Algorithms for Joining R-trees and Linear Region Quadtrees

Article · February 2002

Source: CiteSeer

CITATIONS
6

READS
12

2 authors:

Michael Vassilakopoulos
University of Thessaly
116 PUBLICATIONS **1,006** CITATIONS

SEE PROFILE

Yannis Manolopoulos
Aristotle University of Thessaloniki
526 PUBLICATIONS **9,354** CITATIONS

SEE PROFILE

Algorithms for Joining R-trees and Linear Region Quadtrees

A. Corral[1] M. Vassilakopoulos[2] Y. Manolopoulos

Data Engineering Lab
Department of Informatics
Aristotle University
54006 Thessaloniki, Greece

acorral@ualm.es *mvass@computer.org* *manolopo@csd.auth.gr*

Abstract. The family of R-trees is suitable for storing various kinds of multidimensional objects and is considered an excellent choice for indexing a spatial database. Region Quadtrees are suitable for storing 2-dimensional regional data and their linear variant is used in many Geographical Information Systems for this purpose. In this report, we present five algorithms suitable for processing join queries between these two successful, although very different, access methods. Two of the algorithms are based on heuristics that aim at minimizing I/O cost with a limited amount of main memory. We also present the results of experiments performed with real data that compare the I/O performance of these algorithms.

Index terms: Spatial databases, access methods, R-trees, linear quadtrees, query processing, joins.

1 Introduction

Several spatial access methods have been proposed in the literature for storing multi-dimensional objects (e.g. points, line segments, areas, volumes, and hypervolumes). These methods are classified in one of the following two categories according to the principle guiding the hierarchical decomposition of data regions in each method.

- Data space hierarchy: a region containing data is split (when, for example, a maximum capacity is exceeded) to sub-regions which depend on these data only (for example, each of two sub-regions contains half of the data)
- Embedding space hierarchy: a region containing data is split (when a certain criterion holds) to sub-regions in a regular fashion (for example, a square region is always split in four quadrant sub-regions)

[1] Research performed under the European Union's TMR Chorochronos project, contract number ERBFMRX-CT96-0056 (DG12-BDCN), while on leave from the University of Almeria, Spain.

[2] Post-doctoral Scholar of the State Scholarship-Foundation of Greece.

The book by Samet [20] and the recent survey by Gaede and Guenther [6] provide excellent information sources for the interested reader.

A representative of the first principle that has gained significant appreciation in the scientific and industrial community is the R-tree. There are a number of variations of this structure all of which organize multidimensional data objects by making use of the Minimum Bounding Rectangles (MBRs) of the objects. This is an expression of the "conservative approximation principle". This family of structures is considered an excellent choice for indexing various kinds of data (like points, polygons, 2-d objects, etc) in spatial databases and Geographical Information Systems.

A famous representative of the second principle is the Region Quadtree. This structure is suitable of storing and manipulating 2-dimensional regional data (or binary images). Moreover, many algorithms have been developed based on Quadtrees [20]. The most widely known secondary memory alternative of this structure is the Linear Region Quadtree [21]. Linear Quadtrees have been used for organizing regional data in Geographical Information Systems [19].

These totally different families of popular data structures can co-exist in a Spatial Information System. For example, in his tutorial in the same conference, Sharma [23] refers to spatial and multimedia extensions to the Oracle 8i server that are based on the implementation of a linear quadtree and a modified R*-tree. Each of these structures can be used for answering a number of very useful queries. However, the processing of queries that are based on both structures has not been studied in the literature. In this report, we present a number of algorithms that can be used for processing joins between the two structures.

For example, the R-tree data might be polygonal objects that represent swimming and sun-bathing sites and the Quadtree data a map, where black color represents a decrease of ozon and white color represents ozon safe areas. A user may ask which sites suffer from the ozon problem. The major problem for answering such a query is to make use of the space hierarchy properties of each of the structures, so that not to transfer in main memory irrelevant data, or not to transfer the same data many times. Three of the proposed algorithms are simple and suffer from such unnecessary transfers, when the buffering space provided is limited. We also propose another two more sophisticated algorithms that deal with this problem by making use of heuristics and achieve good performance with a limited amount of main memory.

The organization of the paper is as follows. In Section 2, we present in brief the families of R-trees and Linear Region Quadtrees. In Section 3, we review join processing in spatial databases. In Section 4, we present the algorithms that process R-Quad Joins. More specifically, in Subsections 4.1 to 4.3 we present the three simple algorithms, in Subsection 4.4 our heuristics and the buffering scheme used and in Subsections 4.5 and 4.6 the two sophisticated algorithms. In Section 5, we present our experimental setting and some results of experiments we performed with real data. These experiments compare the I/O performance of the different algorithms. In Section 6, we summarize the contribution of this work and discuss issues that require further research in the future.

2 The Two Structures

2.1 R-trees

R-trees are hierarchical data structures based on B^+-trees. They are used for the dynamic organization of a set of k-dimensional geometric objects representing them by the minimum bounding k-dimensional rectangles (in this paper we focus on 2 dimensions). Each node of the R-tree corresponds to the minimum rectangle that bounds its children. The leaves of the tree contain pointers to the objects of the database, instead of pointers to children nodes. The nodes are implemented as disk pages.

It must be noted that the rectangles that surround different nodes may be overlapping. Besides, a rectangle can be included (in the geometrical sense) in many nodes, but can be associated to only one of them. This means that a spatial search may demand visiting of many nodes, before confirming the existence or not of a given rectangle.

The rules obeyed by the R-tree are as follows. Leaves reside on the same level. Each leaf contains pairs of the form (R, O), such that R is the minimum rectangle that contains spatially object O. Every other node contains pairs of the form (R, P), where P is a pointer to a child of the node and R is the minimum rectangle that contains spatially the rectangles contained in this child. An R-tree of class (m, M) has the characteristic that every node, except possibly for the root, contains between m and M pairs, where $m \leq \lceil M/2 \rceil$. The root contains at least two pairs, if it is not a leaf. Figure 1 depicts some rectangles on the right and the corresponding R-tree on the left. Dotted lines denote the bounding rectangles of the subtrees that are rooted in inner nodes.

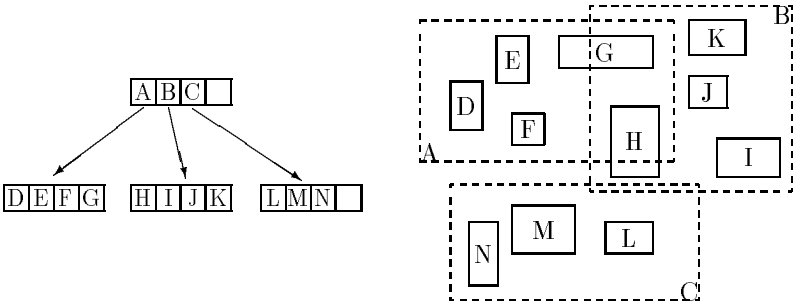

Fig. 1. An example of an R-tree

Many variations of R-trees have appeared. The most important of theses are packed R-trees [18], R^+trees [22] and R*-trees [2]. The R*-tree does not have the limitation for the number of pairs of each node and follows a node split technique that is more sophisticated than that of the simple R-tree. It is considered the most efficient variant of the R-tree family and, as far as searches are concerned,

it can be used in exactly the same way as simple R-trees. This paper refers to simple R-trees or to R*-trees.

2.2 Region Quadtrees

The Region Quadtree is the most popular member in the family of quadtree-based access methods. It is used for the representation of binary images, that is $2^n \times 2^n$ binary arrays (for a positive integer n), where a 1 (0) entry stands for a black (white) picture element. More precisely, it is a degree four tree with height n, at most. Each node corresponds to a square array of pixels (the root corresponds to the whole image). If all of them have the same color (black or white) the node is a leaf of that color. Otherwise, the node is colored gray and has four children. Each of these children corresponds to one of the four square sub-arrays to which the array of that node is partitioned. We assume here, that the first (leftmost) child corresponds to the NW sub-array, the second to the NE sub-array, the third to the SW sub-array and the fourth (rightmost) child to the SE sub-array. For more details regarding Quadtrees see [20]. Figure 2 shows an 8×8 pixel array and the corresponding Quadtree. Note that black (white) squares represent black (white) leaves, while circles represent gray nodes.

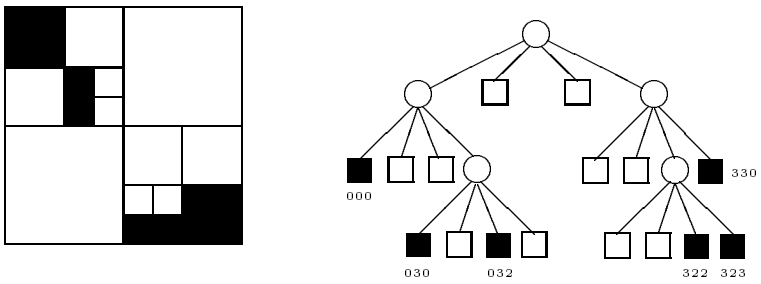

Fig. 2. An image and the corresponding Region Quadtree

Region Quadtrees, as presented above, can be implemented as main memory tree structures (each node being represented as a record that points to its children). Variations of Region Quadtrees have been developed for secondary memory. Linear Region Quadtrees are the ones used most extensively. A Linear Quadtree representation consists of a list of values, where there is one value for each black node of the pointer-based Quadtree. The value of a node is an address describing the position and size of the corresponding block in the image. These addresses can be stored in an efficient structure for secondary memory (such as a B-tree or one of its variations). There are also variations of this representation where white nodes are stored too, or variations which are suitable for multicolor images. Evidently, this representation is very space efficient, although it is not suited to many useful algorithms that are designed for pointer-based Quadtrees.

The most popular linear implementations are the FL (Fixed Length), the FD (Fixed length – Depth) and the VL (Variable length) linear implementations [21].

In the FL implementation, the address of a black Quadtree node is a code-word that consists of n base-5 digits. Codes 0, 1, 2 and 3 denote directions NW, NE, SW and SE, respectively, while code 4 denotes a do-not-care direction. If the black node resides on level i, where $n \geq i \geq 0$, then the first $n - i$ digits express the directions that constitute the path from the root to this node and the last i digits are all equal to 4. In the FD implementation, the address of a black Quadtree node has two parts: the first part is a code-word that consists of n base-4 digits. Codes 0, 1, 2 and 3 denote directions NW, NE, SW and SE, respectively. This code-word is formed in a similar way to the code-word of the FL-linear implementation with the difference that the last i digits are all equal to 0. The second part of the address has $\lceil log_2(n+1) \rceil$ bits and denotes the depth of the black node, or in other words, the number of digits of the first part that express the path to this node. In the VL implementation the address of a black Quadtree node is a code-word that consists of at most n base-5 digits. Code 0 is not used in addresses, while codes 1, 2, 3 and 4 denote one of the four directions each. If the black node resides on level i, where $n \geq i \geq 0$, then its address consists of $n - i$ digits expressing the directions that constitute the path from the root to this node. The depth of a node can be calculated by finding the smallest value equal to a power of 5 that gives 0 quotient when the address of this node is divided (using integer division) with this value.

In the rest of this paper we assume that Linear Quadtrees are represented with FD-codes stored in a B^+-tree (this choice is popular in many applications). The choice of FD linear representation, instead of the other two linear representations, is not accidental. The FD linear representation is made of base-4 digits and is thus easily handled using two bits for each digit. Besides, the sorted sequence of FD linear codes is a depth-first traversal of the tree. Since internal and white nodes are omitted, sibling black nodes are stored consecutively in the B^+-tree or, in general, nodes that are close in space are likely to be stored in the same or consecutive B^+-tree leaves. This property helps at reducing the I/O cost of join processing. Since in the same quadtree two black nodes that are ancestor and descendant cannot co-exist, two FD linear codes that coincide at all the directional digits cannot exist neither. This means that the directional part of the FD-codes is sufficient for building B^+-trees at all the levels. At the leaf-level, the depth of each black node should also be stored so that images are accurately represented. In Figure 2 you can see the directional code of each black node of the depicted tree.

3 Spatial Join Processing

In Spatial Databases and Geographical Information Systems there exists the need for processing a significant number of different spatial queries. For example, such queries are: nearest neighbor finding, similarity queries [16], window queries, content based queries [24], or spatial joins of various kinds. A spatial join consists

in testing every possible pair of data elements belonging to two spatial data sets against a spatial predicate. This predicate might be *overlap, distance within, contain, intersect, etc.* In this paper we mainly focus on the *intersection spatial join* (the most widely used join type), or on spatial joins which are processed in the same way as the intersection join.

There have been developed various methods for processing spatial joins for spatial data using approximate geometry [13, 14], two R-trees [3], PMR quadtrees [5], seeded trees when one [9] or none [10] of the data sets does not have a spatial index, spatial hashing [1, 11, 15], or sort merge join [8].

In this paper, we make the assumption that our spatial information system keeps non-regional data in R-trees or R*-trees and regional data in Linear Region Quadtrees, while users pose queries that involve both these two kinds of data. For example, the non-regional data might be cities and the regional data a map where black represents heavy clouds and white rather sunny areas. The user is very likely to ask which cities are covered with clouds.

Most spatial join processing methods are performed in two steps. The first step, which is called filtering, chooses pairs of data that are likely to satisfy the join predicate. The second step, which is called refinement, examines the predicate satisfaction for all these pairs of data. The algorithms presented in this paper, focus on the function of the filtering step and show how a number of pairs of the form (Quadtree block, MBR of object) can be produced (the two members of each pair produced intersect).

4 Join Algorithms

Before join processing, the correspondence of the spaces covered by the two structures must be established. A level-n Quadtree covers a quadrangle with $2^n \times 2^n$ pixels, while an R-tree covers a rectangle that equals the MBR of its root. Either by asking the user for input, or by normalizing the larger side of the R-tree rectangle in respect to 2^n, the correspondence of spaces may be determined. After this action, the coordinates used in the R-tree are always transformed to Quadtree pixel locations.

Joining of the two structures can be done with very simple ways, if it is ignored that both structures are kept in disk pages as multiway trees. These ways fall in two categories: either we scan the entries of the B^+-tree and perform window queries in the R-tree, or we scan the entries of the R-tree and perform window queries in the B^+-tree. More specifically, we designed and implemented the following three simple algorithms.

4.1 B^+ to R Join

- Descend the B^+-tree from the root to its leftmost leaf.
- Access sequentially (in increasing order) the FDs present in this leaf and for each FD perform a range search in the R-tree (reporting intersections of this FD and MBRs of leaves).

– By making use of the horizontal inter-leaf pointer, access the next B^+-tree leaf and repeat the previous step.

This algorithm may access a number of FDs (and the leaves in which they reside) that do not intersect with any data elements stored in the R-tree. Moreover, this algorithm is very probable to access a number of R-tree nodes several times.

4.2 R to B^+ Join with sequential FD access

– Traverse recursively the R-tree, accessing the MBRs in each node in order of appearence within the node.
– For the MBR of each leaf accessed, search in the B^+-tree for the FD of the NW corner of this MBR, or one of its ancestors.
– Access sequentially (in increasing order) the FDs of the B^+-tree until the FD of the SE corner of this MBR, or one of its ancestors is reached (reporting intersections of FDs and this MBR).

This algorithm may perform unnecessary accesses in both trees, while multiple accesses in B^+-tree leaves are very probable. The unnecessary accesses in the B^+-tree result from the sequential access of FDs. The following algorithm is a variation that deals with B^+-tree accessing differently.

4.3 R to B^+ Join with maximal block decomposition

– Traverse recursively the R-tree, accessing the MBRs in each node in order of appearence within the node.
– For the MBR of each leaf accessed, decompose this MBR in maximal quad-tree blocks.
– For each quadblock, search in the B^+-tree for the FD of the NW corner of this quadblock, or one of its ancestors.
– Access sequentially (in increasing order) the FDs of the B^+-tree until the FD of the SE corner of this quadblock, or one of its ancestors is reached (reporting intersections of FDs and the respective MBR).

Although this algorithm saves many unnecessary FDs accessed, each search for a quadblock descends the tree. Nevertheless, the same intersection may be reported more than once. To eliminate duplicate results, a temporary list of intersections for the current leaf is maintained.

4.4 Heuristics and Buffering Scheme

In order to overcome the unnecessary and/or duplicate accesses of the previous algorithms, we propose a number of heuristics/rationales that focus on the opposite direction, that of increasing I/O performance.

- heuristic 1: Process small enough parts of the R-tree space so that the join processing of each part can be completed (in most cases) with the limited number of FD-codes that can be kept in main memory. At the presented form of the algorithm, each of these parts is a child of the root.
- heuristic 2: Process the children of an R-tree node in order that is close to the order in which the FD-codes (quadtree sub-blocks) are transferred in main memory. This order, called FD-order, is formed by sorting MBRs by the FD-code that corresponds to their NW corner.
- heuristic 3: While processing a part of the R-tree space, keep in memory only the FD-codes that may be needed at a later stage, drop all other FD-codes and fill up buffer with FD-codes that are needed but were not transferred in memory due to the buffer limit.
- heuristic 4: Use a buffer scheme for both trees that reduces the need to transfer in memory multiple times the same disk pages (explained in detail below).

A buffering scheme that obeys Heuristic 4 is presented graphically in Figure 3. In detail, this scheme is as follows.

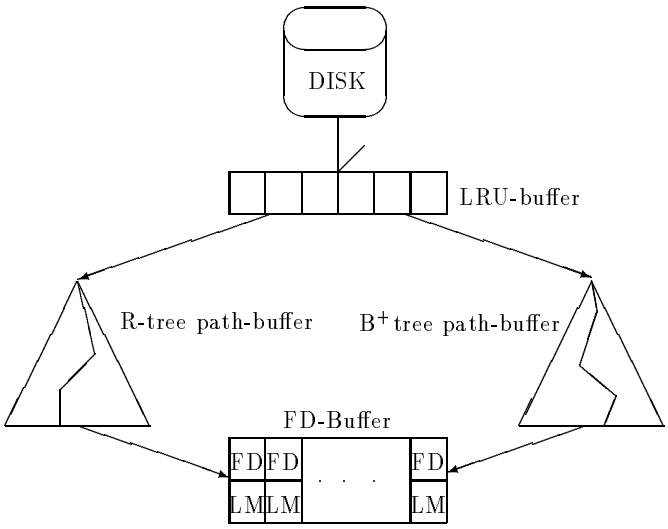

Fig. 3. The buffering scheme

- There is a path-buffer for R-tree node-pages (with number of pages equal to the height of the R-tree). However, the buffer pages of the R-tree buffer are larger than the actual R-tree disk pages, because for each entry (each MBR) an extra point is kept. This point is called START and expresses the pixel where processing of the relevant MBR has stopped (a special value, named MAX, specifies that processing of that MBR has been completed).

This means that during transfers from disk to memory and the opposite an appropriate transformation of the page contents needs to be made.
- There is a path-buffer for B^+-tree node-pages (with number of pages equal to the height of the B^+-tree).
- It is assumed that the operating system keeps a large enough LRU-buffer for disk reads. The same assumption was made in [3]. This buffer is used for pages belonging in paths related to the current path that are likely to be accessed again in subsequent steps of the algorithm.
- The last buffer, called FD-buffer, is not a page buffer but one that holds the FDs needed for the processing of the current R-tree part. Each entry in this buffer contains also a level mark (LM), that is a number that expresses the level of the current R-tree path at which and below which the related FD might be needed for join processing. The size of this buffer is important for the I/O efficiency of the sophisticated algorithms.

Note that, the three simple algorithms described above can be easily made more effective by using a path-buffer and an LRU-buffer for each tree. As will be demonstrated in the experimentation section, by using adequately large LRU-buffers, the performance of the simple algorithms is comparable to that of the sophisticated ones.

The searching method used in the algorithms is as follows. When, for a point belonging in an R-tree MBR, the existence of a Linear Quadtree code that covers this point (its block contains this point) needs to be determined, we search the B^+-tree for the maximum FD-code M that is smaller than or equal to the FD-code P of the pixel related to this point. If $M = P$ and $\text{depth}(M) = \text{depth}(P)$, then this specific black pixel exists in the Quadtree. If $M \leq P$, $\text{depth}(M) < \text{depth}(P)$ and the directional codes of M and P coincide in the first $\text{depth}(M)$ bits, then M is a parent of P (it represents a block that contains P). This searching method is used in lines 32 and 47 of the "One level FD-buffer join" and "Many levels FD-buffer join" algorithms, respectively. In the following, these two algorithms, which are designed according to the above heuristics, are presented.

4.5 One level FD-buffer join

In very abstract terms, this algorithm works as follows:

- Process the children of the R-tree root in FD-order.
- Read as many FDs as possible for the current child and store them in FD-buffer.
- Call recursively the Join routine for this child.
- When the Join routine returns, empty the FD-buffer and repeat the previous two steps until the current child has been completely checked.
- Repeat for the next child of the root.

The Join routine for a node works as follows:

- If the node is a leaf, check intersections and return.

– If not (this is a non-leaf node), for each child of the node that has not been examined in relation to the FDs in FD-buffer, call the Join routine recursively.

In pseudo-code form this algorithm is as follows:

```
01 insert R-tree root in path-buffer;
02 for every MBR x in R-tree root
03     START(x) := NW-corner-of(x);
04 order MBRs in R-tree root according to FD of their START;
05 for every MBR x in R-tree root, in FD-order
06     while START(x) < MAX begin
07     |   read-FDs-in-buffer(x);
08     |   R-Quad-Join(node of x);
09     |   remove every FD from FD-buffer;
10     end;

11 Procedure R-Quad-Join(Z: R-tree node);
12 begin
13     if Z is not in path-buffer
14         insert Z in path-buffer;
15     if Z is internal then begin
16     |   for every MBR x in Z
17     |       START(x) := NW-corner-of(x);
18     |   order MBRs in Z according to FD of their START;
19     |   for every MBR x in Z, in FD-order
20     |       if START(x) < START(MBR of Z) begin
21     |       |   START(x) := first pixel of x after the last FD accessed,
       |       |   or MAX (if no such pixel exists);
22     |       |   if START(x) ≠ MAX or at least one FD in FD-buffer intersects x
23     |       |       R-Quad-Join(node of x);
24     |       end;
25     end
26     else
27         check and report possible intersection of MBR of Z and
           every FD in FD-buffer;
28 end;

29 Procedure read-FDs-in-buffer(Z: MBR);
30 begin
31     while START(Z) < MAX and FD-buffer not full begin
32     |   search in QuadTree for FD f covering START(Z) or
       |   for the next FD (in FD-order);
33     |   if no FD was accessed
34     |       f := MAX;
35     |   if f intersects Z
36     |       store f in FD-buffer;
37     |   START(Z) := first pixel of Z after f, or MAX (if no such pixel exists);
38     end;
39 end;
```

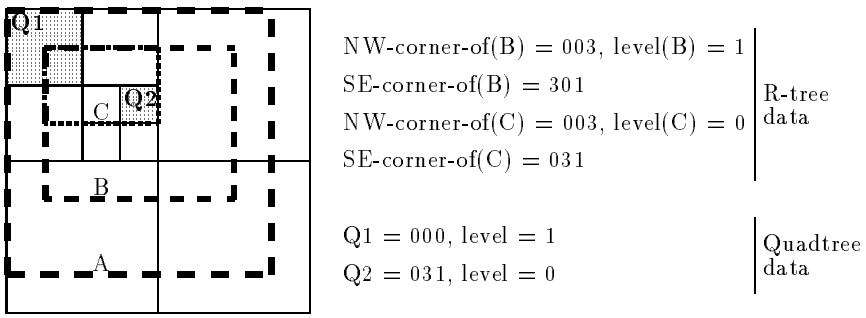

Fig. 4. An example showing Quadtree (thin lines) and R-tree (thick dashed lines) subdivision of space.

In Figure 4, a simple example demonstrating the simultaneous subdivision of space by a Quadtree for an 8×8 image (thin lines) and part of an R-tree (thick dashed lines) is depicted. The Quadtree contains two black quadrangles (the two dotted areas), Q1 and Q2. The MBR of the root of the R-tree is rectangle A. However, in Figure 4, only one of the children of this root is depicted, the one with MBR B. Moreover, a child of B is depicted, the one with MBR C. We suppose that C is a leaf of the R-tree and it has no intersection with the other children of B (not shown in the figure). Consider a trivial situation, where the FD-buffer of the algorithm presented above can host one FD-code and the corresponding level mark (LM), only. Let's trace the most important steps of a partial run of this algorithm for this specific case.

1	START(B) = 003;	(1.3)
2	read-FDs-in-buffer(B);	(1.7)
3	search reaches Q1	(1.32)
4	FD-buffer: f = 000	(1.36)
5	START(B) = 012;	(1.37)
6	R-Quad-Join (B);	(1.8)
7	Since B is internal...	(1.15)
8	START(C) = 003;	(1.17)
9	C is the first MBR in B	(1.18)
10	Since START(C) < START(B)...	(1.20)
11	START(C) = 012;	(1.21)
12	R-Quad-Join(C);	(1.23)
13	Since C is a leaf...	(1.15)
14	report intersection of C and Q1;	(1.27)
15	R-Quad-Join(C) returns;	(1.28)
16	Other children of B are processed	(1.19)
17	R-Quad-Join(B) returns;	(1.28)

18	remove 000 from FD-buffer;	(1.9)
19	read-FDs-in-buffer(B);	(1.7)
20	search reaches Q2	(1.32)
21	FD-buffer: f = 031	(1.36)
22	START(B) := 032;	(1.37)
23	R-Quad-Join (B);	(1.8)
24	Since B is internal...	(1.15)
25	START(C) = 003;	(1.17)
26	C is the first MBR in B	(1.18)
27	Since START(C) < START(B)...	(1.20)
28	START(C) = MAX;	(1.21)
29	R-Quad-Join(C);	(1.23)
30	Since C is a leaf...	(1.15)
31	report intersection of C and Q2;	(1.27)
32	R-Quad-Join(C) returns;	(1.28)
33	Other children of B are processed	(1.19)
34	R-Quad-Join(B) returns;	(1.28)
35	remove 031 from FD-buffer;	(1.9)

The run of the algorithm continues with the next loop for B. The interested reader can trace the same example with FD-buffer size equal to 2 and note the differences. This algorithm only fetches and releases FDs at the level of the children of the root. For such a case, the LM for each FD is not necessary to be kept in the FD-buffer.

4.6 Many levels FD-buffer join

This algorithm, follows the same basic steps, however, it releases from the FD-buffer the FDs that will no longer be needed in the current phase of the algorithm as soon as possible and fills it up. Again, in very abstract terms, this algorithm works as follows:

- Process the children of the R-tree root in FD-order.
- Read as many FDs as possible for the current child and store them in FD-buffer.
- Call recursively the Join routine for this child.
- When the Join routine returns, repeat the previous two steps until the current child has been completely checked.
- Repeat for the next child of the root.

The Join routine for a node works as follows:

- If the node is a leaf, remove from FD-buffer all FDs that will not be needed in the current phase of the algorithm, check intersections and return.
- If not (this is a non-leaf node), for each child of the node that has not been examined in relation to the FDs in FD-buffer, mark the FDs that only affect the results for this child and call the Join routine recursively.

- When all the children of the node have been examined, reorder them and repeat the previous step until all the children have been examined with the current state of the FD-buffer.
- Remove from FD-buffer all FDs that will not be needed in the current phase of the algorithm and return.

Due to the possibility to remove FDs from FD-buffer at any level, an extra variable, called NEXT, that keeps track of the pixel where fetching of FDs has stopped is needed in this algorithm. In pseudo-code form the algorithm is as follows:

```
01 insert R-tree root in path-buffer;
02 for every MBR x in R-tree root
03     START(x) := NW-corner-of(x);
04 order MBRs in R-tree root according to FD of their START;
05 for every MBR x in R-tree root, in FD-order begin
06 |   NEXT := NW-corner-of(x);
07 |   while START(x) < MAX begin
08 |   |   read-FDs-in-buffer(node of x);
09 |   |   R-Quad-Join(node of x);
10 |   end;
11 end;

12 Procedure R-Quad-Join(Z: R-tree node);
13 begin
14     if Z is not in path-buffer
15         insert Z in path-buffer;
16     if Z is internal node then begin
17     |   for every MBR x in Z
18     |       START(x) := first pixel of x ≥ START(MBR of Z),
       |       or MAX (if no such pixel exists);
19     |   order MBRs in Z according to FD of their START;
20     |   repeat-flag := True;
21     |   while repeat-flag begin;
22     |   |   repeat-flag := False;
23     |   |   for every MBR x in Z, in FD-order
24     |   |       if START(x) ≤ SE-corner-of(last FD accessed) and
       |   |       START(x) ≠ MAX begin
25     |   |       |   repeat-flag := True;
26     |   |       |   if FD-buffer not empty
27     |   |       |       for every f in FD-buffer with LM(f) = level-of(Z)
       |   |       |       such that x is intersected by f and
       |   |       |       SE-corner-of(f) < START(y), ∀ y ≠ x intersected by f
28     |   |       |           LM(f) := level-of(node of x);
29     |   |       |   if at least one FD in FD-buffer intersects x begin
30     |   |       |   |   R-Quad-Join(node of x);
31     |   |       |   |   read-FDs-in-buffer(Z);
32     |   |       |   end
33     |   |       |   else
34     |   |       |       START(x) := first pixel of x after the last FD accessed, or
```

```
    |   |       |       MAX (if no such pixel exists);
35  |   |       end;
36  |   |   order MBRs in Z according to FD of their START;
37  |   end;
38  end
39  else
40      check and report possible intersection of MBR in Z and every
        FD f in FD-buffer, such that SE-corner-of(f) ≥ START(MBR of Z);
41  START(MBR of Z) := first pixel of MBR of Z after the last FD accessed, or
    MAX (if no such pixel exists);
42  remove from FD-buffer every FD f with LM(f) = level-of(Z);
43 end;

44 Procedure read-FDs-in-buffer(Z: R-tree node);
45 begin
46  while NEXT < MAX and FD-buffer not full begin
47  |   search in QuadTree for FD f covering NEXT
    |   or for the next FD (in FD-order);
48  |   if no FD was accessed
49  |       f := MAX;
50  |   if f intersects the active MBR in the top path-buffer node begin
51  |   |   LM(f) := level-of(top path-buffer node) - 1;
52  |   |   while LM(f) > level-of(Z) and f intersects only the active MBR
    |   |   in the next lower path-buffer node
53  |   |       LM(f) := LM(f) - 1;
54  |   |   store f in FD-buffer
55  |   end;
56  |   NEXT := first pixel of the active MBR in the top path-buffer node after f,
    |   or MAX (if no such pixel exists);
57  end;
58 end;
```

5 Experimentation

We performed experiments with Sequoia data from the area of California. The point data correspond to specific country sites, while regional data correspond to three different categories: visible, emitted infrared and reflected infrared spectrums. We performed experiments for all the combinations of point and regional data. The query used for all experiments is the intersection join query between these two kinds of data: "which country sites are covered with colored regions?".

There were many parameters that varied in these experiments. Each pixel of the regional data had a range of 256 values. Each image was converted to black and white by choosing a threshold accordingly so as to achieve a requested black-white analogy. This analogy ranged between 20% and 80%. The images in our experiments were 1024×1024 and 2048×2048 pixel large. The cardinality of the point set was 68764. The page size for both trees was 1024 bytes. Under these conditions, both R and B^+ trees had 4 levels (including the leaf level). The FD-buffer size for the two sophisticated algorithms ranged between 150 and

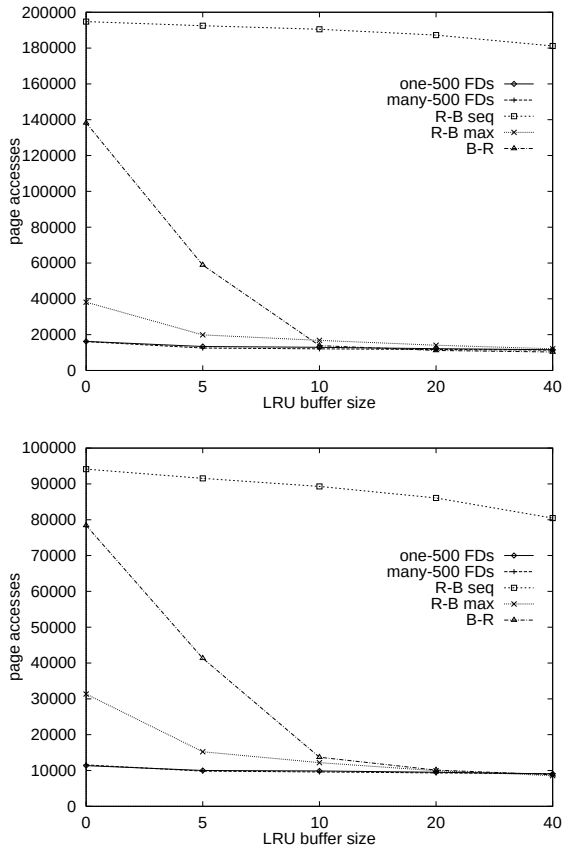

Fig. 5. The performance of the five algorithms as a function of LRU-buffer size for 50% (upper diagram) and 80% (lower diagram) black images.

2500 FDs. The LRU-buffer size for each tree ranged between 0 and 40 pages.

In the following, some characteristic results of the large number of experiments performed for images of 2048×2048 pixels from the visible spectrum are depicted. Note that, since $n = 11$, an FD for such an image requires $2 \times 11 + \lceil log_2(11+1) \rceil = 26$ bits. In the upper (lower) part of Figure 5 the number of disk accesses during join for each of the five algorithms, when images are 50% (80%) black, as a function of LRU-buffer size is shown. More specifically, "one-500 FDs" stands for "One level FD-buffer join" with FD-buffer holding up to 500 FDs, "many-500 FDs" for "Many levels FD-buffer join" with FD-buffer holding up to 500 FDs, "R-B seq" for "R to B^+ Join with sequential FD access", "R-B max" for "R to B^+ Join with maximal block decomposition" and "B-R" for "B^+ to R Join". It is evident that R to B^+ Join with sequential FD access has the worst performance which is not improved as the LRU-buffers size increases.

Nevertheless, R to B^+ Join with maximal block decomposition and B^+ to R Join gets improved and achieves comparable performance to the sophisticated algorithms as the LRU-buffers size increases. The two sophisticated algorithms perform well, even for small LRU-buffers.

To study the situation more closely, in Figure 6 we present performance results for the two simple algorithms that have better performance and two versions (for FD-buffer size equal to 500 and to 1500 FDs) of the sophisticated algorithms. The diagram on the upper (lower) part corresponds to 50% (80%) black images. We can easily see that the sophisticated algorithms perform very well for a wide range of LRU- and FD-buffers sizes, while the two simple algorithms achieve comparable, or even better, performance than the sophisticated ones, when the LRU-buffers size increases.

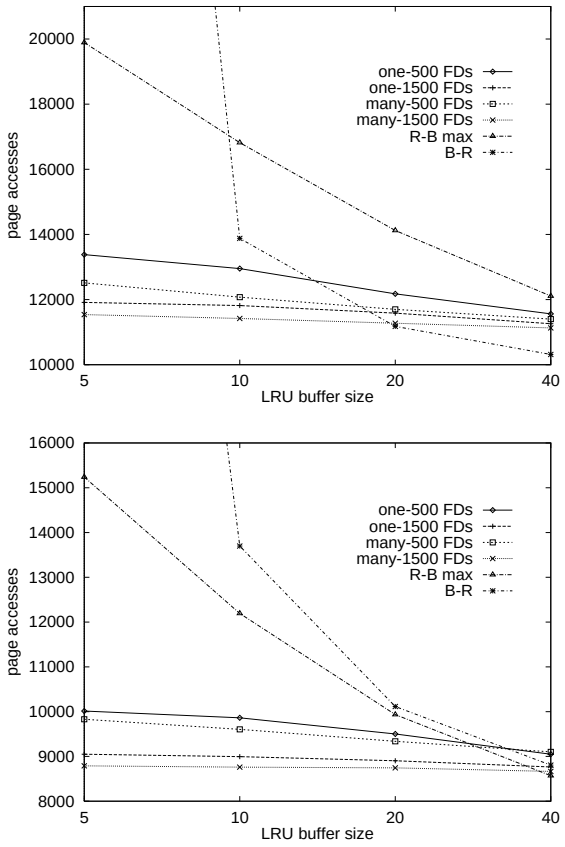

Fig. 6. The performance of two of the simple and two versions of the sophisticated algorithms as a function of LRU-buffer size for 50% (upper diagram) and 80% (lower diagram) black images.

Finally, in Figure 7 the performance of sophisticated algorithms for various kinds of images as a function of a combination of LRU-buffers size and FD-buffers size is depicted. Many levels FD-buffer join performs slightly better than One level FD-buffer join for all cases. Note that the difference gets smaller when FD-buffer size increases.

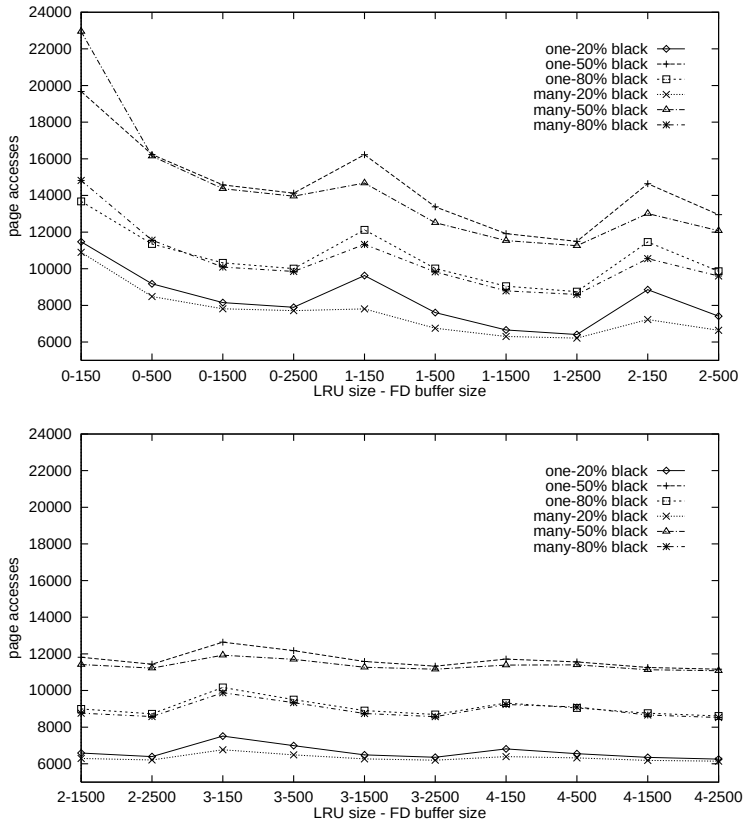

Fig. 7. The performance of the sophisticated algorithms as a function of LRU- and FD-buffers sizes for 20%, 50% and 80% black images.

6 Conclusions

In this report, we presented five algorithms for processing of joins between two popular but different structures used in spatial databases and Geographic Information Systems, R-trees and Linear Region Quadtrees. These are the first

algorithms in the literature that process joins between these two structures. Three of the algorithms are simple, but suffer from unnecessary and/or repeated disk accesses, when the amount of memory supplied for buffering is limited. The other two are more sophisticated and are based on heuristics that aim at minimizing the I/O cost of join processing. That is, they try to minimize the transfer to main memory of irrelevant data or the multiple transfer of the same data. Moreover, we presented results of experiments performed with real data. These experiments investigate the I/O performance of the different join algorithms.

The presented results show that

- better performance is achieved when using the sophisticated algorithms in a system with small LRU-buffer. For example, in a system with many users, where buffer space is used by many processes, the sophisticated algorithms are the best choice. Besides, the sophisticated algorithms are quite stable. That is, their performance is not heavily dependent on the increase of available main memory.
- When there is enough main memory (when the LRU-buffer is big enough), one of the simple algorithms, the B^+ to R Join algorithm, performs very well.

Intuition leads us to believe that the sophisticated algorithms are expected to perform better for data that obey unusual distributions, since they are designed to partially adapt to the data distribution.

The presented algorithms perform only the filtering step of the join. Processing of the refinement step requires the choice and use of Computational Geometry algorithms [12, 17], which is among our future plans in the area of this topic. Each choice should take into account not only worst-case complexity, but expected sizes of data sets, average case complexity and multiplicative constant of complexity as well, for each alternative. In addition, these algorithms could be tested on other kinds of data, e.g. making use of R-trees as a storage medium for region data as well. Moreover, we plan to elaborate the presented heuristics even further and/or examine different policies of page replacement (other than LRU) so as to improve performance, or master the worst case behavior of the join algorithms (when they deal with "pathological" data, deviating from the usual situations in practice). Another route of research would be to examine the parallel processing of this kind of joins, based on ideas presented in [4].

References

1. L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel and J.S. Vitter, "Scalable Sweeping-Based Spatial Join", *Proceedings of the 24th VLDB conference*, New York, 1998, pp. 570-581.
2. N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger, "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles", *Proceedings of the 1990 ACM SIGMOD Conference*, Atlantic City, NJ, pp. 322-331.

3. T. Brinkhoff, H.P. Kriegel and B. Seeger, "Efficient Processing of Spatial Joins Using R-trees", *Proceedings of the 1993 ACM SIGMOD Conference*, Washington, DC, pp. 237-246.
4. T. Brinkhoff, H.P. Kriegel and B. Seeger, "Parallel Processing of Spatial Joins Using R-trees", *International Conference on Data Engineering*, New Orleans, 1996, pp. 258-265.
5. E.G. Hoel and H. Samet, "Benchmarking Spatial Join Operations with Spatial Output", *Proceedings of the 21st VLDB conference*, Zurich, 1995, pp. 606-618.
6. V. Gaede and O. Guenther: "Multidimensional Access Methods", *ACM Computer Surveys*, Vol 30, No 2, 1998, pp. 170-231.
7. A. Guttman: "R-trees - a Dynamic Index Structure for Spatial Searching", *Proceedings of the 1984 ACM SIGMOD Conference*, Boston, MA, pp. 47-57.
8. N. Koudas and K.C. Sevcik, "Size Separation Spatial Join", *Proceedings of the 1997 ACM SIGMOD Conference*, Tuscon, pp. 324-335.
9. M.L. Lo and C.V. Ravishankar, "Spatial Joins Using Seeded Trees", *Proceedings of the 1994 ACM SIGMOD Conference*, Minneapolis, pp. 209-220.
10. M.L. Lo and C.V. Ravishankar, "Generating Seeded Trees From Data Sets", *Proceedings of the 4th International Symposium on Large Spatial Databases*, Portland, ME, 1995, pp. 328-347.
11. M.L. Lo and C.V. Ravishankar, "Spatial Hash-Joins", *Proceedings of the 1996 ACM SIGMOD Conference*, Montreal, Canada, pp. 247-258.
12. K. Mehlhorn, "Data Structures and Algorithms 3: Multi-dimentional Searching and Computational Geometry", *Springer Verlag*, 1984.
13. J.A. Orenstein and F.A. Manola, "PROBE Spatial Data Modeling and Query Processing in an Image Database Application", *IEEE transactions on Software Engineering*, Vol 14, No 5, 1988, pp. 611-629.
14. J.A. Orenstein, "Spatial Query Processing in an Object Oriented Database System", *Proceedings of the 1986 ACM SIGMOD Conference*, pp. 326-336.
15. J.M. Patel and D.J. DeWitt, "Partition Based Spatial Merge-Join", *Proceedings of the 1996 ACM SIGMOD Conference*, Montreal, pp. 259-270.
16. A. Papadopoulos and Y. Manolopoulos, "Similarity Query Processing using Disk Arrays", *Proceedings of the 1998 ACM SIGMOD Conference*, Seattle, pp. 225-236.
17. F.P. Preparata and M.I. Shamos, "Computational Geometry", *Springer Verlag*, 1988.
18. N. Roussopoulos and D. Leifker, "Direct Spatial Search on Pictorial Databases using Packed R-trees", *Proceedings of the 1985 ACM SIGMOD Conference*, Austin, TX, pp. 17-31.
19. H. Samet, C.A. Shaffer, R.C. Nelson, Y.G. Huang, K. Fujimura and A. Rosenfeld, "Recent Developments in Linear Quadtree-based Geographic Information Systems", *Image and Vision Computing*, Vol 5, No 3, 1987, pp. 187-197.
20. H. Samet: "The Design and Analysis of Spatial Data Structures", *Addison-Wesley*, Reading MA, 1990.
21. H. Samet: "Applications of Spatial Data Structures", *Addison-Wesley*, Reading MA, 1990.
22. T. Sellis, N. Roussopoulos and C. Faloutsos: "The R+tree - a Dynamic Index for Multi-Dimensional Objects", *Proceedings of the 13th VLDB conference*, 1987, pp. 507-518.
23. J. Sharma: "Implementation of Spatial and Multimedia Extensions in Commercial Systems", tutorial during the *6th International Symposium on Spatial Databases*, Hong Kong, July 20-23, 1999.

24. M. Vassilakopoulos and Y. Manolopoulos, "Dynamic Inverted Quadtree: a Structure for Pictorial Databases", *Information Systems (special issue on Multimedia)*, Vol 20, No 6, 1995, pp. 483-500.

This article was processed using the LaTeX macro package with LLNCS style