PERFORMANCE EVALUATION OF SPATIAL JOIN STRATEGIES

Ye-In Chang†

†Dept. of Computer Science and Engineering
National Sun Yat-Sen University
Kaohsiung, Taiwan
Republic of China
E-mail: changyi@cse.nsysu.edu.tw

Ming-Jyh Wu‡

‡Dept. of Applied Mathematics
National Sun Yat-Sen University
Kaohsiung, Taiwan
Republic of China

ABSTRACT

Spatial database users frequently need to combine two spatial inputs based on some spatial relationship between the objects in the two inputs, for example, *map overlap*. This operation of combining two inputs based on their spatial relationship is called a *spatial join*, which is an expensive operation. Rotem's algorithm is a well-known approach for *overlap* detection, which is based on a new join index. However, there are some errors in Rotem's algorithm, which can result in some wrong answers. Brinkhoff et al. have proposed a different approach for *overlap* detection which is based on the *R-tree*. In this paper, we first correct the errors in Rotem's algorithm. Next, we study the performance of Rotem's and Brinkhoff et al.'s algorithms for *overlap* detection by simulation. From our simulation results, we show that Brinkhoff et al.'s algorithm needs longer join processing time, a larger number of comparisons and a smaller number of buffers than Rotem's algorithm.

1. INTRODUCTION

Recently, spatial database systems have become more and more important for public administration, science and business. Several spatial database systems (spatial DBSs), particularly designed for organizing spatial data of a geographic information system (GIS), have been developed for applications such as cartography, environmental science and geography [1, 3, 14].

In analogy to relational DBSs, a collection of spatial objects defined on the same attributes is called a *spatial relation*. A typing spatial query is the *window* query where the response set consists of all objects whose geometric component overlaps with a given query rectangle [3]. 'Find all objects which intersect a given window' is such an example. In contrast to a *window* query, the *spatial join* is defined on two or more relations. The spatial join computes a subset of the Cartesian product. It combines spatial objects from these relations according to their geometric attributes, such as distance, intersection, containment, etc [3].

In any case, the term *spatial* usually refers to objects and operators in a space of dimension 2 or higher. That implies, however, that they have the following key feature that makes the computation of spatial operators, including spatial joins and searches, significantly more difficult than the computation of their non-spatial counterparts [4]: *There is no total ordering among spatial objects that preserves spatial proximity.*

Since the representation of a spatial object can be very large, spatial operations, including the spatial join, typically operate in two steps [11]: (1) **Filter Step**: In this step, an approximation of each spatial objects, such as its Minimum Bounding Rectangle, (MBR) is used to eliminate tuples that cannot be part of the result. This step produces *candidates* that are a superset of the actual result. (2) **Refinement Step**: In this step, each candidate is examined to check if it is part of the result. This check generally requires running a CPU-intensive computational geometry algorithm.

Numerous algorithms have been proposed to execute the filter step of a spatial join. Some of the earlier algorithms are based on transforming an approximation of spatial object into another domain (e.g. a 1-dimensional domain), and performing the filter step in the new domain [10, 15]. Naturally, we expect the spatial join operator to take advantage of these indices in the same way as a conventional join is typically performed using R-trees, R^+-trees, or hash tables in most commercial relational database products [2, 4, 5, 7, 9, 12]. Recently, an index called *join index* was proposed as an accelerator for performing join operations between two relations. The idea is to maintain a precomputed structure that indicates which tuples from one relation will match tuples of the other rela-

[1] This research was supported in part by the National Science Council of Republic of China under Grant No. NSC-89-2213-E-110-004

tion based on some join predicate [12, 13]. The other simple and straightforward approach without using any index is to apply the strategies for relational join operations, including nested-loop and hash partition, to spatial join operations [3, 6, 8, 11].

Among those algorithms for spatial joins, Rotem's algorithm [12] is one of the well known algorithms based on a spatial join index approach, while Brinkhoff et al.'s algorithm [2] is one of the well-known algorithms based on the *R-tree* approach. In Rotem's algorithm, it partially precomputes the results of a spatial join. The algorithm for building the spatial join requires grid files for indexing the spatial data, and uses these grid files to compute the spatial join index. However, there are some errors in Rotem's algorithm, which can result in some wrong answers. In Brinkhoff et al.'s algorithm, it shows that the *R-tree* can also be exploited for performing spatial joins efficiently. The algorithm sorts the entries in a node of the R-tree according to the spatial location of the corresponding rectangles.

In this paper, we focus on the problem of region data. Moreover, we focus on the filter step. We first correct the errors in Rotem's overlap detection algorithm. Next, we study the performance of Rotem's and Brinkhoff et al.'s algorithms for *overlap* detection by simulation. From our simulation results, we show that Brinkhoff et al.'s algorithm needs longer join processing time, a larger number of comparisons and a smaller number of buffers than Rotem's algorithm. Moreover, both the algorithms need sorting process first.

2. A SURVEY

2.1 Rotem's Algorithm

In [12], Rotem proposed an algorithm to generate a join index from two given grid files. This algorithm assumes that each file G_i has an associated region directory RD_i, already built for it. In the two dimensional case, each entry includes a *region-id* entry and *two pairs of coordinates* giving the bottom left and the top right corners of each region.

The collection of regions defined by the region directory forms a tiling of the coverage area called a *region map*. Assume that the region map of one grid file is super-imposed on top of the other one to obtain a single map which includes all regions (see Figure 1). The idea is to use a technique from computational geometry called *plane sweeping* in which an imaginary line, parallel to the y axis, scans the super-imposed map from left to right. At each point in time, all regions cut by the line are called *candidate* regions. A region stops being *candidate* when all its points are to the left of the sweeping line. The algorithm observes that if two regions have an overlap they must be simultaneously

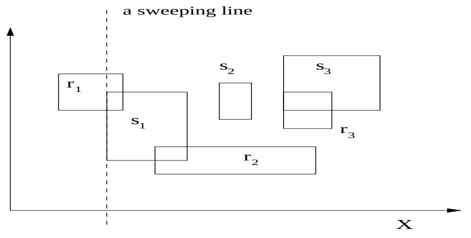

Figure 1: A sweeping line

candidate at some point during the execution of the algorithm as the line must cut their common intersection area [12].

For that reason, it is necessary to maintain a list of the current *candidate* regions for each directory RD_i, and add or delete regions from it as the line enters or leaves them, respectively. Edges are to be the join index whenever it is detected that a region l from RD_i which just became *candidate* has an overlap with any of the other regions on the *candidate* list for RD_j where $j \neq i$. Since the algorithm requires separating x and y coordinates and merging information from both directories, it will scan the directories and construct three auxiliary structures: a set X of vectors, a pair of arrays *Y1* and *Y2* as described below.

From each single x coordinate found in RD_1 *or* RD_2, a vector is built with the structure:
<value, bt, region-id, file-id> with the following interpretation: (1) **value**: a value of a coordinate. (2) **bt**: 0 if this is a left coordinate, otherwise 1. (3) **region-id**: the region-id number for this coordinate. (4) **file-id**: either 1 or 2 depending on the region directory RD_j this coordinate comes from.

The set X, will have these vectors as its elements, and will have a cardinality of *2t* where $t = n_1 + n_2$, as each region in the system contributes two x coordinates to this set. (Note that here, the algorithm assumes that G_1 and G_2 be two grid files with n_1 and n_2 data pages, respectively.)

Next, for the arrays which stores y coordinates, these will be kept in one of two arrays declared as *Y1[n_1][2]* for y coordinates of RD_1 and *Y2[n_2][2]* for y coordinates of RD_2. In each of these arrays, the first subscript indicates *region-id* number and the second indicates whether it is a bottom (subscript equals 0) or top coordinate (subscript equals 1) for the region. For example, assume that an entry in RD_1 has *region id* 8, with bottom left corner (3,6) and top right corner (12,15). This will generate vectors <3,0,8,1> and <12,1,8,1> in X and a pair of entries *Y1[8][0]=6* and *Y1[8][1]=15* in *Y1*.

In this algorithm, it keeps an updated list of all can-

```
Algorithm-1: Join Index Generator
Input: A pair of region directories $RD_1$ and $RD_2$ with $|RD_i| = n_i$;
Output: Join graph edge list;
Begin:
        Scan both directories $RD_1$ and $RD_2$
        and generate a set $X$ of vectors and the
        pair of arrays $Y1$ and $Y2$;
        Sort the set of vectors in $X$ based on the first
        and second component of each vector and
        obtain $\vec{X}$;
        $t := n_1 + n_2$;
        For $i := 1$ to $2t$ Do
            Call DETECT($\vec{x_i}$);
End;
```

Figure 2: Algorithm-1 in Rotem's algorithm

```
Procedure DETECT($\vec{x}$ : vector);
Begin
    $j = \vec{x}[2]$; $k = \vec{x}[3]$; $l = \vec{x}[4]$;
    (* j indicates if this is a left or right coordinate; k
    and l provide the region and file identity *)
    If ($j = 0$) then                (* the left coordinate *)
        Begin
            Insert <Y1[k][0]; Y1[k][1]; k> into binary
            tree $B_l$;
            Call SEARCH(Y1[k][0], Y1[k][1], k, l);
            (* this is a new region, it must become
            candidate and get inserted into the
            appropriate tree *)
        End
    Else              (* $j = 1$, the right coordinate *)
            Delete the node $a_i$ with key Y1[k][0] from
            $B_l$;
            (* this is the end of the region, it must be
            deleted from the tree *)
End;
```

Figure 3: Procedure DETECT in Rotem's algorithm

didate regions for each directory RD_i. First, a new region can become candidate (or stop being one) only when the sweeping line passes over an x coordinate of some region of RD_j for $j = 1, 2$. At such points, called *detection points*, the algorithm updates the candidate list and calls a *detection* routine to find which new overlaps are introduced.

In order to scan systematically the set of detection points, the algorithm sorts the elements in X according to their first component as major and second component as minor both in an increasing order to obtain the sorted set $\vec{X} =< \vec{x_1}, \vec{x_2}, \ldots\ldots \vec{x_{2t}} >$, where each $\vec{x_i}$ is some element of X. The algorithm simulates the sweeping line motion by checking at each successive $\vec{x_i}$ which region has become a candidate or stopped being a candidate. This is done by checking the second component of $\vec{x_i}$, if it is 0, the new element represents a beginning of a new region which must now become candidate; otherwise, it is the end of a region which must now stop being a candidate. The identity of the region and the region directory (RD_1 or RD_2) it comes from, can be found from the last two components of $\vec{x_i}$. The whole algorithms are shown in Figures 2, 3 and 4.

```
Procedure SEARCH(key1, key2, reg, tree-id :
            integer);
Begin
    If tree-id = 1 then $m = 2$ else $m = 1$;
    (* find the correct tree in which to search *)
    If (key1 is larger than the maximum key in $B_m$ or
    key2 is smaller than the minimum key in $B_m$) then
        return;      (* no overlap *) (* Case I *)
    Search for the largest key in $B_m$ which does not
    exceed key1, call that node init;
    If (search unsuccessful) then let init to be the
    leftmost node in $B_m$;
    Perform in-order traversal of $B_m$ from init until
    a node last found for which $TOP_{last}$ (the second
    component of node last) $\geq$ key2 or tree traversal
    completed;          (* Case II *)
    For each node $n$ visited from init to last
        Add an edge (denoting an overlap relationship)
        to the join graph connecting region reg to
        $REG_n$;      (* Case III *)
End;
```

Figure 4: Procedure SEARCH in Rotem's algorithm

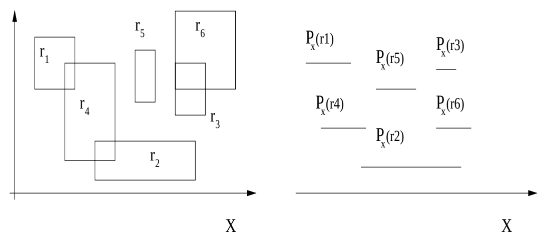

Figure 5: Two sets of rectangles and their projection onto the X-axis

2.2 Brinkhoff et al.'s Algorithm

In [2], Brinkhoff et al. have proposed an approach for improving spatial join which sorts the entries in a node of the *R-tree* according to the spatial location of the corresponding rectangles. Let us consider a sequence $Rseq = < r_1, \ldots, r_n >$ (or $Sseq = < s_1, \ldots, s_n >$) of n rectangles. A rectangle r_i is given by its lower left corner ($r_i.xl$, $r_i.yl$) and its upper right corner ($r_i.xu$, $r_i.yu$). The algorithm uses $P_x(r_i)$ and $P_y(r_i)$ to refer to the projection of r_i onto the X- and Y-axis, if $r_i.xl \leq r_{i+1}.xl$, $1 \leq i < n$. For example, a sorted se-

```
Procedure SortedIntersectionTest($Rseq$, $Sseq$:
            sequence of $rectangle$;
        Var $Output$: sequence of pair of $rectangle$);
    (* $Rseq$ and $Sseq$ are sorted; $|Rseq|$ = number
    of rectangles in $Rseq$ *)
Begin
    $Output$ := <>; $i$ := 1; $j$ := 1;
    While ($i \leq |Rseq|$) and ($j \leq |Sseq|$) Do
    Begin       If ($r_i.xl < s_j.xl$) then
        Begin
            Call InternalLoop($r_i$, $j$, $Sseq$, $Output$);
            $i := i + 1$;
        End
        Else
        Begin
            Call InternalLoop($s_j$, $i$, $Rseq$, $Output$);
            $j := j + 1$;
        End;
    End;
End;
```

Figure 6: Procedure SortedIntersectionTest in Brinkhoff et al.'s algorithm

```
Procedure InternalLoop($t$: $rectangle$; $unmarked$:
            $cardinal$;
            $Sseq$: sequence of $rectangle$; Var $Output$:
            sequence of pair of $rectangle$);
Begin
    $k := unmarked$;
    While ($k \leq |Sseq|$) and ($s_k.xl \leq t.xu$) Do
    Begin
        If ($t.yl < s_k.yu$) and ($t.yu > s_k.yl$) then
            Append ($t$,$s_k$) to $Output$;
        $k := k + 1$;
    End;
End;
```

Figure 7: Procedure InternalLoop in Brinkhoff et al.'s algorithm

quence of 6 rectangles depicted in Figure 5 is $< r_1, r_4, r_2, r_5, r_3, r_6 >$.

Plane sweep is a common techniques for computing intersections. The basic idea is to move a line, the so-called *sweep-line*, perpendicular to one of the axes, e.g. the *x*-axis, from left to right. Given two sequences of rectangles *Rseq* and *Sseq*, they exploit the plane-sweep technique without the overhead of building up any additional dynamic data structure. The formal descriptions of the algorithm are shown in Figures 6 and 7. For example, in Figure 8, the sweep-line stops at rectangles r_1, s_1, r_2, s_2 and r_3. For each step, the pairs of rectangles which are tested for intersection are given on the right hand side of Figure 8.

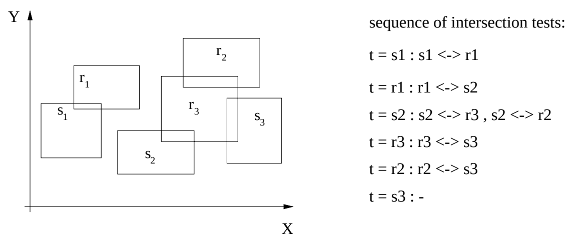

Figure 8: An example for the sorted intersection test

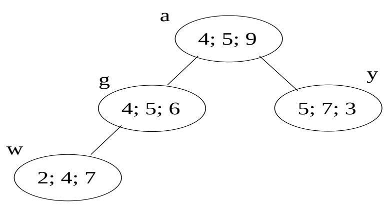

Figure 9: A binary search tree for Figure 10-(b)

3. A CORRECTION TO ROTEM'S OVERLAP DETECTION ALGORITHM

Rotem's algorithm [12] is a well-known approach for *overlap* detection, which is based on a new join index. However, there are some errors in Rotem's algorithm, which can result in some wrong answers. In this section, we correct the errors in Rotem's algorithm.

Following the basic concept about the data structure used in Rotem's algorithm as described in Section 2.1, we now describe Rotem's algorithm [12] in more details. Assume that a region with *region-id* i from RD_1 becomes candidate. Its bottom and top y coordinates can be read from the entries *Y1[i][0]* and *Y1[i][1]*, respectively. In this way, the algorithm can maintain the y coordinates of all regions which are currently candidate in a suitable data structure which will allow us to add or delete such coordinates, and search for overlaps between region i and current candidate regions from RD_2. The algorithm observes that such a region $l \in RD_2$ will overlap region i if conditions (I) or (II) hold:

(I) $Y2[l][0] \leq Y1[i][0] \leq Y2[l][1]$;

(II) $Y1[i][0] \leq Y2[l][0] \leq Y1[i][1]$.

A binary search tree B_1 for RD_1 based on bottom coordinates as keys is suitable for this purpose as shown in Figure 9. More specifically, each node s in this tree contains three elements $< BOT_s; TOP_s; REG_s >$. The first element is the key of the node (in the binary search tree) which is also the bottom coordinate of the region REG_s, and the second element identifies the top coordinate of that region. The third element identifies the *region-id*. A similar binary tree B_2 is constructed for candidate regions of RD_2.

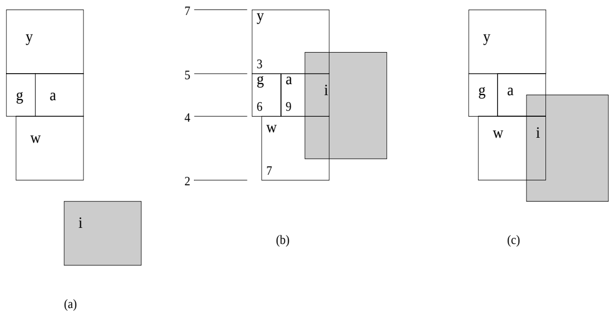

Figure 10: Three possible cases in *overlap* detection: (a) Case I; (b) Case II; (c) Case III.

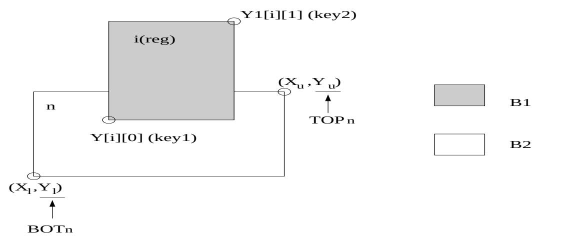

Figure 11: Case I in *overlap* detection

For each new candidate region i from RD_1, the algorithm inserts the triple $<$*Y1[i][0]; Y1[i][1]; i*$>$ into the tree B_1. Next, the following three cases (in which there exist some errors) are considered in Rotem's algorithm:

Case I : *Y1*[*i*][1] is smaller than any key in B_2 or *Y1*[*i*][0] is bigger than any key in B_2. The algorithm denotes that in this case, no overlaps involving region i are possible as shown in Figure 10-(a). However, based on this test as presented in Procedure SEARCH shown in Figure 4, for the example shown in Figure 11, it will conclude that there is no overlap between nodes n and *reg*, since it satisfies the condition "$key1$ (= *Y1*[*i*][0]) is larger than the maximum key in B_2," where the maximum key in B_2 is Bot_n. (Note that the key of the binary search tree is the bottom left y coordinate of node n.) To avoid such a mistake, we should replace this test with the condition "$key1$ (= *Y1* [*i*][0]) is larger than the maximum TOP_n (i.e., the *second* component of node n)."

Case II : A node w exists in B_2 such that $BOT_w \leq$ *Y1[i][0]* $\leq TOP_w$. If such a node is found, we search forward from that node in an increasing order of keys (inorder traversal) until we reach the end of the tree or a node y such that

$BOT_y \leq$ *Y1[i][1]* $\leq TOP_y$.

All the regions whose *region-id* is found on nodes visited by this traversal have an overlap with the region i as shown in Figure 10-(b).

Case III : A node w (as described in Case II) does not exist, i.e., *Y1*[*i*][0] is smaller than any key in B_2. Since we rule out the case of no overlaps (Case I), it follows that i overlaps the region represented by the leftmost node in B_2. The regions which have an overlap with i can be found by an inorder traversal of B_2, starting from the leftmost node until some node a is found where

$BOT_a \leq$ *Y1[i][1]* $\leq TOP_a$,

i.e, the end of the tree is reached. Figure 10-(c) shows such a case.

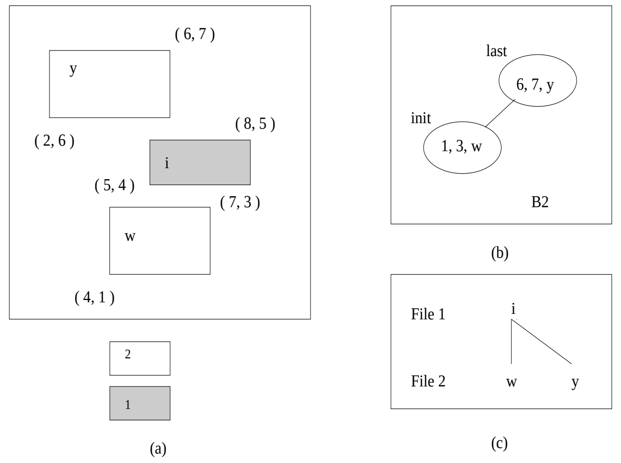

Figure 12: A counterexample

However, in Rotem's procedure SEARCH as presented in Figure 4, for Cases II and III, it only checks whether there exists a node a (in tree B_2) whose second component (i.e., the upper right y coordinate denoted as TOP_a) is larger than *Y1*[*i*][1]. For Case II, it only checks whether there exists a node b (in tree B_m) whose first component (i.e., the bottom left y coordinate denoted as BOT_b) is smaller than *Y1*[*i*][0]. Based on these pseudo-codes, they can result in a wrong answer. Let's see a counterexample. For the example shown in Figure 12, Rotem's procedure SEARCH will find *init* = w and *last* = y, and conclude that there exist an overlap relationship between regions i and w, and an overlap relationship between regions i and y. This wrong decision is caused by the following condition: when we traversal inorder from node *init* and *last*, there may exist nodes a and b on the same path in B_2, such that

$BOT_b \leq TOP_b \leq$ *Y1[i][0]* and

Y1[i][1] $\leq BOT_a \leq TOP_a$.

In this case, there should be no overlap relationship between nodes $a(b)$ and i. To rule out such a case, we have to check each possible overlap case between the incoming node *reg* and each one of those nodes n on the path between node *init* to node *last* shown in Figure 13. Figure 13-(a) means $BOT_n \leq$ *key1* $\leq TOP_n$,

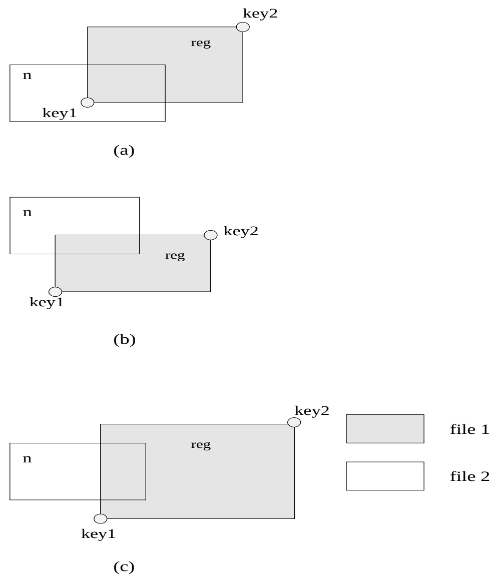

Figure 13: Three overlap cases: (a) $BOT_n \leq key1 \leq TOP_n$; (b) $BOT_n \leq key2 \leq TOP_n$; (c) $key1 \leq BOT_n \leq TOP_n \leq key2$.

Figure 13-(b) means $BOT_n \leq key2 \leq TOP_n$ and Figure 13-(c) means $key1 \leq BOT_n \leq TOP_n \leq key2$.

The correct SEARCH algorithm for overlap detection is shown in Figure 14.

```
Procedure SEARCH(key1, key2, reg, tree-id :
            integer);
Begin
    If tree-id = 1 then $m = 2$ else $m = 1$;
    If (key1 is larger than the maximum $TOP_n$ in
    $B_m$ or key2 is smaller than the minimum key in
    $B_m$) then
        return;   (* no overlap *)   (* Case I *)
    Search for the largest key in $B_m$ which does
    not exceed key1, call that node init;
    If (search unsuccessful) then let init to be
    the leftmost node in $B_m$;
    Perform inorder traversal of $B_m$ from init
    until a node last found for which $TOP_{last} \geq key2$ or
    tree traversal completed;
    For each node n visited from init to last
    (* Case II *)
        If ($BOT_n \leq key1 \leq TOP_n$) or
           ($BOT_n \leq key2 \leq TOP_n$) or
           ($key1 \leq BOT_n \leq TOP_n \leq key2$)then
           (* Case III *)
              Add an edge (denoting an overlap
              relationship) to the join graph
              connecting region reg to $REG_n$;
End;
```

Figure 14: The correct version of Procedure SEARCH

4. PERFORMANCE EVALUATION

4.1 Simulation Study

In our performance model, we make some assumptions. Each rectangles was displaced at random within the given two-dimensional data space; i.e., the data are uniform distribution. All databases used in this performance evaluation are randomly generated sets of rectangles with the *overlap selectivity*, where an *overlap selectivity* is defined as follows:

$$p = \frac{\text{the number of the data with } \textit{overlap} \text{ relationship}}{\text{the total number of data } (N)},$$

where both given files for the spatial join algorithms have the same total number of data N. Moreover, the buffers for join operations are infinite large. There are two major parameters that characterize such a geometric database: the number of data objects (the database size), N, and the *overlap selectivity* p $(0 \leq p \leq 1)$ [4].

In our simulation experiences, we first consider a map with 10000x10000 pixels in which N rectangles with a size between 5x5 and 25x25 pixels are uniformly distribution, where $N =$ 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000 and 10000. Based on such a map A and a given overlap selectivity p, we create the other map B in the following way, where p = 0.3, 0.4, 0.5, 0.6, 0.7 and 0.8. For each rectangle t in map A, we call a random function which returns a random number w. If $w \leq p$, we create a rectangle s in map B such that $s \cap t \neq \emptyset$; otherwise, we create a rectangle s in map B such that $s \cap t = \emptyset$, where $\cap$ means the *overlap* relationship. Therefore, there are totally 60 data files for map A and the related 60 data files for map B.

In this simulation study, we collect the following performance measures: (1) **The total join time (in terms of msec) to process the join operation.** (Note that here, we do not include the sorting time in both algorithms.) (2) **The total number of comparisons between these two input maps.** For example, in a nested join approach with an infinite buffer for files R and S, the total of comparisons is $|R| * |S|$, where $|R|$ is the size of file R. (3) **The number of buffers.** In Rotem's algorithm, we need buffer to store both the binary trees. In Brinkhoff et al.'s algorithm, we need buffers to store the rectangles in the plane-sweep process.

4.2 Simulation Results

Tables 1 and 2 show the join processing time in Rotem's and Brinkhoff et al.'s algorithms, respectively. From these tables, we observe that Brinkhoff et al.'s algorithm needs longer join processing time than Rotem's algorithm. Tables 3 and 4 show the number

N \ p	0.3	0.4	0.5	0.6	0.7	0.8
1000	15	17	16	16	11	16
2000	27	33	33	33	33	33
3000	60	55	66	60	60	66
4000	77	88	83	83	88	88
5000	99	99	110	104	116	110
6000	116	126	126	121	126	138
7000	137	138	143	143	149	159
8000	176	165	154	159	148	171
9000	181	176	176	182	193	175
10000	198	176	175	220	186	198

Table 1: The join processing time (*msec*) in Rotem's algorithm

N \ p	0.3	0.4	0.5	0.6	0.7	0.8
1000	17	22	22	22	22	22
2000	82	82	77	83	83	88
3000	159	154	159	159	165	159
4000	242	242	236	236	241	241
5000	335	335	335	341	335	340
6000	451	451	451	456	450	456
7000	510	599	599	610	599	604
8000	763	758	769	753	830	747
9000	967	928	1022	1022	994	977
10000	1252	1257	1263	1176	1225	1264

Table 2: The join processing time (*msec*) in Brinkhoff et al.'s algorithm

N \ p	0.3	0.4	0.5	0.6	0.7	0.8
1000	2204	2293	2350	2461	2576	2738
2000	7069	7230	7567	7728	7970	8170
3000	13777	14184	14512	14784	15072	15221
4000	21933	22562	22882	23527	23822	24311
5000	31629	31872	32454	32932	33700	33790
6000	42222	42933	43323	43852	44553	44853
7000	53565	54860	55294	56057	56892	57361
8000	66625	67175	67804	69357	69775	71087
9000	80476	81176	82371	83391	84257	85462
10000	96016	96607	97944	99184	100341	103180

Table 3: The number of comparisons in Rotem's algorithm

N \ p	0.3	0.4	0.5	0.6	0.7	0.8
1000	2539	2600	2680	2786	2942	3129
2000	9415	9726	9980	10276	10436	10631
3000	20722	21199	21550	21728	22236	22024
4000	36659	37468	37709	38524	38456	39382
5000	57212	57539	58279	58054	59065	59542
6000	81850	82281	83050	83377	84073	84406
7000	110592	111764	113096	113225	113780	113880
8000	143767	145336	145713	147592	147692	148531
9000	181957	182860	183666	185674	185993	187285
10000	226047	225224	226583	228857	231379	232271

Table 4: The number of comparisons in Brinkhoff et al.'s algorithm

of comparisons in Rotem's and Brinkhoff et al.'s algorithms, respectively. From these tables, we observe that Brinkhoff et al.'s algorithm needs a larger number of comparison than Rotem's algorithm. Tables 5 and 6 show the number of buffers used in Rotem's and Brinkhoff et al.'s algorithms, respectively. From these tables, we observe that Brinkhoff et al.'s algorithm needs a smaller number of buffers than Rotem's algorithm.

Consequently, from our simulation study, we conclude that Brinkhoff et al.'s algorithm needs longer join processing time, larger number of comparisons and a smaller number of buffers than Rotem's algorithm. Moreover, both the algorithms need sorting process first. For the index structure part, Rotem's algorithm creates join index, while Brinkhoff et al.'s algorithm is

N p	0.3	0.4	0.5	0.6	0.7	0.8
1000	11	10	10	10	13	11
2000	15	15	17	18	17	16
3000	20	21	21	22	19	21
4000	24	25	25	26	28	25
5000	27	31	25	27	28	29
6000	33	30	33	31	37	33
7000	35	38	37	36	36	41
8000	38	40	43	38	39	45
9000	45	47	46	46	39	45
10000	49	47	45	47	49	51

Table 5: The number of buffers used in Rotem's algorithm

N p	0.3	0.4	0.5	0.6	0.7	0.8
1000	7	6	7	9	7	7
2000	10	11	13	12	12	11
3000	14	15	13	14	14	15
4000	17	19	17	19	17	20
5000	21	19	22	20	23	21
6000	26	23	24	23	26	22
7000	26	28	28	26	26	27
8000	27	29	29	29	27	29
9000	31	32	32	31	32	31
10000	34	36	33	35	38	34

Table 6: The number of buffers used in Brinkhoff et al.'s algorithm

based on R-tree.

5. CONCLUSION

In this paper, we have corrected the errors in Rotem's overlap detection algorithm. Moreover, we have studied the performance of Rotem's and Brinkhoff et al.'s algorithms for *overlap* detection by simulation. From our simulation study, we have shown that that Brinkhoff et al.'s algorithm needs longer join processing time, larger number of comparisons and a smaller number of buffers than Rotem's algorithm. How to extend those overlap detection algorithms for other region relationships, like *meet, disjoin, cover, contain and equal*, is the possible future research direction.

References

[1] E. Bertino and B. C. Ooi, "The Indispensability of Dispensable Indexes," *IEEE Trans. on Knowledge and Data engineering,* Vol. 11, No. 1, pp. 17-27, Jan./Feb. 1999.

[2] Thomas Brinkhoff, Hans-Peter Kriegel and Bernhard Seeger, "Efficient Processing of Spatial Joins Using R-Trees," *in Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pp. 237-246, 1993.

[3] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider and Bernhard Seeger, "Multi-Step Processing of Spatial Joins," *in Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pp. 197-208, 1994.

[4] Oliver Gunther, "Efficient Computation of Spatial Joins," *Proc. of Int. Conf. on Data Engineering*, pp. 50-59, 1993.

[5] Erik G. Hoel and Hanan Samet, "Benchmarking Spatial Join Operations with Spatial Output," *Proc. of VLDB Conf.*, pp. 606-618, 1995.

[6] Nick Koudas and Kenneth C. Sevcik, "Size Separation Spatial Join." *in Proc. of ACM SIGMOD Int. Conf. on Management of Data.* pp. 324-335, 1997.

[7] Ming-Ling Lo and Chinya V. Ravishanker, "The Design and Implementation of Seeded Trees: An Efficient Method for Spatial Joins," *IEEE Trans. on Knowledge and Data Engineering*, Vol. 10, No. 1, pp. 136-152, Jan./Feb. 1998.

[8] Ming-Ling Lo and Chinya V. Ravishanker, "Spatial Hash-Joins," *in Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pp. 247-258, 1996.

[9] Nikos Mamoulis and Dimitris Papadias, "Integration of Spatial Join Algorithms for Processing Multiple Inputs," *Proc. of ACM SIGMOD'99,* pp. 1-12, 1999.

[10] Jack Orenstein, "An Algorithm for Computing the Overlay of K-Dimensional Spaces," *in Advances in Spatial Databases - 2nd Symp., SSD'91*, pp. 381-400, 1991.

[11] Jignesh M. Patel and David J. DeWitt, "Partition Based Spatial-Merge Join," *in Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pp. 259-270, 1996.

[12] Doron Rotem, "Spatial Join Indices," *in Proc. of Int. Conf. on Data Engineering*, pp. 500-509, 1991.

[13] Kenneth C. Sevcik and Nikos Koudas, "Filter Trees for Managing Spatial Data Over a Range of Size Granularities," *Proc. of VLDB Conf.*, pp. 16-27, 1996.

[14] S. Shekhar, S. Chawla, S. Ravada, A. Fetterer, X. Liu and C. T. Lu, "Spatial Databases – Accomplishments and Research Needs," *IEEE Trans. on Knowledge and Data Engineering,* Vol. 11, No. 1, pp. 45-55, Jan./Feb. 1999.

[15] J. W. Song, K. Y. Whang, Y. K. Lee, M. J. Lee and S. W. Kim, "Spatial Join Processing Using Corner Transformation." *IEEE Trans. on Knowledge and Data Engineering,* Vol. 11, No. 4, pp. 688-695, July/Aug. 1999.