All-Nearest-Neighbors Queries in Spatial Databases

Jun Zhang
School of Computer Engineering
Nanyang Technological University
Nanyang Avenue, Singapore
jzhang@ntu.edu.sg

Nikos Mamoulis
Department of Computer Science and Information Systems
University of Hong Kong
Pokfulam Road, Hong Kong
nikos@csis.hku.hk

Dimitris Papadias
Department of Computer Science
Hong Kong University of Science and Technology
Clearwater Bay, Hong Kong
dimitris@cs.ust.hk

Yufei Tao
Department of Computer Science
City University of Hong Kong
Tat Chee Avenue, Hong Kong
taoyf@cityu.edu.hk

Abstract

Given two sets A and B of multidimensional objects, the all-nearest-neighbors (ANN) query retrieves for each object in A its nearest neighbor in B. Although this operation is common in several applications, it has not received much attention in the database literature. In this paper we study alternative methods for processing ANN queries depending on whether A and B are indexed. Our algorithms are evaluated through extensive experimentation using synthetic and real datasets. The performance studies show that they are an order of magnitude faster than a previous approach based on closest-pairs query processing.

1 Introduction

Let A and B be two spatial datasets and $dist(p,q)$ be a distance metric. Then, the *all-nearest-neighbors* query is defined as: $ANN(A,B) = \{<a_i,b_j>: \forall\ a_i \in A, \exists\ b_j \in B, \neg\exists\ b_k \in B\ \{dist(a_i,b_k) < dist(a_i,b_j)\}\}$. In other words, the query finds for each object in A its nearest neighbor(s) in B. Notice that $ANN(A,B)$ is not commutative, i.e., in general $ANN(A,B) \neq ANN(B,A)$. The ANN problem has been studied in the context of computational geometry [PS85], where several main memory techniques have been proposed (e.g., [Cla83]) for the case where $A=B$, i.e., the nearest neighbors are found in the same dataset. However, limited work has been done in the context of secondary memory algorithms, although this query type is frequent in several database applications:

- *Geographical Information Systems*: Example queries include "find the nearest parking lot for each subway station" or "find the nearest warehouse for each supermarket", common in urban planning and resource allocation problems.
- *Data analysis*: ANN queries have been considered as a core module of clustering [JMF99] and outlier detection [AY01]. For example, the algorithm of [NTM01] owes its efficiency to the use of ANN queries, as opposed to previous quadratic-cost approaches.
- *Computer Architecture/VLSI design*: The operability and speed of very large circuits depends on the relative distance between the various components in them. ANN is applied to detect abnormalities and guide relocation of components [NO97].

Previous methods [HS98, CMTV01] for ANN evaluation in secondary memory are inefficient and applicable only when both A and B are indexed, which is not necessarily true in practice (e.g., one or both query inputs could be intermediate results of complex queries). In this paper, we propose novel techniques for general ANN query processing. Following the common trend in the literature, we assume that the underlying indexes (whenever available) are R-trees [Gut84, BKSS90]. Although, for simplicity, we deal with points and use Euclidean distance, extensions to other data partition access methods, extended objects and other distance metrics are straightforward. The rest of the paper is organized as follows. Section 2 discusses previous work directly related to the ANN problem. Sections 3 and 4 present algorithms for different cases, based on whether A, B, or both are indexed. Section 5 experimentally evaluates the algorithms, and section 6 concludes the paper with a discussion.

2 Related work

ANN queries constitute a hybrid of nearest neighbor search and spatial joins; therefore, in sections 2.1 and 2.2 we review related work for these query types focusing more on the processing techniques that are also employed by our algorithms. Section 2.3 describes methods for closest-pair queries, and section 2.4 discusses existing techniques for ANN query processing.

2.1 Nearest neighbor queries

The goal of nearest neighbor (NN) search is to find the objects in a dataset A that are closest to a query point q. Existing algorithms presume that the dataset is indexed by an R-tree and use various metrics to prune the search space: *mindist(q,M)* is the minimum distance between q

and any point in a minimum bounding rectangle (MBR) *M*. The algorithm of [RKV95] traverses the tree in a depth-first (DF) manner. Assume that we search for the nearest neighbor NN(*q*,*R*) of *q* in R-tree *R*. Starting from the root, all entries are sorted according to their *mindist* from *q*, and the entry with the smallest *mindist* is visited first. The process is repeated recursively until the leaf level where a potential nearest neighbor is found. During backtracking to the upper levels, the algorithm only visits entries whose *mindist* is smaller than the distance of the nearest neighbor found so far. As an example consider the R-tree of Figure 1, where the number in each entry refers to the *mindist* (for intermediate entries) or the actual distance (for leaf entries, i.e., objects) from *q* (these numbers are not stored but computed dynamically during query processing). DF would first visit the node of root entry E_1 (since it has the minimum *mindist*), and then the node pointed by E_4, where the first candidate (*a*) is retrieved. When backtracking to the previous level, entry E_6 is excluded since its *mindist* is greater than the distance of *a*, but E_5 has to be visited before backtracking again to the root level.

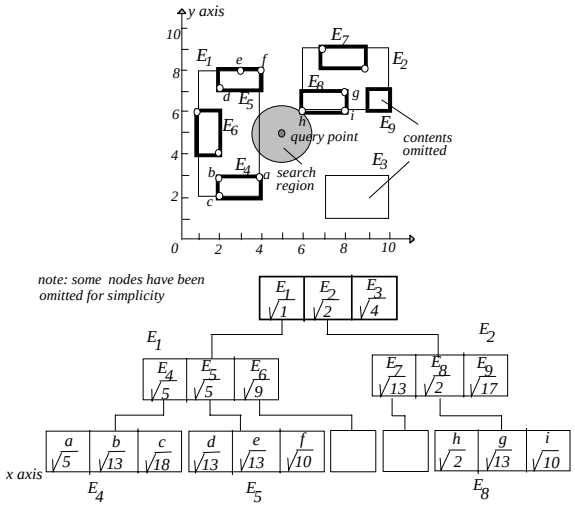


Figure 1: Example R-tree and *mindist* values

The performance of DF was shown to be suboptimal in [PM97], which reveals that an optimal algorithm only needs to visit those nodes whose MBRs intersect the so-called "search region", i.e., a circle centered at the query point with radius equal to the distance between the query and its nearest neighbor (shaded circle in Figure 1). A best-first (BF) algorithm for NN search is proposed in [HS99]. BF keeps a *heap* with the entries of the nodes visited so far. Initially the heap contains the entries of the root sorted according to their *mindist*. When E_1 is visited, it is removed from the heap and the entries of its node (E_4, E_5, E_6) are added together with their *mindist*. The next entry visited is E_2 (it has the minimum *mindist* in the heap), followed by E_8, where the actual result (*h*) is found and the algorithm terminates. BF is I/O optimal because it only visits the nodes necessary for obtaining the nearest neighbor. These methods can be easily extended for finding the *k* nearest neighbors of *q*. Nevertheless in high dimensional spaces the performance of spatial access methods degenerates and specialized techniques are used to solve the problem.

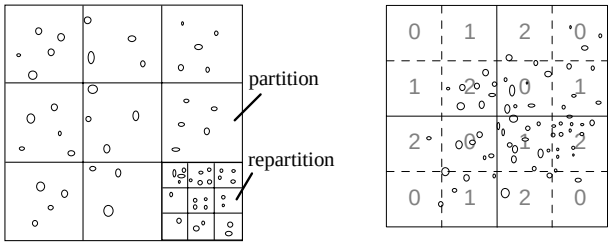
2.2 Spatial joins

The spatial join between two datasets *A* and *B* finds the object pairs in the Cartesian product *A*×*B* which satisfy a spatial predicate, most commonly *intersect* (assuming the datasets contain objects with spatial extent). Depending on the existence of indexes, different spatial join algorithms can be applied. The *R-tree join* algorithm (RJ), proposed in [BKS93], computes the spatial join of two inputs indexed by R-trees. RJ synchronously traverses both trees, starting from the roots and following entry pairs which intersect. Let E_A be a node entry from R-tree R_A, and E_B a node entry from R-tree R_B. RJ is based on the following property: if the MBRs of E_A and E_B do not intersect, there can be no pair $<a_i,b_j>$ of intersecting objects, where a_i and b_j are pointed by E_A and E_B, respectively. If only one dataset (let *A*) is indexed, a common method [LR94] is to build an R-tree for *B* and then apply RJ. In [MP99], a hash-based algorithm is proposed that uses the existing tree (of *A*) to determine the hash partitions. If both datasets are non-indexed, alternative methods include sorting and external memory plane-sweep [APR+98], or spatial hash join algorithms, like partition based spatial merge join (PBSM) [PD96].

PBSM divides the space regularly using an orthogonal grid and hashes objects from both datasets into the partitions (buckets); each object is assigned to the buckets that contain it. Figure 2a illustrates a regular 3×3 partitioning and some hashed data. During the matching phase of the hash join, pairs of buckets from the two datasets that correspond to the same area are loaded and joined in memory (e.g., using plane-sweep). If the data in a bucket do not fit in memory, the algorithm recursively repartitions the cell into smaller parts and re-hashes the objects. In order to alleviate repartitioning of skewed data, which increases the cost of the algorithm, PBSM defines the hash buckets by grouping multiple grid tiles together. A tile numbering with a hash function is defined and tiles with the same hash value are assigned to the same bucket.

Figure 2b shows a 4×4 tiling with some objects hashed in it. Three buckets are used to hash the objects and the tiles are assigned to them according to a round-robin hash function. Skewed data are now distributed more evenly to buckets than if we used three continuous partitions. Objects that span the grid borderlines are assigned to multiple buckets during PBSM (replication), thus the output of the algorithm has to be sorted in order to remove pairs reported more

than once. Duplicate elimination, however, can be combined with the refinement step incurring minimal overhead.



(a) Regular partitioning (b) Hashing using tiles

Figure 2: Regular partitioning by PBSM

2.3 Closest pairs queries

Given two datasets A and B, the closest pairs (CP) query asks for the k closest pairs in $A \times B$. If both A and B are indexed by R-trees, the concept of synchronous tree traversal (employed by RJ) and DF or BF (discussed in section 2.1) can be combined for query processing. As an example consider that k=1 and DF (depth-first) is applied. A CP-DF algorithm would visit the roots of the two R-trees and recursively follow the pair of entries $<E_A, E_B>$, $E_A \in R_A$ and $E_B \in R_B$, whose *mindist* is the minimum among all pairs. The difference from RJ is that sometimes nodes that do not overlap have to be visited, if they can contain points whose distance is smaller than the minimum distance found. The application of BF is also similar to the case of NN queries. A number of optimization techniques, including the application of other metrics (*maxdist*, *minmaxdist*) for pruning, have been proposed in [HS98, CMTV00]. Additional techniques [SML00] include sorting and application of plane-sweep during the expansion of node pairs, using estimates of the k-closest pair distance to suspend unnecessary computations of MBR distances. Since we deal with ANN queries we focus on the relevant techniques.

2.4 Existing techniques for ANN queries

A naïve approach to process an ANN query is to perform one NN query on dataset B for each object in dataset A. In [BEKS00], several optimization techniques are proposed to improve the CPU and I/O performance of multiple similarity queries on a dataset (in our case, we perform multiple nearest neighbor queries on B). The optimizations assume that the queries fit in main memory; thus, they cannot be directly applied to ANN problem when the size of A is larger than the size of the available memory.

[HS98] and [CMTV01] use CP algorithms to process ANN queries. Both papers propose the same processing method. A bitmap S_0 of size $|A|$ indicates the points in A for which the NN has been found. An incremental CP query is executed and this memory-resident bitmap is updated while the closest pairs are computed. Since the closest pairs are output in increasing distance order, we know that if a pair $<a_i, b_j>$ is the first for a_i, then b_j is the NN for a_i; $S_0[a_i]$ is set to 1 to prevent updating the NN of a_i in the future (i.e., when another pair containing a_i is found). The algorithm terminates when $S_0[a_i] = 1$ for all $a_i \in A$.

As shown in the experimental section, this method is even worse than the straightforward application of one NN query for each point in A. Its inherent drawback is that it was developed based on the requirements of CP processing (i.e., multiple pairs for each point in A, incremental output of the results), whereas ANN queries have different characteristics. For example, assume that there is a point a_i in A whose distance $NNdist(a_i,B)$ from its NN in B is large. Since the termination condition for the CP-based algorithm is the identification of the nearest neighbor for all points in A, many node pairs with smaller distance than $NNdist(a_i,B)$ will have to be visited, incurring significant overhead.

An external memory algorithm for ANN is proposed in [GTVV93], suitable for the case where A=B, i.e., the NN are retrieved from the same dataset. This method applies on a non-indexed dataset, and initially hashes all points in vertical stripes. It then performs sorting in each stripe and uses a plane sweep algorithm that concurrently scans all stripes (potentially in parallel) finding the NN of each point in its current or neighbor stripes. The authors do not provide algorithmic details and they study only the worst case I/O performance. [BK02] study the *nearest neighbor join*, which finds for each object in one dataset, its k nearest neighbors in another dataset (when k = 1, this corresponds to an ANN query). Their solution is based on a specialized index structure called multipage index and is inapplicable for general-purpose index structures (e.g., R-trees).

In this paper, we propose alternative techniques for processing ANN queries. Depending on whether A or B are indexed by R-trees, we suggest different evaluation strategies. For simplicity, our methods compute for each point a single nearest neighbor, even if there exist multiple (with equal distance). Extensions to the generalized case are straightforward. Table 1 summarizes the most frequently used symbols throughout the paper.

Symbol	Description
$\|A\|,\|B\|$	Cardinality of A, B
R_A, R_B	R-tree for A, B
N_A, N_B	(Leaf) node of R_A, R_B
E_A, E_B	Node entry of R_A, R_B
G_A	Group of objects from A
H_A, H_B	Hash bucket from A, B
$PS(t)$	Pending set of a hash tile t
$NN(a_i, X)$	NN of $a_i \in A$ in X=B, N_B, H_B, t
$NNdist(a_i, X)$	Distance between a_i and $NN(a_i, X)$

Table 1: Table of Symbols

3 Index-based ANN methods

When B is indexed, we can take advantage of R_B to accelerate search. Starting from the rather straightforward approach that applies one NN query on R_B for each point in A, we propose more sophisticated methods that aim at reducing the processing cost. All methods are based on depth-first traversal due to its lower I/O cost in the presence of buffers [CMTV00]. The extension to the best-first search paradigm is straightforward.

3.1 Multiple nearest neighbor search

The *multiple nearest neighbor* (MNN) algorithm can be considered as the counterpart of index-nested-loops in relational databases. In particular, MNN applies $|A|$ NN queries (one for each point in A) on R-tree R_B. The order of the NN queries is very important since if two consecutive query points (of A) are close to each other, a large percentage of pages from R_B needed during the second query will be in the LRU memory buffer due to the first one. In order to achieve this, we employ the following methods:

If A is not indexed, we sort its points using a space filling curve (e.g., Hilbert order [Bia69]) and visit them in this order, to maximize locality.

If A is indexed, its points are already well-clustered in the leaf nodes of R_A. We exploit this clustering and further increase spatial locality, by traversing the tree, following the entries of each node by Hilbert value (with respect to the center of node's MBR), and using this order to apply NN queries on leaf node entries.

Due to the proximity of successive query points, MNN is expected to be efficient in terms of I/O. However, its CPU cost is high because of the numerous distance calculations for each NN search. Let f_B be the average fanout of R_B, and $NA_{NN}(a_i,R_B)$ the average number of node accesses in R_B for finding the nearest neighbor of $a_i \in A$. Then, the number of distance computations for *MNN(A,B)* is $|A| \cdot f_B \cdot NA_{NN}(a_i,R_B)$. In other words, for each NN query the distance between a_i and all entries in the visited nodes from R_B has to be computed. In addition, these distances have to be sorted (or inserted in a heap) individually for each query (since the entries of a node are visited in increasing distance order).

3.2 Batched nearest neighbor search

In order to reduce the high computational cost of MNN, we propose a *batched* NN (BNN) method, which retrieves the nearest neighbors for multiple points at a time. BNN splits the points from A into a number of n groups G_{A1}, G_{A2}, …, G_{An}, such that $\cup G_{Ai} = A$ and $\forall i,j$, $1 \le i < j \le n$: $G_{Ai} \cap G_{Aj} = \emptyset$. Then for each group, R_B is traversed only once, reducing considerably the number of distance computations.

BNN first initializes information about the NN of each point a_i in G_A (and its distance) and a *globaldist*(G_A, B) parameter, which stores the maximum *NNdist*(a_i, B) for all points $a_i \in G_A$. Then R_B is traversed as in MNN, by recursively visiting the nodes in increasing *mindist* order from G_A. In case of a tie, i.e., two or more entries have the same *mindist* from G_A, we pick the one with the maximum overlap with G_A. Obviously, entries E_B in intermediate nodes of R_B for which *mindist*(G_A, E_B) > *globaldist*(G_A, B) can be pruned from search, since they cannot point to the NN of any point in G_A.

When a leaf node N_B of R_B is visited, each $a_i \in G_A$ updates its NN in the points of NB. A brute-force approach would compute the distance *dist*(a_i, b_j) $\forall$ $a_i \in G_A$, $b_j \in N_B$. In order to reduce this quadratic cost, we remove from consideration (i) all a_i, for which *NNdist*(a_i, B) < *mindist*(a_i, NB) and (ii) all b_j for which *mindist*(G_A, b_j) > *globaldist*(G_A, B). For example consider the G_A and N_B instances of Figure 3, where the radii of the circles centered at each $a_i \in G_A$ correspond to the current *NNdist*(a_i, B). Point a_1 will be pruned at this preprocessing step because *NNdist*(a_1, B)<*mindist*(a_1, NB). The same is true for b_6 and b_7 because their *mindist* from G_A is larger than the current *globaldist*(G_A, B) = *NNdist*(a_4,B), i.e., the largest radius of the points in G_A.

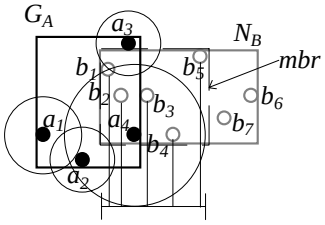


Figure 3: Optimizing *updateNN*

After pruning some points, we can avoid checking the Cartesian product of the remaining ones by employing a method similar to that of [SML00] for CP queries. Let *mbr* be the MBR of the remaining points in N_B (the MBR of b_1,…,b_5 in the example). The *axis* with the largest *mbr* projection (let x) is chosen and all remaining b_j's are sorted in increasing order of their x-coordinate (i.e., b_1, b_2, b_3, b_4, b_5). Observe that if a point $a_i \in G_A$ is on the left of some b_j and *NNdist*(a_i, B)<*dist*($a_i.x$, $b_j.x$) then no point b_m, $m>j$ can be the NN of a_i. Similarly, if a_i is on the right of b_j, all b_m, $m<j$ can be pruned. We use this observation to accelerate search as follows. For each a_i, binary search is applied to find the point b_j with the smallest x-distance from a_i. The actual *dist*(a_i, b_j) is then computed and *NN*(a_i, B) is updated if necessary. We continue checking (in both x-directions) the other points in the list in increasing x-distance order from a_i, until a point b_j for which *NNdist*(a_i, B)<*dist*($a_i.x$, $b_j.x$) is found. Consider for instance Figure 3, and assume that we want to update *NN*(a_2, B). The closest point in the x-axis is b_1, which is checked first. Then b_2 is visited and we find that

NNdist(a_2, B)<*dist*($a_2.x$, $b_2.x$). Thus, we need not compute a_2's distance from b_2 and subsequent points. Similarly, for a_3 we (bi-directionally) check b_2, b_3 and b_1 and avoid checking b_4 and b_5. These optimization methods reduce the computational cost of BNN considerably (50%-100% for our experimental data), so we include them in the implementation of BNN.

Although a bound similar to *globaldist*(E_A, B) could reduce the cost of CP, in practice it cannot be applied because it requires knowledge of *NNdist*(a_i, B) for each $a_i \in A$, which cannot be kept in memory. On the other hand, *NNdist*(a_i, B) for each $a_i \in G_A$ can be easily maintained by BNN. This further increases the value of BNN in processing ANN queries; better pruning bounds lead to better performance.

3.3 Choosing groups in BNN

Optimization of BNN must take into account several criteria: (i) the number of points in each group G_A should be maximized in order to decrease the number of NN queries, (ii) the MBR of each group should be minimized in order to reduce the cost of each NN query; (iii) each group should be small enough to fit in memory. Notice that criterion (i) is in conflict with (ii) and (iii). In the extreme case where only criterion (i) is considered, BNN transforms to an algorithm that would attempt to read the whole A dataset in memory and compute the NN of all points simultaneously. In the opposite case (the MBR of each group is minimized), BNN degenerates to MNN. An optimal trade-off between these criteria is difficult to obtain.

Although the problem resembles clustering, it is essentially different. First, clustering does not have a MBR size constraint for each cluster. Second, clustering usually involves a parameter, which is the number of desired clusters. In our case the number of groups can be arbitrary. Third, there is a maximum allowed population for each group (but not for each cluster). Finally, clustering methods are too expensive to be considered for this problem; grouping should be performed with minimal overhead on the overall performance of BNN.

Assume first that dataset A is not indexed. To group the points in this case, we first sort them by to their Hilbert value to maximize proximity. Then consecutive points in this order are grouped[1] until (i) the area of the group's MBR exceeds a maximum threshold (*max_area*), or (ii) the number of points in the group exceeds a threshold (*max_num*). Notice that since successive groups are formed according the Hilbert order, they are likely to access similar pages and utilize the buffer.

[1] We have also tested another grouping condition that considers a maximum allowed density $|G_A|$/MBR(G_A) for each group, but found that it is not appropriate for this problem.

The values of *max_area* and *max_num* are such that the expected nodes of R_B accessed by BNN(G_A, R_B) fit in memory, together with the points of the current group G_A. In this way, we maximize the probability that a page of R_B required by the next group will be in the buffer. In addition, the MBR of G_A should not be too large, in order to avoid accessing unnecessary nodes of R_B in areas where A is much sparser than B. Using global statistics about R_B to tune the thresholds is not effective for real-life skewed data, since the local characteristics of R_B may vary. Therefore, we employ the following heuristic to optimize grouping: when a new group G_A is initialized, a NN query is performed for the next point a_i and the average area *avg_area* of the accessed leaf nodes from R_B is computed and used as an estimate of the leaf nodes of R_B close to G_A. If MBR(G_A) becomes larger than *avg_area*, G_A accesses more nodes than the available memory. Moreover, G_A may be much sparser than B, so nodes will be unnecessarily accessed. In our implementation we use *max_area* = *avg_area*, which provides a good trade-off between CPU and I/O efficiency. Threshold *max_num* is violated earlier than *max_area* in regions where A is very dense, and the points are not expected to fit in memory together with the nodes accessed by BNN. The value of *max_num* is estimated by cost models [PM97, BBKK97] for NN search.

If dataset A is indexed we utilize the R-tree as follows. Starting from the root, BNN follows entries recursively according to their Hilbert value with respect to the center of their container node. If a leaf node is reached, its points are inserted into the current group, until a grouping threshold is violated. Therefore, a leaf node may be split in several groups and in some cases (e.g., dense regions), a group may contain points from more than one leaf node. Figure 4 shows four leaf nodes of R_A, sorted by Hilbert value of their center with respect to the MBR of their container node (not shown). Assuming that the *max_area* is as shown in the figure, the first three points a_1, a_2, a_3 of node A_1 are grouped into G_1. Adding the fourth point a_4 would cause a *max_area* violation; thus, a_4 is a group by itself (G_2), whereas the fifth point a_5 together with the points of A_2, A_3, and A_4 form the third group G_3. Observe how irregular a good distribution of points to groups can be.

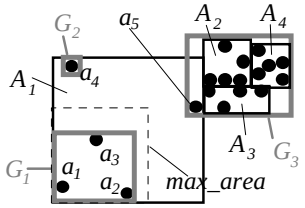


Figure 4: Adaptive grouping in the presence of R_A

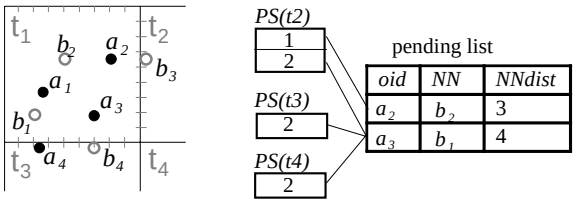
4 ANN on non-indexed datasets

If R_B is not present, we cannot directly use index-based ANN algorithms. A plausible solution is to build the

tree on-the-fly and apply the algorithms of section 3.1. Although bulk loading can accelerate the construction of R_B, the cost of external sorting can be high. Therefore we also investigate the application of an alternative technique based on hashing.

4.1 A hash-based ANN algorithm

We propose a two-phase hash-based ANN algorithm which (i) hashes the points from A and B in spatial partitions, (ii) loads pairs $<H_A, H_B>$, $H_A \in A$, $H_B \in B$ of buckets covering the same region and searches for each $a_i \in H_A$ its NN in H_B. In order to distribute the points evenly to the hash buckets, we consider the spatial hashing method used by PBSM [PD96] (described in section 2.2). A fine regular grid that contains more cells than the number of hash buckets is used to partition the points. Each bucket contains multiple tiles (i.e., grid cells) at different areas of the space. Since H_A and H_B cover the same area, the NN of each point $a_i \in H_A$ is likely to be in H_B. However, there are cases where NN(a_i, B) may not be in H_B. This holds for points that lie close to the border of a tile. In Figure 5a, for instance, the nearest neighbor b_3 of point $a_2 \in t_1$ lies in the neighbor tile t_2. Moreover, there may be a tile containing points from A, but no point from B.



(a) hash-based ANN (b) pending list and sets

Figure 5: Finding NN in different tiles

In order to handle these cases, we maintain in memory a list that contains information about the *border* points, the NN of which is not guaranteed to be in the same tile that contains them. An entry in this *pending list* contains the point, its current NN and its distance *NNdist* from it. We also assign to each tile t an initially empty list of points, called *pending set PS(t)*, which keeps references to points from other hash buckets that may have their NN in that tile. When a pair of buckets $<H_A, H_B>$ is processed and a border point $a_i \in H_A$ is found (i) it is inserted into the pending list and (ii) a reference to it is inserted into the pending set *PS(t)* of each tile t for which $mindist(a_i,t) < NNdist(a_i,H_B)$. When a future hash bucket from B that contains t is loaded, all points in $PS(t)$ will seek a potentially closer neighbor there. If at some stage no pending set is pointing to a point, then its actual NN has been found and it is removed from the pending list.

Consider again the example of Figure 5a and assume that the current bucket pair to be processed includes t_1, but not t_2, t_3 and t_4. The nearest neighbors of all points are computed in t_1: NN(a_1, t_1) = b_1, NN(a_2, t_1) = b_2, and NN(a_3, t_1) = b_1. Since NN(a_2,t_1) = b_2 and $mindist(a_2, t_2) < dist(a_2, b_2)$, a_2 is inserted in the pending list and in $PS(t_2)$. For the same reason, a_3 is also inserted into the list and in $PS(t_2)$, $PS(t_3)$ and $PS(t_4)$, because a neighbor closer than b_1 could be in any of these three tiles. Later, when t_2 is processed, the actual NN of a_2 (i.e., b_3) will be discovered. Figure 5b shows the pending list and sets after processing t_1.

A point may enter the pending set of a tile after the tile's bucket has been processed. Assume that bucket H_B containing t_1 is already processed, and t_3 is the current tile. The NN of a_4 in t_3 (point b_4) is found to be further than t_1. If H_B is still in memory, we immediately search for the NN of a_4 there. However, if H_B has been evicted, a_4 is inserted into $PS(t_1)$. Therefore, after processing all bucket pairs, there may still be non-empty pending sets. The corresponding buckets are then loaded in a second pass; in the worst case, B will be read twice during the matching phase of the algorithm, assuming that the pending list and sets fit in memory.

4.2 Optimization of HANN

The computational cost of HANN can be minimized by some optimization techniques. When loading buckets $<H_A, H_B>$, their contents are hashed in memory according to the tiles contained in them. Thus, ANN(H_A, H_B) is broken to multiple problems, one for each tile of the bucket. These problems have much smaller size and processing cost. We also use the CPU optimization techniques of BNN described in section 3.2 to decrease the quadratic number of distance computations.

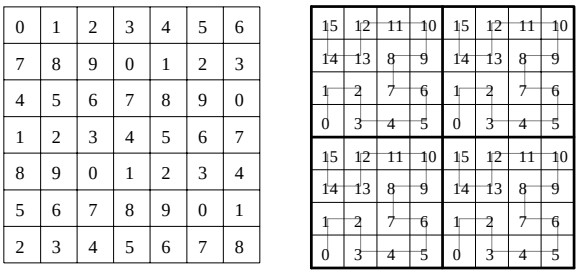
In order to reduce the I/O cost of HANN we need to minimize the number of page accesses required by the 2nd pass of the algorithm. In other words, we have to minimize the number of buckets from B that need to be loaded twice. A straightforward policy is to keep in memory pages from B as long as possible. If the memory is large enough to fit many partitions, border points whose potential NNs are in previous buckets may be processed immediately. Therefore, it is important to process buckets in an order such that the tiles in memory are close to each other with high probability. Consider Figure 5a and assume that each tile is a hash bucket. If we process t_1 first and keep its contents from B in memory while processing tiles t_2, t_3, and t_4, we can be sure that this tile will never be needed again in the future, since no points from tiles other than t_2, t_3, and t_4 can affect its pending set.

If the buckets contain multiple tiles, such a schedule is difficult to achieve, unless the neighbors of all tiles in a specific bucket belong to a small number of *neighbor* buckets. Then HANN can process the pairs $<H_A, H_B>$ in an order such that part of the neighbors of the current bucket H_B are processed immediately before H_B (and thus they are in memory), and the rest immediately after H_B. In this way: (i) border points of H_B close to tiles

processed before will find their NNs in memory, (ii) border points of H_B close to tiles not processed yet will find their NNs into the buckets processed immediately after H_B and memory will be freed.

The (round-robin) tiling scheme, described in section 2.2 for PBSM, has a nice tile neighborhood property. Assume that the number of tiles is $n \times n$ and let m be the number of buckets. If a tile belongs to bucket b, the tile on its left belongs to bucket b-1 mod m, the one on its right to bucket b+1 mod m, the tile above to bucket b-n mod m, and the one below to bucket b+n mod m. Thus, the neighbor buckets of b are fixed for each tile in b. Figure 6a illustrates this tiling scheme with n=7 and m=10. The numbers correspond to bucket ids. Observe that all tiles belonging to a partition have the same neighbors at the same directions. Thus, if the neighbor buckets of the currently processed pair are loaded either just before or just after it, the data from B that need to be read during the 2nd pass will be minimized.

However, defining a good bucket ordering on the round-robin tiling is hard. Therefore we propose an alternative tiling scheme. The space is divided using large regular windows, called *supertiles*, of m tiles each. Thus, there are $\lceil n \times n/m \rceil$ supertiles. The m tiles in each of them are ordered according to their Hilbert value with respect to the center of the supertile and assigned to partitions in this order. Figure 6b shows an example where n=8 and m=16. The hash buckets during the matching phase of the algorithm are also loaded in this (Hilbert) order. This *Hilbert tiling* has several benefits. First, the buckets have the same neighbors in all tiles; the nice property of round-robin tiling is preserved. Second, the order in which buckets are visited preserves spatial locality. Finally, the tiles of a corresponding bucket are scattered at different areas of the space and points are evenly distributed; skewed data are hashed effectively.



(a) round-robin tiling (b) Hilbert tiling

Figure 6: Tiling schemes for HANN

If the regular grid is very fine or the data distribution is very skewed, the pending list and sets can grow larger than the available memory. In this case, we first perform a cache clean-up by removing points from pending sets, for which the NN has already been found and their references are still in some lists. In Figure 5, assume that t_3 is processed after t_1 and the NN of a_3 is found in t_3. The normal lazy policy of HANN leaves a_3 in $PS(t_2)$ and $PS(t_4)$. Later, when these tiles are processed, updating a_3 will be found unnecessary. However, if HANN runs out of memory a cache clean-up function immediately visits the pending list and sets and removes redundant entries. Notice that an eager policy which performs cache clean-up every time a NN is updated is expensive (the pending sets need to be scanned very often). If after the clean-up there is still not enough memory, the pending sets of tiles that already have been processed are flushed to disk (since they will be needed only during the second pass) and memory is freed. The same is done for points in the pending list that seek their NN only in such tiles. In general, these techniques have high cost, so it is important to use a good tiling scheme and bucket ordering in order to avoid them.

A challenging problem is to choose an appropriate number of tiles. If a very fine grid is used, then the probability that the points are evenly distributed to buckets increases. On the other hand, the number of border points also increases, and so do the chances that a bucket needs to be reloaded at a second pass. Therefore there is a trade-off in the choice of the grid. In our experimental instances, the best results of HANN are achieved when a small number of tiles (in the order of 10) correspond to each partition. The number of partitions is such that the expected buckets from B that fit in memory are between 5-10. Of course this number is also constrained by the available memory (at most M-1 buckets can be defined if M is the number of memory pages).

Finally, we need to comment on how we handle cases where tiles are empty (in A or B). Obviously empty tiles in A can be handled trivially. On the other hand, empty tiles in B need special consideration. During hashing, we construct a memory-resident bitmap indicating the empty tiles in B. Consider a tile t containing points from A, which is empty in B. Instead of putting all these points into multiple pending sets, we keep them in memory and wait for one from the closest non-empty tiles from B to be loaded in order to handle them then. The closest non-empty tiles can be easily determined from the bitmap and the encoding of the tile. We also use the bitmap to avoid assigning points to pending sets of empty tiles. In the example of Figure 5, if t_4 is empty in B, we need not put a_3 in $PS(t_4)$.

5 Experimental Evaluation

In this section we study the performance of the methods proposed in sections 3 and 4, under various conditions. We also evaluate an implementation of the best (to our knowledge) CP-based ANN algorithm from previous work [HS98], described in section 2.4. All methods were implemented in C++ and the experiments were executed using a PC with a Pentium III 733MHz processor, running Windows NT. The page and R*-tree

node size is set to 4K. Each leaf node entry contains the coordinates of a point (in two double precision numbers) and an object id, summing to 20 bytes. Thus, the capacity of a leaf node is 204. Unless otherwise stated in all problem instances we set the size of the LRU buffer to 512K.

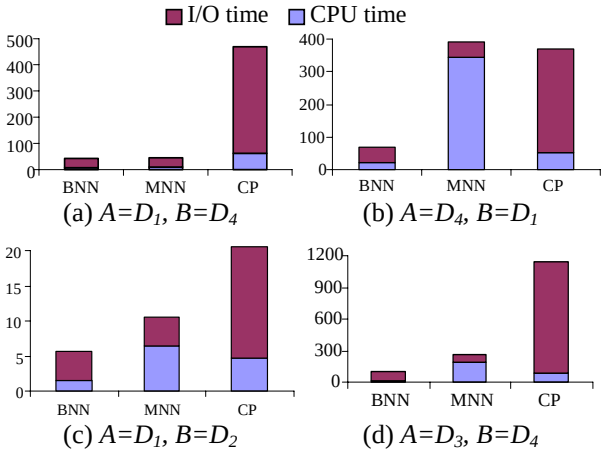
For the experiments we employed four datasets [Map] representing different layers of North America's map, described in Table 2. Datasets containing line segments were transformed to point datasets, by taking the middle point of each segment. We also used uniform datasets whenever we needed to test algorithmic performance based on parameters for which real datasets were not available.

Dataset	Cardinality	Description
D_1	9,203	Cultural landmarks
D_2	24,493	Populated places
D_3	191,637	Railroad segments
D_4	569,120	Road segments

Table 2: Description of real datasets

5.1 Experiments with indexed datasets

In the first set of experiments we compare the performance of index-based ANN algorithms. In order to include the CP-based method, we assume that both datasets are indexed. Figure 7 shows the response time of all methods (split into I/O and CPU cost) for four ANN queries, representing different problem size cases; in the first query A is small and B is large, in the second query A is large and B is small, and in the last two cases both A and B have sizes in the same order, i.e., they are both small or large.



(a) $A=D_1$, $B=D_4$ (b) $A=D_4$, $B=D_1$

(c) $A=D_1$, $B=D_2$ (d) $A=D_3$, $B=D_4$

Figure 7: Response time (sec) for indexed data

In all cases BNN is the best method, since it retains the low I/O cost of MNN and at the same time reduces the distance computations. CP is clearly inappropriate for ANN queries; it is outperformed by both MNN and BNN in all cases, except for case (b), where the number of NN queries performed by MNN is huge, with high computational cost. A comparison between case (c) and case (d) reveals that the improvement of BNN over MNN is independent of the problem size. This observation is confirmed by subsequent experiments.

On the other hand, the I/O cost of MNN is nearly optimal. The algorithm incurs at most 10% more I/Os than the total number of pages in trees R_A, R_B. BNN has marginally higher I/O cost, which is by far compensated by the large computational savings. Only in the first case both MNN and BNN have similar cost. We observed that in this case BNN reduces to MNN; each G_A has 3 points on the average. A is very sparse compared to B and grouping many points results in low I/O performance. The adaptive nature of BNN predicts this and chooses small point groups. We validated the importance of adaptive grouping by rerunning the experiment for $A=D_1$, $B=D_4$, and comparing BNN with another version of the algorithm that selects as (intuitive) groups the leaf nodes N_A of R_A. Figure 8 shows that the non-adaptive version has indeed high I/O cost; the leaf nodes of R_A have very large MBRs compared to the leaf nodes of R_B and a large part of R_B must be loaded.

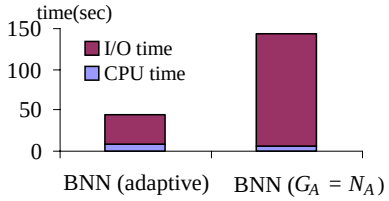


Figure 8: Performance of BNN versions ($A=D_1$, $B=D_4$)

In order to test the robustness of BNN to the available memory, we executed the most expensive query of Figure 7d varying the memory buffer size. As shown in Figure 9, MNN and BNN are not sensitive to the memory size, since their efficiency is based on the locality of two consecutive NN or BNN queries. A small buffer of 128K suffices to maximize the probability that the access path of a query is in memory. On the other hand, CP accesses an excessive number of node pairs and the size of the buffer affects the I/O cost. Even when the datasets are relatively small and the I/O cost is reduced, CP is outperformed by BNN, since the latter has much lower computational cost (see Figure 7c).

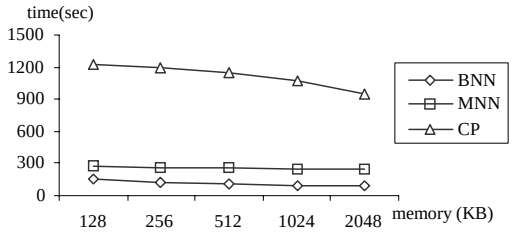


Figure 9: ANN($A=D_3$, $B=D_4$) varying buffer size

We performed another experiment to test the scalability of BNN, using synthetic datasets of uniform points. In the experimental instances the cardinality of both A and

B was the same and varied from 10^4 to 10^6 points. Figure 10 shows the response time of BNN and MNN. Observe that BNN is consistently around 4 times faster than MNN.

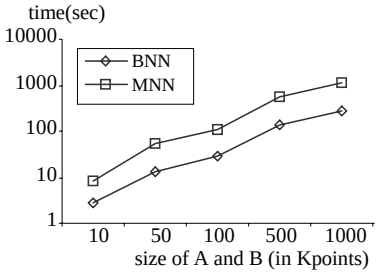
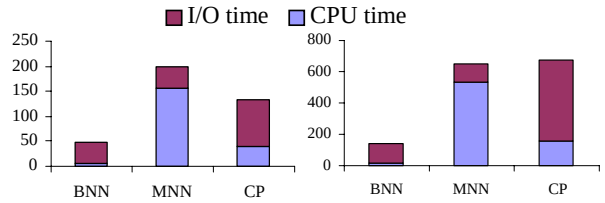


Figure 10: Scalability of BNN

Finally, we compare the algorithms for the case where A=B. This query is frequent in clustering applications, where all nearest neighbors and their distances need to be identified in a single dataset. We used the largest datasets D_3 and D_4 and executed BNN, MNN and CP, after adapting them in order not to report a point as the nearest neighbor of itself. Figure 11 shows the response time of the algorithms. Again BNN outperforms MNN and CP by far. Interestingly, for ANN(D_3, D_3), MNN is slower than CP, due to its excessive CPU cost. Notice that results of the small datasets are meaningless since they fit in memory.



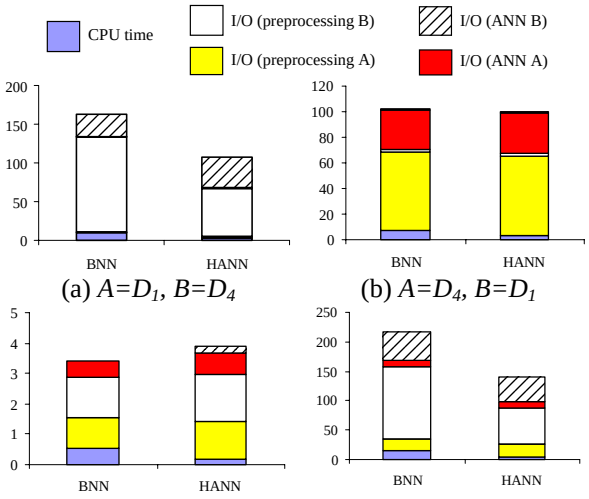
(a) A=D_3 (b) A=D_4

Figure 11: Response time (sec) for ANN(A,A) queries

We have also performed experiments assuming that A is not indexed, in which case CP is not applicable. For MNN and BNN, the only difference is the Hilbert sorting, which is the same for both methods. Beyond this, the cost is similar to that shown in Figure 7. We omit the plots, since they essentially do not contain any additional information.

5.2 Experiments with non-indexed datasets

In order to test the efficiency of HANN, we implemented another method that builds R_B on-the-fly and applies BNN, the most efficient index-based ANN algorithm. For bulk-loading R-trees we use sort-tile-recursive [LEL97], an effective algorithm that creates leaf nodes with small overlap. Figure 12 shows the response time of the algorithms for the four ANN queries of Figure 7. The costs are broken to five parts for easier analysis; the CPU cost, the I/O cost of preprocessing (i.e., sorting or hashing) each dataset, and the cost for ANN (i.e., reading the datasets during BNN or the match phase of HANN). Observe that both methods are I/O bound, due to the multiple passes over the data (for bulk loading and hashing). The relative performance of the algorithms is different in the various cases, so we will interpret them individually.



(a) A=D_1, B=D_4 (b) A=D_4, B=D_1

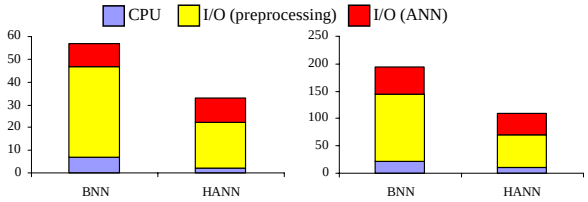
(c) A=D_1, B=D_2 (d) A=D_3, B=D_4

Figure 12: Response time (sec) for non-indexed data

First, consider cases (a) and (d), where B is much larger than the memory buffer. Bulk loading R_B requires two passes (2 reads + 2 writes). On the other hand, HANN hashes B in only one pass. In these cases, the cost for ANN is similar for both algorithms; BNN accesses 90%-150% of the pages from R_B and HANN reloads on the average around 25% of the hash buckets from B during the second pass. In case (b), A is much larger than B, which fits in memory. The performance of the algorithms is then almost the same. R_B is constructed by reading B once and remains in memory, and so are the hash buckets H_B during HANN. Sorting and hashing A have similar cost, because we combine the second pass of externally sorting A with the application of BNN. In (c) set B fits in memory, as well, so BNN has minimal cost. However, in this case the hash buckets of HANN occupy more space than the original datasets, since many of them are half-full after hashing. As a result, few pages of B are flushed to disk and loaded again, and HANN is slightly more expensive than BNN. HANN is computationally cheaper than BNN in all cases because the tiles in HANN contain more objects than the groups in BNN and the CPU optimization techniques are more effective.

We also compared BNN and HANN for the case where A=B. Figure 13 shows the performance of the algorithms on datasets D_3 and D_4. As expected, HANN is faster than BNN, since the query dataset does not fit in memory in either case. In general, we expect the hash-based algorithm to do better than bulk-loading + BNN, in problems where B is much larger than the available memory. However, this does not decrease the value of BNN, which is a simple, easy to implement and very efficient in the presence of spatial indexes. We also observed that BNN has robust behavior for special

cases with skewed data distributions, e.g., when A covers different part of the space than B. In this case HANN generates large pending sets, which may not fit in memory during its execution.



(a) $A=D_3$ (b) $A=D_4$

Figure 13: Response time(sec) for ANN(A,A) queries

6 Conclusions

This paper studies ANN queries in spatial databases. Multiple Nearest Neighbor (MNN) and Batched Nearest Neighbor (BNN) presume the existence of an R-tree on the inner dataset B, and take advantage of the structure to accelerate search. These approaches are up to an order of magnitude faster than CP-based methods. BNN is more efficient than MNN, since (i) it retains its low I/O cost by applying consecutive NN queries with spatial locality (ii) it significantly reduces the computational cost by minimizing the number of NN queries. If B is not indexed, we compare the straightforward approach of bulk-loading R_B and applying BNN with HANN, an alternative approach based on spatial hashing. The hash-based algorithm is more efficient for large problems, where building the tree requires multiples passes over B.

It would be interesting to see how ANN evaluation methods scale with dimensionality. R-trees are not appropriate for high-dimensional data, but there are structures with similar properties [SYUK00], which scale well with dimensionality. Our index-based methods can be easily employed with these structures. On the other hand, HANN is not expected to perform well for high dimensional data, since many tiles will be empty and the expected number of border points increases fast with dimensionality.

Acknowledgements

This work was supported by grants HKUST 6180/03E and HKU 7149/03E from Hong Kong RGC.

References

[APR+98] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, J.S. Vitter, Scalable Sweeping-Based Spatial Join, VLDB, 1998.

[AY01] C. C. Aggarwal, P. S. Yu, Outlier Detection for High Dimensional Data, SIGMOD, 2001.

[BBKK97] S. Berchtold, C. Böhm, D. A. Keim, H.P Kriegel, A Cost Model For Nearest Neighbor Search in High-Dimensional Data Space, PODS, 1997.

[BEKS00] B. Braumuller, M. Ester, H. Kriegel, J. Sander, Efficiently Supporting Multiple Similarity Queries for Mining in Metric Databases, ICDE, 2000.

[Bia69] T. Bially, Space-Filling Curves: Their Generation and Their Application to Bandwidth Reduction, IEEE TIT 15(6): 658-664, 1969.

[BK02] C. Böhm, F. Krebs, High Performance Data Mining Using the Nearest Neighbor Join, ICDM, 2002.

[BKS93] T. Brinkhoff, H.P. Kriegel, B. Seeger, Efficient Processing of Spatial Joins Using R-trees, SIGMOD, 1993.

[BKSS90] N. Beckmann, H.P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an Efficient and Robust Access Method for Points and Rectangles, SIGMOD, 1990.

[Cla83] K. Clarkson. Fast Algorithms for the All-nearest-neighbors Problem, FOCS, 1983.

[CMTV00] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Closest Pair Queries in Spatial Databases, SIGMOD, 2000.

[CMTV01] A. Corral, Y. Manolopoulos, Y. Theodoridis, M. Vassilakopoulos, Algorithms for Processing Closest Pair Queries in Spatial Databases, www-de.csd.auth.gr/publications.html), 2001.

[Gut84] A. Guttman, R-trees: A Dynamic Index Structure for Spatial Searching, SIGMOD, 1984.

[GTVV93] M.T. Goodrich, J.J. Tsay, D.E. Vengroff, and S. Vitter, External Memory Computational Geometry, FOCS, 1993.

[HS98] G. Hjaltason, H. Samet, Incremental Distance Join Algorithms for Spatial Databases, SIGMOD, 1998.

[JMF99] A. Jain, M. Murthy, P. Flynn, Data Clustering: A Review, ACM Computing Surveys, 31(3): 264-323, 1999.

[LEL97] S. T. Leutenegger, J. M. Edgington, M. A. Lopez, STR: A Simple and Efficient Algorithm for R-Tree Packing, ICDE, 1997.

[LR94] M.L. Lo, C.V. Ravishankar, Spatial Joins Using Seeded Trees, SIGMOD, 1994.

[Map] http://www.maproom.psu.edu/dcw.

[MP99] N. Mamoulis, D. Papadias, Integration of Spatial Join Algorithms for Processing Multiple Inputs, SIGMOD, 1999.

[NO97] K. Nakano, S. Olariu, An Optimal Algorithm for the Angle-Restricted All Nearest Neighbor Problem on the Reconfigurable Mesh, with Applications, IEEE TPDS 8(9): 983-990, 1997.

[NTM01] A. Nanopoulos, Y. Theodoridis, Y. Manolopoulos, C2P: Clustering based on Closest Pairs, VLDB, 2001.

[PD96] J.M. Patel, D.J. DeWitt, Partition Based Spatial-Merge Join, SIGMOD, 1996.

[PM97] A. Papadopoulos, Y. Manolopoulos, Performance of Nearest Neighbor Queries in R-Trees, ICDT, 1997.

[PS85] F. Preparata, M. Shamos, Computational Geometry, Springer, 1985.

[RKV95] N. Roussopoulos, F. Kelley, F. Vincent, Nearest Neighbour Queries, SIGMOD, 1995.

[SML00] H. Shin, B. Moon, S. Lee, Adaptive Multi-Stage Distance Join Processing, SIGMOD, 2000.

[SYUK00] Y. Sakurai, M. Yoshikawa, S. Uemura, H. Kojima, The A-tree: An Index Structure for High-Dimensional Spaces Using Relative Approximation, VLDB, 2000.