

Closest Pair Queries in Spatial Databases *

Antonio Corral

Dept. of Languages &
Computation,
University of Almeria,
04120 Almeria,
Spain.

acorral@ualm.es

Yannis Manolopoulos

Data Eng. Lab,
Dept. of Informatics,
Aristotle University
of Thessaloniki,
GR-54006 Greece.

manolopo@csd.auth.gr

Yannis Theodoridis

Computer Technology
Institute,
P.O. Box 1122,
GR-26110 Patras,
Greece.

ytheod@cti.gr

Michael Vassilakopoulos

Data Eng. Lab,
Dept. of Informatics,
Aristotle University
of Thessaloniki,
GR-54006 Greece.

mvass@computer.org

Abstract

This paper addresses the problem of finding the K closest pairs between two spatial data sets, where each set is stored in a structure belonging in the R-tree family. Five different algorithms (four recursive and one iterative) are presented for solving this problem. The case of 1 closest pair is treated as a special case. An extensive study, based on experiments performed with synthetic as well as with real point data sets, is presented. A wide range of values for the basic parameters affecting the performance of the algorithms, especially the effect of overlap between the two data sets, is explored. Moreover, an algorithmic as well as an experimental comparison with existing incremental algorithms addressing the same problem is presented. In most settings, the new algorithms proposed clearly outperform the existing ones.

1 Introduction

The role of spatial databases is continuously increasing in many modern applications during last years. Mapping, urban planning, transportation planning, resource management, geomarketing, archeology and environmental modeling are just some of these applications.

The key characteristic that makes a spatial database a powerful tool is its ability to manipulate spatial data, rather than simply to store and represent them. The most basic form of such a manipulation is answering queries related to the spatial properties of data. Some typical spatial queries are the following:

- a “Point Location Query” seeks for the spatial objects that fall on a given point.

*Research performed under the European Union’s TMR Chorochronos project, contract number ERBFMRX-CT96-0056 (DG12-BDCN).

- a “Range Query” seeks for the spatial objects that are contained within a given region (usually expressed as a rectangle).
- a “Join Query” may take many forms. It involves two or more spatial data sets and discovers pairs (or tuples, in case of more than two data sets) of spatial objects that satisfy a given predicate. For example, a join query that acts on two data sets, may discover all pairs of spatial objects that intersect each other.
- Finally, a very common spatial query is the “Nearest Neighbor Query” that seeks for the spatial objects residing more closely to a given object. In its simplest form, it discovers one such object (the Nearest Neighbor). Its generalization discovers K such objects (K Nearest Neighbors), for a given K .

In this paper, a spatial query that combines join and nearest neighbor queries is examined. It is called “ K Closest Pairs Query” (K -CPQ) and it discovers the K pairs of spatial objects formed from two data sets that have the K smallest distances between them, where $K \geq 1$. Like a join query, all pairs of objects are candidates for the result. Like a nearest neighbor query, the K nearest neighbor property is the basis for the final ordering. In the degenerate case of $K = 1$, the closest pair of spatial objects is discovered. This problem is a rather novel one. Although, the CP problem is well honored in Computational Geometry, to the authors knowledge, there is only one paper in the literature that has addressed it in the context of spatial databases [11] by presenting a number of incremental algorithms for its solution. A similar problem is the “all nearest neighbor” problem which has been investigated in [9].

K -CPQs are very useful in many applications that use spatial data for decision making and other demanding data handling operations. For example, consider a case where one data set represents the locations of the numerous archeological sites of Greece, while the second set stands for the most important holiday resorts. A K -CPQ will discover the K pairs of sites and holiday resorts that have the K smaller distances so that tourists accommodated in a resort can easily visit the

archeological site of each pair of the result. This information could be utilized by the tourist authorities for advertising purposes. The value of K is dependent on the advertising budget of the tourist authorities.

The fundamental assumption is that the two spatial data sets are stored in structures belonging in the family of R-trees [10]. R-trees and their variants (see [8, 15]), are considered an excellent choice for indexing various kinds of spatial data (like points, polygons, 2-d objects, etc) and have already been adopted in commercial systems (Informix, Oracle, etc). In this paper we focus on sets of point data. Five different algorithms are presented for solving the problem of K -CPQ. Four of these algorithms are recursive and one is iterative. The problem of 1-CPQ is treated as a special case, since increased performance can be achieved by making use of properties holding for this case. Moreover, an extensive performance study, based on experiments performed with synthetic as well as with real point data sets, is presented. A wide range of values for the basic parameters affecting the performance of the algorithms is examined. In addition, an algorithmic as well as a comparative performance study with the incremental algorithms of [11] is presented. The finding of the above studies is the determination of the algorithm outperforming all the others for each set of parameter values. As it turns out, the new (non-incremental) algorithms outperform the existing incremental algorithms, under various settings.

The organization of this paper is the following. In Section 2 the problem of K -CPQ, a brief description of the family of R-trees and some useful metrics between R-tree nodes and distances of closest pairs are presented. In Section 3 the five new algorithms are introduced and they are compared algorithmically with the algorithm of [11]. Sections 4 and 5 exhibit a detailed performance study of all the algorithms for 1- and K -CPQs, respectively. Moreover, in Section 5 a comparative performance study between the newly introduced algorithms and the algorithm of [11] is presented. In Section 6 conclusions on the contribution of this paper and related future research plans are presented.

2 Closest Pair Queries and R-trees

2.1 Definition of Problem

Let two point sets, $P = \{p_1, p_2, \dots, p_{NP}\}$ and $Q = \{q_1, q_2, \dots, q_{NQ}\}$, be stored in two R-trees, R_P and R_Q , respectively. As 1-CP (One Closest Pair) of these two point sets we define a pair

$$(p_z, q_l), \quad p_z \in P \wedge q_l \in Q$$

such that

$$\text{dist}(p_i, q_j) \geq \text{dist}(p_z, q_l), \quad \forall p_i \in P \wedge \forall q_j \in Q$$

In other words, an 1-CP of P and Q is a pair that has the smallest distance between all pairs of points that can be formed by choosing one point of P and one point of Q .¹ As K -CPs (K Closest Pairs) of P and Q we define a collection of K ordered pairs

$$(p_{z_1}, q_{l_1}), (p_{z_2}, q_{l_2}), \dots, (p_{z_K}, q_{l_K}),$$

$$p_{z_1}, p_{z_2}, \dots, p_{z_K} \in P \wedge q_{l_1}, q_{l_2}, \dots, q_{l_K} \in Q$$

such that

$$\begin{aligned} \text{dist}(p_i, q_j) &\geq \text{dist}(p_{z_K}, q_{l_K}) \geq \\ \text{dist}(p_{z_{(K-1)}}, q_{l_{(K-1)}}) &\geq \dots \geq \text{dist}(p_{z_1}, q_{l_1}), \end{aligned}$$

$$\forall (p_i, q_j) \in (P \times Q - \{(p_{z_1}, q_{l_1}), (p_{z_2}, q_{l_2}), \dots, (p_{z_K}, q_{l_K})\})$$

In other words, K -CPs of P and Q are K pairs that have the K smallest distances between all pairs of points that can be formed by choosing one point of P and one point of Q . K must be smaller than $|P| \cdot |Q|$, i.e. the number of pairs that can be formed from P and Q .

Note that, due to ties of distances, the result of 1-CPQ or the K -CPQ may not be unique for a specific pair of P and Q . The aim of the presented algorithms is to find one of the possible instances. Note also that in the context of this paper “dist” stands for Euclidean Distance, although the presented methods can be easily adapted to any Minkowski metric. We also focus on 2-dimensional space, but the extension to k -dimensional space is straightforward.

2.2 R-trees

R-trees are hierarchical data structures based on B⁺-trees. They are used for the dynamic organization of a set of k -dimensional geometric objects representing them by the minimum bounding k -dimensional rectangles. Each R-tree node corresponds to the MBR that contains its children. The tree leaves contain pointers to the objects of the database, instead of pointers to children nodes. The nodes are implemented as disk pages.

Many variations of R-trees have appeared in the literature (an exhaustive survey can be found in [8]). One of the most popular variations is the R*-tree [1]. The R*-tree follows a node split technique that is more sophisticated than that of the simple R-tree and is considered the most efficient variant of the R-tree family, since, as far as searches are concerned, it can be used in exactly the same way as simple R-trees. In this paper, we choose R*-trees to perform our experimental study.

¹The 1-CP problem addressed in this paper is an extension of the popular closest pair problem that appears in computational geometry [20] for two point sets.

2.3 Useful Metrics

Since the different algorithms for CPQs act on pairs of R-trees, some metrics between MBRs (that can be used to increase performance of the algorithms) will be defined. Let N_P and N_Q be two internal nodes of R_P and R_Q , respectively. Each of these nodes has an MBR that contains all the points that reside in the respective subtree. In order for this rectangle to be the minimum bounding one, at least one point is located at each edge of the rectangle. Let M_P and M_Q represent the MBRs of N_P and N_Q , respectively. Let r_1, r_2, r_3 and r_4 be the four edges of M_P and s_1, s_2, s_3 and s_4 be the four edges of M_Q . By $\text{MINDIST}(r_i, s_i)$ we denote the minimum distance between two points falling on r_i and s_i . Accordingly, by $\text{MAXDIST}(r_i, s_i)$ we denote the maximum distance between two points falling on r_i and s_i . In the sequel, we extend definitions of metrics between a point and an MBR that appear in [21] and define a set of useful metrics between two MBRs. In case M_P and M_Q are disjoint we can define a metric that expresses the minimum possible distance of two points contained in different MBRs:

$$\text{MINMINDIST}(M_P, M_Q) = \min_{i,j} \{\text{MINDIST}(r_i, s_j)\}$$

In case the MBRs of the two nodes intersect, then $\text{MINMINDIST}(M_P, M_Q)$ equals 0. In any case (intersecting or disjoint MBRs) we can define the metrics

$$\text{MINMAXDIST}(M_P, M_Q) = \min_{i,j} \{\text{MAXDIST}(r_i, s_j)\}$$

and

$$\text{MAXMAXDIST}(M_P, M_Q) = \max_{i,j} \{\text{MAXDIST}(r_i, s_j)\}$$

MAXMAXDIST expresses the maximum possible distance of any two points contained in different MBRs. MINMAXDIST expresses an upper bound of distance for at least one pair of points. More specifically, there exists at least one pair of points (contained in different MBRs) with distance smaller than or equal to MINMAXDIST . In Figure 1, two MBRs and their MINMINDIST , MINMAXDIST and MAXMAXDIST distances are depicted. Recall that at least one point is located on each edge of each MBR. To summarize, for each pair (p_i, q_j) of points, p_i enclosed by M_P and q_j enclosed by M_Q , it holds that

$$\begin{aligned} \text{MINMINDIST}(M_P, M_Q) &\leq \text{dist}(p_i, q_j) \\ &\leq \text{MAXMAXDIST}(M_P, M_Q) \end{aligned} \quad (1)$$

Moreover, there exists at least one pair (p_i, q_j) of points, p_i enclosed by M_P and q_j enclosed by M_Q , such that

$$\text{dist}(p_i, q_j) \leq \text{MINMAXDIST}(M_P, M_Q) \quad (2)$$

These metrics can be calculated by formulae analogous to the ones presented in [18, 21], where metrics between a point and an MBR are defined.

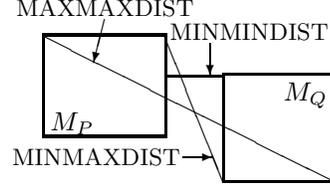


Figure 1: Two MBRs and their MINMINDIST , MINMAXDIST and MAXMAXDIST .

3 Algorithms for CPQs

In the following, a number of different algorithmic approaches for discovering the 1-CP and the K -CPs between points stored in two R-trees are presented. Since the height of an R-tree depends on the number of points inserted (as well as in the order of insertions), the two R-trees may have the same, or different heights. Besides, an algorithm for such a problem may be recursive or iterative. All these different possibilities are examined in the next sections. For the ease of exposition, we proceed from simpler to more complicated material.

3.1 Naive Algorithm

The simplest approach to the problem of Closest Pair Queries is to follow a recursive naive solution for the 1-CP subproblem and for two R-trees of the same height. Such an algorithm consists of the following steps.

- CP1 Start from the roots of the two R-trees and set the minimum distance found so far, T , to ∞ .
- CP2 If you access a pair of internal nodes, propagate downwards recursively for every possible pair of MBRs.
- CP3 If you access two leaves, calculate the distance of each possible pair of points. If this distance is smaller than T , update T .

The desired 1-CP is the one that corresponds to the final value of T .

3.2 Exhaustive Algorithm

An improvement of the previous algorithm is to make use of the left part of Inequality 1 and prune some paths in the two trees that are not likely to lead to a better solution. That is, to propagate downwards only for those pairs of MBRs that satisfy this property. The CP2 step of the previous algorithm would now be:

- CP2 If you access a pair of internal nodes, calculate MINMINDIST for each possible pair of MBRs. Propagate downwards recursively only for those pairs that have $\text{MINMINDIST} \leq T$.

3.3 Simple Recursive Algorithm

A further improvement is to try to minimize the value of T as soon as possible. This can be done by making use of Inequality 2. That is, when a pair of internal nodes is visited, to examine if Inequality 2 applied to every pair of MBRs, can give a smaller T value. Since Inequality 2 holds for at least one pair of points, this improvement is sound for the 1-CP problem. The CP2 step would now be:

CP2 If you access a pair of internal nodes, calculate the minimum of MINMAXDIST for all possible pairs of MBRs. If this minimum is smaller than T , update T . Calculate MINMINDIST for each possible pair of MBRs. Propagate downwards recursively only for those pairs that have $\text{MINMINDIST} \leq T$.

3.4 Sorted Distances Recursive Algorithm

Pairs of MBRs that have smaller MINMINDIST are more likely to contain the 1-CP and to lead to a smaller T . A heuristic that aims at improving our algorithms even more when two internal nodes are accessed, is to sort the pairs of MBRs according to ascending order of MINMINDIST and to obey this order in propagating downwards recursively. This order of processing is expected to improve pruning of paths. The CP2 step of the previous algorithm would be:

CP2 If you access a pair of internal nodes, calculate the minimum of MINMAXDIST for all possible pairs of MBRs. If this minimum is smaller than T , update T . Calculate MINMINDIST for each possible pair of MBRs and sort these pairs in ascending order of MINMINDIST. Following this order, propagate downwards recursively only for those pairs that have $\text{MINMINDIST} \leq T$.

3.5 Heap Algorithm

Unlike the previous ones, this algorithm is non recursive. In order to overcome recursion and to keep track of propagation downwards while accessing the two trees, a heap is used. This heap holds pairs of MBRs according to their MINMINDIST. The pair with the smallest value resides on top of the heap. This pair is the next candidate for visiting. The overall algorithm is as follows.

CP1 Start from the roots of the two R-trees, set T to ∞ and initialize the heap.

CP2 If you access a pair of internal nodes, calculate the minimum of MINMAXDIST for all possible pairs of MBRs. If this minimum is smaller than T , update T . Calculate MINMINDIST for each possible pair of MBRs. Insert into the heap those pairs that have $\text{MINMINDIST} \leq T$.

CP3 If you access two leaves, calculate the distance of each possible pair of points. If this distance is smaller than T , update T .

CP4 If the heap is empty then stop.

CP5 Get the pair on top of the heap. If this pair has $\text{MINMINDIST} > T$, then stop. Else, repeat the algorithm from CP2 for this pair.

The 1-CP is the pair that has distance T .

3.6 Treatment of Ties

In the algorithms where ties of MINMINDIST values may appear (the Sorted Distances and the Heap algorithms), it is possible to get a further improvement by choosing the next pair in case of a tie using some heuristic (and not following the order produced by the sorting or the heap handling algorithm). The following list presents five criteria that are likely to improve performance. Thus, in case of a tie of two or more pairs, choose the pair that has:

1. as one of its elements the largest MBR (the area of an MBR is expressed as a percentage of the area of the relevant root),
2. the smallest MINMAXDIST between its two elements,
3. the largest sum of the areas of its two elements,
4. the smallest difference of areas between the MBR that embeds both its elements and these elements,
5. the largest area of intersection between its two elements.

In case the criterion we use can not resolve the tie (provide a winner), another criterion may be used at a second stage.

3.7 Treatment of different heights

When the two R-trees storing the two point sets have different heights the algorithms are slightly more complicated. In recursive algorithms, there are two approaches for treating different heights.

- The first approach is called “fix-at-root”. The idea is, when the algorithm is called with a pair of internal nodes at different levels, downwards propagation stops in the tree of the lower level node, while propagation in the other tree continues, until a pair of nodes at the same level is reached. Then, propagation continues in both subtrees as usual.
- The second approach is called “fix-at-leaves” and works in the opposite way. Recursion propagates downwards as usual. When the algorithm is called with a leaf node on the one hand and an internal

node on the other hand, downwards propagation stops in the tree of the leaf node, while propagation in the other tree continues as usual.

For example, for the Simple algorithm, application of the “fix-at-leaves” strategy results in the following extra step:

CP2.1 If you access a pair with one leaf and one internal node, calculate the minimum of MINMAXDIST for all possible pairs of MBRs that consist of the MBR of this leaf and an MBR contained in the internal node. If this minimum is smaller than T , update T . Calculate also MINMINDIST for all these pairs. For each pair that has $\text{MINMINDIST} \leq T$, make a recursive call with node parameters the nodes corresponding to the MBRs of the pair (note that the one parameter will be the leaf node and the other parameter will be a child of the internal node).

While application of the “fix-at-root” strategy, for the same algorithm results in the following extra step:

CP2.1 If you access two nodes residing at different levels, calculate the minimum of MINMAXDIST for all possible pairs of MBRs that consist of an MBR contained in the higher level node and the MBR of the lower level node. If this minimum is smaller than T , update T . Calculate also MINMINDIST for all these pairs. For each pair that has $\text{MINMINDIST} \leq T$, make a recursive call with node parameters the nodes corresponding to the MBRs of the pair (note that the one parameter will be the node residing at the lower level).

The Heap algorithm can be easily modified to deal with different heights by the “fix-at-leaves” or the “fix-at-root” strategy. The extra steps that needs to be added to the Heap algorithm is analogous to one of the steps presented above (depending on the strategy). The only difference is that a recursive call is replaced by an insertion in the heap.

3.8 Extending to K Closest Pairs

In order to solve the K -CPs problem an extra structure that holds the K Closest Pairs is necessary. This structure is organized as a max heap (called K -heap) and holds pairs of points according to their distance. The pair of points with the largest distance resides on top of the heap.

- Initially the K -heap is empty.
- The pairs of points discovered in step CP3 are inserted in the K -heap until it gets full.
- Then, when a new pair of points is discovered in step CP3 and if its distance is smaller than the top of the K -heap, the top is deleted and this pair is inserted in the K -heap.

The above additions are the only ones needed for the Naive and the Exhaustive algorithms to solve the K -CPs problem and were used in the implementation of the K -CP versions of these algorithms.

However, these additions are not sufficient for the Simple, the Sorted Distances and the Heap algorithms. These algorithms make use of Inequality 2 that, in general, does not hold for more than one pairs of points. This means that updating of T based on MINMAXDIST values must be discarded. A simple modification that can make these three algorithms suitable for solving the K -CPs problem is to use the distance of the top of the K -heap as the value of T , after the K -heap has become full. While the K -heap has empty slots, infinity should be used as T . An alternative, although more complicated, modification (used in the implementation of the K -CP versions of the three algorithms) is to make use of the right part of Inequality 1, while pruning unnecessary paths in the two trees. That is, among a number of pairs of MBRs to find the one with MAXMAXDIST that might update the value of T . Details are outlined in [5].

3.9 Related Work

Hjaltason and Samet [11] also presented algorithms for closest pair queries in spatial databases (called “distance join algorithms” in [11]). These algorithms are based on a priority queue and resemble the functionality of the Heap algorithm. However, there are significant differences between the algorithms of [11], the Heap and the other algorithms presented in the present paper. The key differences are outlined in the following paragraphs.

The algorithms of [11] store in the priority queue item pairs of the following types: node/node, node/obr, obr/node and object/object (where “node” is an R-tree node, “obr” is an object bounding rectangle and “object” is an actual data item). The Heap algorithm stores pair items that are of internal-node/internal-node type (where with “node” an R-tree node is referred). While processing a pair of subtrees between the two data sets, the algorithms of [11] are likely to insert in the priority queue a significant portion of all the above four types of item pairs that can be formed from these subtrees. On the contrary, while processing the same pair of subtrees, the Heap algorithm inserts only a portion of the internal-node pairs that can be formed from these subtrees (a small fraction of the pairs that are likely to be inserted in the priority queue of [11]). This fact advocates for the creation of a significantly larger priority queue for [11] in comparison to the structure of our Heap algorithm.

Due to its expected large size, the priority queue of [11] is stored partially in main memory (with one part as a heap and another part as an unordered list)

and partially in secondary memory (as a number of linked lists). The distinction of the size of each part is crucial for performance (since pairs that are unlikely to be accessed should be stored on disk) and depends on the arbitrary choice of a constant called D_T . As the authors of [11] state, a policy for choosing D_T is a subject for further investigation. On the contrary, the queue of the Heap algorithm is completely stored in main memory, since its size is significantly smaller.

The basic algorithm of [11] is incremental, in the sense that an unlimited number of closest pairs is produced in ascending order of distance. To reduce the size of the priority queue and increase the performance of the algorithm, in [11], an extra structure is introduced and an upper bound K is set for the number of closest pairs that can be produced. After this modification, the algorithm becomes incremental up to K , only. The algorithms of the present paper calculate all K closest pairs together. The main idea behind them is to increase performance by achieving the highest pruning possible, while tree paths are followed.

The algorithms of [11] solve ties of distances using one of two approaches: depth-first (a pair with a node at a deeper level has priority) and breadth-first (the opposite). Moreover, the algorithms of [11] traverse trees according to one of three policies: basic (priority is given to one of the trees, arbitrarily), even (priority is given to the node at shallower depth) and simultaneous (all possible pairs of nodes are candidates for traversal). On the other hand, all our algorithms follow the simultaneous approach.

Finally, in the algorithms of [11], there is no distinction for the 1-CPQ (1-CPQ is just a case of K -CPQ). In our algorithms, the 1-CPQ is a special case and the use of Inequality 2 increases pruning and performance.

After presenting our proposals for efficient CPQ processing as well as related work, an extensive experimentation follows for 1- and K -CPQs. The goal of the experiments is to trace the pros and cons of each alternate solution and provide guidelines for query optimization purposes. Due to space limitations, in the presentation to follow some charts are omitted; interested readers can refer to [5] for the complete performance study.

4 1-CPQs Performance Comparison

This section provides the results of an extensive experimentation study aiming at measuring and evaluating the efficiency of the four CP algorithms (the Naive one excluded) proposed in Section 3 for 1-CP queries, namely the Exhaustive, the Simple, the Sorted Distances,² and the Heap algorithm (in the sequel, de-

²We have experimented with six sorting methods (Bubble-, Selection-, Insertion-, Heap-, Quick-, MergeSort) and chosen MergeSort because it obtained the best performance in terms of both I/O and CPU cost.

noted by EXH, SIM, STD, and HEAP, respectively). As already discussed, two parameters that need to be further evaluated deal with (i) the treatment of ties when STD and HEAP need to choose among several node pairs with equal MINMINDIST and (ii) the treatment of R-trees with different heights. After fixing these two techniques, we proceed with an extensive comparison of the proposed algorithms. A major part of the experimentation consists of detecting and evaluating the effect of two crucial factors involved: (i) the portion of overlapping between the two data sets and (ii) the size of the underlying buffering scheme. For our experiments we have built several R*-trees using the following data sets:

- a group of random data sets of cardinality 20K, 40K, 60K, and 80K points following a uniform-like distribution,
- a real data set from the Sequoia database [22] consisting of 62,536 points that represent sites in California, and
- a uniformly distributed data set consisting of 62,536 points, too.

All experiments have run on a workstation of 64 Mbytes RAM and several Gbytes of secondary storage. The page size was set to 1 Kbyte thus resulting to R*-tree node capacity $M = 21$ (minimum occupancy was set to $m = M/3 = 7$, a reasonable choice according to [1]).

4.1 Treatment of ties

According to the discussion in Subsection 3.6, five alternative techniques are presented to deal with ties between two or more node/node pairs that are likely to appear during the sorting phase of the STD or the HEAP algorithm. We call them T1 - T5 and evaluate them in Figures 2.a and 2.b for the STD algorithm and the HEAP algorithms, respectively. 60K/60K data sets have been used. By fixing the performance of T1 to 100% we present the relative gain or loss of the other alternatives. Several other combinations were also tested but the trends are similar and thus are not presented here. For this set of experiments

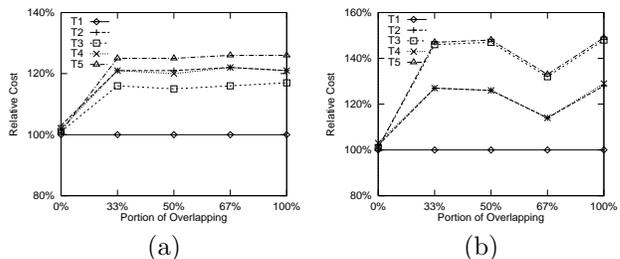


Figure 2: Comparison of Different Tie Treatment Approaches in the (a) STD and (b) HEAP Algorithms with Random Data Sets (60K/60K).

the buffer was disabled. The conclusion is that T1 is the clear winner. It always outperforms all other alternatives since the other techniques lead to a performance deterioration of up to 50% with respect to T1. Obviously, the differences are clear for overlapping data sets since for disjoint ones (overlapping = 0%) ties appear rarely and thus almost all alternatives appear to be equivalent.

4.2 R-trees with different heights

The “classic” join procedure propagates the two R-trees and fixes the level of the shorter tree when it reaches the leaves (“fix-at-leaves”) [3, 14, 16, 23]. In Subsection 3.7 we presented an alternative, called “fix-at-root”, where the level of the shorter tree is fixed at the root as long as the algorithm propagates downwards in the taller tree until both reach the same level. We compare the two approaches and the results are illustrated in Figures 3.a and 3.b for the STD and HEAP algorithms, respectively. The cardinality of the taller R-tree was fixed to 80K random data (height $h = 5$) while the shorter R-tree consisted of 20K - 60K random data (all with $h = 4$). Three configurations were considered: 0%, 50%, and 100% overlapping between the two data sets. The buffer size was again set to zero.³ Although

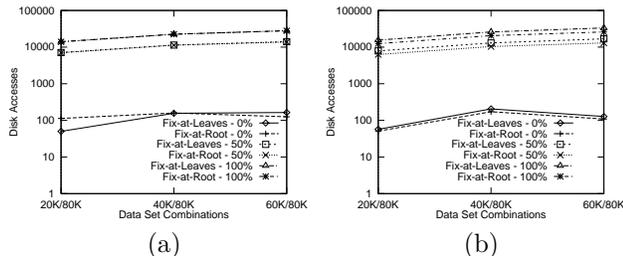


Figure 3: Comparison of Different Height Treatment Approaches in the (a) STD and (b) HEAP Algorithms. Log Scale.

neglected in the literature of spatial join processing, the “fix-at-root” approach turns out to be very efficient for the purposes of closest-pair algorithms. After many experiments we concluded that within the SIM and HEAP algorithms, it always performs better than the traditional “fix-at-leaves” approach with a relative performance gain usually ranging from 10% up to 40%. On the other hand, in the STD algorithm, the two are more or less equivalent except the case of 0% overlapping where “fix-at-leaves” performs much better than the “fix-at-root” approach. This is explained by the fact that pruning with “fix-at-root” is likely to appear at higher levels in the taller tree than the “fix-at-leaves”, thus excluding larger subtrees from further processing.

³We have run the same experiments for several buffer sizes and the conclusions were similar.

4.3 Comparison of 1-CP algorithms

We now proceed with the performance comparison of the four algorithms, which were proposed as improvements to the naive solution. We first assume zero buffer (Subsection 4.3.1). As intuitively shown in Section 2, closest pair queries are very sensitive to the relative location of the two data sets involved, especially the portion of overlapping. In other words, the higher the overlap between the two workspaces the more expensive the query, in terms of disk accesses. To measure that, we track the sensitivity of the algorithms on that parameter (Subsection 4.3.2) and then we introduce a buffer of predefined size following the least-recently-used (LRU) policy (Subsection 4.3.3).

4.3.1 The effect of zero buffer capacity

All four algorithms were evaluated with respect to the cardinality of the data sets, the distribution of data and the portion of overlapping between the two data sets. For this set of experiments the buffer was set to zero. Figure 4 illustrates the performance of each algorithm on 1-CPQ between real Sequoia data set and random data of varying cardinality and (a) 0% and (b) 100% overlapping workspaces. Based on these experimental

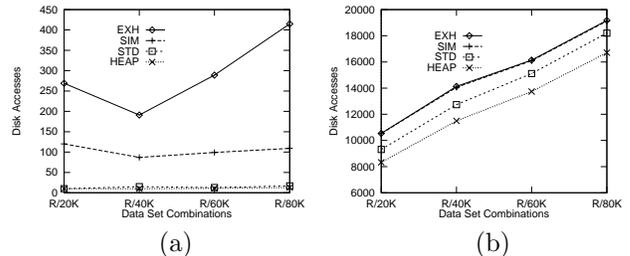


Figure 4: Comparison of the four CP Algorithms: Real vs. Random Data in (a) 0% and (b) 100% Overlapping Workspaces.

outcomes (for more outcomes see [5]) we conclude that for 0% overlapping, the cost of HEAP and STD is one order of magnitude lower than that of SIM and EXH. Also, for overlapping workspaces, HEAP and STD are the winners with an average relative gap of 20% and 10%, respectively. In general, we derive that STD and HEAP appear to be the most promising ones, since they almost always outperform the other two. Another conclusion is that the overlap factor should be further investigated since all 1-CP algorithms are very sensitive to this parameter. In other words, the key question that needs to be addressed is the following: is there a threshold (if yes, which) that makes the performance of the three algorithms distinct enough?

4.3.2 The effect of the overlap

This sensitivity is quantitatively measured in the set of experiments that follow. In particular, we measured

the relative cost of the three algorithms (SIM, STD, and HEAP) with respect to the cost of EXH, with the portion of overlapping ranging from 0% to 100%. For each experiment, on the one hand, it was the real data set, denoted by “R”, consisting of 62,536 entries (Figure 5) and, on the other hand, a random data set of 40K or 80K cardinality. Especially for the R/80K experiments where the two R*-trees have different heights, we adopted the “fix-at-root” treatment which was shown to be efficient in Subsection 4.2. Indeed and in accordance to our intuition, overlap between the data sets is crucial for the performance of all 1-CP algorithms. The cost for a query involving fully overlapping data sets is orders of magnitude higher than that involving disjoint workspaces. The behavior of the three algorithms is

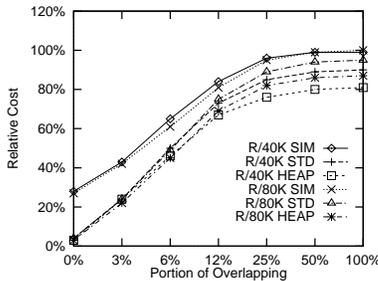


Figure 5: Finding a Threshold on the Overlap Factor: Real vs. Random Data.

surprisingly similar. For small overlap (at most 5%), they all achieve a significant improvement being 2-20 times faster than EXH when a real data set is joined with a uniform one. A justification for this huge improvement when real data is involved is the fact that node rectangles between the two R*-trees are likely to be disjoint (or low overlapping) even for high overlapping data sets. As a conclusion, zero or low overlap gives a serious advantage to the three non-exhaustive algorithms. It also turns out that this parameter must be taken into account very seriously when performing experimentation on CP algorithms.

4.3.3 Introducing the LRU-buffer

Cache policies considerably affect the performance of the algorithms dealing with secondary storage. [13, 4] have already studied the effect of the LRU-buffer size in spatial selection, spatial join and spatial join between different kinds of data, respectively. In this subsection we present how the LRU buffer affects the performance of each algorithm. We used LRU buffer varying from $B = 0 \dots 256$ pages (dedicated to each R-tree as two equal portions of $B/2$ pages). For each experiment, on the one hand, it was the real data set, denoted by “R”, and, on the other hand, a random data set (of cardinality 40K or 80K). Figure 6 illustrates the

results for the two configurations considered between the two data sets (0% and 100% overlapping). As before, for the R/80K experiment where the two R-trees have different heights, it was the “fix-at-root” approach adopted. Case (a) is clear: although EXH

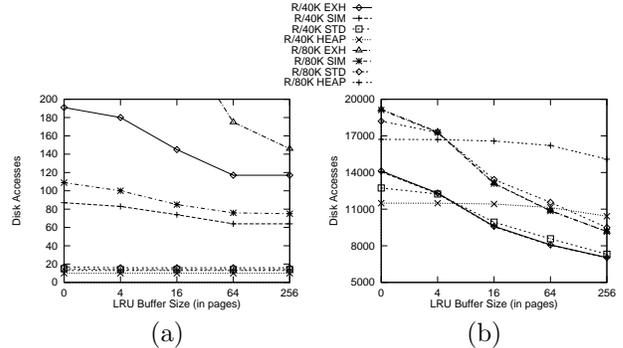


Figure 6: Comparison of the four 1-CP Algorithms for Varying LRU Buffer Size: Real vs. Random Data in (a) 0% and (b) 100% Overlapping Workspaces.

and SIM are getting improved (by a factor that reaches up to 2-3 for $B = 256$ pages) as long as the buffer size grows, they never come close to STD and HEAP. Interestingly, the latter ones are not sensitive to the buffer size. On the other hand, overlapping adds a lot of complexity to the question about the most efficient algorithm. The conclusions that are drawn for this case are the following:

- EXH and SIM are again affected by the LRU buffer by improving their performance up to a factor of 2. Contrary to the previous case, STD also takes gain of the buffer while HEAP remains non-sensitive (only a 10% improvement is shown). The result of that behavior is that HEAP quickly loses its relative advantage and, after the threshold of $B = 4$ pages, the others outperform it.
- For large LRU buffer sizes, the three recursive algorithms (EXH, SIM, and STD) show interesting similarities. First, the degree of improvement is similar as the buffer grows (5%-10% for each duplication of buffer size). Second, their behavior is independent from the cardinality of the data sets.

Regarding sensitivity to buffer size, the non-recursive HEAP algorithm is quite stable because it processes the nodes from each R-tree in a simultaneous way. On the other hand, the recursive EXH, SIM and STD algorithms are more sensitive to the capacity of the LRU buffer because they process a 1-CP query by following a depth-first traversal pattern.

4.4 General guidelines

General guidelines that arise through this experimentation are the following:

- STD and HEAP are the most efficient, since they usually outperform the other two up to one order of magnitude (for zero buffer size).
- The overlap factor between joined data sets deserves serious consideration when performing experimentation on CP algorithms since the cost for fully overlapping workspaces is several orders of magnitude higher than that for disjoint workspaces. According to our experiments, zero or small (at most 5%) overlap gives a serious advantage to the three non-exhaustive algorithms since they turn out to be 2-20 times faster than the exhaustive one.
- Although HEAP is the winner for zero buffer, it quickly loses its relative advantage as buffer size increases and, after the threshold of $B = 4$, the other three algorithms outperform it.

5 K -CPQs Performance Comparison

We proceed with evaluating the performance of the four algorithms for K -CPQs also taking into consideration several parameters that affect performance (Subsection 5.1). A comparison with an incremental approach, already found in the literature [11], is also included (Subsection 5.2).

5.1 Comparison of the four algorithms

In correspondence to Section 4, we first assume zero buffer (Subsection 5.1.1) and then measure the sensitivity of the algorithms on the overlap (Subsection 5.1.2) and the buffer size (Subsection 5.1.3).

5.1.1 Experiments with zero buffer

For this set of experiments, we have chosen to run K -CPQs between the real and the uniform data set, with K varying from 1 up to 100,000. Figure 7 illustrates the performance of each algorithm assuming (a) 0% and (b) 100% overlapping workspaces. Obviously, the

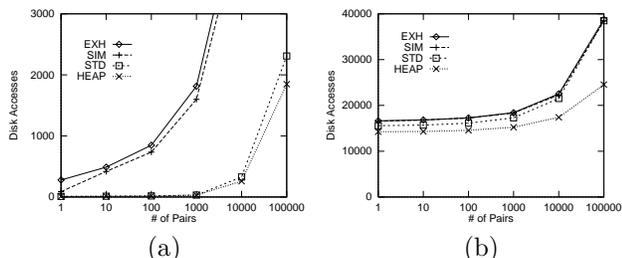


Figure 7: Comparison of the four K -CP Algorithms for Varying K : Real vs. Uniform Data in (a) 0% and (b) 100% Overlapping Workspaces.

cost of each algorithm gets higher as K increases. Interestingly, the deterioration is not smooth; after a threshold the cost increases exponentially. This threshold is usually between $K = 100$ and 1,000. Another observation is that the portion of overlapping

again plays an important role in the ranking of the four algorithms, as already detected in Section 4 (for $K = 1$). For non-overlapping data sets (Figure 7a), STD and HEAP are 10 - 50 times faster than EXH, whereas SIM cannot achieve a significant improvement. On the other hand, for overlapping workspaces (Figure 7b), it is only HEAP that clearly improves performance with respect to EXH (although less than before: 10%-30%). Since the overlap factor again turns out to be crucial for the relative performance of the K -CP algorithms we further investigate it in the sequel.

5.1.2 The effect of the overlap

The goal of this set of experiments is to detect a threshold (if such exists) that would be used as a guideline for an effective choice among the four algorithms. To reach such a conclusion, we illustrate in Figure 8 the relative cost of the STD and HEAP algorithms with respect to the cost of EXH. For each experiment, it was again the real and the uniform data sets chosen, with K varying from 1 up to 100,000. The results detect the winner for each configuration:

- SIM cannot improve the performance more than 20% except the case where $K = 1$ and overlap less than 25% (see also [5]).
- STD and HEAP are almost equivalent for overlap less than 10% (being from 5 up to 50 times faster than EXH) and, then, HEAP clearly outperforms STD with a relative gap increasing with K . For overlap more than 50%, HEAP achieves 15% (for small K values) up to 35% (for large K values) savings in I/O compared with the rest algorithms.

As a guideline, for disjoint workspaces STD and HEAP are both very efficient while for overlapping workspaces HEAP is the algorithm to be preferred, and this holds for an arbitrary K value.

5.1.3 The effect of the LRU-buffer

Similar to Subsection 4.3.3, an LRU buffer varying from $B = 0$ up to 256 pages was dedicated to the two R-trees in two equal portions of $B/2$ pages. For each experiment, it was again, on the one hand, the real data set and, on the other hand, the uniform data set. Part of the results is illustrated in Figure 9 (for the (a) STD and (b) HEAP algorithms). The charts correspond to overlap 0% between the two data sets. According to these results (appearing in more detail in [5]), for 0% overlap between two data sets, SIM and STD take advantage of the buffer and their cost is significantly reduced as the buffer size increases (up to one order of magnitude for $K = 100,000$ and $B = 256$ pages). On the other hand, HEAP is sensitive to the buffer size only for large K values saving more than half of its cost when $K \geq 10,000$ and $B > 16$ pages. When comparing

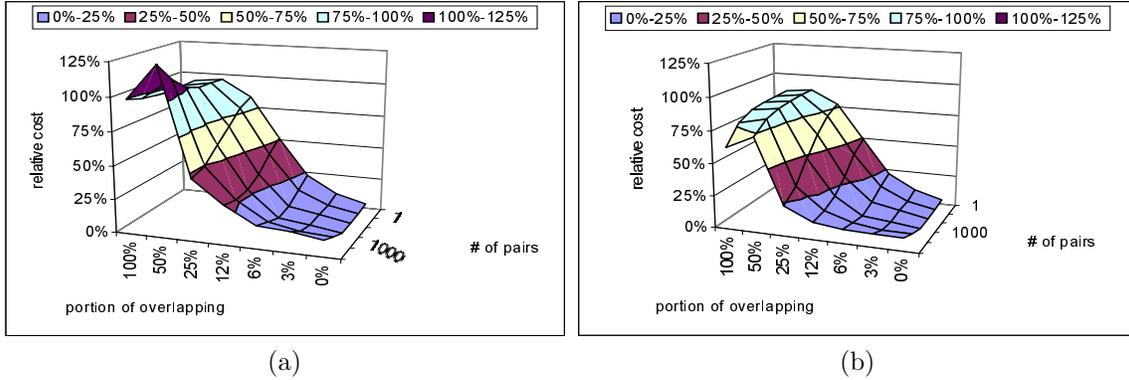


Figure 8: Finding a Threshold on the Overlap Factor for Varying K in (a) STD, and (b) HEAP.

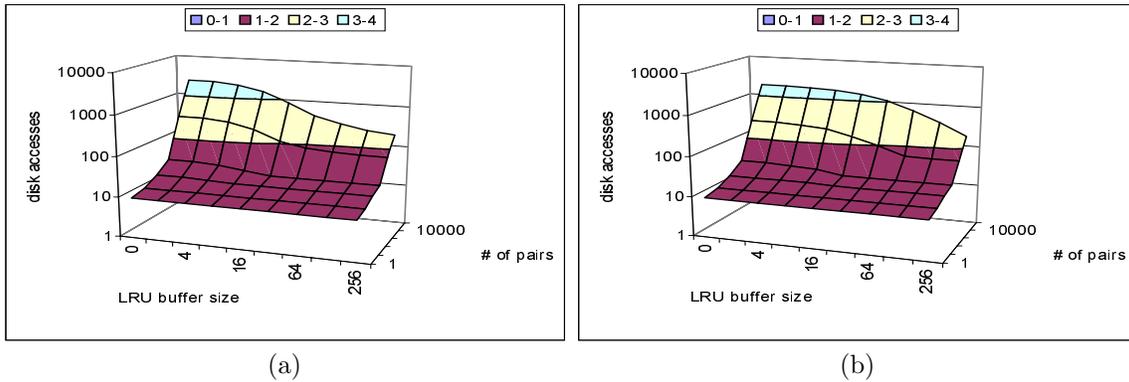


Figure 9: The Effect of the LRU Buffer Size for Varying K in (a) STD and (b) HEAP. Log Scale.

it with the other two algorithms, it turns out that the superiority of HEAP (as already shown in Figure 7) is overcome by its lack of sensitivity to the existence of the LRU buffer. Thus, STD quickly outperforms HEAP, i.e., for $B > 4$ pages. We have also run this set of experiments for overlapping workspaces as well and reached to similar conclusions, therefore they are not illustrated. A remark that deserves more attention and does not appear for 0% overlap, concerns the behavior of SIM with respect to STD and HEAP: for large buffer sizes SIM competes both of them and, in some cases, even outperforms them, although marginally. Overall, as a general hint, we suggest that HEAP (respectively, STD) is the algorithm to be preferred when the LRU buffer is small enough (respectively, medium to large).

5.2 Comparison with the incremental approach

In [11] three alternative tree traversal policies were presented: basic, even, and simultaneous tree traversals (in the sequel, BAS, EVN, and SML, respectively), as already discussed in Section 3. We have implemented all three alternatives and, although our algorithms are not incremental, in this subsection we provide

a performance comparison in order to extract useful conclusions about the behavior of each one. In the rest of the section, we compare the performance of STD and HEAP, on the one hand, with EVN and SML, on the other hand, since BAS turned out to be inefficient for most settings of our experiments (included in [5]). Figure 10 illustrates the performance of two algorithms (STD and HEAP) proposed in Section 3 and two algorithms (EVN and SML) proposed in [11] for all combinations of two settings for buffer size (0 and 128 pages) and two settings for overlapping factor (0% and 100%). According to these experiments, EVN is competitive for small K values but turns out to be inefficient for $K \geq 10,000$. For zero buffer size, it is HEAP and SML that outperform the others in most cases while for large buffer size, STD is the most efficient. The advantage of STD due to large buffer has been also mentioned in Subsection 5.1.3. It is also worth mentioning that for disjoint workspaces HEAP and SML appear to have identical behavior.⁴

⁴Both algorithms work in a simultaneous way. Apart from their policy in filling the heap structure, a major difference consists of the policy in treatment of ties. However, this is not reflected in the overall I/O cost when the workspaces are disjoint, since ties are rare in that case.

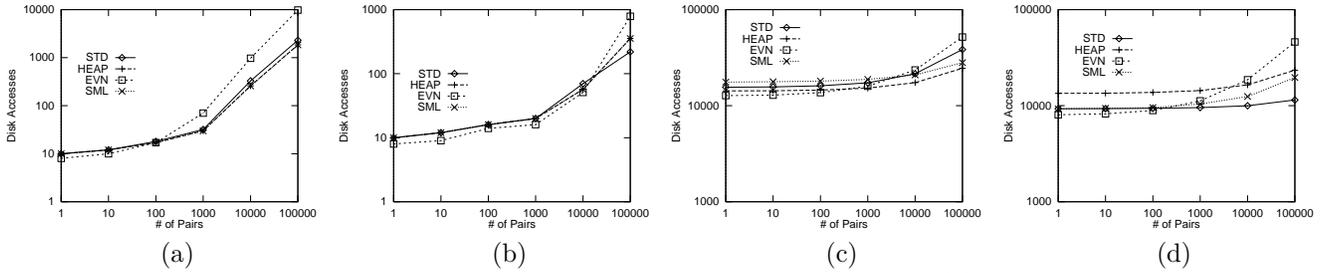


Figure 10: Comparison of two Incremental and two Non-incremental Algorithms: (a) No Buffer, Overlap = 0%, (b) 128 Page Buffer, Overlap = 0%, (c) No Buffer, Overlap = 100%, (d) 128 Page Buffer, Overlap = 100%. Log Scale.

Additional results, exploring the effect of the buffer size in the relative performance of SML, STD and HEAP, are presented in [5].

5.3 General guidelines

Overall, processing K -CPQs is a very expensive operation. Among the four proposed algorithms we can choose the most robust ones following some guidelines:

- Assuming no buffer, STD and HEAP are both very efficient when the workspaces of the two data sets do not overlap while, for overlapping workspaces, HEAP is the most efficient choice.
- When assuming a buffer of reasonable size (e.g., $B > 4$ pages), STD outperforms HEAP, since the latter one turned out to be insensitive to the existence of the LRU buffer.

Regarding the incremental algorithms proposed in [11], although SML is a competitive approach for several configurations, it is again HEAP (for zero or small buffer) and STD (for large buffer) that usually outperform SML with a gap reaching up to 20% and 50%, respectively. On the other hand, unlike the three competitors (SML, STD, and HEAP), EVN is not stable to the variety of K values we experimented with.

6 Conclusions

CPQs are important in spatial databases; [19] includes a variation of this query in a benchmark for Paradise [6]. However, unlike in computational geometry literature [7, 12], efficient processing of the CP problem has not gained special attention in spatial database research. Certain other problems of computational geometry, including the “all nearest neighbor” problem (that is related to the CP problem), have been solved for external memory systems [9]. To the authors’ knowledge, [11] is the only reference to this type of queries in the spatial database literature. In this paper, we presented a naive and four improved algorithms to process both 1- and K -CPQs. Three out of them are recursive (the exhaustive, the simple, and the sorted distances) and one is iterative (heap). An extensive experimentation

was also included, which resulted to several conclusions about the efficiency of each algorithm with respect to K , the size of the underlying buffer and the portion of overlapping between the workspaces of the two data sets. They are listed as follows:

- Sorted distances and Heap are the most promising algorithms, since they perform well in the vast majority of different configurations.
- Buffer size plays an important role, as presented in detail, since it gives bonus to simple and sorted distances rather than heap.
- K does not radically affect the relative performance and the previous conclusions generally hold even for large K values.

We have also presented the novel “fix-at-root” technique for treating R-trees with different heights as an efficient, as turned out to be, alternative of the popular “fix-at-leaves” technique. In comparison with the algorithms proposed in [11], our algorithms are more stable with K and (especially, sorted distances) usually outperform them by 10% - 50%. Our work is also the first that addresses and evaluates the effect of the portion of overlapping in the performance of CP algorithms ([11] considered fully overlapping workspaces only). As presented through the experiments, a small increase in the overlap between the data sets may cause performance deterioration of orders of magnitude, in terms of I/O cost and this is a key issue for effective query optimization. Moreover, this behavior raises the issue of the ‘meaning’ of CPQs under some conditions. Following [2], we plan to explore the effect of overlap on the CP problem, taking into account the (geometric) distinction between the closest and the most remote pairs.

One direction of future work is to study two special cases of CPQs: “self-CPQ” and “Semi-CPQ”. In the first case, both data sets actually refer to the same entity ($P \equiv Q$) [5]. In the second case, a set of point pairs is produced, where the first point of each pair appears only once in the result (i.e. for each point in P , the nearest point in Q is discovered) [11]. Other directions of future work also include (a) the study of

multi-way CPQs where tuples of objects are expected to be the answers, extending related work in multi-way spatial joins [14, 16] and (b) the analytical study of CPQs, extending related work in spatial joins [23] and nearest-neighbor queries [17].

References

- [1] N. Beckmann, H.P. Kriegel, R. Schneider and B. Seeger: "The R*-tree: an Efficient and Robust Access Method for Points and Rectangles", *Proc. 1990 ACM SIGMOD Conf.*, pp.322-331, Atlantic City, NJ, 1990.
- [2] K.S. Beyer, J. Goldstein, R. Ramakrishnan and U. Shaft: "When Is 'Nearest Neighbor' Meaningful?", *Proc. 7th Int. Conf. on Database Theory (ICDT'99)*, pp.217-235, Jerusalem, Israel, 1999.
- [3] T. Brinkhoff, H.-P. Kriegel and B. Seeger: "Efficient Processing of Spatial Joins Using R-trees", *Proc. 1993 ACM SIGMOD Conf.*, pp.237-246, Washington, DC, 1993.
- [4] A. Corral, M. Vassilakopoulos and Y. Manolopoulos: "Algorithms for Joining R-trees and Linear Region Quadtrees", *Proc. 6th Int. Symp. on Spatial Databases (SSD'99)*, pp.251-269, Hong Kong, China, 1999.
- [5] A. Corral, Y. Manolopoulos, Y. Theodoridis and M. Vassilakopoulos: "Closest Pair Queries in Spatial Databases", Technical Report, Data Engineering Lab, Dept. of Informatics, Aristotle Univ. of Thessaloniki, Greece, 1999 (available from URL: <http://delab.csd.auth.gr/~michalis/cpq.html>).
- [6] D.J. DeWitt, N. Kabra, J. Luo, J.M. Patel and J.-B. Yu: "Client-Server Paradise", *Proc. 20th VLDB Conf.*, pp. 558-569, Santiago, Chile, 1994.
- [7] M. Dietzfelbinger, T. Hagerup, J. Katajainen and M. Penttonen: "A Reliable Randomized Algorithm for the Closest-Pair Problem", *Journal of Algorithms*, Vol.25, No.1, pp.19-51, 1997.
- [8] V. Gaede and O. Guenther: "Multidimensional Access Methods", *ACM Computer Surveys*, Vol.30, No.2, pp.170-231, 1998.
- [9] M.T. Goodrich, J.-J. Tsay, D.E. Vengroff and J.S. Vitter: "External-Memory Computational Geometry", *Proc. 34th Annual IEEE Symp. on Foundations of Comp. Science (FOCS'93)*, pp.714-723, Palo Alto, CA, 1993.
- [10] A. Guttman: "R-trees - a Dynamic Index Structure for Spatial Searching", *Proc. 1984 ACM SIGMOD Conf.*, pp.47-57, Boston, MA, 1984.
- [11] G.R. Hjaltason and H. Samet: "Incremental Distance Join Algorithms for Spatial Databases", *Proc. 1998 ACM SIGMOD Conf.*, pp.237-248, Seattle, WA, 1998.
- [12] S. Khuller and Y. Matias: "A Simple Randomized Sieve Algorithm for the Closest-Pair Problem", *Information and Computation*, Vol.118, No.1, pp.34-37, 1995.
- [13] S.T. Leutenegger and M.A. Lopez: "The Effect of Buffering on the Performance of R-Trees", *Proc. 14th IEEE Int. Conf. on Data Engineering (ICDE'98)*, pp.164-171, Orlando, FL, 1998.
- [14] N. Mamoulis and D. Papadias: "Integration of Spatial Join Algorithms for Processing Multiple Inputs", *Proc. 1999 ACM SIGMOD Conf.*, pp.1-12, Philadelphia, PA, 1999.
- [15] Y. Manolopoulos, Y. Theodoridis and V. Tsotras: *Advanced Database Indexing*, Kluwer Academic Publishers, 1999.
- [16] D. Papadias, N. Mamoulis and Y. Theodoridis: "Processing and Optimization of Multi-way Spatial Joins Using R-trees", *Proc. 18th ACM PODS Symp. (PODS'99)*, pp.44-55, Philadelphia, PA, 1999.
- [17] A.N. Papadopoulos and Y. Manolopoulos: "Performance of Nearest Neighbor Queries in R-Trees", *Proc. 6th Int. Conf. on Database Theory (ICDT'97)*, pp.394-408, Delphi, Greece, 1997.
- [18] A.N. Papadopoulos and Y. Manolopoulos: "Nearest Neighbor Queries in Shared-Nothing Environments", *Geoinformatica*, Vol.1, No.4, pp.369-392, 1997.
- [19] J.M. Patel et al.: "Building a Scalable Geo-Spatial DBMS: Technology, Implementation, and Evaluation", *Proc. 1997 ACM SIGMOD Conf.*, pp.336-347, Tucson, AZ, 1997.
- [20] F.P. Preparata and M.I. Shamos: *Computational Geometry: an Introduction*, Springer-Verlag, 1985.
- [21] N. Roussopoulos, S. Kelley and F. Vincent: "Nearest Neighbor Queries", *Proc. 1995 ACM SIGMOD Conf.*, pp.71-79, San Jose, CA, 1995.
- [22] M. Stonebraker, J. Frew, K. Gardels and J. Meredith: "The Sequoia 2000 Benchmark", *Proc. 1993 ACM SIGMOD Conf.*, pp.2-11, Washington, DC, 1993.
- [23] Y. Theodoridis, E. Stefanakis and T. Sellis: "Cost Models for Join Queries in Spatial Databases", *Proc. 14th IEEE Int. Conf. on Data Engineering (ICDE'98)*, pp.476-483, Orlando, FL, 1998.