Efficient Processing of Spatial Joins Using R-trees

Thomas Brinkhoff
Hans-Peter Kriegel
Bernhard Seeger

Institute of Computer Science, University of Munich
Leopoldstr. 11 B, W-8000 München 40, Germany

e-mail: {brink,kriegel,bseeger}@dbs.informatik.uni-muenchen.de

Abstract

Spatial joins are one of the most important operations for combining spatial objects of several relations. The efficient processing of a spatial join is extremely important since its execution time is superlinear in the number of spatial objects of the participating relations, and this number of objects may be very high. In this paper, we present a first detailed study of spatial join processing using R-trees, particularly R*-trees. R-trees are very suitable for supporting spatial queries and the R*-tree is one of the most efficient members of the R-tree family. Starting from a straightforward approach, we present several techniques for improving its execution time with respect to both, CPU- and I/O-time. Eventually, we end up with an algorithm whose total execution time is improved over the first approach by an order of magnitude. Using a buffer of reasonable size, I/O-time is almost optimal, i.e. it almost corresponds to the time for reading each required page of the relations exactly once. The performance of the various approaches is investigated in an experimental performance comparison where several large data sets from real applications are used.

1 Introduction

During the last decade, spatial database systems have become more and more important for industry and researchers. For example, substantial progress has been made with respect to developing spatial data models and designing efficient spatial access methods [22]. Spatial data consists of points, lines, rectangles, polygons, surfaces and even more complex objects composed from simple ones. In order to support spatial queries such as "find all objects which intersect a given window", spatial access methods cluster objects on disks according to their spatial location in space. Consequently, if objects are close together in space, they are stored close together on disk with high probability. Due to high performance requirements on spatial database systems, spatial access methods should be used on every spatial relation for supporting spatial queries efficiently.

In a database system (DBS), we can distinguish between two different types of queries. The one type of queries, called *single-scan queries*, require at most one access to an object and therefore, the execution time is at most linear in the number of objects stored in the corresponding relation. Window queries and point queries are prime examples for single-scan queries in a spatial database. Several approaches for an efficient processing of such queries are discussed in great detail. Samet provides an excellent survey [22] of almost all of these methods. The other type of queries are *multiple-scan queries* where objects have to be accessed several times and therefore, execution time is generally not linear but superlinear in the number of objects. For example, a join-operation in a relational DBS is a multiple-scan query. The most important multiple-scan queries in a spatial DBS are the *spatial join* [18] and the *map overlay* [3]. The spatial join operation is used to combine spatial objects of two sets according to some spatial properties. For example, consider the spatial relations "Forests" and "Cities" where an attribute in each relation represents the borders of forests and cities, respectively. The query "find all forests which are in a city" is an example of a spatial join.

In this paper, we focus on the problem of designing and implementing algorithms for spatial joins using spatial access methods. Our goal is to use already existing access method rather than presenting another special purpose data structure tailor-cut for this operation. There are several reasons for using access methods to perform a spatial join:

- In spatial applications, the assumption is almost always true that a spatial index exists on a spatial relation. Therefore, it seems to be an interesting approach to exploit these indices for spatial joins.
- Many efficient algorithms designed for natural-joins, such as hash-based join algorithms, cannot efficiently be applied to spatial joins. Note, that hashing as used in the natural-join does not preserve the order of data and therefore, objects close together in data space are not in the same bucket with high probability.
- For todays spatial DBSs, one of the most challenging requirement is to maintain seamless spatial databases. This results in very large spatial relations. Then, a spatial join almost always requires access to objects in a given window. For example, let us assume that the relations Forests and Cities contain all information over Europe. A user in Munich might be interested in the query "For all cities not further away than 100 km from Munich, find all forests which are in a city". An efficient processing of the window query is important for the efficiency of the spatial join.

In the following, we restrict our considerations to R-trees [10], particularly R*-trees [2], used as the underlying spatial access method for processing spatial joins. It has been shown in [2] that the R*-tree is one of the most efficient members of the R-tree family with respect to several single-scan queries. Since several geographic information systems, e.g. Intergraph's GIS, and DBSs, e.g. Postgres [23], use R-trees as their basic spatial access method. there is also considerable interest in efficient join algorithms using R-trees.

In this paper, we focus on exploiting R-trees for the efficient processing of spatial joins. With respect to the authors knowledge, spatial joins supported by R-trees have not been examined until now. Previous approaches for processing joins are based on access methods others than R-trees or address the problem in a quite different context. Orenstein [18] has proposed B^+-trees in combination with z-ordering as the underlying access method for performing spatial joins. For a class of tree structures, Günther [9] has presented a general model for estimating the cost of spatial joins. In a different setting, Rotem [21] has discussed the issue of spatial join indices.

The paper is organized as follows. We first introduce to the basic ideas of spatial access methods and spatial joins in section 2. Next in section 3, we review the most important design goals of the R*-tree. In section 4, we study spatial joins using R*-trees. First, we present a straightforward approach for spatial join processing using R-trees. After discussing the most important drawbacks, we present several approaches for improving CPU-time and I/O-time and experimentally identify their characteristics. In section 5, we report the results of an experimental performance comparison. Section 6 concludes the paper and gives an outlook to future work.

2 Spatial Query Processing

Spatial DBSs have to cope with huge collections of multidimensional data objects, such as polygons in two-dimensional data space. It is well understood, that traditional index structures do not efficiently support queries on spatial data. Therefore, several so-called spatial access methods have been suggested over the last decade. Spatial access methods are primarily designed to support single-scan queries in a spatial database. An example for a single-scan query is the *window query*: Let A be a set of spatial objects and r be a rectilinear rectangle. Find all objects $a \in A$ with $a \cap r \neq \emptyset$.

In order to support spatial queries like the window query efficiently, a spatial access method attempts to store objects which are close together in the data space on a common page. Obviously, this policy reduces the number of disk accesses which is most commonly used for measuring response time of queries, particularly I/O-intensive spatial queries.

The size of spatial objects in a relation varies extremely. In particular, the size of an object is not limited by the page size. In order to deal with such large objects and at the same time to preserve spatial locality in pages, spatial access methods organize only approximations of objects as an index to the exact representation. The *minimum bounding rectilinear rectangle (MBR)* of an object is a very common approach for an approximation. Then the window query is processed in two steps, called filter and refinement step in [18]:

(i) *filter step*: Find all objects whose MBR intersects the query rectangle.
(ii) *refinement step*: For those objects, check whether they really fulfil the query condition (if necessary, make use of the exact representation).

Most popular with respect to spatial data handling are access methods based on linear quadtrees [8] or, almost equivalently, z-ordering [17] and other space-filling curves [6]. The basic idea of these methods is to decompose the data space into non-overlapping cells which can be computed by recursively cutting the space into two or four parts of equal size. Cells can be identified by a location code, called *z-value*, and a size code, called *level*. On secondary storage, cells can be organized according to their z-value and level in a B^+-tree. A disadvantage of this approach is that the same reference might be stored several times in different cells. The ratio of the number of references to the number of spatial objects is called *redundancy factor*. Note, that a high redundancy factor implies an accurate approximation of the object. Consequently, the number of objects participating in the refinement step is low and, the cost for the filter step is high. A detailed discussion on the effect of redundancy is given in [19].

Other access methods, such as R-trees [10], represent an object only once generally using a MBR as an approximation. Then, a data entry of an access method consists of the MBR and a pointer to the corresponding object. An R-tree is a balanced tree built up in a similar way as a B^+-tree. Data entries are grouped together in a data page according to the spatial proximity of their rectangles. A directory entry in the R-tree is referring to this data page. In order to direct a spatial query, a directory entry consists of the minimum bounding rectangle of those rectangles stored in the data pages of the corresponding subtree. R-trees have the advantage to represent objects uniquely. However, R-trees cannot guarantee a disjoint decomposition, i.e. rectangles of directory entries may overlap. A high overlap results in poor query performance. Nevertheless, in [2] it has been shown that the R*-tree is very efficient for spatial query processing, particularly in comparison to other members of the R-tree family.

2.1 Joins with Spatial Objects

In the following, we consider two sets $A = \{a_1,\ldots,a_n\}$ and $B = \{b_1,\ldots,b_m\}$ of spatial objects. We refer to these sets as *spatial relations*. The reader may think of the objects in a spatial relation as polygons representing a surface of a spatial object. Let *Id* be a function that assigns a unique identifier to each spatial object in a database. Furthermore, let *Mbr* be a function that computes the minimum bounding rectilinear rectangle of an object. For example, $Id(a_1)$ and $Mbr(a_1)$ denote the identifier and minimum bounding rectangle of a_1, respectively. The most important spatial joins are given in the following:

1.) *MBR-spatial-join*: Compute all pairs $(Id(a_i), Id(b_j))$ with $Mbr(a_i) \cap Mbr(b_j) \neq \emptyset$
2.) *ID-spatial-join*: Compute all pairs $(Id(a_i), Id(b_j))$ with $a_i \cap b_j \neq \emptyset$
3.) *Object-spatial-join*: Compute $a_i \cap b_j$ with $a_i \cap b_j \neq \emptyset$

Note that the MBR-spatial-join can be used for implementing the filter step of the ID- and object-spatial-join. This explains the importance of the MBR-spatial-join. The object-spatial-join does not only compute the identifiers of the objects in the response set, but also the resulting objects. Obviously, this is more expensive than an ID-spatial-join. In addition to these three joins, we can introduce other types of joins, if we use other spatial operators than intersection, e.g. containment, or if we consider more than two spatial relations for processing a join. The problem of spatial joins with more than two spatial relations is similarly defined and its solution can make use of the techniques that will be presented in this paper.

Almost all methods designed for an efficient join processing of non-spatial relations, see [15] for a survey, cannot be used for spatial joins. Using the simple nested loop approach, every object of the one relation has to be checked against all objects of the other relation. Since we consider very large relations of spatial objects, the performance of the nested loop algorithm is not acceptable. As already indicated previously, hashed-based join algorithms are suitable only for natural and equi-joins, but not for spatial joins. Another approach is similar to a sort-merge join. Here, spatial objects are sorted according to their spatial proximity. This approach for processing a spatial join may be considered, if there is not already a spatial index on the spatial relations. However, our basic assumption is that a spatial access method efficiently supports access to the spatial relations. Thus, we are interested in exploiting the spatial access method for an efficient join processing.

There are several other approaches for performing joins using multidimensional point access methods, e.g. grid files [1] and kd-trees [11]. These methods can be used for spatial joins, if spatial objects are transformed to higher-dimensional points. However, the most serious disadvantage is that the use of transformation does not preserve proximity.

Methods for computing the spatial join are discussed in great detail for quadtrees and similar access methods [18]. The cells of quadtrees can be represented in a linear fashion in order of z-value and level. A spatial join of two quadtrees can be performed by an "almost linear" scan through the corresponding sequences of cells. Note that the meaning of linear is not related to the number of objects, but to the number of pointers referring to the objects. The number of pointers might be essentially higher than the number of objects. Without going into details of the algorithms, during the linear scan of the cells it might even be necessary to jump back to a previous position in the sequence and start scanning over again. The length of the jump depends on the redundancy factor and on the overlap of the spatial objects. To avoid redundancy, objects must be uniquely represented as suggested for R-trees. It is still an open question which type of access method is most suitable for performing spatial joins and for other operations [12].

3 R*-tree

An R-tree [10] is a B^+-tree like access method that stores multidimensional rectangles as complete objects without clipping them or transforming them to higher dimensional points. Until now the most efficient variant of the R-trees is experimentally shown to be the R*-tree [2]. The R*-tree uses more sophisticated insertion and splitting algorithms than the original R-tree. However, there is almost no difference in the data structure.

3.1 Data Structure of R-trees

A non-leaf node contains entries of the form *(ref, rect)* where *ref* is the address of a child node and *rect* is the minimum bounding rectangle of all rectangles which are entries in that child node. A leaf node contains entries of the same form where *ref* refers to a spatial object in the database and *rect* is the minimum bounding rectangle of that spatial object. Let us mention, that storing spatial objects instead of references in leaf nodes does not affect the basic algorithms of the R-tree. Given an entry E of a node, E.rect and E.ref denote the corresponding rectangle and reference, respectively.

Let M be the number of entries that fit in a node and let m be a parameter specifying the minimum number of entries in a node, $2 \leq m \leq \lceil M/2 \rceil$. An R-tree satisfies the following properties:

- The root has at least two children unless it is a leaf.
- Every node contains between m and M entries unless it is the root.
- The tree is balanced, i.e. every leaf node has the same distance from the root
- Every rectangle of a non-leaf node covers all rectangles in its corresponding child node. In general, the rectangle is even the minimum bounding rectangle.

Since one node of the data structure exactly corresponds to one page on secondary storage, we will use both terms synonymously in the following.

An R-tree is completely dynamic; insertions and deletions can be intermixed with queries without any global reorganization. Although data entries are grouped together according to the location of their rectangles in space, R-trees have to allow overlap in directory nodes, i.e. rectangles of different entries may have a common intersection. Since a high overlap results in poor query performance, one of the most important design goals of the R*-tree was the reduction of overlap. Similar to B-trees, R-trees guarantee that storage utilization is at least m/M and that the height grows logarithmically in the number of data records. The reader is referred to the original papers [10] and [2] for a more detailed discussion.

3.2 Basic Algorithms

Following the similarities in the data structure, there is almost no difference between R-tree and R*-tree with respect to specific queries, like the window query. Let S be a query rectangle of a window query. Then, the query is performed by starting in the root and computing all entries whose rectangle intersects S. For these entries, the corresponding child nodes are read into main memory and the query is performed like in the root node unless it is a leaf node.

An example for an R-tree is given in Figure 1. The tree consists of three data pages and a directory page. Note, that the rectangles of the directory page are the minimum bounding rectangles of those rectangles that are stored in the corresponding child node. The query window is depicted by the gray colored rectangle. First, the query is performed against the root of the R-tree where rectangle r and t intersect the window. Thus, the corresponding two data pages are read into memory and their entries are checked for a common intersection with the window. Eventually, rectangle a1 is found to be an answer of the window query.

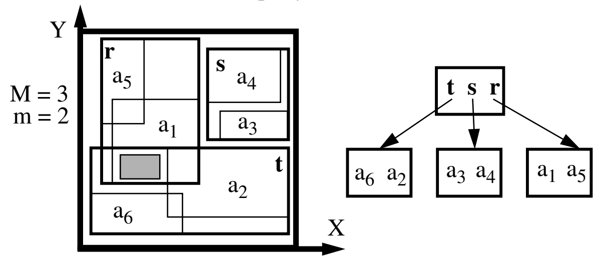


Figure 1: Example of an R-tree

The efficiency of queries depends on the goodness how R-trees assign rectangles to nodes. Thus, the insertion algorithm of the R-tree is most crucial to query performance. The insertion algorithm of the R*-tree is completely different and superior to the ones of the other R-trees. There are basically three reasons for the performance improvements.

First, an insertion of an entry with rectangle R might be guided to a non-leaf node that does not contain any entry whose rectangle covers R completely. An algorithm has to select a child node to which the insertion process is forwarded to. In case of the R*-tree, the entry is chosen whose rectangle has the minimum increase of overlap with its siblings.

Second, whenever an entry is inserted into a full node in the R*-tree, the node is generally not split, but some fraction of its entries is deleted and re-inserted on the same level in the R*-tree. The entries for re-insertion are selected such that they have the largest distance from the centre of the original minimum bounding rectangle of the node. Re-insertion of entries increases storage utilization, improves the quality of the partition (e.g. reducing overlap) and makes performance almost independent of the sequence of insertions. If, during a re-insertion process on level *l* of the R*-tree, an entry should be inserted into a full node on level *l*, the node is split into two.

Third, the algorithm for splitting a node in an R*-tree is completely different from the one of the original R-tree. First, we must determine the axis where the split has to be performed. For each axis, all entries are sorted according to the left corner (with respect to the considered axis) of their rectangles, all possible M-2m+1 splits are considered with respect to the ordering in the sequence, and eventually, we sum up the perimeters of the resulting nodes overall possible splits. The same process is repeated with the entries ordered according to the right corner of their rectangles. The axis with the minimum overall sum is chosen as the split-axis. Then, we consider again the sequence of entries sorted with respect to the left corner of the interval obtained by projecting the rectangle on the split-axis. This sequence is grouped into two subsequences. The number of entries in the left and right subsequence varies between m and M-m+1. Among these M-2m+1 possibilities, we choose the split resulting in a minimum of overlap between the minimum bounding rectangles of the two subsequences. Each of the subsequences is stored separately in a node of the R-tree.

4 Spatial Join

In the following we investigate how to perform the MBR-spatial-join using R*-trees. In the remainder of the paper, we use spatial join as a synonym for the MBR-spatial join. Furthermore, for an R*-tree R we use the following notations:

$|R|_{dir}$ ($|R|_{dat}$) = number of directory (data) pages
$||R||_{dir}$ ($||R||_{dat}$) = number of directory (data) entries

Obviously, the total number |R| of pages is given by $|R|_{dat} + |R|_{dir}$. Analogously, the total number ||R|| of entries is given by $||R||_{dat}+||R||_{dir}$.

Now, let us consider two sets of rectangles each of them being organized by an R*-tree. In the following, we call the R*-trees R and S. As a performance measure for the execution time of a spatial join between R and S, we use CPU-time as well as I/O-time. Only the number of disk accesses is not sufficient to measure execution time of joins, particularly of spatial joins, appropriately. Already in case of natural-joins, the number of tests for the join condition has essential influence on the execution time. To check the join condition in case of spatial joins is far more expensive than in case of natural-joins. Therefore, a good measure for performance consists of both, the number of disk accesses and the number of comparisons.

The I/O-time is usually measured in the number of disk accesses required for performing the join. A similar performance measure is used in the following. Since the I/O-time for reading and writing a page slightly depends on the size of the page, we differentiate between the positioning cost (including seek cost and rotational latency) and transfer cost. Obviously, a buffer of pages can essentially reduce the I/O-cost. Independent of the size of a buffer, an optimal processing of a spatial join will read each required page only once into main memory. If all pages are required for a join, a lower bound for the I/O-cost is $|R|_{dat} + |S|_{dat}$. This result is well-known as the minimum cost for a natural-join using two B-trees. In case of R*-trees, however, it might be possible that a spatial join requires less than $|R|_{dat} + |S|_{dat}$ pages. The reason is that the union of all directory rectangles with the same distance from the root of the R*-tree does not completely cover the underlying data space.

The CPU-time of a spatial join is measured in the number of floating point comparisons. Comparisons are required for checking the join condition, i.e. whether two rectilinear rectangles intersect. Note, that for a pair of rectilinear rectangles four comparisons are exactly required to determine that the join condition is fulfilled. If the rectangles do not fulfill the join condition, less than four comparisons might be required. There are many other operations affecting CPU-time, but in general they do not asymptotically influence CPU-time.

An analytical investigation of the execution time of a spatial join performed with R*-trees seems to be almost impossible. Not surprisingly, there are only a few analytical results known for spatial access methods. Most of these analytical results are restricted to multidimensional points, to single-scan queries, and to uniformly distributed data set very rarely occurring in real applications. In the following, we investigate spatial joins in an empirical comparison using carthographic maps from real applications. We consider two files with 131,461 and 128,971 line objects in an area of California. The data is drawn from the TIGER/Line files used by the US Bureau of the Census [4]. The first map represents streets whereas the second map represents rivers and railway tracks. For each of the maps, the minimum bounding rectangles of the polygonal objects are stored in an R*-tree. The join of R and S results in a response set of 86,094 pairs of intersecting rectangles. For various page sizes, the most important properties of R*-trees R and S are reported in Table 1.

page size	M	R*-tree R			R*-tree S			|R|+|S|
		height	$\|R\|_{dir}$	$\|R\|_{data}$	height	$\|S\|_{dir}$	$\|S\|_{data}$	
1 KByte	51	4	127	4,202	4	117	3,996	8,442
2 KByte	102	3	33	2,143	3	30	1,991	4,197
4 KByte	204	3	9	1,069	3	8	1,005	2,091
8 KByte	409	3	3	541	3	3	495	1,042

Table 1: Properties of R*-trees R and S

4.1 A First Approach of a Spatial Join for R-trees

The basic idea of performing a spatial join with R*-trees is to use the property that directory rectangles form the minimum bounding box of the data rectangles in the corresponding subtrees. Thus, if the rectangles of two directory entries E_R and E_S do not have a common intersection, there will be no pair ($rect_R$, $rect_S$) of intersecting data rectangles where $rect_R$ is in the subtree of E_R and $rect_S$ is in the subtree of E_S. Otherwise, there might be a pair of intersecting data rectangles in the corresponding subtrees. In the following three subsections we assume that both trees are of the same height. The case of joining R*-trees of different height is discussed in section 4.4. The following algorithm presents our first approach:

```
SpatialJoin1 (R,S: R_Node);        (* height of R is equal height of S *)
  FOR (all E_S ∈ S) DO
    FOR (all E_R ∈ R with E_R.rect ∩ E_S.rect ≠ ∅) DO
      IF (R is a leaf page) THEN     (* (S is also a leaf page) *)
        output (E_R,E_S)
      ELSE
        ReadPage(E_R.ref);  ReadPage(E_S.ref);
        SpatialJoin1(E_R.ref,E_S.ref)
      END
    END
  END
END SpatialJoin1;
```

Here it is assumed that *R_Node* is the type of a node in an R*-tree and that each node accommodates a collection of entries. As previously introduced, an entry *E* consists of a pointer *ref* and a rectangle *rect*. A procedure *ReadPage* is assumed to read the required page from the buffer or, if the page is not in the buffer, from secondary storage. The algorithm recursively traverses both of the trees in top-down fashion.

The R*-tree makes use of a so-called path buffer accommodating all nodes of the path which was accessed last. In order to be more efficient with respect to I/O, an additional buffer is used for single pages, not complete paths, independently of the path buffer. The buffer, called LRU-buffer, follows the *last recently used* policy. The reason for two different buffers is that the path buffer exclusively belongs to the data structure (i.e. R*-tree), whereas the LRU-buffer can be considered as a buffer of the underlying system. Note, that in a multiuser environment, a spatial join operation can only occupy a "small" fraction of the LRU-buffer. To which extend the LRU-buffer is used in a spatial join, depends on the system load. In our experiments, we assume that the R*-trees involved in the spatial join exclusively use all pages of the LRU-buffer.

		Size of pages			
		1 KByte	2 KByte	4 KByte	8 KByte
Size of LRU-buffer (KByte)	0	24,727	12,479	5,720	2,837
	8	20,318	12,010	5,720	2,837
	32	13,803	9,589	5,454	2,822
	128	11,359	6,299	4,474	2,676
	512	10,372	4,964	2,768	2,181
opt. buffer size		8,442	4,197	2,091	1,042
# comparisons		33,566,961	65,807,555	118,864,748	242,728,164

Table 2: Number of disk accesses and comparisons of *SpatialJoin1*

In Table 2, we report the results of the algorithm *SpatialJoin1* using R*-trees R and S as input. The spatial join is performed with various page sizes and various buffer sizes. The largest considered buffer size corresponds to keeping about 6% of both R*-trees in main memory. For todays computer system, this seems to be a realistic assumption. For each setting, we report the number of disk accesses required to compute the join. Additionally, we keep track of the number of comparisons required for checking the join condition. This number is reported in the last row of Table 2. Note, that this number is independent of the size of the LRU-buffer.

Let us first discuss the results without using an LRU-buffer, i.e. buffer size = 0. For all sizes of pages, a page of the R*-tree is read into main memory on the average only about three times. For these specific files, the overlap of the R*-tree seems to have not much influence on the performance of the join. Now let us consider the cases of using small LRU-buffers. For a small page, the LRU-buffer pays off even if it is very small. For a 32 KByte LRU-buffer, the number of disk accesses is only 55% of the number required by the join without any LRU-buffer. For larger page sizes, the same effect can only be observed in case of also using a large buffer.

In order to determine whether the spatial join is CPU-bound or I/O-bound, we have estimated the execution time of the spatial join charging $1.5*10^{-2}$ seconds for positioning the disk arm, $5*10^{-3}$ seconds for transferring 1 KByte of data from disk and, $3.9*10^{-6}$ seconds for a floating point comparison (including necessary overhead). The last number is experimentally determined and obviously depends on the underlying hardware. For performing the experiments, we used HP720 workstations which deliver on 57 MIPs and 17.9 MFlops. Moreover, the positioning cost for moving the disk arm to the position of the desired page depends on the specific disk, how well the spatial relations are clustered on disk and on the load of the disk during performing the join. Using the results of Table 2, we have computed the time estimates presented in Figure 2.

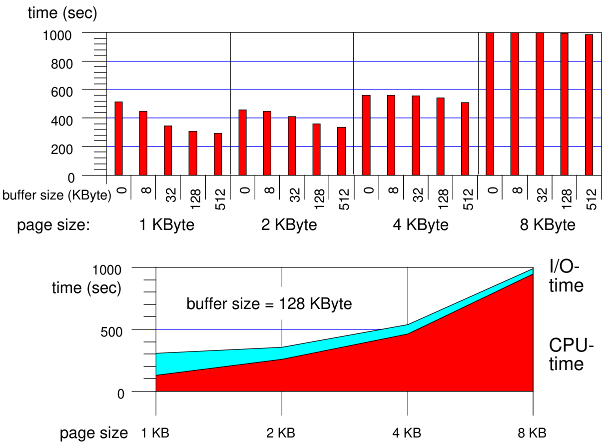


Figure 2: Etimation of the execution time of *SpatialJoin1*

As demonstrated in the upper diagram of Figure 2, best overall performance is achieved for page sizes of 1 and 2 KByte. The lower diagram shows that the spatial join is slightly I/O-bound for a page size of 1 KByte. However, with increasing page size, the spatial join is becoming more and more CPU-bound. This observation is true for small and large LRU-buffers. Hence, we can state that cost optimization of a spatial join has to take into account both, CPU- and I/O-time.

There are at least two parts in the algorithm *SpatialJoin1* which are worth to be improved. First, CPU-time consumption is rather high since each entry of the one node is checked against all entries of the other node. Second, pages are selected for the next call of *SpatialJoin1* without taking into account the I/O-cost for reading these pages. A better approach would be to compute the "best" sequence of pairs of pages required for computing the join on the next level of the R*-tree.

4.2 CPU-Time Tuning

The consumption of CPU-time is proportional to the number of floating point comparisons required for computing the join condition (i.e. the test whether two rectangles intersect). Hence, our primary goal is to reduce the number of comparisons for each call of *SpatialJoin1*, i.e. for each pair of qualifying nodes. However, several constraints have been taken into account:

- The storage utilization and the query performance of the original R*-tree should not be affected. In particular, the node capacity should not be reduced.
- *Expensive* preprocessing steps for the nodes of the R*-tree should be avoided. Since pages are kept in main memory only temporarily, such a preprocessing step of a page would be required each time the page is read into the buffer.
- The algorithm should be robust and easy to implement. In general, we are not interested in the algorithm with best asymptotic performance, but in the one which is efficient for realistic problem sizes which corresponds to the number of entries in the nodes.

In order to reduce CPU-time we examine two approaches: First, for a given pair of nodes, we restrict the search space of the join such that in *SpatialJoin1* only a small number of entries has to be considered. Second, entries are sorted according to their spatial location and thereafter, an algorithm based on the plane-sweep paradigm [20] is used to compute the desired pairs of intersecting entries.

Restricting the search space

In algorithm *SpatialJoin1*, each entry of the one node is tested for the join condition against all entries of the other node. Let us consider two directory entries E_R and E_S which fulfil the join-condition, i.e $E_R.rect \cap E_S.rect \neq \emptyset$. Then, *SpatialJoin1* is invoked using $E_R.ref$ and $E_S.ref$ as input. The following property is very important with respect to reducing CPU-time:

Only the entries of $E_1.ref$ and $E_2.ref$ which intersect the intersection rectangle $E_1.rect \cap E_2.rect$ may have a common intersection.

This property can easily be used for improving the join. A linear scan through each of the two nodes marks all entries which are required for performing the join, i.e. which intersect the intersecting rectangles of the two nodes. Then, each marked entry of the one node is tested against all marked entries of the other node.

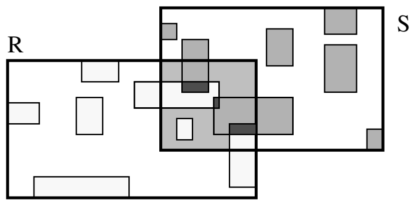


Figure 3: Restricting the search space of a spatial join

An example is illustrated in Figure 3. Algorithm *SpatialJoin1* checks the join condition 49 times, whereas our new approach only requires 6 tests of the join condition and additionally, it must scan each of the nodes which corresponds to 14 times checking the join condition. The modified algorithm, called *SpatialJoin2* (short *SJ2*), is specified as follows:

SpatialJoin2 (R,S: R_Node; rect: Rectangle);
  (* rect is the intersection of the MBRs of nodes R and S *)
BEGIN
  SpatialJoin1a ({E_i| ($E_i \in$ R) $\wedge$ (E_i.rect $\cap$ rect $\neq \emptyset$)},
                        {E_i| ($E_i \in$ S) $\wedge$ (E_i.rect $\cap$ rect $\neq \emptyset$)})
END SpatialJoin2;
(* *SpatialJoin1a* is *SpatialJoin1* where line
  "*SpatialJoin1*(E_R.ref,E_S.ref)" is replaced by
  "*SpatialJoin2*(E_R.ref, E_S.ref, E_R.rect $\cap$ E_S.rect)" *)

Table 3 reports the results for the number of our comparisons for both *SpatialJoin2* and to *SpatialJoin1*. The results show a huge improvement in the number of comparisons. The factor of improvement varies between 4.6 and 8.9. Nevertheless, there is still a strong dependence between the required CPU-time and the page size. More precisely, the effect is clearly visible that the number of comparisons increases superlinearly in the page size.

	Size of pages			
	1 KByte	2 KByte	4 KByte	8 KByte
SpatialJoin1	33,566,961	65,807,555	118,864,748	242,728,164
SpatialJoin2	7,316,389	10,347,688	15,796,183	27,219,893
performance gain	4.59	6.36	7.52	8.92

Table 3: Comparisons with/without restricting the search space

Spatial sorting and plane sweep

Our second approach for improving spatial join is to sort the entries in a node of the R*-tree according to the spatial location of the corresponding rectangles. Here, the obvious problem occurs that two-dimensional rectangles cannot be sorted (i.e. mapped into an 1-dimensional sequence) without any loss of locality. A suitable solution with respect to computing the intersection is the following approach. Let us consider a sequence Rseq = $<r_1,...,r_n>$ 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). We use $\pi_X(r_i)$ and $\pi_Y(r_i)$ to refer to the projection of r_i onto the X- and Y-axis, respectively. A sequence Rseq = $<r_1,...,r_n>$ is sorted with respect to the X-axis, if r_i.xl $\leq r_{i+1}$.xl, $1 \leq i < n$. For example, a sorted sequence of 6 rectangles is depicted in Figure 4 is $<r_1, r_4, r_2, r_5, r_3, r_6>$.

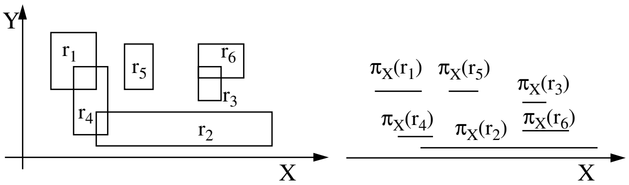


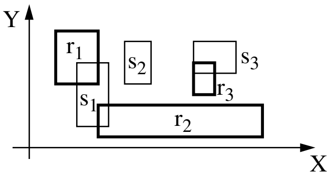
Figure 4: Two sets of rectangles & their projection onto the X-axis

Plane sweep is a common technique for computing intersections [20]. 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, we exploit the plane-sweep technique without the overhead of building up any additional dynamic data structure. Let Rseq = $<r_1,\ldots,r_n>$ and Sseq = $<s_1,\ldots,s_m>$ be sequences of rectangles. First, Rseq and Sseq are sorted as described above. Hence, r_i.xl $\leq r_{i+1}$.xl, $1 \leq i < n$, and s_j.xl $\leq s_{j+1}$.xl, $1 \leq j < m$. Then, we move the sweep-line to the rectangle, say t, in Rseq $\cup$ Sseq with the lowest xl-value. If the rectangle t is in Rseq, we sequentially traverse Sseq starting from its first rectangle until a rectangle, say s_h, in Sseq is found whose xl-value is greater than t.xu. Now, we know that interval $\pi_X(t)$ intersects the interval $\pi_X(s_j)$ for all j with $1 \leq j < h$. If also interval $\pi_Y(t)$ intersects interval $\pi_Y(s_j)$, then rectangle t also intersects rectangle s_j. If rectangle t is in Sseq, Rseq is traversed analogously. Thereafter, rectangle t is marked to be processed. Then, the sweep-line is moved to the next unmarked rectangle in Rseq $\cup$ Sseq with the lowest xl-value and the same step as described above is repeated for all unmarked rectangles. When the last entry from R or S was processed, all intersections are computed. The formal description of the algorithm is given as follows:

```
SortedIntersectionTest (Rseq, Sseq: SEQUENCE OF Rectangle;
                        VAR Output: SEQUENCE OF PAIR OF Rectangle);
(* Rseq and Sseq are sorted; ||Rseq|| = number of rectangles in Rseq *)
   Output := <>; i := 1; j := 1;
   WHILE (i ≤ ||Rseq||) ∧ (j ≤ ||Sseq||) DO
      IF r_i.xl < s_j.xl THEN          (* ⇒ t = r_i *)
         InternalLoop (r_i, j, Sseq, Output);   i := i+1
      ELSE                             (* ⇒ t = s_j *)
         InternalLoop (s_j, i, Rseq, Output);   j := j+1
      END
   END
END SortedIntersectionTest;
```

```
InternalLoop (t : Rectangle; unmarked: CARDINAL;
              Sseq: SEQUENCE OF Rectangle;
              VAR Output : SEQUENCE OF Pair OF Rectangle);
   k := unmarked;
   WHILE (k ≤ ||Sseq||) ∧ (s_k.xl ≤ t.xu) DO      (* ⇒ X-intersection *)
      IF (t.yl < s_k.yu) ∧ (t.yu > s_k.yl) THEN   (* ⇒ Y-intersection *)
         Append (Output, (t,s_k) )
      END;
      k := k+1
   END
END InternalLoop;
```

An example how the algorithm proceeds is illustrated in Figure 5. The sweep-line stops at rectangles r_1, s_1, r_2, s_2 and r_3. For each stop, the pairs of rectangles which are tested for intersection are given on the right hand side of Figure 5.



sequence of intersection tests:
$t = r_1$: $r_1 \leftrightarrow s_1$
$t = s_1$: $s_1 \leftrightarrow r_2$
$t = r_2$: $r_2 \leftrightarrow s_2$, $r_2 \leftrightarrow s_3$
$t = s_2$: -
$t = r_3$: $r_3 \leftrightarrow s_3$

Figure 5: Example for the sorted intersection test.

We want to emphasize that this algorithm can easily be implemented with two pointers, but without using any additional data structures. The algorithm can be performed in O(||Rseq|| + ||Sseq|| + k_X) time where k_X denotes the number of pairs of intersecting intervals created by projecting the rectangles of Rseq and Sseq onto the X-axis. There are several computational geometry algorithms that can solve the same problem in O((||R||+||S||) $\cdot$ log(max(||R||,||S||)) + k) time where k denotes the size of the response set. However, these approaches require rather complex data structures, see [20] and [5]. Moreover, several (but a constant number of) sweeps over the data are necessary which leads to rather high constants. These solutions are worthwhile to be considered for large problem sizes, but their overhead is too high for a rather small problem size as in our case for commonly used page sizes.

More interesting than the worst time complexity of the algorithm is the average time needed by the algorithm for real spatial join operations. There are two interesting questions: First, is it still worth to restrict the search space like for the straightforward algorithm? Second, does the preprocessing step (i.e. sorting) still pay off, even if a page has to be sorted each time it is read into the buffer? This is an important question because a number of pages may be transferred into main memory more than once.

		Size of pages			
		1 KByte	2 KByte	4 KByte	8 KByte
version (I) without restricting the search space	join	4,906,048	6,079,544	7,202.892	9,651,854
	sorting	2,818,729	3,307,217	3,796,240	4,318,238
	join-ratio of (I) to SJ1	6.84	10.82	16.50	25.15
version (II) with restricting the search space	join	5,124,435	5,521,254	5,769,313	6,662,370
	sorting	768,551	880,171	993,419	1,120,404
	join-ratio of (II) to SJ1	6.55	11.92	20.60	36.43
	join-ratio of (II) to SJ2	1.43	1.87	2.74	4.09
	repeat-factor to SJ2	2.85	5.48	10.09	18.35

Table 4: Comparisons of spatial joins with/without sorting

The results of Table 4 illustrate the effect of sorting on the CPU-time (measured in the number of comparisons) required for the spatial join between the two R*- trees R and S. We investigated two versions: version I is without restricting the search space and version II with restricting the search space. In both versions we distinguish between the time spent for the join and the time spent for sorting all the nodes of the tree.

First, let us assume that the entries in the R*-tree are sorted as desired. This assumption is true if the insert and delete algorithms maintain the nodes of the R*-tree sorted or if we sort all nodes of the R*-trees once and then perform only queries and joins. Then, the rows 'join-ratio' give the ratio of the number of comparisons required for the corresponding version to the number of comparisons required for the specified algorithm (*SJ1* or *SJ2*). As illustrated in Table 4, there is a huge improvement if the entries of the nodes are sorted. The join-ratio increases with an increasing size of the pages. With restricting the search space the results are even better. The number of comparisons does not vary considerably in the page size. There is also a clear gain over the unsorted spatial join with restricting the search space. The join-ratio varies between 1.4 and 4.1.

In general, the sorted order of entries is not maintained in the nodes of an R*-tree during insertions and deletions of data objects. Then we suggest that a page is sorted immediately after it is read from disk. Due to a limited size of the buffer, a page may have to be read and sorted more than once. The repeat-factor in Table 4 gives the number how often a page can be sorted on the average until the performance of the join without sorting is better than the performance of the join with sorting (both joins with restricting the search space). For example, in case of 4Kbyte pages, a page has to be read and sorted more than 10 times until the sorted version of the join is less efficient than the version without sorting. As shown in Table 2, even for *SpatialJoin1* the number how often a page is read is essentially less than the repeat-factor.

Overall, we have shown that for realistic page sizes the combination of both approaches (spatial sorting and restriction of the search space) reduces the CPU-time consumption of a spatial join by factors.

4.3 I/O-Time Tuning

In this section we address the problem of improving the I/O-performance of a spatial join performed on R*-trees. We assume the availability of an LRU-buffer organized by the underlying system in which the R*-tree is embedded. Multiple users share the LRU-buffer and therefore, the R*-tree might occupy only a small fraction of the buffer. Therefore, our goal is to achieve good I/O-performance with a buffer size as small as possible. In section 4.1, we have already shown in some experiments that for an LRU-buffer of reasonable size algorithm *SpatialJoin1* reads a page around 1.5 times on the average. Under the assumption that every page is required to answer the join, this means that *SpatialJoin1* is already close to optimum.

In order to reduce I/O-cost, the problem occurs to compute a read schedule of the pages required for the spatial join of two R*-trees such that for a given buffer the number of disk accesses is minimal. The read schedule determines the sequence how the pages are read form disk into the buffer. In [16] the problem of determining the optimal read schedule for a fixed size buffer has been shown to be NP-hard. Thus, we are interested in a heuristic solution that comes close to the optimum.

In a similar setting, heuristic solutions have already been discussed in [7]. However, these solutions assume that the required pairs of (data) pages are known before performing the join. Moreover, expensive preprocessing steps are necessary to build up some additional data structures. We call this approach a *global optimization policy*. In a spatial access method such as the R*-tree, a *local optimization* should be preferred such that the spatial partitioning and clustering of the R*-tree is exploited to determine the read schedule. A local optimization is less expensive than a global optimization with respect to CPU-time and main memory. A third optimization policy is the *algorithm-driven optimization*. In this approach, the read schedule is determined by the algorithm processing the spatial data. An example is the plane-sweep map overlay presented in [13]. But the algorithm-driven optimization is not generally applicable because it is restricted to a class of algorithms exploiting spatial order.

Our solution presented here uses a *local optimization policy based on spatial locality* as a main criterion for computing the read schedule. Such a policy tries to fill the buffer with those pages whose minimum bounding rectangles are close together in data space. Because of the overlap in the R*-tree, the corresponding entries referring to those pages might be spread over several paths of the tree. Thus, we decided to restrict our policy to one pair of pages.

In the following, we discuss three local optimization policies based on spatial locality. We discuss their I/O-performance using the results of experimental tests. These results are compared to the ones of algorithm *SpatialJoin1*, see section 4.1.

Local plane-sweep order

In section 4.2, we already described a plane-sweep algorithm to determine the intersections of two sets of rectangles. Based on spatial ordering this algorithm creates a sequence of pairs of intersecting rectangles. Obviously, this sequence can also be used to determine the read schedule of the spatial join. In addition to preserving spatial locality in the buffer, this approach can be used without any extra cost. In the following, we call this approach *local plane-sweep order* and the corresponding join algorithm *SpatialJoin3* (in short: *SJ3*). Note, that we assume in the following that algorithm *SortedIntersectionTest* does not use sequences of rectangles as introduced originally, but sequences of entries for in- and output.

```
SpatialJoin3 (R,S: R_Node; rect: Rectangle);
  R := {Ei| (Ei ∈ R) ∧ (Ei.rect ∩ rect ≠ ∅)};
  S := {Ei| (Ei ∈ S) ∧ (Ei.rect ∩ rect ≠ ∅)};
  Sort(R); Sort(S);
  SortedIntersectionTest (R,S, Seq);
  FOR i := 1 TO ||Seq|| DO
    (ER,ES) := Seq[i];
    IF (R is a leaf page) THEN (* (S is also a leaf page) *)
      Output (ER,ES)
    ELSE
      ReadPage(ER.ref); ReadPage(ES.ref);
      SpatialJoin3 (ER.ref,ES.ref, ER.rect ∩ ES.rect)
    END
  END
END SpatialJoin3;
```

Let us discuss the properties of the algorithm using the examples illustrated in Figure 6. Let us assume that a buffer consists of two pages and that the references belonging to the rectangles are pointing to data pages. In both examples two sequences of minimal bounding rectangles of subtrees (r_1, r_2, r_3, r_4) and (s_1, s_2) are processed. Note that s_1 is the only rectangle that differentiates the two examples. In the first example the sequence of intersections using local plane-sweep order is (I, II, IV, III). Thus, the read schedule is $<s_1,r_2,r_1,s_2,r_4,r_3>$. For this sequence, each data page on the next level of the tree has to be read into main memory once.

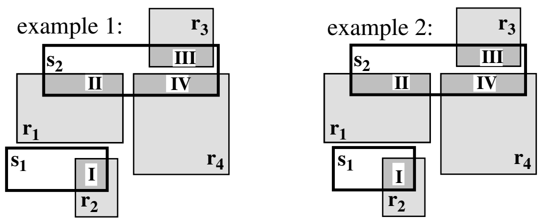


Figure 6: Examples for join situations

In example 2, the situation is very similar to example 1 but the xl-value of s_1 is now greater than the xl-value of r_1. This slightly different situation changes the read schedule. Now, the sequence of intersections is (II, I, IV, III) and the corresponding read schedule is <r_1, s_2, s_1, r_2, s_2, r_3, r_4>. Thus, the data page belonging to rectangle s_2 must be read twice into the buffer. The problem is that a page, whose corresponding rectangle frequently intersects other rectangles, is not completely processed in one step. In order to avoid such situations we complete the local plane-sweep order with a *pinning mechanism*.

Local plane-sweep order with pinning

The concept of *pinning* is based on the following idea. First, we determine a pair (E_R, E_S) of entries with respect to the local plane-sweep order. After processing the corresponding subtrees E_R.ref and E_S.ref, we compute the *degree* of the rectangles of both entries. The *degree* of an rectangle of an entry E, short *deg(E.rect)*, is given by the number of intersections between rectangle E.rect and the rectangles which belong to entries of the other tree not processed until now. Second, we pin the page in the buffer whose corresponding rectangle has a maximal degree. Third, the spatial join is performed on the pinned page with all other pages. Then, we determine the next pair of entries using the local plane-sweep order again. In example 2 of Figure 6, we obtain $deg(r_1) = 0$ and $deg(s_2) = 2$. Thus, the read schedule is <r_1, s_2, r_4, r_3, s_1, r_2>. A similar idea to make use of the degree of an entry was already presented in [7].

The local plane-sweep order approach extended by pinning works like algorithm *SpatialJoin3* with the following exception: If a pair of intersecting pages is in the directory, we compute the degree of the corresponding rectangles after joining the pair. The page whose rectangle has a maximal degree is pinned and the spatial join is completely performed for the pinned page. Then the algorithm continues as before. In the following, we call the corresponding join algorithm *SpatialJoin4* (in short: *SJ4*).

Local z-order

Until now, we have presented two approaches based on the plane-sweep algorithm which has already been used to reduce CPU-time. Let us now consider the spatial join of two nodes with respect to spatial sorting. First, we compute the intersections between each rectangle of the one node and all rectangles of the other node. Then, we sort the rectangles according to the spatial location of their centers. Thus, the problem occurs to sort two-dimensional points. A common way for sorting multidimensional points is based on space-filling curves, for example the Peano-curve, also called z-ordering [17]. The basic idea is to decompose the underlying space into cells of equal size and provide an ordering on this set of cells.

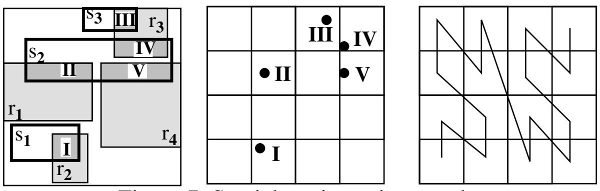


Figure 7: Spatial sorting using z-order

An example is given in Figure 7. First, all intersections are computed between the rectangles (r_1,…,r_4) and (s_1,…,s_3). This results in a set of rectangles (I,…,V). The centers of the rectangles are represented by the dots in the grid. The order of the cells of the grid is determined by z-ordering which is illustrated in the rightmost diagram of Figure 7. Thus the sequence of intersection is (I, II, III, V, IV). As introduced for the local plane-sweep order, we use this approach with pinning of page and therefore, the read schedule is <s_1,r_2,r_1,s_2,r_4,r_3,s_3>. The corresponding algorithm is named *SpatialJoin5*, short *SJ5*.

Comparison

In Table 5, we compare the I/O-performance of the different approaches presented in this subsection. Several experiments were performed for different page and buffer sizes. For a given page size of 4 KByte, the columns entitled "SJ3", "SJ4" and "SJ5" show the number of disk accesses for the algorithms *SJ3*, *SJ4* and *SJ5*, respectively. The results vary in these columns in the size of the available LRU-buffer. We observe the following effects:

- The local plane-sweep order approach (*SJ3*) considerably profits from pinning (*SJ4*) of pages, particularly, if the buffer is small. With a buffer size of 512 KByte the I/O-performance is nearly the same. Because the CPU-performance of both approaches is the same, the algorithm using local plane-sweep order with pinning (*SJ4*) is clearly better than *SJ3*.
- For small buffer sizes the z-order approach (*SJ5*) is slightly better than the plane-sweep approach (*SJ4*), for large buffer sizes it is vice versa. But in order to compute the z-order of the intersections, we need CPU-time. Even, if there is a small I/O-gain, the loss of CPU-time is not compensated.

buffer size	SJ3	SJ4	SJ5
0 KByte	6,085	5,384	5,290
8 KByte	6,062	5,366	5,248
32 KByte	4,678	4,246	4,178
128 KByte	3,117	3,008	2,947
512 KByte	2,399	2,373	2,392

Table 5: Number of disk accesses of algorithms *SJ3*, *SJ4* and *SJ5*

In Table 6, we compared the I/O-performance of algorithm *SpatialJoin1* with the one of *SpatialJoin4*. Column "SJ4" refers to the number of disk accesses for algorithm *SJ4* and column "(%)" gives the ratio of the number of accesses required for *SJ4* to the number of accesses required for *SJ1* expressed in per cent. Row "optimum" gives the optimum number of disk accesses which corresponds to |R|+|S|, the total number of pages in both R*-trees. Results depend on the buffer size and on the page size. Table 6 shows the following results and effects:

- The approach using algorithm *SJ4* requires up to 45% less disk accesses than the algorithm *SJ1*. For a reasonable size of the buffer, the number of accesses is close to the optimum. Moreover, algorithm *SJ4* achieves about the same I/O-performance as *SJ1* using a buffer which is smaller by a factor of 3 to 6.
- The performance gain of *SJ4* in comparison to *SJ1* depends on both, buffer and page size. In detail, the larger the page, the bigger is the buffer size with the greatest gain over algorithm *SJ1*, indicated by the gray colored cells in Table 6. The reason is that the buffer performance does not only depend on the total buffer size, but also on the number of buffer pages. This number decreases with increasing page size.

page size:		1 KByte		2 KByte		4 KByte		8 KByte	
		SJ4	(%)	SJ4	(%)	SJ4	(%)	SJ4	(%)
buffer size (KByte)	0	23,088	93.4	11,530	92.4	5,384	94.1	2,703	95.3
	8	17,513	86.2	10,632	88.5	5,366	93.8	2,703	95.3
	32	12,704	92.0	7,436	77.5	4,246	77.9	2,552	90.4
	128	10,856	95.6	5,685	90.3	3,008	67.2	1,857	69.4
	512	9,385	90.5	5,108	102.9	2,373	85.7	1,186	54.4
optimum		8,442		4,197		2,091		1,042	

Table 6: I/O-performance given by number of dissk accesses

Overall, local plane-sweep ordering in combination with pinning of pages, i.e. algorithm *SJ4*, seems to be the most efficient approach. The I/O-performance is improved in comparison to the original algorithm *SJ1* and to the algorithm *SJ3* without pinning. Moreover, the I/O-performance is almost the same as the one for algorithm *SJ5* which makes use of z-ordering. However, the preprocessing step for algorithm *SJ5* is more expensive than the one for algorithm *SJ4*. Nevertheless, the I/O-performance of both approaches is very close to the optimum number of disk accesses.

4.4 Spatial Join of R*-trees with Different Height

In our previous discussion on spatial joins, we made the assumption that the R*-trees are of the same height. In this subsection, we discuss the special case of different heights. In the following, we assume that R*-tree R is higher than R*-tree S.

There is no difference in join processing to our previous algorithms as long as both nodes contain directory entries. Let N_R be a directory node of R*-tree R, N_S be data node of R*-tree S and, let us consider the spatial join between node N_R and N_S.

Now, the idea is to perform window queries on the subtrees rooted in N_S using the data rectangles of N_R as query rectangles. First, we compute all intersecting pairs of entries (E_R, E_S) with E_R.rect $\cap$ E_S.rect $\neq \emptyset$. We have considered three different methods for performing the window queries: *(a)* for each entry E_S, we perform a window query on E_R.ref, if E_R.rect $\cap$ E_S.rect $\neq \emptyset$, *(b)* for each entry E_R, all window queries with query rectangles E_S.rect, where E_S.rect $\cap$ E_R.rect $\neq \emptyset$, are performed in the subtree rooted in E_R.ref in one step, or *(c)* we use an order based on spatial locality such as local plane-sweep ordering with pinning. The difference between the first two policies is that for policy (a) the nodes of the subtree rooted in N_R are read into the buffer whereas for policy (b) the nodes of the subtree are read only once. Thus, as long as all pages of the subtree can be kept in the buffer, there is no difference in I/O-performance. Otherwise, we expect that policy (b) performs better than policy (a).

Table 7 presents the results of the experiments comparing these three versions. We have used similar data sets as before with the exception that R*-tree R stores 598,677 records instead of 131,461. With a page capacity of 2 KByte the height of R*-tree R is 4 and the height of R*-tree S is 3. The spatial join was performed using *SpatialJoin4*, but the join of directory and data nodes followed the policies, described above.

buffer size	(a)	(b)	(c)
0 KByte	111,140	24,111	27,679
8 KByte	27,586	23,288	23,822
32 KByte	18,019	17,936	17,954
128 KByte	14,453	14,453	14,454
512 KByte	13,038	13,038	13,038

Table 7: I/O-performance in case of R*-trees of different height

As expected, policies (b) and (c) outperform policy (a). For very small buffers, version (b) is superior to policy (c). The reason is that policy (b) guarantees that each page of the subtrees is read only once into the buffer, whereas policy (c) shows similar performance only in the case of a medium-sized and large buffer.

Overall, we get the best I/O-performance by joining the entries of two directory pages according to the local plane-sweep order with pinning *(SpatialJoin4)* and by following policy (b) in case of the join of a directory and a data page.

5 Summary of the Performance Comparison

In the previous sections, we compared the performance of various spatial join algorithms using two sample spatial relations. Most interesting in our concluding comparison is the validation of our previous results using other spatial relations for performing spatial joins. Furthermore, we present the results with respect to the overall execution time. Under the same assumption as in section 4.1, the estimation for the execution time of *SpatialJoin4* is presented in Figure 8. In analogy, we charged $1.5*10^{-2}$ seconds for positioning the disk arm, $5*10^{-3}$ seconds for transferring 1 KByte of data from disk and $3.9*10^{-6}$ seconds for a floating point comparison. Contrary to *SpatialJoin1*, *SJ4* achieves best performance for a page size of 8 KByte and we expect even better performance for a page size of 16 KByte. This observation can be made independent of the buffer size, as shown in the upper diagram of Figure 8. Originally, the execution time of *SpatialJoin1* was clearly determined by CPU-time. However, the lower diagram of Figure 8 shows that the execution time of *SpatialJoin4* is now I/O-bound. Only if the page size is rather large, the execution time will be CPU-bound again.

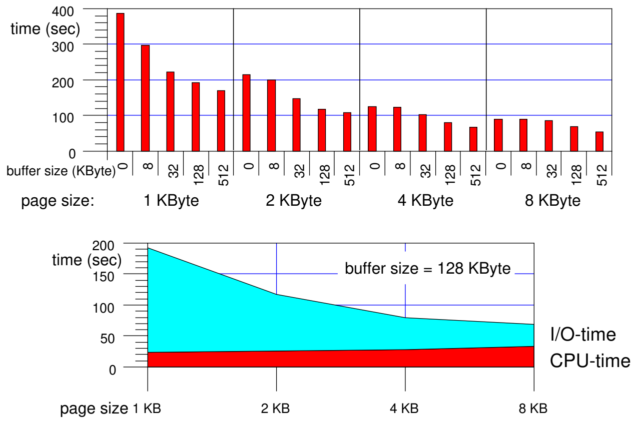


Figure 8: Total join time and ratio between CPU- & I/O-time

The factor of the performance improvement of algorithm *SpatialJoin4* in comparison to algorithm *SpatialJoin1* is illustrated in the upper diagram of Figure 9. For a page size of 4 KByte, *SpatialJoin4* performs about 5 times faster than *SpatialJoin1*. For larger page sizes, the factor increases, whereas for smaller page sizes, e.g. 1 KByte, the factor decreases. The lower diagram of Figure 9 depicts the factors of the performance improvement of *SJ4* in comparison to algorithm *SpatialJoin2*.

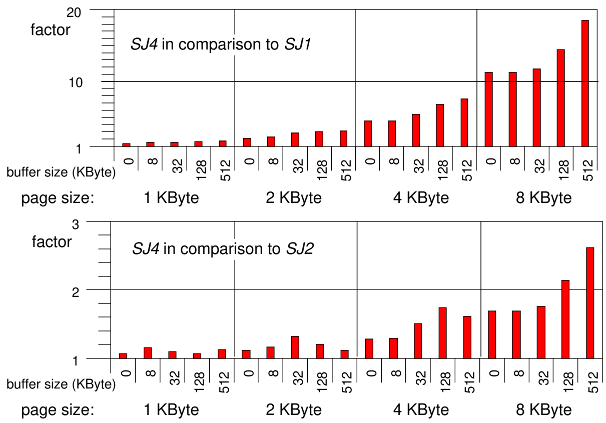


Figure 9: Overall improvements in respect to total join time

In order to investigate whether our previous results depend on the particular data, we have performed several spatial joins with various spatial relations obtained from real geographic applications. In Table 8, we report the most import characteristics of our tests (A) to (E). For test (D), we have performed a spatial join with two identical R*-trees. Nevertheless, our algorithms treated the R*-trees as if they would be different. For test (E), region data instead of line data was used for performing spatial joins [24].

	R*-tree R		R*-tree S		inter-sections
	$\|\|R\|\|_{dat}$	subject of map	$\|\|S\|\|_{dat}$	subject of map	
(A)	131,461	streets	128,971	rivers & railways	86,094
(B)	131,461	streets	131,192	streets	154,262
(C)	598,677	streets	128,971	rivers & railways	395,189
(D)	128,971	rivers & railways	128,971	rivers & railways	505,583
(E)	67,527	region data	33,696	region data	543,069

Table 8: Characteristics of R*-trees in various test A-E

In Figure 10, we depict the improvement factor how many times algorithm *SpatialJoin4* performs better than *SpatialJoin1* for different page sizes. We have assumed a buffer size of 128 KByte. The results confirm the factors of performance improvement over test (A) already frequently being quoted in the previous sections. There is only one exception: test (C) with 2 KByte pages has a lower improvement because the R*-trees have different height and therefore, the window queries do not profit very much from sorting.

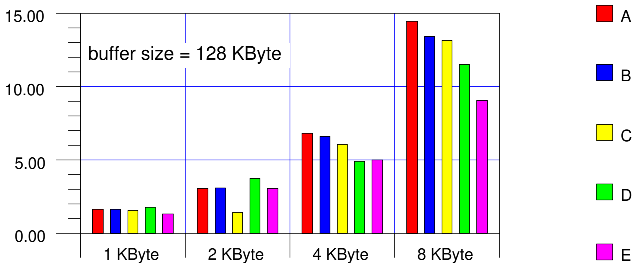


Figure 10: Improvement factors for different real test data

6 Conclusions and Future Work

In a spatial database system, spatial joins are one of the most important operations for combining spatial objects of several relations. Contrary to window queries, execution time is superlinear in the number of objects and therefore, spatial joins may require more than one access to each of the objects. Moreover, approaches suggested for performing traditional joins, e.g. natural-joins, are not applicable to spatial joins.

In this paper, we have presented the first detailed study of spatial join processing using R-trees, particularly R*-trees. The R*-tree is one of the most efficient members of the R-tree family. Several database and geographic information systems already use R-trees as their basic spatial access method. Therefore, it is of considerable interest to design efficient join algorithms particularly for R*-trees.

First, we have presented a straightforward join algorithm for R*-trees using already an LRU-buffer to reduce the I/O-time. We have shown that this approach requires optimization with respect to both CPU- and I/O-time. In order to improve CPU-time, we suggest two techniques which make use of *spatial sorting* and *restricting the search space*. Moreover, I/O-performance has been improved by determining how the required pages are read into the buffer. Our suggested policies are based on *spatial locality*, such that the pages required for computing an answer to the spatial join are already in the buffer with high probability. In an experimental performance comparison using large relations of real data, we showed that our suggested techniques improve the execution time of the first approach by factors. With respect to the results of our experiments, we can state that the technique of restricting the search space improves the number of comparisons by a factor of 4 to 8, and the technique of spatial locality reduces the number of disk accesses up to 45%. Independent of the time required for one comparison and of the time for one disk access, the execution time of the approach using all of our suggested techniques is considerable better than the approaches using only a few or even none of the suggested techniques.

In our future work, we are particularly interested in a more detailed study of spatial joins. In this paper, we put our emphasis on the MBR-spatial-join which computes the spatial join of the minimum bounding rectangles of the spatial objects. We are in the process of investigating more complex types of spatial joins which actually operate on the real spatial objects. Until now, all our approaches assume a conventional computer architecture. However, parallel computer systems and disk arrays are very interesting for performing spatial joins and window queries, for example using parallel R-trees [14].

References

[1] Becker, L. A.: '*A New Algorithm and a Cost Model for Join Processing with Grid Files*', PhD-thesis, University of Siegen, 1992.

[2] Beckmann N., Kriegel H.-P., Schneider R., Seeger B.: *'The R*-tree: An Efficient and Robust Access Method for Points and Rectangles',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Atlantic City, N.J., 1990, pp. 322-331.

[3] Burrough P. A.: *'Principles of Geographical Information Systems for Land Resources Assessment',* Oxford University Press, 1986.

[4] Bureau of the Census: *'Tiger/Line Precensus Files: 1990 technical documentation'*, Bureau of the Census, Washington, DC, 1989.

[5] Bentley J.L., Wood D.: *'An Optimal Worst Case Algorithm for Reporting Intersections of Rectangles'*, IEEE Trans. on Computers, Vol. C-29, No. 7, 1980, pp. 571-577.

[6] Faloutsos, C.: *'Gray Codes for Partial Match and Range Queries'*, IEEE Trans. on Software Engineering, Vol. 14, No. 10, 1988, pp. 1381-1393.

[7] Fotouhi F., Pramanik S.: *'Optimal Secondary Storage Access Sequence for Performing Relational Join'*, IEEE Trans. on Knowledge and Data Engineering, Vol. 1, No. 3, 1989, pp. 318-328.

[8] Gargantini, I. : *'An Effective Way to Represent Quadtrees'*, Comm. of the ACM, Vol. 25, No. 12, 1982, pp. 905-910.

[9] Günther, O.: *'Efficient Computations of Spatial Joins'*, Proc. 9th Int. Conf. on Data Engineering, Vienna, Austria, 1993.

[10] Guttman A.: *'R-trees: A Dynamic Index Structure for Spatial Searching',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Boston, MA., 1984, pp. 47-57.

[11] Harada L., Nakano M., Kitsuregawa M., Takagi M.: *'Query Processing Methods for Multi-Attribute Clustered Relations'*, Proc. 16th Int. Conf. on Very Large Data Bases, Brisbane, 1990, pp. 59-70.

[12] Hoel E. G., Samet H.: *'A Qualitative Comparison Study of Data Structures for Large Line Segment Databases',* Proc. ACM SIGMOD Int. Conf. on Management of Data, San Diego, CA., 1992, pp. 205-214.

[13] Kriegel H.-P., Brinkhoff T., Schneider R.: *'An Efficient Map Overlay Algorithm based on Spatial Access Methods and Computational Geometry',* Proc. Int. Workshop on Database Management Systems for Geographical Applications, Capri, Italy, 1991, in: Geographic Database Management Systems, Springer, 1992, pp. 194-211.

[14] Kamel, I., Faloutsos, C.: *'Parallel R-Trees'*, Proc. ACM SIGMOD Int. Conf. on Management of Data, San Diego, CA., 1992, pp. 195-204.

[15] Mishra P., Eich M.H.: *'Join Processing in Relational Databases'*, ACM Computing Surveys, Vol. 24, No. 1, 1992, pp. 63-113.

[16] Merret T., Kambayashi Y., Yasuura H.: *'Scheduling of Page-Fetches in Join-Operations'*, Proc. 7th Int. Conf. on Very Large Data Bases, Cannes, 1981, pp. 488-498.

[17] Orenstein J. A., Merrett T. H.: *'A Class of Data Structures for Associative Searching',* Proc. 3rd ACM SIGACT/SIGMOD Symp. on Principles of Database Systems, 1984, pp. 181-190.

[18] Orenstein J. A.: *'Spatial Query Processing in an Object-Oriented Database System',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Washington D.C., 1986, pp. 326-333.

[19] Orenstein J. A.: *'Redundancy in Spatial Databases',* Proc. ACM SIGMOD Int. Conf. on Management of Data, Portland, Oreg., 1989, pp. 294-305.

[20] Preparata F. P., Shamos M. I.: *'Computational Geometry',* Springer, 1988.

[21] Rotem D.: '*Spatial Join Indices*', Proc. Int. Conf. on Data Engineering, 1991, pp. 500-509.

[22] Samet H.: *'The Design and Analysis of Spatial Data Structures',* Addison Wesley, 1990.

[23] Stonebraker M., Rowe L., Hirohama M.: *'The Implementation of POSTGRES'*, IEEE Trans. on Knowledge and Data Engineering, Vol. 2, No. 1, 1990, pp. 125-142.

[24] Statistical Office of the European Communities: *'Regions'*, 1990.