Spatial joins and R-trees[*]

M.G. Martynov
St.-Petersburg university
E-mail: ban@niimm.spb.su

Abstract

One of the most important operations in spatial access needed systems are spatial joins. Using for processing such operations R-tree like structures is intensively studies now. The paper gives contemporary state in this research area and presents effective algorithm for spatial joins which combines both I/O and CPU cost optimization. This algorithm is designed like a multi-step process [4] and uses combination of the algorithms described in the previous paper [12]. Besides that several optimization of the basic R*-tree algorithms and some spatial join heuristics are described to reduce CPU time. The experimental results and analysis of compared methods are presented.

1 Introduction

Spatial database systems and other applications such as CAD [1], computer vision [6], GIS [2], temporal and geometric databases are quickly developed now and gets more attention. This systems store and operate with large volumes of data and require for access methods are able to perform spatial queries in reasonable time. Several methods were proposed [11], including R-tree like ones which preserve spatial relations for indexed objects and use object approximations in their native space.

The most time-intensive type of a spatial query is a spatial join which usually defined as all pairs of intersected objects of two spatial relations. In general more than one relation may be involved into join process and more complex spatial predicates may be used, but in many cases it is possible to decompose such queries in a sequence of some simple steps.

There are wide range of methods for processing traditional joins [13], but they cannot be applied in spatial case efficiently because of not preserving spatial relations. Several methods using spatial indices were proposed [8, 7, 10, 9, 5], including those based on tree-like access methods which usually assume existence indices for both participating data sets. The situation when this assumption is not true is considered in [10] where dynamically constructed index structure uses knowledge of the data sets are proposed to speed up join process.

[*]This work was partially supported by Russian Foundation for Basic Research under grant 95-01-00636, and UrbanSoft Ltd. under contract 35/95.

Proceedings of the International Workshop on Advances in Databases and Information Systems. Moscow, June 27–30, 1995.

This paper investigates performing spatial join on two data sets with pre-existing R*-tree like indices (see [4]). The ability of applying polygons with oriented sides as approximation objects are considered. The designed algorithm has two steps. At the first step all pairs of data objects with intersected description objects are defined. At the second step pairs are filtered and checked to matching join predicate.

Some optimizations of R*-tree algorithms are proposed which is basic for spatial trees.

The paper is organized as follows. In section 2 some optimizations of R*-tree algorithms are described. Spatial join features and proposed algorithms are considered in section 3. Section 4 concludes the paper.

2 Optimizations of R*-tree algorithms

2.1 Local coordinates

One of the fundamental property of a spatial tree is a proximity of spatial objects storing in a single node. This feature may be used for decreasing number of bits needed for object representation by saving them in local coordinates of their nodes.

Thus usual node is a set of entries of the form $(Sobj, Cp)$, where $Sobj$ is a description of spatial object, and Cp is a pointer to a child node. In many cases $Sobj$ may be considered as fixed number of values which are coordinates along predefined linear axis. For example, in R-tree spatial objects are rectangles represented by two values along each axis of n-dimensional space. In other case $Sobj$ may be number of points describing border of the object or its convex hull.

Then modified node will have the following structure: $(fr, Ncp, Nb_1, \ldots, Nb_n, (Sobj', Cp)^*)$, where Ncp is a basis of local coordinates, Nb_i is a number of bits for values of i-th axes, and $Sobj'$ is $Sobj$ in a new coordinates. Every tree page will have flag fr showing what node representation uses — usual or modified. At the beginning tree will have one empty page with usual representation. Spatial objects from entries of modified nodes translates to global coordinates before using in all tree algorithms. Local coordinates are introduced or changes (then recalculation performs) after split for both nodes. Ncp are set to be the center of enclosing object of a node and Nb_i are chosen to be as less as possible to represent all node entries. If after adding new entry node becomes grater than coordinates domain, it should be spread by increment respective Nb_i and recalculating objects coordinates. This process may initiate split.

The described algorithm has following properties:

- It may be easy implemented and has low CPU cost (only additions during translation).

- Recalculations of coordinates are performed not often (when split or coordinates domain overflow), and do not impact CPU cost.
- Performance gain is higher when spatial index has large size and contains small objects.

Performance gain of using this algorithm can be estimated in a following way. Let M be the maximum number of entries in a node, e be the entry length in bits, n_i be the number of objects in i-th level, where $i = 0, 1, \ldots$ from leave level and s be storage utilization coefficient. Then without using local coordinates $n_{i+1} \approx \lceil \frac{n_i}{Ms} \rceil$, and when it is applied

$$n_{i+1} \approx \left\lceil \frac{n_i(e - 4\log_2 \sqrt{\frac{n_i}{M}})}{Ms\,e} \right\rceil .$$

Let us consider the example: $n_0 = 250000$, $M = 50$, $s = 0.7$, $e = 160$. The result is shown in the next table.

i	no l.c.	with l.c.
0	250000	250000
1	7142	6045
2	204	157
3	6	5
4	1	1

The table shows that with local coordinates engaged disk space is reduced by about 1 Mb, and I/O cost for large window queries decrease by about 10% when top three levels of the tree is in main memory, that is rather realistic. M, e and s in example are borrowed from experiments when page size is 1 Kb and entry length 20 bytes.

2.2 Forced reinsert modification

Forced reinsert [3] is R*-tree algorithm performed during the insertion routine to do the tree structure better. It consists in deleting and reinserting entries removed from the center of a node in which split must be performed. These method leads to more exact distribution of the entries and improves storage utilization. During insertion of one data entry forced reinsert may be done only once on every tree level in original algorithm.

The modification is to allow performing forced reinsert on every level fixed number of times. If after one reinsert all entries went to the same node chain of reinsertion is cut and split is performed. Table 1 shows comparison of forced reinsert methods with 0,1,2,3 possible reinsertions on every level. 1 respects to original algorithm. In experiment there was 30000 data objects with mixed uniform distribution.

Mfr	Build	Stor	Qavr
0	3.01	66.01	8.60
1	5.79	69.02	8.14
2	5.88	69.31	8.10
3	5.98	69.17	8.08

Table 1: Forced reinsert methods comparison

In this table Build — is average number of disk accesses during tree construction, Stor — is storage utilization, Qavr — is average number of disk accesses through all queries.

This modification gives slightly better performance, but its implementation is very easy, and gain may arise with uneven data and bad distribution.

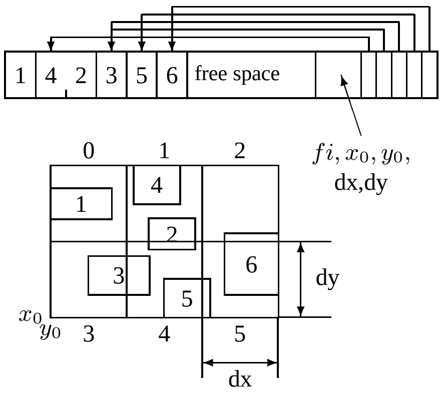

Figure 1: Grid index example

2.3 Local temporary grid index

Free space of a node may be used to speed up choosing insertion path. For this purpose local temporary grid index was designed. It is created after split every time and engages last bytes of a node, which has the following structure $(k, e_1, e_2, \ldots, e_k, —, fi, x_0, y_0, dx, dy, b_1, \ldots, b_m)$, where e_i usual entries, fi — is flag showing weather temporary index are used, x_0, y_0 are coordinates of left-bottom corner of a bounding object of a node, dx, dy are x and y sizes of the one cell, b_i — are beginnings of the groups of entries which centers get into the same cell of the grid. The figure 1 shows example of 3×2 grid index for node with 6 entries.

When index are created at first x_0, y_0, dx, dy are defined, then entries are sorted to groups according their centers and b_i are set. During node growth cell parameters are not changed. When one child node splits new entry are added to our node to appropriate group.

Insertion of a new data object in the tree includes choosing insertion path. It assumes evaluation 'distance' between data object and all objects of the processed node. If we have local grid index we first define cells covered with enhanced by factor δ data object, and then we only check entries in chosen cells.

Method	Time	Stor	Qavr
0	100	68.4	8.1
1	83.3	68.4	8.3
2	79.6	69.2	8.7
3	78.0	69.5	8.7
4	77.5	70.7	8.5
5	77.3	70.7	8.4

Table 2: Local grid index testing

The table 2 contains experimental results of testing 5 methods: 0 — without local index, 1 — grid 2×2, 2 — grid 3×3, 3 — grid 4×4, 4 — grid 4×4 and one of split entries are regrouped, 5 — grid 4×4 and both of split entries are regrouped. In all methods $\delta = 0.3$. The other parameters are as in previous subsection. Time depicts CPU time consumption in relative units.

Proposed algorithm has following properties:

- Local index engages little space of the node and almost always works.
- It is not often recalculated, especially in the top tree levels.
- The lager page size the higher advantage of the method is.

3 Spatial join

Spatial join is one of the non-trivial and time-intensive operations in spatial DBMs from CPU and I/O points of view. In compare with window and point queries spatial join involves into processing several relations and in general case has superlinear execution time in the number of the objects.

In this section processing of spatial join using spatial trees are considered. At the beginning general features of such processing and its realization for complex spatial objects in [5, 4] will be briefly described. Then development of the ideas of previous work will be presented.

3.1 Spatial join features

We will consider the situation when we have two sets of spatial objects which initially described as a sequence of two-dimensional points. It is assumed that data objects are rather complex i.e. have many points in representation and non-even border. Besides, they may contain some holes inside. It also assumed that for both data sets we have tree index constructed before and need to perform intersection join.

Under this assumptions join algorithm should have two steps:

1. *Filter step* — find all pairs of the objects which approximation objects (rectangles in R-trees) are intersected.
2. *Refinement step* — for those objects check whether they really intersected.

First step algorithm is described in [5]. It concludes in tree matching, beginning from the roots in a depth-first tree traversing order. To decrease CPU time not all objects of two nodes are checked for intersection but only those intersected common area of the nodes. It improves performance by factor between 4.6 and 8.9. Then objects are sorted on one axis, and *plane-sweeping* technique used to further reducing CPU cost. I/O cost are proposed to decrease by choosing schedule of reading pages to manage the external LRU buffer more efficiently. For this purpose *page-pinning* technique based on *degrees* (number of objects of the page of one tree that intersect considered object in the page of another tree) was also used.

Processing of the second step is described in [4] and includes two substeps: *geometric filter* which allows to define quickly for large per sent of pairs if they fulfill join predicate or not, and *exact geometric processor* which checks remainder of the pairs.

Geometric filter uses different internal and external object approximations which intersection tests have no high CPU cost and describe object more precisely.

Exact geometric processor uses TR*-tree (R*-tree in memory with node capacity of 3 to 5) representations of decomposed objects for intersection test. In this case test concludes in using first step algorithm on TR*-trees which described before. There were considered decompositions by convex polygons, by trapezoids, by triangles. Using *plane-sweep algorithm* in exact geometric processor are not considered here due to worse performance.

3.2 Combined spatial join algorithm

In this subsection combined algorithm for first step will be described. It is based on two algorithms from my previous [12] work. First algorithm defines right order of reading pages to minimize I/O cost, and reads each page only once. It stores entries in memory non-relatively to containing them nodes, and when entry has no intersections with entries of the other tree in working space it immediately rejected from buffer. Entries are divided in classes of equivalence, and every class are separately processed. This method has following features:

1. Buffer space sets free at once when entry in it is not needed.
2. Every tree page reads only once.
3. Stack order of processing classes of equivalence provides minimum loading of the buffer.
4. CPU time is enough high because we must compute estimating function for each pair of entries in class and after reading one page define division by classes.
5. Order of reading are defined not on **pairs** but on **entries** so that after reading of one page from the pair we will probably have not to read the second page.
6. Situation when buffer is overflowed has not been considered.

The second algorithm performs tree matching working with two access paths for both trees and using interleaving descent so that for this algorithm item 5 in last enumeration is also true. This algorithm has following features:

1. Buffer stores only two access paths.
2. Pages may be read not once.
3. CPU time is relatively high without any heuristics.
4. The same as item 5 above.

The first algorithm provides low I/O cost, but buffer may overflow if data objects close enough to each other (in this case classes of equivalence may be very large). This situation takes place in upper levels of the tree when data objects not clustered i.g. uniformly distributed.

The second algorithm need little buffer space although gives less I/O cost optimization. Combination of the algorithms allowing to avoid buffer overflow and to use advantages of the first method is described below.

In upper levels where node objects close to each other are used second algorithm, and then starting from predefined level we begin using first algorithm to better I/O cost optimization. Due to stack order of processing classes maximum load of buffer is function of tree levels and maximum size of classes. Therefore we must limit the class size. It reaches by processing large classes (classes with number of entries greater than value being defined before tree matching) by tree matching algorithm which takes pairs of class consequently and joins respective subtrees. The second way is in breaking large classes in two parts by processing only some pairs of the class.

Combined algorithm allows to avoid buffer overflow and may reduce I/O cost in case of high clustered and low overlap data. Experiments on real data are needed to estimate and compare performance of the algorithm.

3.3 CPU cost decreasing

For reducing number of comparisons in definition of intersected objects of two nodes some heuristics were proposed, including checking in common area of the nodes, applying plane-sweep algorithm, etc. Below the *grid* heuristic will be described.

Let us have two nodes with intersected bounding objects, then we define their common area, divide it by the grid and allocate array of respective sizes with bit strings of length M, where M is maximum number of objects in one node. At first all bits sets to 0. Then for every object of one node we evaluate touched cells of grid and set in strings respective them i-th bit in 1 (i is number of processed object). After that elements of array shows for each cell which object touched them. Then for objects of another node we evaluate touched cells and OR'ing their elements of array get string, according which define candidate pairs.

This algorithm allows to avoid checking for intersection distant objects.

The results of experiments of comparing different heuristics are presented in table 3. First column of the table contains number of data objects inserted in both trees, second shows number of intersected pairs resulting from the join, and last three contains number of object intersection tests for methods with following heuristics applied: *Common area* heuristic for all three methods; *plain-sweep* heuristic for Meth2 and *grid* heuristic for Meth3.

Upper part of the table respects uniformly distributed data objects with uniformly distributed radii in $(0, 0.0002)$ and lower part contains experimental results for mixed-uniform distribution when in 79 cases radii distributed in $(0, 0.0002)$ and in one case in $(0, 0.007)$.

Data objects	Inters. pairs	Intersections		
		Meth1	Meth2	Meth3
12000	4351	259920	50760	43533
24000	17240	525731	201096	119467
36000	38681	797126	448030	217573
48000	68506	1093014	789956	336658
12000	5692	268415	64353	44565
24000	22461	545021	227811	120999
36000	51059	856743	494187	230207
48000	90718	1198308	859647	360964

Table 3: Comparison of Spatial join heuristics

This table shows that number of object intersection tests reduces with using grid heuristic instead plain-sweep heuristic by factor 0.85–0.41 with respect to data object distribution and spatial join parameters. Advantage of the grid heuristic against plain-sweep algorithm is the higher the less *input/output* ratio, because plain-sweep algorithm performs more unnecessary intersection tests when size of node objects increases in compare with bounding node object size.

In experiments sizes of the grid were 8×8 so that sizes of the grid cells was enough little but number of cells was not too big. In common case naturally may be used grid $x \times y$, where $x = y = \lceil \sqrt{M} \rceil$.

3.4 Oriented polygons

Polygons with oriented sides may be used for internal and external approximations of the objects and in TR*-tree for decomposition. This type of description is attractive, because its intersection test has low evaluation cost ($2 * n$-comparisons for oriented n-angles in worst case), and changing n we can choose precision of description.

Polygon with oriented sides is described by number of values which is minimal coordinates of enclosed spatial object along predefined axes. For instance, rectangles commonly used may be thought as oriented polygons with four base axes which normal vectors are $(1, 0), (0, 1), (-1, 0), (0, -1)$. Coordinates of a such polygon are equal to $xl, yl, -xr, -yr$ of a corresponding rectangle.

Description with oriented polygons is effective, because we can perform operations with it like with rectangles, especially when a set of normals of predefined axes contains both $\vec{v}$ and $-\vec{v}$ as is in example. If it is not satisfied we will have to compute maximum values along some axes evaluating coordinates of some vertices when searching for polygons intersection.

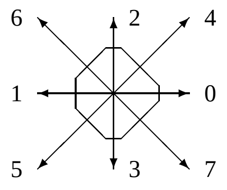

Figure 2: Order of comparisons

Intersection test of oriented polygons has one feature: it's speed depends on the *order* in which axes are taken for comparison respective intervals of objects. Speed is higher when in every step we choose axis with maximal minimal angle to already chosen axes. Picture 2 shows right (in sense of direction of base axes) 8-angle and order of axes for comparing.

The following experiment was done to compare using different approximations in *exact geometry processor* test. The best way to perform it according [4] is as follows: first we decompose spatial objects to trapezoids, then build-up TR*-trees and join them till finding out first intersected pair of trapezoids (then objects intersect). If there is no such a pair then objects do not intersect. As a description objects in TR*-trees were tested right 4,6,8,10-angles. Three parameters were selected for comparing: TR*comp is number of *interval* comparisons in searching for intersected leave objects of the TR*-trees; Tcomp is number of compared trapezoids and Ecomp is number of compared edges of trapezoids. Table 4 presented experimental results shows that 6-angle is better than rectangle in all parameters. Natural feature of N-angle description with big N is more consumption of memory and disk space.

Edges	TR*comp	Ecomp	Tcomp
4	49.4	6.28	1.47
6	48.5	4.82	1.26
8	54.4	4.32	1.20
10	61.7	3.75	1.11

Table 4: Oriented polygons testing

The parameters of experiment are: number of edges of generated objects were distributed uniformly from 100 to 150; there was done 100 generations and then every generated pair of objects were tested in different positions relatively each other 10 times; positions were selected such that bounding rectangles of the objects are intersected.

4 Conclusions

Some optimizations of R*-tree algorithms, including local temporary grid index and forced reinsert modification have been described in the paper. There was also described combined spatial join algorithm using R*-tree indices for data and based on previous work. Also grid heuristic was proposed which is better than sweep-line heuristic in optimization of spatial join CPU time. Since spatial DBSs and other systems used spatial processing are intensively developed now more problems in future will be linked with integrating spatial access methods in real systems with other type of data processing. For example, in [10] situation are investigated when spatial join input is a result of a non-spatial queries.

References

[1] A.Guttman. New features for relational database systems to

support cad applications. In *PhD thesis*, University of California, Berkeley, 1984.

[2] K. K. Al-Taha and R. Barrera. Temporal data and GIS: An overview. In *Proceedings of GIS/LIS '90*, November 1990.

[3] N. Beekmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, Atlantic City, NJ, 1990.

[4] T. Brinkhoff, H.-P. Kriegel, R. Schneider, and B. Seeger. Multi-step processing of spatial joins. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 197–208, Minneapolis, Minn., 1994.

[5] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 237–246, 1993.

[6] D.Ballard and C.Brown. Computer vision. In *Prentice Hall*, 1982.

[7] O. Guenther. Efficient computation of spatial joins. In *The Ninth IEEE International Conference on Data Engineering*, Vienna, Austria, April 1993.

[8] Orenstein J.A. Spatial query processing in an object-oriented database systems. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 326–333, Washington, DC, 1986.

[9] I. Kamel and C. Fauloutsos. Parallel R-trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, volume 21:2 of *SIGMOD Record*, pages 195–204, San Diego, Calif., 1992.

[10] Ming-Ling Lo and C.V. Ravishankar. Spatial join using seeded trees. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 209–220, Minneapolis, Minn., 1994.

[11] D. Lomet. A review of recent work on multi-attribute access methods. *ACM SIGMOD Record*, 21(3):56–63, 1992.

[12] M.G. Martynov. Variations of R-tree structure for indexing of spatial objects. In *Proc. of the Intnl. Workshop on Advances in Databases and Information Systems - ADBIS'94*, pages 217–221, Moscow, May 23–26 1994.

[13] P. Mishra and M. Eich. Join processing in relational databases. *compsurv*, 24(1):63–113, March 1992.