The International Conference in Knowledge Management (CIKM'94), Gaithersburg, MD, November 1994.

Hashing by Proximity to Process Duplicates in Spatial Databases

Walid G. Aref
Matsushita Information Technology Laboratory
Two Research Way
Princeton, New Jersey 08540
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
hjs@cs.umd.edu

Abstract

In a spatial database, an object may extend arbitrarily in space. As a result, many spatial data structures (e.g., the quadtree, the cell tree, the R^+-tree) represent an object by partitioning it into multiple, yet simple, pieces, each of which is stored separately inside the data structure. Many operations on these data structures are likely to produce duplicate results because of the multiplicity of object pieces. A novel approach for duplicate processing based on proximity of spatial objects is presented. This is different from conventional duplicate elimination in database systems because, with spatial databases, different pieces of the same object can span multiple buckets of the underlying data structure. Example algorithms are presented to perform duplicate processing using proximity for a quadtree representation of line segments and arbitrary rectangles. The complexity of the algorithms is seen to depend on a geometric classification of different instances of the spatial objects. By using proximity and the spatial properties of the objects, the number of disk-I/O requests as well as the run-time storage during duplicate processing can be reduced.

1 Introduction

Spatial databases are usually organized in data structures that provide efficient access and flexible manipulation of data. There are several ways of representing a spatial object inside a data structure. Some data structures (e.g., the R-tree [9] and the Grid File [12]) represent a spatial object by just one entity inside the data structure (e.g., by the object's bounding rectangle in the case of the R-tree and by a point in a higher dimensional space in the case of the Grid File). On the other hand, another family of data structures (e.g., the quadtree [10], cell tree [8], and R^+-tree [6]) represent a spatial object by partitioning it into more than one piece, each of which is stored separately inside the data structure. In this paper, we focus on the latter family of data structures. We attempt to take advantage of the spatial characteristics of objects as a guide to duplicate processing (see [5] for an overview of techniques for duplicate processing in database systems) that result from spatial database operations. It is worth mentioning that duplicate processing in spatial databases is slightly different from conventional duplicate elimination in database systems because with spatial databases, different pieces of the same object can span multiple buckets of the underlying data structure. For example, in the case of the R^+-tree and the quadtree, duplicates arise because the object is partitioned into pieces. We can benefit from knowing that these pieces are contiguous in space (if this is really the case). In particular, we define an operation, termed *Report_Unique*, which manages to process duplicates and report each object in the data structure just once, regardless of the multiplicity of the partitions of the object inside the spatial data structure.

Report_Unique can be viewed as an alternative approach to hashing (see [7] for a good coverage of hashing techniques) when dealing with spatial data. Report_Unique maintains a dynamic data structure, termed the *active border* [15], that serves as a repository for the objects currently being processed, termed *active objects*. The active border data structure resembles a dynamic hash table. When conventional hashing techniques are used there is no way of predicting when an object will not be referenced again and hence can be safely deleted from the hash table. Using spatial properties of the underlying objects, the active border can grow and shrink in constant time. By knowing the extent and proximity of the objects, we can detect when all the pieces comprising the object have been processed and hence can delete the object entry from the active border. This way we can avoid the situation that the hash table grows until it reaches its maximum limit, i.e., O(number of objects in the database). In fact, by using the active border technique we can limit the storage complexity to be in the order of the active objects only. A look-up function, also based on spatial proximity, can be used to access entries in the active border. The look-up function is the spatial analog of a hash function. As we demonstrate in the paper, an important advantage of proximity-based look-up functions is that as a result of their use, buckets in the active border are guaranteed not to overflow. In addition, we also show how we can utilize the proximity information to reduce the number of disk-I/O requests during duplicate processing.

The rest of the paper is organized as follows. In Section 2 we describe the data structures that we use for storing the underlying spatial database. The remaining sections describe techniques to process duplicates using spatial characteristics of the underlying spatial objects. Section 3 describes an algorithm that uses spatial proximity to eliminate the space requirements that result from using other duplicate elimination techniques (e.g., hash tables). Section 4 demonstrates that supporting the Report_Unique operation for a realistic variety of objects requires classifying spatial objects into categories of different complexity. Section 5 shows how these classes affect the performance and correctness of the alternative approaches to Report_Unique. Section 6 contains some concluding remarks.

2 The Underlying Spatial Data Structures

We assume that the spatial objects are stored in data structures that partition the objects into multiple pieces. We use the quadtree as an example to demonstrate our algorithms for duplicate processing. The quadtree partitions each object into multiple non-overlapping pieces. Here, we show how we organize databases of line segments and rectangles using variants of the quadtree. For line segments we use the *PMR quadtree* [11], while for rectangles we use a variant of the region quadtree [10], termed the *rectangle quadtree*.

In the PMR quadtree, a line segment is divided into several pieces, where each piece is associated with the quadtree block that it intersects. Figure 1 shows one example of the PMR quadtree. A block in the PMR quadtree is permitted

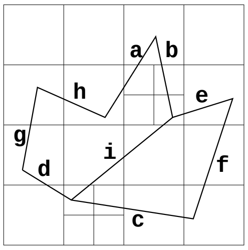

Figure 1: An example of a PMR quadtree.

to contain a variable number of line segments. It is constructed by inserting the line segments one-by-one into an initially empty structure consisting of one block. Each line segment is inserted into all of the blocks that it intersects or occupies in its entirety. During this process, the occupancy of each affected block is checked to see if the insertion caused it to exceed a predetermined splitting threshold. If the splitting threshold is exceeded, then the block is split once, and only once, into four blocks of equal size.

Each block in the PMR quadtree stores the object identifiers of the lines passing through it. The full description of the line segments (e.g., the start and end coordinate values of the end-points) is stored in what is called the *feature table*. The index of line l in the feature table is l's object identifier. Notice that we cannot simply traverse the feature table and report the lines that we find since not all the lines in the feature table have to belong to the data structure in question, i.e., that the feature table may be shared by more than one PMR quadtree.

Rectangles are stored in a variant of the region quadtree [10] which we term a *rectangle quadtree*. Each rectangle is decomposed into the quadtree blocks that lie inside the rectangle. For example, Figure 2a shows the decomposition of rectangle `r` into its constituent quadtree blocks. We assume that the rectangles need not be disjoint. We also adopt the restriction that each quadtree block is completely inside all of the rectangles that overlap it. In other words, the case

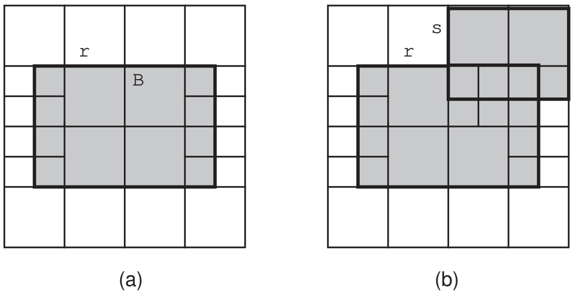

Figure 2: (a) The decomposition of rectangle `r` into its constituent quadtree blocks. (b) Block `B` is decomposed after rectangle `s` is inserted.

that only part of a quadtree block lies in a rectangle is not allowed. For example, this restriction means that block `B` of Figure 2a must be decomposed when rectangle `s` is added to yield the decomposition given in Figure 2b. The rectangle quadtree representation assumes that all the blocks that are internal to a rectangle store an object identifier which is an offset in a feature table where the upper-left and bottom-right coordinate values of the rectangle are stored.

A central operation in spatial databases is the window operation. A window can be in the form of a rectangle or a polygon. This operation serves as a building block for a number of queries. 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. 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. More details about window queries and algorithms for answering them can be found in [1].

When a rectangle is clipped (e.g., as a result of a window operation), the rectangle quadtree does not update the coordinate values of the the clipped rectangle in the feature table. Notice that we cannot simply traverse the feature table and report the rectangles that we find since not all the rectangles in the feature table necessarily belong to the data structure in question.

3 Avoiding the Extra Space - A First Attempt

In this section we present a simple duplicate processing algorithm that does not require additional space (e.g., in contrast to the $O(n)$ space required for hashing in the case of a database of n objects). We also discuss the limitations of this algorithm. In the remaining sections of the paper, we will address these limitations and try to tackle them. We present the algorithm for the case of a database of line segment objects. A similar algorithm can be described for a database of rectangular objects.

In order to exploit the proximity of object pieces we define a test function, say t, such that given a spatial object, say o, with k pieces, p_1, p_2, ..., p_k, t has a value of false for only one of the pieces, say p_j of o. Application of t to the rest of the pieces yields the value of true. Any function t that satisfies this criterion can be used as a means of suppressing all pieces of o other than p_j from reporting o's object identifier, and hence can be adopted by the Report_Unique operation. For example, in the case of hashing, a function t_1 is defined as follows (a is a bit array with all its elements initialized to have a value of false, and $oid(p_i)$ is the identifier of the object to which piece p_i belongs):

$$\begin{aligned} t_1(p_i) \quad = \quad & ret\_value \leftarrow a[oid(p_i)]; \\ & if(not\ a[oid(p_i)])\ then \\ & \quad a[oid(p_i)] \leftarrow true; \\ & return\ (ret\_value); \end{aligned}$$

We can get rid of the $O(n)$ space requirements of the function t_1 in the following way. This gives rise to algorithm *Report_Unique1* described below. We traverse the blocks of the PMR quadtree, and for each block, say B, we perform the following actions:

1. For each object identifier i stored in B (that corresponds to piece p_i), retrieve the line segment description, say l_i, from the feature table, and
2. perform the following testing function t_2

$$\begin{aligned} t_2(l_i, B) \quad = \quad & if(start\_point(l_i)\ in\ B) \\ & then\ return\ (false) \\ & else\ return\ (true) \end{aligned}$$

3. If $t_2(l_i, B)$ is false, then report line segment l_i.

We use Figure 3 for illustration. Notice that for each line segment `l`, t_2 is false only when the block containing the start point of `l` is processed. Application of t_2 to all blocks containing pieces of `r` other than the upper-left corner of `r` yields true (t_2 is referred to as the *point-in-block test*). Notice also that the algorithm works only in the case where the pieces comprising the spatial object are contiguous. The case where the pieces of an object are disjoint (e.g., due to clipping parts of the object by a window operation, described in Section 2) is handled in Section 5.

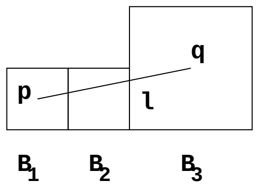

Figure 3: Line `l` is stored in blocks B_1, B_2, and B_3 where only block B_1 causes the point-in-block test to succeed since the starting point `p` of `l` only lies inside B_1 and not in B_2 nor in B_3.

The algorithm Report_Unique1 does not require any additional space. However, its CPU and disk-I/O costs are high. The algorithm applies a point-in-block test per object piece, which requires four comparisons (each of cost C_{cpu}). In particular, the point (x, y) is inside block $B((bl_x, bl_y), (ur_x, ur_y))$, where bl and ur denote the bottom left and upper right corners of the block, respectively, *iff* $bl_x \le x \le ur_x \wedge bl_y \le y \le ur_y$. t_2 is performed nk times, where n is the number of objects in the database, and k is the average number of pieces per object in the database.

Notice that in order to perform the point-in-block test (t_2) we must access the feature table (via the object identifier) each time we encounter a piece of a line segment. This is necessary in order to retrieve the start point of the line segment. Assuming that the cost of retrieving a segment from the feature table is C_{io}, then the execution time of the algorithm is $4knC_{cpu} + knC_{io}$.

The cost term knC_{io} involves redundant disk-I/O requests. Basically, it represents the cost of retrieving line segments from the feature table (once for every line partition, amounting to k disk requests per line segment). A better algorithm would perform only n such requests, i.e., one request per line segment in the database instead of one request for each piece of each line segment. Thus, we would like to develop an algorithm that does not need the $O(n)$ extra storage (or at least having an asymptotic storage cost less than $O(n)$), yet one that still performs only n disk requests (e.g., n accesses to the feature table). In the following sections, we show how we can achieve this goal. However, this depends on a closer scrutiny of the nature of the spatial objects which is the subject of the next section.

4 Object Classification

An important factor affecting the performance of the Report_Unique operation is the nature of the objects stored in the database. For example, what is the effect of restricting the lines to be rectilinear, in contrast to lines with arbitrary slopes? As another example, suppose that objects are partially clipped as is the situation after a window operation (described in Section 2). Is it more difficult to report these clipped objects than reporting non-clipped objects?

As an illustration of the second example, note that due to the way the PMR quadtree is defined, Report_Unique1 does not work properly if the line segments are partially clipped. This is shown in in Figure 4. In particular, when a line is clipped (e.g., as a result of a window operation), the PMR quadtree does not update the starting and ending points of the clipped line in the feature table. This is done since the line may be shared by other indexes in the database. In addition, this ensures consistency when portions of line segments are removed and then are added back later in the processing as a result of set operations [16]. Thus the feature table is not updated and the clipped line segment is stored implicitly. As a result, if a line, say `l`, in Figure 4 is clipped so that only `s` remains and if `s` does not contain `l`'s starting point, then `s` will not be reported by Report_Unique1. This case suggests that we have to consider the alternative classes of objects (e.g., clipped objects) when developing algorithms for Report_Unique since it affects the correctness of the algorithm. Due to space limitations, we restrict our discussion to rectangular objects only. More details about the classification and duplicate processing algorithms for line segment objects can be found in [2].

Spatial objects of type rectangle can be classified in the following way (the different classes are given in Figure 5):

- Class-1 rectangles: a rectangle having no clipped or missing portions as illustrated in Figure 5a, and termed a *regular* rectangle.
- Class-2 rectangles: a rectangle where only a rectangular portion of it is included and the rest of the rectangle is clipped as illustrated in Figure 5b, and termed a *clipped* rectangle.
- Class-3 rectangles: a rectangle where several portions (holes) may be missing, but still the rest of the rectangle is connected, as illustrated in Figure 5c, and termed a *clipped-out* rectangle.
- Class-4 rectangles: a rectangle where several portions (holes) of arbitrary shape may be missing that result

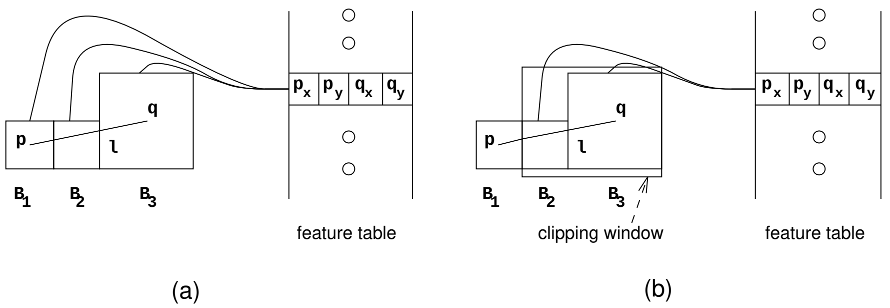

Figure 4: (a) Line l as stored in a PMR quadtree. (b) The result of clipping l by a window operation in a PMR quadtree. The original starting point of l in the feature table is neither in blocks B_2 nor in B_3. Notice that the feature table still stores point p as part of the description of the clipped line.

in the rectangle being decomposed into several disjoint pieces, as illustrated in Figure 5d, and termed a *disjoint* rectangle. Although disjoint, all the portions of the rectangle refer to, and represent, just one object of type rectangle.

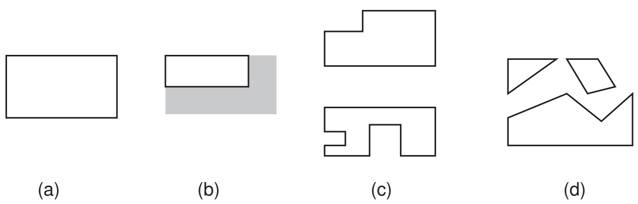

Figure 5: Classification of rectangles (a) Class-1, (b) Class-2, (c) Class-3, (d) Class-4.

This classification of rectangles is of practical use. Class-1 represents regular rectangles (e.g., bounding boxes of some spatial objects in a map). Class-2 represents the parts of a rectangle that remain after a rectangular window operation (Figure 6a). Class-3 represents the parts of a rectangle

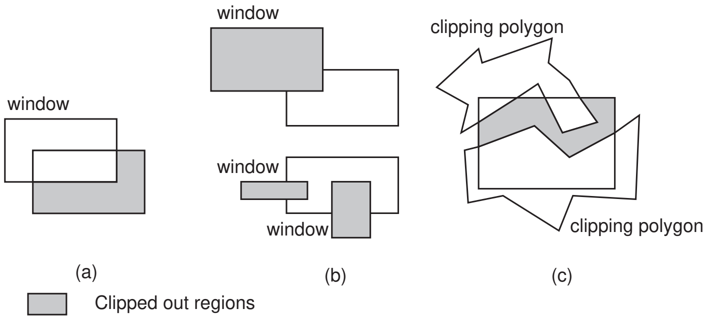

Figure 6: Example illustrating how some of the rectangle classes are created: (a) Class-2, (b) Class-3, (c) Class-4.

that lie outside one or more rectangular query windows (Figure 6b). These rectangles result from clipping-out regular rectangles against the query rectangles. Notice that each of the resulting rectangles is still representable as a four-connected region. Class-4 represents the parts of a rectangle that lie inside (or outside) one or more query windows of arbitrary shape. For example, Figure 6c is the result of intersecting and clipping a rectangle with a number of simple polygons. We further classify rectangles of each class into the following types according to their orientation:

- Type-1 rectangles: rectilinear rectangles, i.e., whose sides are parallel to the axes (Figure 7a).
- Type-2 rectangles: arbitrary rectangles - i.e., rectangles whose orthogonal sides have arbitrary orientations (slopes) (Figure 7b).

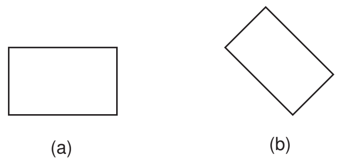

Figure 7: Two types of rectangles: (a) Type-1, (b) Type-2.

5 Duplicate Processing for Rectangular Objects

Below, we present algorithms for the Report_Unique problem that can handle several class/type combinations of rectangular objects given in Section 4, and see how these combinations affect the complexity of the algorithms. It is important to observe that rectangles of the underlying database are allowed to overlap in space. Let q be the maximum number of overlapping rectangles at any point in space, i.e., for any block, there are at most q overlapping rectangles.

5.1 Class-1 Type-1 and Class-2 Type-1 Rectangles

The rectangles that correspond to these class/type combinations are ones whose sides are parallel to the x and y axes where some of the rectangles may be clipped (Class-2) but are still of rectangular form. Our goal for this Class/Type combination is not to perform any disk I/O to the feature table. In this case, the only information at hand are the rectangle identifiers in the quadtree blocks.

The algorithm traverses the rectangle quadtree block-by-block and maintains the *active set* of rectangle identifiers. These correspond to the rectangles that intersect the block. A rectangle identifier, say *rid*, which corresponds to a rectangle, say r, is added to the active set when *rid* is first encountered during the traversal. *rid* is deleted from the active set once all the quadtree blocks that r intersects have been visited. The necessary storage is bounded by the maximum size of the active set during the execution of the algorithm. The size of the active set is considerably smaller than n, the number of rectangles in the spatial database.

In order to execute the algorithm efficiently, we need to organize the active set of rectangles. The organization of the active set depends on the operations that are to be performed on the set. The algorithm performs the following three primitive operations on an active set, say A: (1) test if a rectangle identifier exists in A (notice that this is needed in order to decide whether a rectangle has already been encountered or if it is encountered for the first time in the traversal), (2) insert a rectangle identifier into A, and (3) delete a rectangle identifier from A.

Before providing more details about the algorithm, observe that if we are able to perform each of the above three operations in $O(1)$ time and maintain the underlying data structure that stores the active set in $O(1)$ time, then the overall complexity of the algorithm would be $O(nk)$ time with $O(\textit{active set size})$ space.

The algorithm visits the blocks of the quadtree in NW, NE, SW, SE scanning order. It maintains one basic data structure: an *active border* [15]. In the context of duplicate processing and hashing, the active border is the spatial analog of a hash table. The active border represents the border between those quadtree blocks that have been processed and those that have not. The elements of the active border form a "staircase" of vertical and horizontal edges, moving from southwest to northeast, as shown by the heavy line in Figure 8. Each element of the active border corresponds to an edge (border) in the active border. Initially, the active border consists of two elements that correspond to the north and west borders of the image. When the algorithm terminates, the active border consists of elements that correspond to the south and east borders of the entire image. In other words, whenever a node is visited by the algorithm, the elements of the active border are updated accordingly to include the border of this new block.

We define an active rectangle as a rectangle that is partially processed, i.e., at least one of the quadtree blocks overlapping with this rectangle has not been processed yet. A rectangle is inactive if either all or none of the blocks through which it passes have been processed by the traversal algorithm. There is a data bucket associated with each element of the active border. Each bucket is of capacity q, which is the same as the bucket capacity of the underlying quadtree block. Every element of the active border stores in its bucket the set of rectangle identifiers which correspond to the active rectangles that intersect the portion of the active border corresponding to this element.

Rectangle identifiers are added to the active border when they are encountered during the traversal, and are propagated into the active border until all the pieces of a rectangle have been visited. A rectangle is reported when it is being deleted from the active border. The problem is how to detect that a rectangle has been processed in its entirety when there is no means for the algorithm to know the coordinate values of the end-points of the rectangle. In order to achieve this, the algorithm maintains some temporary information in the active border to help it determine that the lower-right corner of the rectangle has been reached and that the rectangle identifier can be reported. For example, consider the rectangle `r` given in Figure 2a. We use Figure 8 to illustrate the process of reporting its presence. Heavy shading is used to indicate that the block has already been traversed. When block `A` is encountered during the traversal (Figure 8a), `r`'s identifier (associated with `A`) is inserted into the active border. When block `B` is visited (Figure 8b), `r`'s identifier is propagated into the active border. Upon encountering a western or southern boundary of a rectangle for the first time (e.g., when processing block `C` in Figure 8c) a special marker symbol, say $\mathtt{e_r}$, is associated with the active border element that led to the detection of this fact (e.g., the active border element that is adjacent to block `C`). Notice that $\mathtt{e_r}$ is propagated into the active border when block `D` is processed (Figure 8d). As another example, we observe that a special marker symbol, say $\mathtt{s_r}$ is added to the active border element contiguous to block `E` (Figure 8e) and is propagated into the active border elements to the east of `E`, when their adjacent blocks in the database are processed. At the time block `F` is processed (Figure 8f), the active border elements to the north and west of `F` will contain the special markers $\mathtt{e_r}$ and $\mathtt{s_r}$, respectively. This situation serves to indicate that we are through. At this point, both $\mathtt{e_r}$ and $\mathtt{s_r}$ are deleted from the active border and `r`'s identifier is reported. Notice that `r` is reported once all the blocks comprising `r` have been visited by the traversal procedure. For example, in Figure 8f, at the time block `F` is visited, we are sure that all the blocks inside `r` are already processed by the algorithm. This is because a NW, NE, SW, SE traversal order is admissible [4].

The complexity of the algorithm can be evaluated as follows. When a block in the quadtree that corresponds to a leaf node is processed, the portion of the active border that is adjacent to the block must be located. In [3], a technique is described that enables the blocks to be located in the active border in constant time. This is achieved by traversing (and updating) the active border along with the quadtree traversal while passing and stacking pointers to guarantee that the exact location in the active border is available ($O(1)$ time) whenever needed, so that searching in the active border is entirely avoided. The reader is referred to [3] for further details. Once the appropriate active border element is located, testing this element for the existence of a rectangle as well as insertion and deletion of a rectangle can each be done in $O(q)$ time which is really a constant or $O(1)$. As demonstrated in this section, our algorithm does not need to perform any disk I/O requests to access the feature table. As a result, the time complexity of this algorithm is $O(nkq)$ which is proportional to the number of object pieces in the database that are encountered during the traversal. Notice that the factor q is included because all the procedures that access the active border perform in $O(q)$ time, as described in the introduction of Section 5.

The space complexity is $O(\textit{active rectangles})$, where the active rectangles are the ones that intersect the active border at any given point. The size of the active set in the worst case is $O(qT)$ where q is the maximum number of objects that can be stored in a quadtree block before it overflows (i.e., the bucket size) and T is the width (e.g., pixels) of the space comprising the underlying spatial database. In practice, it is expected that the size of the active set is considerably smaller than $O(qT)$. Observe that it is guaranteed that the active border buckets will never overflow since there is one-to-one correspondence between elements of the active border and blocks of the underlying quadtree. Since each block can hold up to q rectangles, then at worst each bucket will hold the same number of rectangles and hence cannot overflow. Notice that the algorithm does not depend on the actual coordinate values of the rectangle since it does not access the entries in the feature table and hence it works correctly even if rectangles are clipped due to some intersection with a rectangular window (Class-2 rectangles).

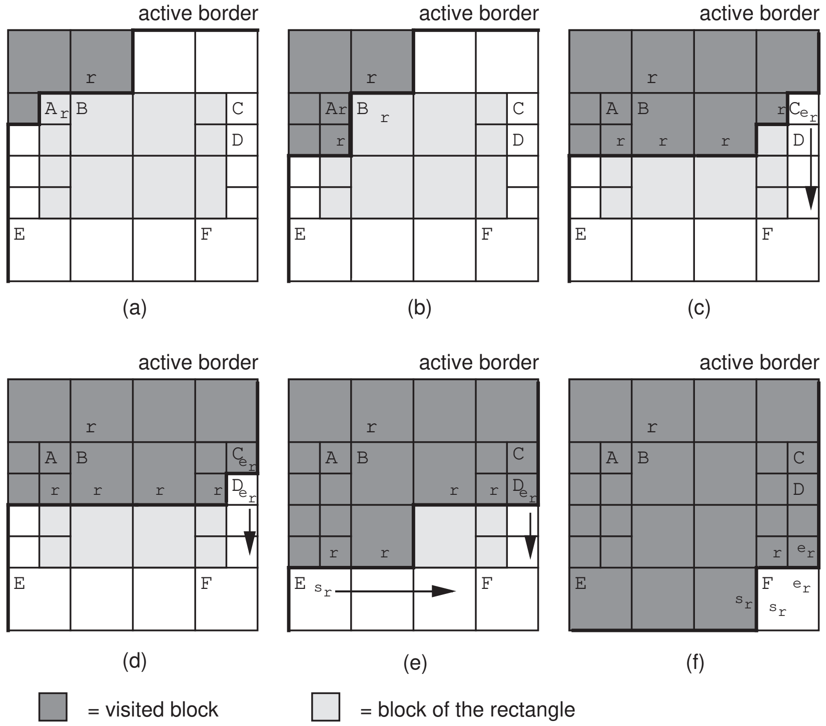

Figure 8: (a) When block `A` is processed `r` is inserted into the active border. (b) `r` is propagated into the active border when block `B` is processed. (c) When block `C` is processed a special marker symbol `er` is inserted into the active border. (d) `er` is propagated into the active border when block `D` is processed. (e) When block `E` is processed a special marker symbol `sr` is inserted into the active border. (f) `r` is reported at the time block `F` is processed and both `er` and `sr` are deleted from the active border.

5.2 Class-3 Type-1 and Class-4 Type-1 Rectangles

Class-3 and Class-4 imply that the rectangles may have notches on them or may even consist of disjoint pieces. For these classes, we cannot avoid accessing their entries in the feature table in order to know the real extent of the rectangles while traversing them. Rectangle identifiers are propagated in the active border (regardless of whether we are traversing a part of the rectangle or a clipped out portion of the rectangle) until the lower-right corner of the rectangle is visited by the traversal. In this case, the rectangle is reported and deleted from the active border. This implies that the algorithm has to issue some disk I/O requests to the feature table. The algorithm proceeds as follows:

1. Traverse the spatial database as well as the active border simultaneously in the NW, NE, SW, SE order.
2. Maintain in the active border:
   (a) the identifiers of the rectangles that intersect the active border or whose pieces have not yet been processed in their entirety by the algorithm, and
   (b) the coordinate values of the end-points of the rectangles in (a).
3. When the block containing the lower-right corner of rectangle r is reached, report the identifier of r, and delete r's information from the active border.

Figure 9 illustrates the locations at which a disk I/O request is issued and when a rectangle is reported for several possible rectangles in Class-3 and Class-4. Notice that for Class-3 rectangles one, or more disk I/O requests can be issued. For example Figures 9a-c and e have only one disk I/O request, Figure 9d has two disk I/O requests as the upper-left corner of the rectangle is clipped-out, while Figure 9f has many disk I/O requests. For Class-4 rectangles, more than one disk I/O request is possible (e.g., Figure 9g) depending on the number of blocks in one of the four line segments of the window boundary that has a NE-SW orientation and is closest to the origin of the underlying space. Notice that for Class-4 rectangles, if the upper-left corner of the rectangle is not clipped out, then only one disk I/O request is issued regardless of the number of disjoint pieces of the rectangle (e.g., Figure 9h). Since the upper-left corner of the rectangle is processed first, a disk I/O request is issued to the feature table and the coordinate values of the end-points of the rectangle are made known to the active border. This helps avoid any further disk I/O requests. Observe that this is not the case when the upper-left corner of the rectangle is clipped out (e.g., Figure 9f).

Figures 9f and 9g illustrate the cases where the maximum number of disk I/O requests for Class-3 and Class-4 rectangles can be generated, respectively. Let k_w be the number of blocks in the edge of the window with a NE–SW orientation that is closest to the origin of the underlying space. Figure 10 illustrates the worst case number of disk I/O requests that are required with lines of this orientation. Observe that a disk I/O request occurs with every other piece in the object. Therefore, $\lceil \frac{k_w}{2} \rceil$ disk I/O requests are issued. Then, the maximum number of disk I/O requests to the feature table for Class-3 and Class-4 is $\lceil \frac{k_w}{2} \rceil$.

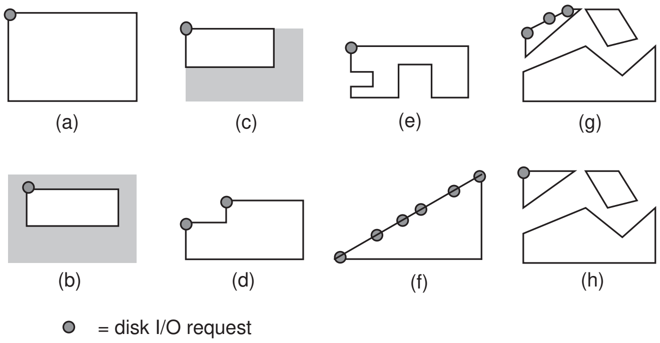

Figure 9: (a)-(c) When the rectangle is first encountered a disk I/O request is issued to retrieve the rectangle's coordinate values. (d) Two disk requests are issued whenever, the upper-left corner of the rectangle is clipped out (Class-3). (e) Only one disk request is issued since the upper-left corner of the rectangle is not clipped out (Class-3). (f) The worst-case for Class-3 rectangles occurs when many query windows clip out parts of the rectangle such that the upper-left piece of the remaining portion of the rectangle forms a line segment of NE-SW orientation. (g) Redundant disk requests are issued when the upper-left piece of the rectangle is processed since the upper-left corner of the rectangle is clipped out (Class-4). (h) Only one disk request is issued although the rectangle is partitioned into more than one disjoint piece (Class-4). Notice that the upper-left corner of the rectangle is not clipped out.

It is worth mentioning that the task of uniquely reporting spatial objects is different from that of connected component labeling [13, 14]. In the latter task we are interested in assigning a unique identifier to each four- (or eight-) connected region in the underlying space. The principal difference is that, generally speaking, the inputs of the two tasks are different. More specifically, in Report_Unique we can always perform a disk I/O request to the feature table and retrieve the object's exact description. This information is not made available to the connected component labeling algorithms. In fact, the main purpose of the connected component labeling algorithms is to build the object's exact description from the local information given in the underlying space.

As a result of the difference in the nature of the type of input and the goals of the two tasks, their complexities are different. For example, Figure 11 demonstrates the worst-case complexity for the connected component labeling algorithm (measured in terms of the new labels that have to be introduced). On the other hand, the shape of the re-

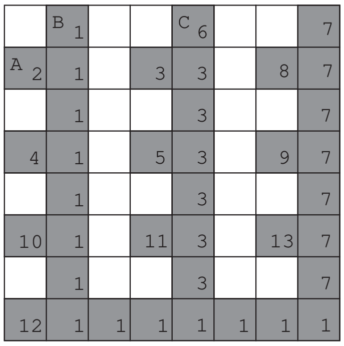

Figure 11: An 8×8 image that results in generating a maximum number of equivalence pairs for the connected component labeling algorithm.

gion in Figure 11 does not represent the worst case disk I/O complexity for Report_Unique. In particular, only two disk I/O requests will be issued (i.e., when blocks `A` and `B` are visited). The worst case number of disk I/O requests for Report_Unique for clipped-out rectangles (Class-3) is given in Figure 9f. Notice that having a saw-tooth-like shape (e.g., as in Figure 11) does not change the worst case since at block `B` of Figure 11, a disk I/O request is issued to retrieve the exact description of the rectangle from the feature table, and this description will be propagated in the active border in the eastern and southern directions. This covers all the saw-tooths to the east and south of block `B`, thereby obviating the need for additional disk I/O requests (e.g., at block `C`).

5.3 Rectangles of Type-2

In order to ensure that rectangles of arbitrary orientations are reported only once, we maintain their rectilinear bounding box in the active border (Figure 12a) and report the rectangle identifier once the lower-right corner of the bounding box is covered by the traversal. As described in Section 5.2

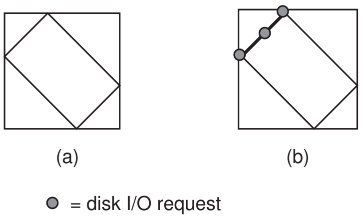

Figure 12: (a) A bounding box is stored in the active border for rectangles of Type-2. (b) Several disk requests occur when processing the line segment closest to the upper-left corner.

(also in Figures 9f, 9g, and 10) the number of disk I/O requests to the feature table is proportional to $\lceil \frac{k_w}{2} \rceil$, where k_w is the number of blocks in the line segment of the window boundary with a NE-SW orientation that is closest to the origin of the underlying space (Figure 12b). Notice that once this set of disk I/O requests is issued, the algorithm does not need to access the feature table again since the

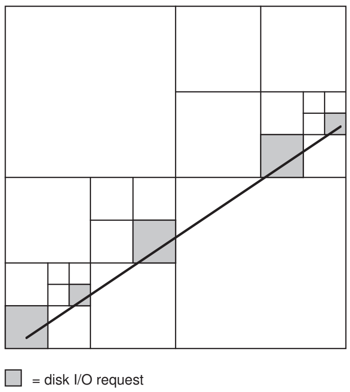

Figure 10: The maximum number of disk requests that can arise for a line with NE-SW orientation.

propagation of the rectangle identifier and coordinate values in the active border will help avoid any further disk I/O requests to the feature table.

6 Conclusion and Future Research

Table 6 gives a summary of our results. It shows the different complexities of the algorithms developed for the various class/type object combinations. As we can see, duplicate

	Type-1	Type-2
Class-1	$O(knq)$ avg. cpu, no i/o $O(qT)$ space, O(active objects) avg. space	$O(knq)$ avg. cpu, $\lceil\frac{kw}{2}\rceil n$ i/o $O(qT)$ space O(active objects) avg. space
Class-2	$O(knq)$ avg. cpu, no i/o $O(qT)$ space, O(active objects) avg. space	$O(knq)$ avg. cpu, $\lceil\frac{kw}{2}\rceil n$ i/o $O(qT)$ space O(active objects) avg. space
Class-3	$O(knq)$ avg. cpu, $\lceil\frac{kw}{2}\rceil n$ i/o $O(qT)$ space O(active objects) avg. space	$O(knq)$ avg. cpu, $\lceil\frac{kw}{2}\rceil n$ i/o $O(qT)$ space O(active objects) avg. space
Class-4	$O(knq)$ avg. cpu, $\lceil\frac{kw}{2}\rceil n$ i/o $O(qT)$ space O(active objects) avg. space	$O(knq)$ avg. cpu, $\lceil\frac{kw}{2}\rceil n$ i/o $O(qT)$ space O(active objects) avg. space

Table 1: Summary of time and space complexities of Report_Unique for class/type combinations of rectangle objects.

processing using the spatial characteristics of objects is a challenging problem. In particular, using the spatial characteristics of the objects enables us to adapt techniques used with hashing. For example, we were able to reduce the number of disk I/O requests (e.g., in comparison to $O(nk)$ disk I/O requests when using hashing), sometimes to 0 or $O(n)$ which is a significant improvement. In addition, by using the extent and proximity of spatial objects we were able to detect when an object is no longer needed in the active border (analogous to a hash table), and hence we were able to bound the size of the table. This was achieved by deleting the objects that are entirely processed by the algorithm. Future work involves developing better algorithms for other class/type combinations, classifying spatial objects besides line segments and rectangles as well as developing algorithms for them, and considering other spatial data structures that partition objects besides the quadtree variants.

7 Acknowledgements

The first author conducted most of this research while at The University of Maryland, College Park. The support of the National Science Foundation under Grant IRI 9216970 is gratefully acknowledged.

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. (also in Proceedings of the Fifth Brazilian Symposium on Databases, Rio de Janeiro, Brazil, April 1990, 15–26).

[2] 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*, pages 178–189, Charleston, SC, August 1992.

[3] M. B. Dillencourt and H. Samet. Using topological sweep to extract the boundaries of regions in maps represented by region quadtrees. *Submitted for publication*, 1991.

[4] M.B. Dillencourt, H. Samet, and M. Tamminen. A general approach to connected-component labeling for arbitrary image representations. *Journal of the ACM*, 39(2):253–280, April 1992.

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

[6] C. Faloutsos, T. Sellis, and N. Roussopoulos. Analysis of object oriented spatial access methods. In *Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data*, pages 426–439, San Francisco, May 1987.

[7] Michael J. Folk and Bill Zoellick. *File Structures - Second Edition.* Addison-Wesley, Reading, MA, 1992.

[8] O. Günther. The design of the cell tree: an object-oriented index structure for geometric databases. In *Proceedings of the Fifth IEEE International Conference on Data Engineering*, pages 598–605, Los Angeles, February 1989.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, pages 47–57, Boston, June 1984.

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

[11] 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).

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

[13] A. Rosenfeld and A. C. Kak. *Digital Picture Processing.* Academic Press, New York, second edition, 1982.

[14] H. Samet. Connected component labeling using quadtrees. *Journal of the ACM*, 28(3):487–501, July 1981.

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

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