorithm. This relation is only onto, rather than one-to-one onto, because a window block, say b, may overlap more than one block in the underlying spatial database (i.e., the overlapped blocks are smaller than b), in which case several blocks in the underlying spatial database will be retrieved. However, they will only be retrieved once.

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` (which is based on the window decomposition algorithm [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 US provided by the Bureau of the Census. A sample map containing line segments is given in Figure 14. The maps are represented using the PMR-quadtree [9], a variant of a quadtree for storing vector data.

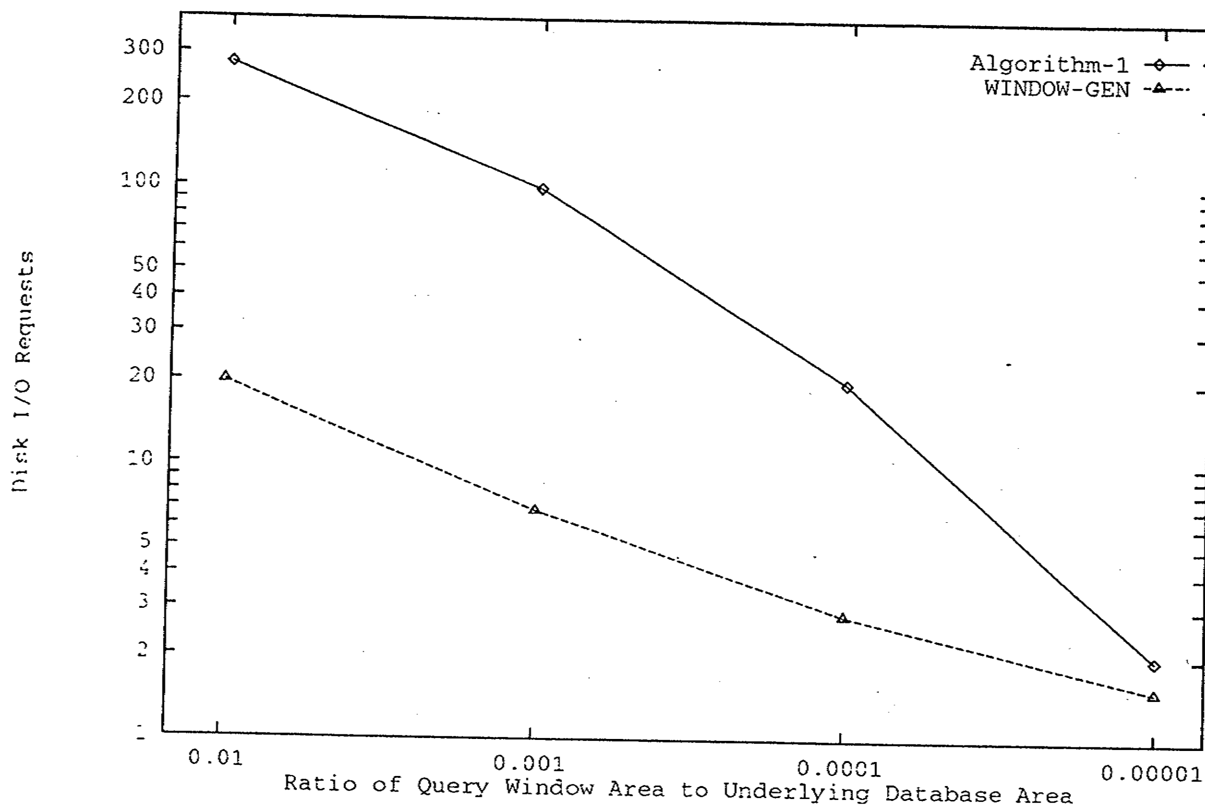


Figure 13: Results of an experiment to compare the disk I/O performance of the algorithm presented here (i.e., `WINDOW_RETRIEVE`) and the algorithm presented in [2] (i.e., `Algorithm-1`).

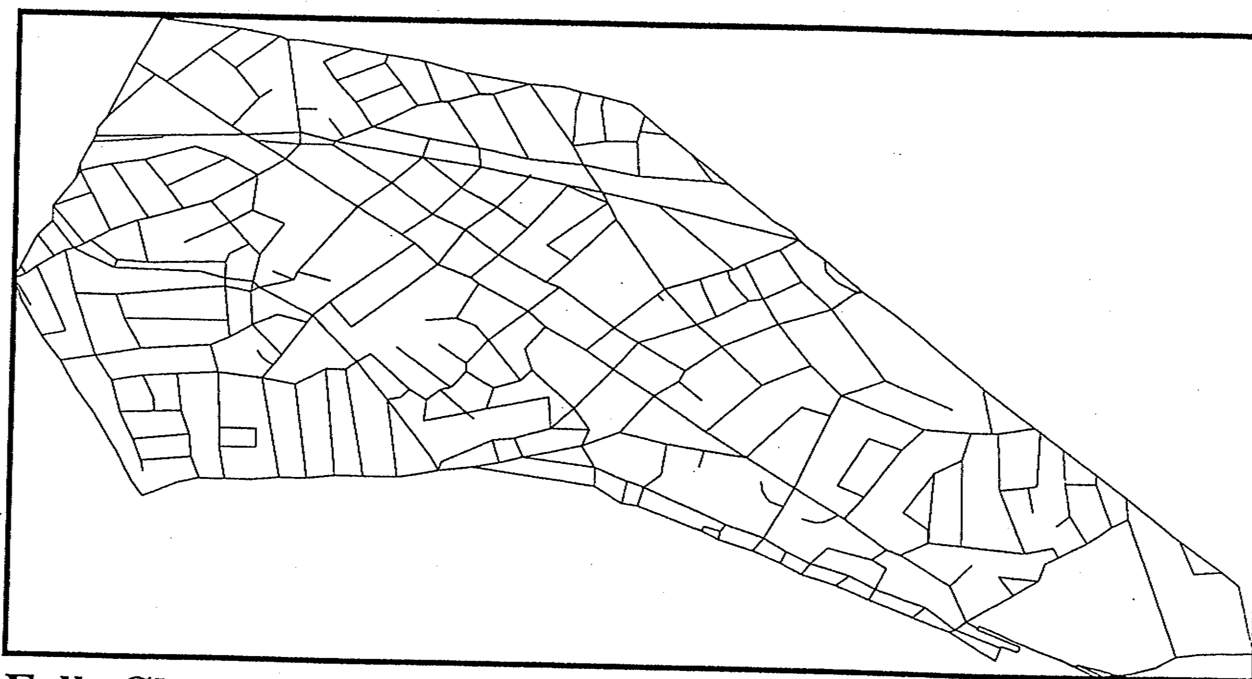


Figure 14: A sample data set: A road network in Falls Church, Virginia.

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-base spatial database environment 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 underlying spatial database is only retrieved once. The algorithm is proven (analytically and experimentally) to have an improvement in disk I/O performance over another algorithm [2] that is based on the decomposition of a window into its maximal quadtree blocks and then retrieving the covering blocks in the underlying spatial database. The window decomposition parts of the two algorithms have the same worst-case execution time complexity, although the algorithm of this paper does require more space (i.e., on the order of the height of the window). This extra space is relatively small.

References

[1] W. G. Aref and H. Samet. Efficient processing of window queries in the pyramid data structure. In *Proceedings of the 9th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 265–272, Nashville, TN, April 1990.

[2] W. G. Aref and H. Samet. Decomposing a window into maximal quadtree blocks. *Information Processing Letters*, 1992. To appear. Also available as Technical Report CS-TR-2771, University of Maryland, College Park, MD, October 1991.

[3] W. G. Aref and H. Samet. Uniquely reporting spatial objects: yet another operation for comparing spatial data structures. In *Proceedings of the 5th International Symposium on Spatial Data Handling*, Charleston, SC, August 1992.

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

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

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

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

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

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

[10] H. Samet and M. Tamminen. Computing geometric properties of images represented by linear quadtrees. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 7(2):229–240, March 1985. (Also available as Technical Report CS-TR-1359, University of Maryland, College Park, MD, December 1983.)

[11] 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.

7 Appendix: Code for the Retrieval Algorithm

```
procedure WINDOW_RETRIEVE(S,W,F);
/* Retrieve the quadtree blocks of the underlying spatial database that overlap
   window W. The window is represented by a record of type window with four fields ROW,
   COL, WIDTH, and HEIGHT corresponding to the y coordinate value of its upper-leftmost
   pixel, the x coordinate value of its upper-leftmost pixel, its width, and its
   height, respectively. The origin is at the upper-left corner of the image and the
   positive x and y directions are to the right and down, respectively. In order to
   perform the window retrieval query, WINDOW_RETRIEVE generates the maximal blocks
   that comprise window W. Then, for each maximal window block, window query function
   F is applied which accesses the underlying spatial database, retrieves the corresponding
```

```
   block of the underlying spatial database, say DbBlock, that overlaps with the maximal
   window block, and applies the desired window operation to it. DbBlock is also
   returned to WINDOW_RETRIEVE to be used in updating the active border. The active
   border is maintained in order to avoid accessing a block of the underlying spatial
   database more than once. Arrays WestList and EastList, which represent the east and
   west borders of the active border, are of size equal to the window height and are
   of type array of pocket. Pocket is a record of two fields: LEN and COL, denoting
   the size of the intersecting block of the underlying spatial database, and the x
   coordinate value of its east or west boundaries, depending on whether it is stored
   in WestList or EastList, respectively. The quadtree blocks inside the query window
   are determined by repeatedly finding southern neighbors and keeping them in a linked
   list whose first and last elements are pointed at by NextTopList and EndNextTopList,
   respectively. All blocks are represented by records of type block with three fields
   ROW, COL, and LEN corresponding to the y coordinate value of its upper-leftmost
   pixel, the x coordinate value of its upper-leftmost pixel, and its length, respectively.
   The block currently being processed is pointed at by TopList.  Initially, the southern
   neighboring blocks of a block of length WIDTH(W) with an upper-leftmost pixel at
   (COL(W),ROW(W)-WIDTH(W)) are generated. */
begin
  reference spatial database S;
  value pointer window W;
  pointer block function F();
  pointer list TopList,NextTopList,EndNextTopList;
  pointer block Current,DbBlock;
  pocket array WestList[ROW(W):ROW(W)+HEIGHT(W)-1];
  pocket array EastList[ROW(W):ROW(W)+HEIGHT(W)-1];
  integer I;

  for I := ROW(W) step 1 until ROW(W)+HEIGHT(W)-1 do
    begin
      COL(WestList[I]) := COL(W);
      COL(EastList[I]) := COL(W) + WIDTH(W);
      LEN(WestList[I]) := 0;
      LEN(EastList[I]) := 0;
    end;
  Current:=create(block); /* Initially we find the southern maximal neighbors of a block
                              as wide as the entire window. */
  ROW(Current):=ROW(W)-WIDTH(W);
  COL(Current):=COL(W);
  LEN(Current):=WIDTH(W);
  NextTopList:=NIL;
  GEN_SOUTHERN_MAXIMAL(NextTopList,Current,W,EndNextTopList,WestList,EastList);
  do
    begin
      TopList:=NextTopList;
      NextTopList:=EndNextTopList:=NIL;
      while not(null(TopList)) do
        begin
          Current:=DATA(TopList);
          TopList:=NEXT(TopList);
          while not(null(TopList)) and CONTAINED(DATA(TopList),Current) do
            TopList:=NEXT(TopList); /* Skip non-maximal blocks inside current */
          /* If Current is already covered by the West or East boundaries of the
             border, then skip it. */
          if((COL(Current)>=COL(WestList[ROW(Current)]) and
             (COL(Current)<COL(EastList[ROW(Current)])) then
            begin /* Current is a maximal window block */
              DbBlock:=F(Current,S);
              if(LEN(DbBlock)>LEN(Current))
```

```
            begin
              if(ROW(DbBlock)<ROW(W)) then
                begin /* Database block intersects the window's top boundary */
                  ROW(Current):=MIN(ROW(W)+HEIGHT(W),ROW(DbBlock)+LEN(DbBlock));
                  LEN(Current):=MIN(COL(DbBlock)+LEN(DbBlock), COL(W)+WIDTH(W))
                                -MAX(COL(DbBlock),COL(W));
                  while not(null(TopList)) and CONTAINED(DATA(TopList),DbBlock) do
                    TopList:=NEXT(TopList);
                end
              else if(COL(DbBlock)<COL(W)) then
                begin /* Database block intersects the window's west boundary */
                  COL(WestList[ROW(Current)]):=COL(DbBlock)+LEN(DbBlock);
                  LEN(WestList[ROW(Current)]):=LEN(DbBlock);
                  ROW(Current):=MIN(ROW(DbBlock)+LEN(DbBlock),ROW(W)+HEIGHT(W));
                end
              else if(COL(DbBlock)+LEN(DbBlock)>COL(W)+WIDTH(W)) then
                begin /* Database block intersects the window's east boundary */
                  COL(EastList[ROW(Current)]):=COL(DbBlock);
                  LEN(EastList[ROW(Current)]):=LEN(DbBlock);
                  ROW(Current):=MIN(ROW(DbBlock)+LEN(DbBlock),ROW(W)+HEIGHT(W));
                end
              else if(ROW(DbBlock)+LEN(DbBlock)>ROW(W)+HEIGHT(W)) then
                begin /* Database block intersects the window's bottom boundary */
                  ROW(Current):=ROW(DbBlock)+LEN(DbBlock);
                  while not(null(TopList)) and CONTAINED(DATA(TopList),DbBlock) do
                    TopList:=NEXT(TopList);
                end
            end
        end
      GEN_SOUTHERN_MAXIMAL(NextTopList,Current,W,EndNextTopList,WestList,EastList);
    end;
  end
  until null(NextTopList);
end;

procedure GEN_SOUTHERN_MAXIMAL(NextTopList,B,W,EndNextTopList,WestList,EastList);
/* Find the maximal blocks to the south of block B in window W and add them to the end
   of the list which starts at NextTopList and ends at EndNextTopList.  If NextTopList
   is NIL, then set it to the first block that is added. WestList and EastList are
   used to avoid the generation of any blocks to the south of B that overlap with a
   block of the underlying spatial database that has already been retrieved. */
begin
  reference pointer list NextTopList,EndNextTopList;
  reference pointer block B;
  value pointer window W;
  reference pointer array WestList, EastList;
  pointer block T;
  integer LEFT,RIGHT;
  while(COL(B)<COL(WestList[ROW(B)]) do /* Check for a west pocket */
    ROW(B):=ROW(B)+LEN(WestList[ROW(B)]);
  while(COL(B)>=COL(EastList[ROW(B)]) do /* Check for an east pocket */
    ROW(B):=ROW(B)+LEN(EastList[ROW(B)]);
  T:=MAX_BLOCK(ROW(B)+LEN(B),COL(B),W); /* Allocate first block. */
  if null(T) then return
  else
    begin /* Allocate first block and initialize start of NextTopList. */
      if null(NextTopList) then NextTopList:=EndNextTopList:=create(list)
      else EndNextTopList:=NEXT(EndNextTopList):=create(list);
      DATA(EndNextTopList):=T;
```

```
        LEFT:=COL(B)+LEN(T);
        RIGHT:=COL(B)+LEN(B);
        while LEFT < RIGHT do /* Generate rest of blocks. */
          begin
            EndNextTopList:=NEXT(EndNextTopList):=create(list);
            DATA(EndNextTopList):=MAX_BLOCK(ROW(B)+LEN(B),LEFT,W);
            LEFT:=LEFT+LEN(DATA(EndNextTopList));
          end;
        NEXT(EndNextTopList):=NIL; /* Set pointer at the end of the list to NIL. */
      end;
end;

pointer block procedure MAX_BLOCK(ROW,COL,W);
/* Find the largest square block inside window W for which (ROW,COL) is the first
   (upper-leftmost) pixel. The length of the side of the block is a power of 2. */
begin
  value integer ROW,COL;
  value pointer window W;
  integer I;
  pointer block B;
  I:=0;
  while IN_WINDOW(ROW+2**I-1,COL+2**I-1,W) and ((ROW mod 2**I)=0)
        and ((COL mod 2**I)=0)
    do I:=I+1;
  if I=0 then return(NIL) /* No maximal block exists. */
  else
    begin
      B:=create(block);
      ROW(B):=ROW;
      COL(B):=COL;
      LEN(B):=2**(I-1);
      return(B);
    end;
end;
```