Published in the Proceedings of 16th International Conference on Data Engineering (ICDE 2000)

Data Redundancy and Duplicate Detection in Spatial Join Processing

Jens-Peter Dittrich   Bernhard Seeger

Department of Mathematics and Computer Science
University of Marburg
Hans-Meerwein-Straße, 35032 Marburg, Germany
{dittrich,seeger}@mathematik.uni-marburg.de

Abstract

The Partitioned Based Spatial-Merge Join (PBSM) of Patel and DeWitt and the Size Separation Spatial Join (S^3J) of Koudas and Sevcik are considered to be among the most efficient methods for processing spatial (intersection) joins on two or more spatial relations. Both methods do not assume the presence of pre-existing spatial indices on the relations. In this paper, we propose several improvements of these join algorithms. In particular, we deal with the impact of data redundancy and duplicate detection on the performance of theses methods. For PBSM, we present a simple and inexpensive on-line method to detect duplicates in the response set. There is no need anymore for eliminating duplicates in a final sorting phase as it has been suggested originally. We also investigate in this paper the impact of different internal algorithms on the total runtime of PBSM. For S^3J, we break with the original design goal and introduce controlled redundancy of data objects. Results of a large set of experiments with real datasets reveal that our suggested modifications of PBSM and S^3J result in substantial performance improvements where PBSM is generally superior to S^3J.

1. Introduction

With the increasing availability of multidimensional data in various forms, advanced database management systems (DBMS) such as spatial DBMS have been emerged during the last decade. One of the most important operations in a spatial DBMS [Güt 94] is the *spatial join (SJ)* which is the counterpart to the intersection join in a relational DBMS. The spatial join operation combines two (or more) sets of spatial objects according to a *spatial predicate* (e.g.: “intersection”, “distance within”). In this paper, we are primarily interested in spatial intersection joins which are considered to be the most important type of spatial join. In order to reduce its complexity, the spatial join is processed in two steps [Ore 86]. In the *filter step*, a spatial join is performed using conservative approximations of the spatial objects such as the rectilinear minimum bounding rectangle (*MBR*). The filter step produces a candidate set that contains all answers of the spatial join. In the *refinement step*, the candidates are tested using their exact geometry whether they really satisfy the join predicate. According to the availability of indices the methods for processing the filter step can be classified into three classes: availability of indices on both relations, on one of the relations or on none of the relations. The last class of methods has received most of the research attention, recently. The Partitioned Based Spatial-Merge Join (*PBSM*) of Patel and DeWitt [PD 96] and the Size Separation Spatial Join (S^3J) of Koudas and Sevcik [KS 97] are considered to be among the most efficient join methods that do not assume the availability of pre-existing indices.

In this paper, we put our focus on PBSM and S^3J. We present several modifications of the original algorithms which lead to considerable runtime improvements. In particular, we deal with the impact of redundancy and duplicate detection on the performance of theses methods. For PBSM, we present a simple and inexpensive on-line method to detect duplicates in the response set. There is no need anymore for sorting the candidate set as it has been suggested originally. One of the most important building blocks of PBSM is its internal algorithm for processing the join on a pair of partitions in memory. In this paper, we also investigate the impact of different internal algorithms on the total processing cost. For S^3J, we introduce replication of data objects and show that then the total processing cost can be reduced considerably. Duplicates in the response set can be detected at very little cost using a slightly

modified version of the method suggested for PBSM. Moreover, we also address the problem of choosing an efficient internal algorithm for S^3J.

Most work on spatial join processing focuses on the efficient computation of the filter step where MBRs are used for approximating the objects. According to the classification introduced above spatial joins can be in one of the following three classes: whether an index exists for both of the input relations, for exactly one of the relations or none of the relations.

- Index on both relations:
  In [BKS 93] a spatial join algorithm was presented where each of the relations is indexed by an R*-tree. This algorithm synchronously traverses both trees and joins all pairs of overlapping regions. This join method is considered as one of the most important ones due to its efficiency and the availability of R-trees in advanced DBMS. A different traversal strategy was presented in [HJR 97] which is superior to the one of [BKS 93] when a large buffer is available. [HS 95] considered the problem of spatial joins when PR quadtrees are on the input relations. A generalization of join processing to a broader class of index structures has been presented in [Gün 93].
- Index on one relation:
  The work on seeded trees [LR 94] was the first that addressed the problem of processing spatial joins when only one R-tree is available. It was suggested to build up the second R-tree using the available tree as a skeleton and to use then one of the algorithms for processing a spatial join on two R-trees. An improvement of this algorithm has been suggested in [MP 99] where the available main memory is exploited more efficiently.
- No indices:
  Recently, the problem of join processing has been examined under the assumption that no index is available. One of the first proposals [GS 87] suggests to use an external version of a computational geometry algorithm (using external segment trees). The spatial-hash join [LR 96] and PBSM [PD 96] divides the datasets into smaller partitions and apply a join algorithm to each pair of partitions. PBSM replicates some of the data of both input relations to improve join processing, whereas the spatial-hash join only allows replication on one relation. Experimental results in [KS 97] have revealed that the performance of PBSM is comparable to the spatial-hash join. The details of PBSM are presented later in Section 3. A method without data replication is S^3J that performs the join by exploiting a variant of quadtrees. S^3J is fully described in Section 4. Another approach without data replication, called Scalable Sweeping-Based Join (SSSJ) [APR+ 98], is an external sweep-line algorithm which tries to keep the status of the sweep-line in main memory. Similar to the sort-merge algorithm in case of equi joins, SSSJ is the most promising algorithm for processing spatial joins w.r.t. worst-case efficiency. A serious deficiency of SSSJ, when it is used in a DBMS, is however that both input relations have to be sorted first before producing the first output tuple. In [Gra 93] it is argued that such algorithms are not suitable for a DBMS where they would block a pipelined processing in an operator tree. Moreover, a comparison of SSSJ and PBSM showed that both methods perform similarly efficient for real datasets, whereas only for artificial, highly skewed datasets SSSJ is generally superior.

Other work on spatial joins deals with improving the refinement step [BKSS 94, ZS 98]; using other approximations in the filter step [Ore 86]; exploiting parallelism [BKS 96, Pat 98] or using join algorithms for more than two dimensions [KS 98]. These issues are however beyond the scope of this paper.

The remainder of this paper is organized as follows. Section 2 introduces our model and the datasets we used in our experiments. In Section 3, we put our focus on PBSM. We first describe the algorithms as they have been described originally. Thereafter, we present our modifications and show that they really pay off. Section 4 is dedicated to S^3J. After a full description of the original algorithm we show that data replication can also be very beneficial for S^3J. Section 5 presents a comparison of the new versions of PBSM and S^3J.

2. Preliminaries

In the following, we consider two spatial relations R and S being used as input to the spatial join. Each of the relations corresponds to a set of so-called *key-pointer elements* (KPE) where a KPE consists of an identifier of an object and its rectilinear minimum bounding rectangle (MBR). A MBR r is represented by its lower left corner (r.xl, r.yl) and its upper right corner (r.xh, r.yh). We assume that R and S are too large for being kept together in main memory. The total runtime of an algorithm for processing a spatial join is therefore also determined by the number of required I/O operations. Moreover, we assume that the spatial join is performed within an operator tree where a node represents an operator which satisfies the open-next-close interface [Gra 93]. We do not take into account the cost of the other operators in the operator tree, i. e. reading the input relations R and S and writing the output of the spatial join are assumed to be free of charge in our model.

In our I/O model we assume that data is transferred between main memory and secondary storage in pages of fixed size. The cost for reading a page

dataset	description	number of MBRs	cover-age
LA_RR	railways and rivers LA	128,971	0.22
LA_ST	streets, LA	131,461	0.03
LA_RR(*p*)	LA_RR increased by p^2	128,971	0.22 p^2
LA_ST(*p*)	LA_ST (see above)	131,461	0.03 p^2
CAL_ST	streets, california	1,888,012	0.12

Table 1. Datasets used in the experiments

join	R	S	number of results	selectivity
J1	LA_RR	LA_ST	85.854	$5.06 \cdot 10^{-6}$
J2	LA_RR(2)	LA_ST(2)	305.537	$1.60 \cdot 10^{-5}$
J3	LA_RR(3)	LA_ST(3)	671.775	$3.96 \cdot 10^{-5}$
J4	LA_RR(4)	LA_ST(4)	1.195.527	$7.05 \cdot 10^{-5}$
J5	CAL_ST	CAL_ST	9.784.072	$2.74 \cdot 10^{-6}$

Table 2. The spatial joins of the experiments

consists of positioning the disk arm and transferring the page. Moreover, we also allow a sequence of n contiguous pages, $n \geq 1$, being read (written) with positioning the disk arm only once. Let PT be the ratio of positioning time to transfer time. The I/O cost of such a request is then simply $PT + n$ (page transfer units).

In the following, we investigate the performance of spatial join algorithms in different experiments using real datasets. The performance of the algorithms is simply measured by the total runtime (in seconds) of the corresponding C++ implementations on a Sun Sparcstation 20 (with a Supersparc II processor, 196 MB main memory and a 2 GB Seagate disk) under Solaris 2.6. Our current implementations do not support an overlapped processing of I/O and CPU. Recall also that the time for reading the input relations and for writing the output does not contribute to the total runtime. The join algorithms were allowed to use only a fraction of main memory. In order to turn off the buffers of the operating system, the I/O operations are performed on a file system where the direct I/O option was turned on.

In our experiments we used different real datasets derived from the TIGER files [Bur 89] (see Table 1). The datasets LA_RR and LA_ST contain the MBRs of different types of lines from the LA region. These datasets have already been used in previous experiments [BKS 93]. The coverage of the data files (which is defined by the sum of the area of the rectangles divided by the area of the MBR of all rectangles) is rather small for these files. In order to generate files with a larger cover we increased both edges of the rectangles in the files LA_RR and LA_ST by a factor of *p*. The resulting datasets are called LA_RR(*p*) and LA_ST(*p*). Moreover, we also used a very large file CA_ST in our experiments that contains about 1.9 million MBRs. The file is derived from the total set of street lines of California. In Table 2 some of the spatial joins are described which we performed in our experiments. The selectivity refers to the number of results divided by the product of the number of MBRs of the input relations.

3. PBSM

Partition Based Spatial Merge Join (PBSM) [PD 96] is a divide & conquer algorithm that breaks up the input relations R and S into partitions using an equidistant grid. Similar to hash joins it is then sufficient to compute the join for pairs of partitions (where one belongs to R and the other belongs to S). In this section, we first present PBSM as it was originally proposed in [PD 96]. Then, we present different modifications and show that they really pay off in comparison to the original proposal.

3.1. Review of PBSM

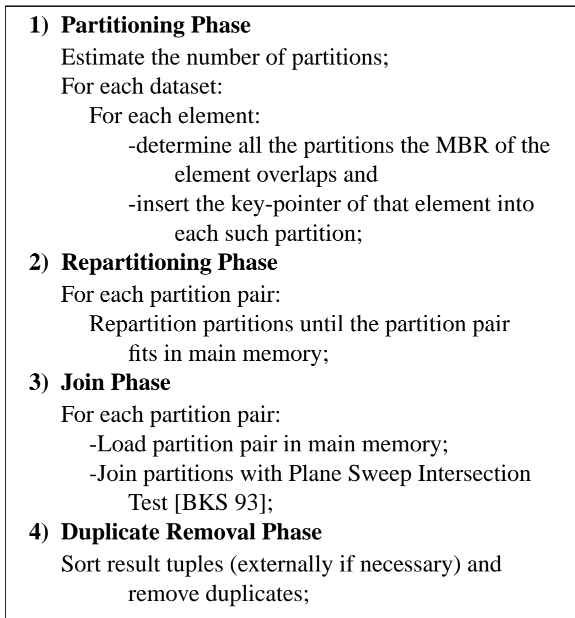
PBSM performs in four phases which are described briefly in the following. In the first phase, the number of partitions P is computed such that the join of a pair of partitions can be processed in main memory (with high probability). Let M be the size of the available main memory and $\|R\|$ ($\|S\|$) be the number of records in a relation *R (S)*. The size of a KPE is denoted by *sizeof(KPE)*. Then, P is given by the following formula:

$$P = \left\lceil \frac{(\|R\| + \|S\|) \cdot sizeof(KPE)}{M} \right\rceil \quad (1)$$

For each of the input relations, P partitions are created by using an equidistant grid with NT cells, $NT \geq P$. A cell of the grid, also termed *tile*, is then assigned to one of the partitions. A KPE of a relation is inserted into a partition of the relation if its rectangle intersects with one of the tiles that belong to the partition. This rule obviously results in replication of KPEs, i. e. a KPE can be stored in more than one partition. The advantage of assigning multiple tiles to a partition is that the KPEs are almost uniformly distributed among the partitions. Patel and DeWitt [PD 96] suggest to use a hash function for mapping the tiles to partitions.

Let $R_1,\ldots,R_P$ and $S_1,\ldots,S_P$ be the partitions of the relations R and S, respectively. In the second phase, pairs (R_i, S_i) of partitions are treated which do not fit into main memory. For these pairs of partitions, repartitioning has to be performed in a recursive fashion. In the third phase, a corresponding pair of parti-

tions is loaded into main memory and an in-memory join is performed using the plane-sweep method originally presented in [BKS 93]. Note that due to the replication of KPEs in different partitions the same result can be produced more than once. In order to eliminate these duplicates in the response set, the results obtained from the three phases are sorted in a final phase. The algorithm skeleton of Figure 1 gives a summary of the four phases.



1) **Partitioning Phase**
   Estimate the number of partitions;
   For each dataset:
      For each element:
         -determine all the partitions the MBR of the element overlaps and
         -insert the key-pointer of that element into each such partition;
2) **Repartitioning Phase**
   For each partition pair:
      Repartition partitions until the partition pair fits in main memory;
3) **Join Phase**
   For each partition pair:
      -Load partition pair in main memory;
      -Join partitions with Plane Sweep Intersection Test [BKS 93];
4) **Duplicate Removal Phase**
   Sort result tuples (externally if necessary) and remove duplicates;

Figure 1. The PBSM Algorithm

In [PD 96] it was shown that the duplicate removal phase of PBSM has a substantial impact on the total runtime. The authors suggest to delay removal until the tuples are entirely processed in the filter step. In the refinement step, tuples can be sorted w.r.t. the physical position of the objects which additionally results in a considerable reduction of the number of random disk accesses. This however causes a few serious problems. First of all, there is overhead because redundant result tuples will lead to additional I/O- and CPU-operations in a DBMS. In an operator-tree environment, it might be advantageous to have separate operators for the filter step and the refinement step. Due to query optimization it would then be likely that the refinement operator does not immediately follow the filter operator. However, duplicated records have to be processed in the intermediate operators which are on the path from the filter operator to the refinement operator. Second, PBSM cannot take full advantage of kernel approximations [BKSS 94] which are important to identify a candidate being an answer without investigating the exact geometry. Third, this algorithm also prevents a pipelined processing since the first output can be produced in the fourth phase only after the candidate set is completely sorted. The algorithm is therefore in this form not suitable for a DBMS (see for example the discussion in [Gra 93]). In the next subsection, we present a new inexpensive method for eliminating the duplicates of PBSM in its filter step.

3.2. Improvements

In this section, we first present an efficient on-line method for detecting duplicates in the response set immediately when they are produced. Results of our experiments show that the new method gives considerable performance improvements. Second, we discuss different internal sweep line algorithms and compare their performance. Third, important implementation details like the re-partitioning phase are discussed.

3.2.1. On-line duplicate removal. The technique we suggest is known from eliminating duplicates when queries are supported on a spatial access method based on the technique of clipping. It is called *Reference Point Method (RPM)* [See 91]. Let us consider

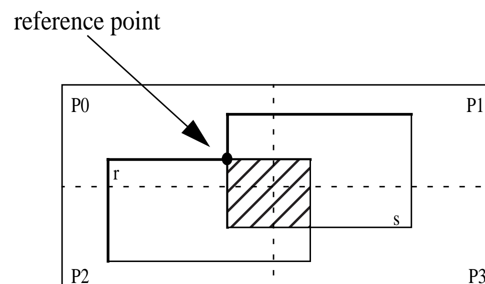


Figure 2. Reference Point Method

the situation in Figure 2, where PBSM creates for each of the relations R and S four partitions. Let r and s be rectangles which belong to relation R and S, respectively. Rectangles r and s are stored in each of the four corresponding partitions. Therefore, PBSM would generate the result (r,s) four times in the join phase. The basic idea of RPM is to assign each element of the result set to one partition using a unique point of the intersecting rectangle. For a pair of intersecting rectangles (r, s) we define the reference point as follows:

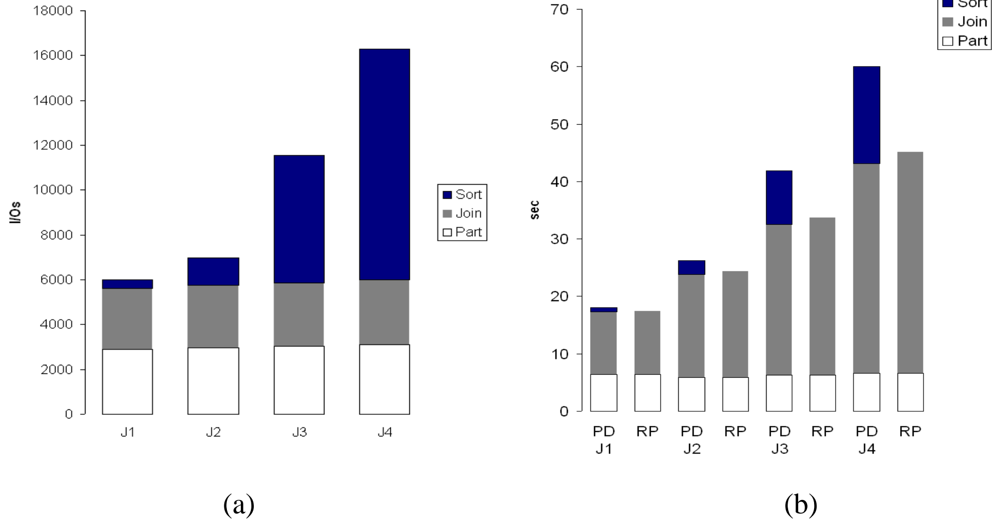
$$x = (max(r.xl, s.xl), min(r.yh, s.yh))$$

The x-coordinate of x is the maximum of the *left edges* of r and s and the y-coordinate is the minimum of the *upper edges* of r and s.

We suggest the following modification of PBSM. A *result tuple will be reported only if the point x is inside the region of the partitions being actually processed.* Let us consider again the example in Figure 2. Since point x is inside of the region P0, the result will only be reported when the corresponding partitions R_0 and S_0 are processed. The same pair will also be detected when the other partitions are processed, but it will not reported anymore as a result.

This technique is generally applicable in situations in which the data space is divided into disjoint parti-

tions. For example, S^3J also fulfills this requirement and therefore, RPM can also be applied as it will be shown in Section 4. It ensures that each result tuple will be reported exactly *once*. Temporary storing result tuples and duplicate removal in a separate phase is avoided at the additional cost of at most six comparisons (when a result is detected) which are required for computing the reference point and testing the point being in the region of the partition.



(a) (b)

Figure 3. Reference Point Method

In order to show the impact of the new method for duplicate removal we present the results of the experiments of spatial joins J1-J4 (see Table 2). We assume here that the size of the available main memory is 2.5 MB. Figure 3a and Figure 3b shows the I/O costs and response times of PBSM for the different joins, respectively. The upper parts of the boxes in Figure 3a show the *I/O-overhead* of PBSM when the results of the join phase have to be sorted in order to remove duplicates. The I/O-cost of the duplicate removal step depends on the size of the result set. The larger the result set the higher is the cost for sorting. PBSM with RPM avoids this overhead completely. In Figure 3b, the total response times of PBSM are plotted for joins J1-J4, where PD refers to the original approach and RP refers to PBSM with RPM. It reveals that PBSM with RPM is considerably faster. Moreover, using RPM for duplicate removal has the following additional benefits: kernel approximations can be used to produce the first results already in the filter step and a pipelined processing within an operator tree is supported more efficiently.

3.2.2. Internal plane sweep algorithms. One of the most important building blocks of PBSM is its algorithm for processing a spatial join in main memory. Originally, PBSM uses the plane-sweep method of [BKS 93] called *Plane Sweep Intersection-Test* which implicitly organizes the status of the sweep line in a list. The advantage of this method is that its implementation is straightforward and that it gives reasonable performance when applied to joining the partitions of PBSM. Note that the algorithm performs poorly when it is directly applied to joining two sets of rectangles in main memory.

The following simple analysis clarifies that the plane-sweep method generally performs well for PBSM. Let n be the number of records in relations R and S. Let P be the number of partitions. We make the common assumption [GS 87] that the sweep-line will intersect with $O(\sqrt{n})$ rectangles. The runtime of the Plane Sweep Intersection-Test applied to the total dataset is then $O(\sqrt{n} \cdot n)$. Now let us consider the situation when a join is performed on a partition of PBSM. We can expect that $O(\sqrt{n}/P)$ rectangles intersect with the sweep line and therefore the running time for joining a partition is $O((\sqrt{n}/P) \cdot n/P)$. Hence, the total running time of the algorithm applied to all partitions is by a factor of P less than the runtime when it is applied to the total dataset. It seems that P should then be as large as possible, but for large P our analysis does not hold since the duplication rate of the data objects will have an important impact on the performance. From this analysis we can make the following interesting observation: the runtime of PBSM will generally not improve with an increasing main memory (which results in smaller values of P). This observation is also confirmed in our experiments (see for example Figure 5).

At least for a large main memory it is therefore important to choose a different internal algorithm for processing the spatial join. For example, [APR+ 98] suggests to use a special type of dynamic interval trees [CLR 90] for organizing the status of the sweep line. In our implementation we decided to use interval *tries* [Knu 70] for organizing the status of the sweep

line. Compared to interval trees the advantage of interval tries is that they avoid the expensive dynamic reorganization of nodes.

In our experiments we compared the performance of the two plane sweep algorithms which organize the sweep-line status in a list (Plane Sweep Intersection-Test, L) and in a trie (T). Figure 4 shows the runtime of the different algorithms when they are applied to processing the spatial joins J1-J4 in main memory. Our plane sweep implementation with interval tries is superior for all joins. Figure 4 also shows that the performance gain of our algorithm increases with an increasing selectivity of the join (recall that the size of the input relations is the same for J1-J4). For join J5 the runtime of the trie algorithm was 236 sec. which was more than a factor of three lower than the runtime of the list algorithm (768 sec.).

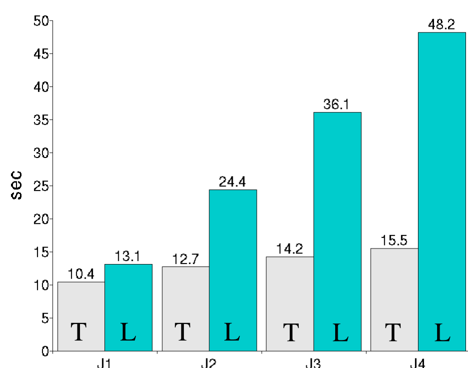


Figure 4. Total runtime (in sec) of internal join algorithms for processing J1-J4

In a second experiment, we compared the performance of PSBM for processing join J5 (using the data from state California) when the sweep-line status is organized as a list and as a trie. The results are plotted as a function of the available main memory in Figure 5. If the available main memory is smaller than 25 MB (which is about 30% of the size of the input relations) the list organization is slightly more efficient than the trie. For larger main memories, the trie organization is superior. Note that the total runtime of PBSM using a list organization increases with an increasing main memory.

Overall, the interval-trie method seems to perform better when partitions of PBSM are large or the join selectivity is high. For small partitions, however, the list method is superior.

3.2.3. Further improvements and implementation details. In this subsection we present some minor improvements of PBSM which are related to estimating the number of partitions and performing the re-partitioning.

In our experiments we observed that formula (1) does not always give the ideal number of partitions. Consider for example the case when the formula without the ceiling returns 1.99 and thus, P = 2. It is

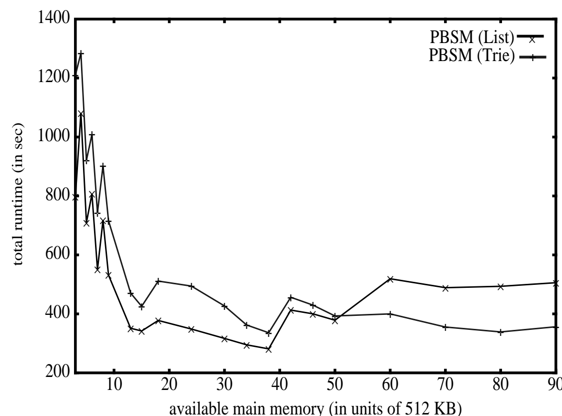


Figure 5. The runtime of two versions of PBSM for processing join J5

then very unlikely that both partitions will fit into main memory and therefore, re-partitioning will be performed on one of them. In order to avoid such a situation formula (1) should be modified such that the formula without the ceiling is multiplied by a factor of t, t > 1. As discussed in [KS 97] computing the number of partitions is generally difficult when the input relations do not refer to base relation of the underlying DBMS. Then, the DBMS has to provide statistics about the intermediate results of operators. This problem is also known from other partition-based join algorithms like the broad family of hash-join algorithms.

In [PD 96] the re-partitioning phase is not treated. Our re-partitioning strategy performs in the following way. If a pair of partitions (PR, PS) does not fit in main memory, we re-partition the larger partition (e.g. PR) into $PR_0,\dots,PR_n$ and check whether each of the partition pairs (PR_0,S), $(PR_1,S),\dots,(PR_n,S)$ fit in main memory. If not, re-partition of S is also performed. This strategy is then recursively performed until all pairs of partitions fit in memory. The results of our experiments show that the re-partitioning phase has only a minor impact on the total runtime of PBSM. This is illustrated in Figure 6 where the fraction of the total runtime for re-partitioning (for processing J5) is plotted as a function of the available main memory. Only for small main memories, re-partitioning contributes about 20% to the total runtime, whereas the influence of re-partitioning diminishes for an increasing main memory.

4. S^3J

The basic idea of Size Separation Spatial Join (S^3J) [KS 97] is to partition the input relations using a hierarchy of equidistant grids with 2^{2k} cells, k=0,1,…, and assign each of the objects to one cell of one of these grids. The join phase basically consists of a synchronized traversal of the different grids. In contrast to PBSM, S^3J does not replicate the data objects and therefore does not need any mechanism

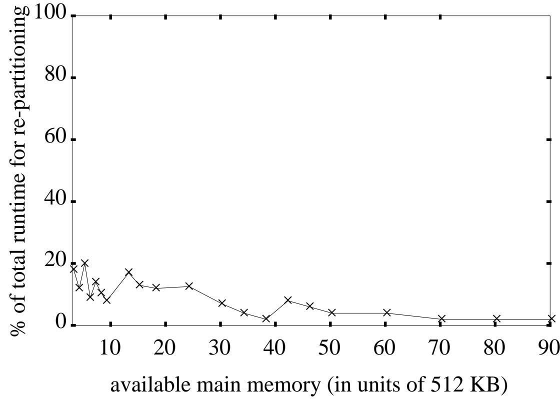


Figure 6. The fraction of the total runtime of PBSM spent for repartitioning when processing J5

for duplicate removal. In the following we first discuss a similar join algorithm based on MX-CIF quadtrees which can be viewed as the internal version of S³J. Thereafter, we briefly review the original approach. Next, we suggest to replicate data objects which results in considerable runtime improvement compared to the original approach. In particular, we present an inexpensive online-algorithm for duplicate removal. Thereafter, we discuss several implementation issues which are important for an efficient join processing.

4.1. Internal join processing using MX-CIF quadtrees

S³J can be viewed as an external version of a join algorithm that is performed on MX-CIF quadtrees [Sam 90]. Therefore, we first would like to illustrate the basic idea of S³J introducing an internal spatial join algorithm based on MX-CIF quadtrees.

The MX-CIF quadtree is designed for organizing a set of rectangles in a dynamic environment. In analogy to other quadtrees (see [Sam 90] for a survey), the MX-CIF quadtree contains different levels where the l-th level consists of 4^l ($l \in \{0, 1, 2, \dots\}$) nodes (We refer to level 0 as the "lowest level" of the tree). Associated to each node is a region of the data space. The region of the root is the complete data space, whereas the region of another node is one of the four quadrants of the region that is associated to the parent node. A rectangle r is inserted into the node on the lowest level such that the region of the node covers the rectangle. Note that in comparison to other types of quadtrees, several rectangles can be associated with a node which need not be a leaf node. Moreover, there is no limit on the number of rectangles associated to a node.

A spatial join is processed on MX-CIF quadtrees in the following way. First, a MX-CIF quadtree is created for each of the input relations. Thereafter, a synchronized pre-order traversal of the quadtrees is performed. Let (N_R, N_S) be a pair of visited nodes

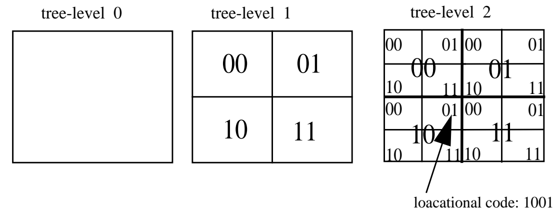


Figure 7. Locational codes of linear quadtrees

during the traversal. We compute the spatial join between N_R and all nodes on the path to N_S and the join between N_S (including N_S) and all nodes on the path to N_R (not including N_R).

It is obvious, that this algorithm performs poorly when data does not fit in main memory. S³J can be viewed as a smart version of this algorithm suitable for external storage.

4.2. Review of S³J

S³J performs in three phases. In the first phase, so-called level-files are generated for each input relation. A level-file of level k contains all rectangles associated to the nodes of the k-th level of the MX-CIF quadtree. In the second phase, the level-files are sorted w.r.t. a space filling curve and in the last phase, a linear scan is performed on the level-files.

In order to support an efficient spatial join processing on the level-files, Koudas & Sevcik suggest to order the data in a file with respect to a recursive space filling curve like the *Hilbert-curve* [Hil 1891]. The space filling curve defines an order on the cells of a level and, since each of the rectangles is covered by one cell, this also defines an order on the rectangles that belongs to the level. The idea of using a space filling curve to order the regions of a quadtree is also known from linear quadtrees [Gar 82] where the values of the space filling curve are called *locational codes*. An example for an other space filling curve is the *Peano-curve* [Pea 1890], also called z-curve or morton-ordering, see Figure 7. More details on space-filling curves are given for example in [Bia 69, Jag 90]. We will discuss the suitability of different curves for S³J later.

In order to reduce the total runtime, it is advisable to compute the locational code only once and attach it to the KPE. Note however that the storage requirements (and hence the computational overhead) of a locational code depends on the level. At level i only $2i$ bits are required. Moreover, the locational code can also be used to determine the level to which the KPE belongs. The level of a KPE is the maximum number of initial bits which are in common for the locational code of the left lower corner of the rectangle and of the right upper corner divided by 2.

In the *join phase* S³J performs a synchronized linear scan of the sorted level-files. For each level-file, only those rectangles have to be kept in main memory which belong to one cell of a level-file (a cell of a

1. Partitioning Phase

For each dataset:

For each element:

a) compute *tree-level* and *locational code*

b) insert KPE in corresponding level-file

2. Sorting Phase

For each level-file:

sort level-file (if necessary externally) w.r.t. locational code

3. Join Phase

Perform a synchronized linear scan of the level-files and join the corresponding pairs of partitions

Figure 8. The S^3J Algorithm

level file corresponds to a node of the MX-CIF quadtree). This set of rectangles are also called a partition in [KS 97]. If these partitions do not exceed the available main memory, each partition of the level-files has to be read only once. The linear scan of the sorted level files simulates the synchronized pre-order traversal of the MX-CIF quadtree. A summary of the algorithm is given in Figure 8.

In our experiments we observed that S^3J does only partially fulfill the design goal that large rectangles are kept in the lower levels and the smaller ones are kept in the upper levels. For example, the vast majority of rectangles in the lowest level-file (level 0) were very small. Though these small rectangles do not provide many results, they have to be tested for intersection with all rectangles from the other relation. This observation was our starting point to rethink the design decision of [KS 97] that redundancy should be completely avoided.

4.3. S^3J with data replication

In the following, we first discuss a simple strategy for introducing data replication in S^3J. Next, we present our on-line method to detect duplicates in the response set.

A strategy for data replication should basically fulfill the following design choices:

- Replicating a rectangle should lead to a substantial reduction of the number of tests for intersection. Thus, small rectangles should not be in low level-files.
- The overall replication rate should be kept sufficiently low. Otherwise, the additional I/O- and CPU-overhead will outweigh the performance gains.

There are several strategies for introducing replication which we have evaluated for the purpose of join processing. Some of the strategies are also known from index structures that are based on the principle of MX-CIF quadtrees, see for example [AS 83]. Due to space limitations we discuss here one of our most efficient strategies that separates the rectangles by the size of their edges. The basic idea is that we first compute the level-file to which a rectangle belongs and then assign the rectangle to the partitions of the level-file that have some overlap with the rectangle. A rectangle r given by its left lower point (xl, yl) and right upper point (xh, yh) is assigned to level

$$\max \{k \mid xh - xl \le 2^{-k} \wedge yh - yl \le 2^{-k}\}$$

The level of a rectangle can be easily computed by using the original level-function of S^3J. Before calling the original level-function, the rectangle is shifted to the origin of the data space. Consider for example rectangles r1 and r2 in Figure 9. In the original approach, r1 and r2 are assigned to level 0 and 1, respectively, whereas in our approach, both rectangles are assigned to level 2. Since rectangles now can intersect with more than one cell they can be stored multiple times in a level file (where for each copy a different locational code is attached to). It is however easy to see that a rectangle is replicated in a level-file at most four times.

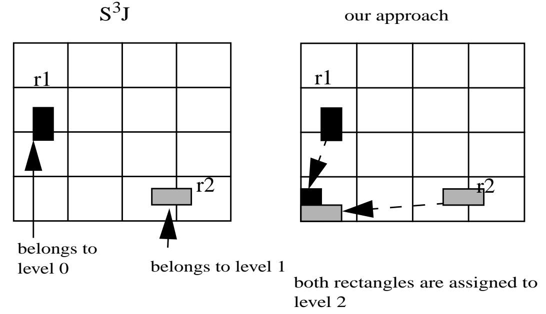
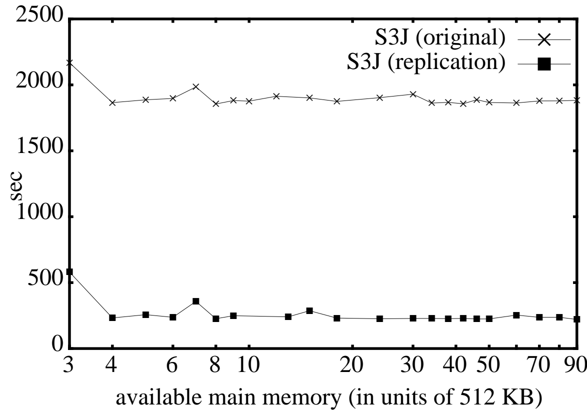


Figure 9. Assignment of rectangles to level-files

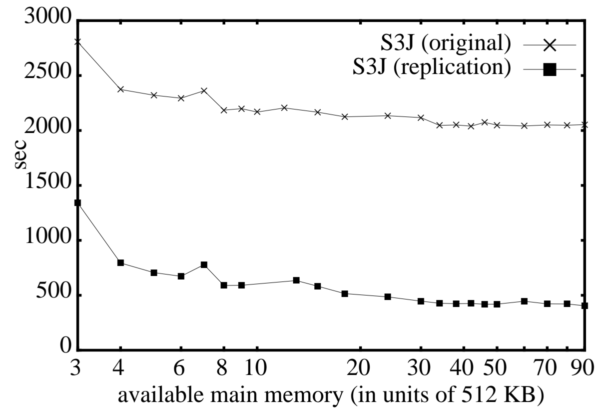
It is obvious that due to the replication of data objects the original join algorithm of S^3J can produce duplicates in the response set. In order to detect these duplicates (at the time they are produced), we present a modified version of the reference point method which has been used for PBSM.

Let us consider that we are processing the join on partitions P_R and P_S which belong to relation R and S, respectively. Let l_R and l_S denote the corresponding level of partitions P_R and P_S. Without loss of generality we assume that $l_R \le l_S$. Furthermore, let r be a rectangle of P_R and s be a rectangle of P_S such that there is an overlap between r and s. We compute the reference point x and test whether x is in the (smaller) region of partition P_S. If so, we report (r,s) as answer. Otherwise, (r,s) is rejected. An example is illustrated in Figure 10 where the level of partition P_R is one less than the level of partition $P1_S$ ($P2_S$). Rectangle s is stored in partition $P1_S$ and $P2_S$ and therefore, the result (r,s) of the join is produced twice. Since the reference point x is however only in $P1_S$ the result is reported exactly once.

In our experiments we compared the performance of the original S^3J to S^3J with replication. The results of the experiments showed that S^3J with replication



a) CPU-time



b) total runtime

Figure 11. Comparison of the required CPU-time and total runtime of S^3J for processing join J5

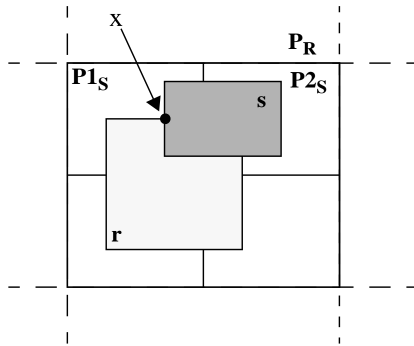


Figure 10. Reference Point Method in S^3J

clearly outperforms the original approach of S^3J which did not allow replication. In Figure 11, we show the results as a function of the available main memory for processing join J5. On the left hand side, we plot the curves for the required CPU-time, whereas on the right hand side the total runtime is depicted. With respect to CPU-time, S^3J with replication is an order of magnitude faster than the original approach. With respect to the total runtime, S^3J with replication is by a factor 2.5 to 4 faster than the original approach. Thus, the overhead of replication does not outweigh the performance gains.

4.4. Further improvements and implementation details

In this subsection we first discuss what kind of internal join algorithm should be used for S^3J. Next, we deal with an efficient computation of the locational codes. Eventually, we describe more details of the traversal scan of the level-files.

4.4.1. Internal join algorithms. Since the partitions of S^3J are considerably smaller than the ones of PBSM we cannot draw the same conclusions about the suitability of internal join algorithms. We therefore performed experiments where we compared three versions of S^3J each of them using one of the following internal join algorithms: nested-loops, Plane Sweep Intersection-Test and trie-based plane sweep. In Figure 12, the total runtime of S^3J is plotted as a function of the available main memory for processing spatial join J5. The one curve shows the performance when a simple nested-loop algorithm is used for joining two partitions, whereas the other curve gives the runtime for the list-based plane sweep method [BKS 93]. Surprisingly, the plane sweep method performs only slightly faster than nested loops. The reason is that the partitions of S^3J are so small that plane-sweep methods do not result in substantial runtime improvements. We even observed that the overhead of a trie-based plane sweep (which is an excellent method for PBSM) is so high that the total runtime is far beyond the one of nested-loops. We therefore did not consider it in Figure 12.

4.4.2. Locational codes. In the sorting phase of S^3J the level-files are sorted w.r.t. a space-filling curve. In [KS 97] it has been suggested to use Hilbert-curve, but other curves like the Peano-curve might also be used. Note that different space-filling curves neither have an impact on the I/O-performance nor on the number of required intersection tests. Therefore, we

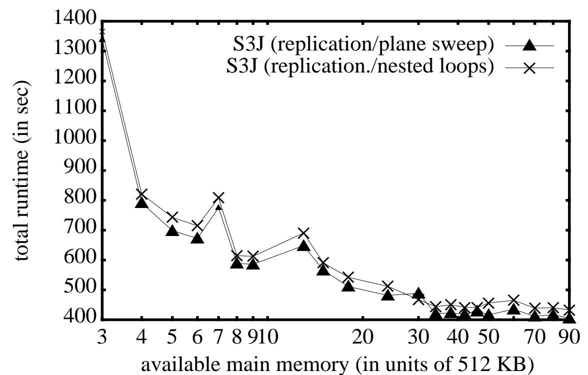


Figure 12. A comparison of the performance of S^3J using two different internal algorithms

suggest to use the curve whose values are computed most efficiently. Since the computation of a Hilbert-value is more expensive than the one of a Peano-value, we use the Peano-curve in our implementation of S^3J. Moreover, our implementation also takes into account that for a rectangle stored in the *i*-th level-file only *2i* bits have to be computed. In particular, there is no need to compute the locational code for rectangles in the lowest level-file (level 0).

4.4.3. Synchronized scan of level-files During join processing (phase 3 of S3J) a synchronized scan through the level-files is performed. One of the problems is that for a large number of levels many of the partitions are empty. It is therefore very inefficient to perform the join by investigating all pairs of partitions. In our implementation we use a heap to avoid the overhead of empty partitions during join processing. The heap is organized in ascending order of the locational codes and it contains the front element of each level-file. The scan is then performed in a similar way as a merge. A detailed description of this method is given in [Dit 99].

5. Comparison

In this section, we compare our new versions of PBSM and S^3J. In particular, we discuss the results of our experiments.

5.1. Analytical comparison

The new versions of S^3J and PBSM show great similarities, but there are still some fundamental differences of these methods. PBSM can be considered as an optimistic (or lazy) method: A grid is used for partitioning the datasets into large chunks. If a pair of related chunks does not fit in main memory, re-partitioning has to be applied. S^3J can be considered as a pessimistic (or eager) method which produces very small partitions which will fit into main memory (almost always). An advantage of our improved verion of S^3J is that the degree of redundancy is at most four, whereas PBSM does not provide an upper bound. Important for the performance of both methods is their internal join algorithm. From our previous discussion it follows that an internal algorithm which is suitable to S^3J is not always suitable to PBSM and vice versa.

	PBSM	S3J
Partitioning Phase	1	1
Repartitioning/Sorting Phase	+	2+
Join Phase	1	1

Table 3. Minimum I/O-operations needed

The I/O-performance of PBSM and S^3J is sketched in Table 3. For the partitioning phase both methods have to write the entire dataset once. Re-partitioning has to be performed for PBSM occasionally. In our experiments we observed that the I/O overhead is rather low for this phase of PBSM. S^3J needs to sort the level files. Therefore, each level-file has to be read and written once. This can be implemented such that random I/Os are almost avoided. When a level-file, however, does not fit in main memory, external sorting has to be applied. Then, the dataset has to be read and written more than twice. In the join phase, both methods have to read the data once from disk.

5.2. Experimental comparison

Although we have already shown the runtime of the improved versions of PBSM and S^3J in the previous sections, we present here a direct comparison of these methods for different spatial joins.

In our first set of experiments we discuss the performance for processing the spatial join of LA_RR(*p*) and LA_ST(*p*), *p =1,...,10*. Recall that the coverage of these datasets growth quadratically in *p*. Thus, the fraction of redundancy for PBSM is very high for a large value of *p*. In Figure 13, the total runtime of S^3J, PBSM (list) and PBSM (trie) is depicted as a function of *p*. We used a total of 2.5 MB as main memory for each of these methods. For small values of *p*, PBSM (list) and PBSM (trie) gives similar performance, whereas S^3J is substantially slower. For larger values of *p*, S^3J performs similar to PBSM (list), but PBSM (trie) still remains to be the clear winner.

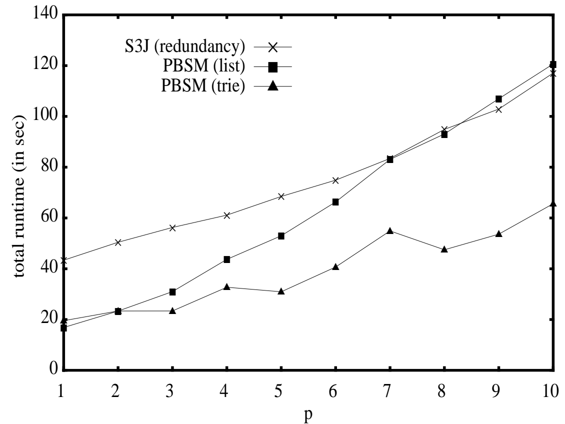


Figure 13. The runtime of S^3J, PBSM (List), PBSM (Trie) for joining LA_RR(p) and LA_ST(p)

Figure 14 shows the total runtime as a function of the available main memory for processing J5. S^3J performs well for small main memories, PBSM (list) is most efficient for medium-sized main memories and PBSM (trie) is most suitable for large main memories.

6. Conclusions

The spatial join is the most important binary operation in a spatial database system. In this paper, we examined two recently proposed spatial join algo-

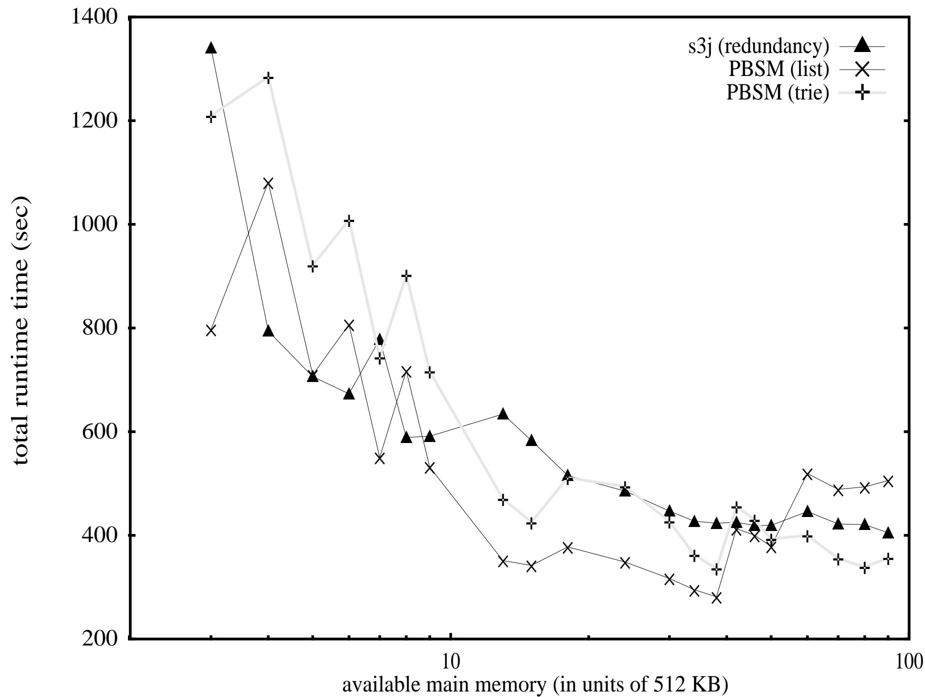


Figure 14. The runtime of S^3J, PBSM (List), PBSM (Trie) for join J5

rithms, PBSM and S^3J, which do not require that a pre-existing index is available on the input. Indices do not exist for example when the input relations corresponds to the result set of other operators. Both algorithms follow the divide & conquer-paradigm, but in contrast to S^3J, PBSM generates redundant data objects to speed up the processing of a join. One of the disadvantages of redundancy is that duplicates in the response set are unavoidable. So far, it was suggested that these duplicates should be eliminated in a final sorting phase. In addition to the runtime overhead of sorting, such a separate phase has several disadvantages when the spatial join is part of a more complex query. For example, such an implementation would obviously block a pipelined processing in an operator-tree since the first result of the spatial join can be produced only after both inputs are completely processed.

In order to overcome the problems with duplicate elimination, we presented a new and inexpensive on-line method for identifying duplicates in the response set of PBSM. Whenever an answer is produced the method decides whether it is a duplicate. Our experiments showed that this method gives substantial runtime improvements. S^3J has been proposed as an algorithm that avoids the problem of duplicates in the response set by simply avoiding the generation of redundant data objects. In this paper, however, we showed that data redundancy results in substantial performance improvements for S^3J. The unavoidable duplicates in the response set can easily be identified by using a slightly modified version of the on-line method we have used for PBSM. In our experiments with real datasets we observed savings in CPU-time up to an order of magnitude and savings in the total runtime up to a factor of 4.

An important building block of join processing is the internal join algorithm which is used for joining partitions in main memory. We showed in our experiments the impact of these algorithms on the performance of PBSM and S^3J. Since the partitions of S^3J are very small and the partitions of PBSM are very large (ideally half of the main memory) both methods require different join algorithms. We showed for PBSM that when the size of the available main memory is beyond a given point the runtime even increases with the amount of available main memory. The reason for this surprising result is the poor performance of the internal join algorithm. We therefore have suggested a different plane-sweep algorithm where the sweep line status is kept in a trie. For S^3J, we showed that a trie-based sweep-line algorithm is inferior to a simple nested-loops algorithm. Overall, we observed in our experiments that our best version of PBSM still outperforms S^3J on the average by a factor of two.

In our future work we are interested in a generalization of our work for multidimensional similarity joins [KS 98]. Moreover, we are currently integrating the different join algorithms into an extensible library of query processing frameworks.

References

[APR+ 98] Lars Arge, Octavian Procopiuc, Sridhar Ramaswamy, Torsten Suel, Jeffrey Scott Vitter:

Scalable Sweeping-Based Spatial Join. VLDB 1998: 570-581

[AS 83] David J. Abel, J. Smith: A data structure and algorithms based on a linear key for a rectangle retrieval problem. Computer Vision 24. 1-13.

[BFH 93] Ludger Becker, Klaus Hinrichs, Ulrich Finke: A New Algorithm for Computing Joins with Grid Files. ICDE 1993: 190-197

[Bia 69] Bially, T.: Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction. IEEE Transactions On Information Theory. IT-15(6). pages 658-664. November 1969.

[BKS 93] Thomas Brinkhoff, Hans-Peter Kriegel, Bernhard Seeger: Efficient Processing of Spatial Joins Using R-Trees. SIGMOD Conference 1993: 237-246

[BKSS 94] Thomas Brinkhoff, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger: Multi-Step Processing of Spatial Joins. SIGMOD Conference 1994: 197-208

[BSS 00] Jochen Van den Bercken, Martin Schneider, Bernhard Seeger: Plug&Join: An easy-to-use Generic Algorithm for Efficiently Processing Equi and Non-Equi Joins. EDBT 2000 (to appear).

[Bur 89] Bureau of the Census: Tiger/Line Precensus Files: 1990 technical documentation. Bureau of the Census. Washington DC. 1989.

[CLR 90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest: Introduction to Algorithms, The MIT Press and McGraw-Hill, 1990.

[Dit 99] Jens-P. Dittrich: Implementierung und Vergleich von Verfahren zur Berechnung des räumlichen Verbunds (in german). Diplomarbeit. Philipps-Universität Marburg. January 1999.

[Gar 82] Irene Gargantini: An Effective Way to Represent Quadtrees. CACM 25(12): 905-910 (198

[Gra 93] Goetz Graefe: Query Evaluation Techniques for Large Databases. Computing Surveys 25(2): 73-170 (1993)

[GS 87] Ralf Hartmut Güting, W. Schilling: A practical Divide-and-Conquer algorithm for the rectangle intersection problem. Information Sciences. pages 95-112. 1987.

[Gün 93] Oliver Günther: Efficient Computation of Spatial Joins. ICDE 1993: 50-59

[Güt 94] Ralf Hartmut Güting: An Introduction to Spatial Database Systems. VLDB Journal 3(4): 357-399 (1994)

[Hil 1891] David Hilbert: Über die stetige Abbildung einer Linie auf ein Flächenstück. Mathematische Annalen. vol. 38. 1891.

[HJR 97] Yun-Wu Huang, Ning Jing, Elke A. Rundensteiner: Spatial Joins Using R-trees: Breadth-First Traversal with Global Optimizations. VLDB 1997: 396-405

[HS 95] Erik G. Hoel, Hanan Samet: Benchmarking Spatial Join Operations with Spatial Output. VLDB 1995: 606-618

[Jag 90] H. V. Jagadish: Linear Clustering of Objects with Multiple Atributes. SIGMOD Conference 1990: 332-342

[Knu 70] Donald E. Knuth: The Art of Computer Programing. Vol. 3. Sorting and Searching.

[KS 97] Nick Koudas, Kenneth C. Sevcik: Size Separation Spatial Join. SIGMOD Conference 1997: 324-335

[KS 98] Nick Koudas, Kenneth C. Sevcik: High Dimensional Similarity Joins: Algorithms and Performance Evaluation. ICDE 1998: 466-475

[LR 94] Ming-Ling Lo, Chinya V. Ravishankar: Spatial Joins Using Seeded Trees. SIGMOD Conference 1994: 209-220

[LR 96] Ming-Ling Lo, Chinya V. Ravishankar: Spatial Hash-Joins. SIGMOD Conf. 1996: 247-258

[MP 99] Nikos Mamoulis, Dimitris Papadias: Integration of Spatial Join Algorithms for Processing Multiple Inputs. SIGMOD Conference 1999: 1-12

[Ore 86] Jack A. Orenstein: Spatial Query Processing in an Object-Oriented Database System. SIGMOD Conference 1986: 326-336

[Pea 1890] G. Peano: Sur une courbe qui remplit toute une aire plane. Mathematische Annalen. vol. 36. 1890.

[PD 96] Jignesh M. Patel, David J. DeWitt: Partition Based Spatial-Merge Join. SIGMOD Conf. 1996: 259-270

[Pat 98] Jignesh M. Patel: Efficient Database Support for Spatial Applications, PhD thesis, Univ. of of Wisconsin-Madison, 1998.

[Rot 91] Doron Rotem: Spatial Join Indices. ICDE 1991: 500-509

[Sam 90] Hana Samet: The Design and Analysis of Spatial Data Structures. Addison Wesley. 1990.

[See 91] Bernhard Seeger: Performance Comparison of Segment Access Methods Implemented on Top of the Buddy-Tree. SSD 1991: 277-296

[ZS 98] Geraldo Zimbrao, Jano Moreira de Souza: A Raster Approximation For Processing of Spatial Joins. VLDB 1998: 558-569