Range Nearest-Neighbor Query

Haibo Hu and Dik Lun Lee

Abstract—A range nearest-neighbor (RNN) query retrieves the nearest neighbor (NN) for every point in a range. It is a natural generalization of point and continuous nearest-neighbor queries and has many applications. In this paper, we consider the ranges as (hyper)rectangles and propose efficient in-memory processing and secondary memory pruning techniques for RNN queries in both 2D and high-dimensional spaces. These techniques are generalized for kRNN queries, which return the k nearest neighbors for every point in the range. In addition, we devise an auxiliary solution-based index EXO-tree to speed up any type of NN query. EXO-tree is orthogonal to any existing NN processing algorithm and, thus, can be transparently integrated. An extensive empirical study was conducted to evaluate the CPU and I/O performance of these techniques, and the study showed that they are efficient and robust under various data sets, query ranges, numbers of nearest neighbors, dimensions, and cache sizes.

Index Terms—Spatial database, nearest-neighbor search.

1 Introduction

Nearest-neighbor (NN) query is an important query type supported by most spatial databases [1], [2], [3], [4]. Traditional NN queries require input as points from which the NNs are computed. Thus, they are also called point nearest-neighbor (PNN) queries. Although Tao et al. [5] extended PNN queries to "continuous nearest-neighbor" (CNN) queries, which retrieve the NNs for all points on a line segment, the query input is still limited to a 1-dimensional line. In this paper, we propose a more general query type—"range nearest-neighbor"(RNN) queries. Given a d-dimensional data set, an RNN query retrieves the nearest neighbors for every point in a d-dimensional hyperrectangle. Such a generalization eliminates the dimensionality limitation on the query input. RNN queries have many applications:

1. In mobile environments, users do not have the accurate knowledge about their locations to specify the query points because all location identification methods have errors. Even if they have such knowledge, they may not want to expose these locations to the service providers for privacy reasons. RNN queries address these issues by allowing users to specify ranges rather than points for NN queries. They are particularly appealing to the large number of 2G/3G mobile subscribers whose devices are incapable of pinpointing locations. While these devices cannot support conventional NN queries, they can issue RNN queries through text messages such as "find the nearest hotels to the City Park."
2. A user may continuously ask for nearest neighbors while moving around. It is inefficient to submit many PNN queries individually to the server. A better alternative is to submit a single RNN query around the current location to fetch all possible nearest neighbors for this area. Any PNN query issued in this area is then processed locally by a nearest-neighbor search in the prefetched set, saving both computation and communication costs.
3. The above case can be generalized to the situation in which these PNN queries are submitted from different users. Processing them individually is inefficient because spatially adjacent PNN queries often access the same R-tree nodes. It is more efficient to group and process them in a batch by first issuing an RNN query whose range is the bounding box of all the PNN query points and then resolving the PNN results within the RNN results.

This paper focuses on RNN query processing techniques. In general, processing an NN query on a spatial index (an R-tree, for example) involves two interleaving phases: secondary memory pruning of distant index nodes and in-memory computation of the nearest neighbors. For the first phase, we develop efficient pruning heuristics for state-of-the-art NN searching paradigms such as depth-first search (DFS) [6] and best-first search (BFS) [7]. The second phase is trivial for PNN queries: The distances between all the objects in a leaf index node and the query point are calculated and the object with the shortest distance is recorded as the PNN candidate. However, this cannot be applied to RNN queries since the number of query points in an RNN query is infinite. Therefore, we propose new algorithms based on either planar geometry for two-dimensional spaces or linear programming for high-dimensional spaces. We then extend both the secondary memory and in-memory techniques to kRNN queries, which retrieve the k nearest neighbors for every point in the range. In addition to these techniques, we propose a CNN-based algorithm for two-dimensional RNN queries.

Another contribution of this paper is a solution-based auxiliary index, called EXO-tree. It is designed to accelerate all types of NN (PNN, CNN, and RNN) searches. The basic idea is that, for each index node, EXO-tree indexes some

• *The authors are with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Kowloon, Hong Kong. E-mail: {haibo, dlee}@cs.ust.hk.*

Manuscript received 23 Dec. 2004; revised 22 May 2005; accepted 12 July 2005; published online 18 Nov. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0525-1204.

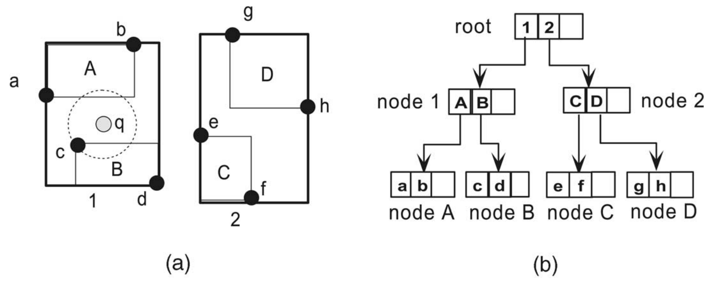


Fig. 1. An example of a nearest-neighbor query on an R-tree index. (a) Objects placement and (b) corresponding R-tree.

"valuable" objects to avoid visiting this node when we process NN queries. EXO-tree is built on top of the primary spatial index (for example, an R-tree) and is separately stored. Therefore, it does not affect the processing of other types of queries, such as range queries or spatial joins. EXO-tree is also orthogonal to other NN processing techniques. Thus, it can be incorporated into the aforementioned secondary memory pruning techniques to further speed up the RNN search.

The remainder of the paper is organized as follows: Section 2 reviews DFS, BFS, CNN, and some specialized NN search indexes. Section 3 presents the formal definition of RNN queries. In Sections 4 and 5, various in-memory and secondary memory RNN (and, more generally, kRNN) processing techniques are proposed. The EXO-tree is proposed in Section 6. The intensive experimental results, in terms of I/O and CPU costs, are analyzed in Section 7 to justify all these techniques. Finally, we suggest some directions for future research.

2 Related Work

2.1 DFS and BFS Paradigms

Given a hierarchical spatial index, for example, R-tree [8] and its variants [9], [10] depth-first search (DFS) [6], and best-first search (BFS) [7] are the most common branch-and-bound paradigms for processing nearest-neighbor queries. DFS recursively visits the index nodes to search for nearest-neighbor candidates. More specifically, after visiting a node, DFS schedules visits to its child nodes in a certain order. The order is determined by sorting the nodes according to their $MINDIST$ distances, which is the minimum distance between the query point q and the node's minimum bounding rectangle (MBR). When a leaf node is visited, the objects are retrieved and the nearest-neighbor candidate is updated. Fig. 1a illustrates an example: $\mathrm{a}, \mathrm{b}, \ldots, \mathrm{h}$ are spatial objects and q is the query point. Fig. 1b shows the corresponding R-tree, including the root, intermediate nodes 1 and 2 and leaf nodes A, B, C, and D. Starting from the root, DFS visits node 1 (because its $MINDIST = 0$) and then node B (because $B.MINDIST < A.MINDIST$), and finally obtains object c as the candidate NN. In subsequent searches, DFS can avoid visiting nodes whose $MINDIST$ is greater than $dist(c, q)$, the distance between the candidate NN, c, and q. In this example, DFS avoids visiting nodes C, D, and 2 because their MBRs are completely outside the dotted circle. Only node A is not pruned, but as the visit to A does not yield a better candidate than c, c is the final result.

BFS uses a priority queue to store entries to be explored during the search. The entries are sorted by their $MINDIST$ distances. BFS pops up the top entry in the queue, pushes its child entries into the queue, and then repeats the process. When a leaf entry (i.e., an entry in a leaf node) is popped, the corresponding object becomes the nearest neighbor. In the example shown in Fig. 1, BFS first pushes the root into the queue. After the root is popped, BFS pushes nodes 1 and 2 into the queue. Then, node 1 is popped and A, B are pushed. Then, B is popped and leaf entries, that is, objects c and d, are pushed. Finally, object c is popped as the nearest neighbor.

2.2 Continuous Nearest Neighbor

Song et al. first proposed the notion of "continuous nearest neighbor" (CNN) queries in [11]. A CNN query searches the nearest neighbors for a moving object. More specifically, it searches the nearest neighbors for all points on a line segment. To process a CNN query, Song proposed repeatedly issuing NN queries on some sample points on the line. In [5], Tao et al. elaborated on the CNN processing algorithm. They proved that the line segment contains a set of "split points,"each of which has two equally nearest objects and that these objects comprise the CNN set. Thus, the CNN problem is equivalent to finding and maintaining the split points when traversing the R-tree. To prune unnecessary node accesses, they proposed three heuristics:

1. prune nodes whose $MINDIST > SL_{MAXD}$, the maximum distance between a split point and its NN,
2. prune nodes whose minimum distance to every split point p is greater than the distance between p and p's NN, and
3. sort nodes to be visited by their $MINDIST$ values.

Tao et al. applied these heuristics to both DFS and BFS paradigms and further generalized the algorithms to answer kCNN queries, which retrieves the k nearest neighbors for all points on a line segment.

2.3 Indexes for Nearest-Neighbor Search

Some dedicated spatial indexes have been proposed in the literature to speed up NN search, especially for high-dimensional data set. SS-tree [12] and SR-tree [13] are early attempts of this kind. SS-tree employs bounding spheres instead of bounding rectangles as the shape of an index node. The shape of an SR-tree node, on the other hand, is the intersection of a bounding sphere and a bounding

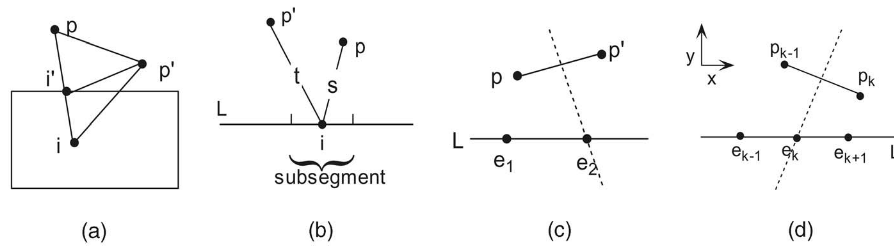


Fig. 2. Proofs of Lemmas 1, 2, 3, and 4. (a) Lemma 1, (b) Lemma 2, (c) Lemma 3, and (d) Lemma 4.

rectangle. Such a shape results in a better subdivision of the space and more disjoint nodes. Subsequent research led to more sophisticated NN indexes. Kim et al. clustered the objects and encapsulated the clusters into geometrically regular shapes [14]. As these shapes are indexed, NN queries will search only a portion of the clusters. In [15], Yu et al. also adopted clustering. In each cluster, they chose a reference point and indexed all objects' distances to this point (called *iDistance*) by a B^+-tree.

There are relatively few solution-based NN indexes. Berchtold et al. approximated the solution space of all NN queries, that is, the Voronoi cells for high-dimensional data set [16], [17]. The objective is to avoid computing the exact Voronoi diagram for a large data set, which is forbiddingly costly in terms of CPU and memory. Their approximation is based on some regular geometric shapes and their unions. Zheng et al. more recently proposed a grid-partition index to approximate these Voronoi cells for wireless environments [18]. The grid index speeds up the searching as the initial NN candidate can be located at constant time. However, both indexes only support 1NN queries, which is the common drawback of a solution-based index.

3 Range Nearest-Neighbor Query

We formally define the range nearest-neighbor query as follows:

Definition 1. *Given a data set in the d-dimensional space $\mathcal{R}^d$, the set of* ***range nearest neighbors*** *(RNN) for a hyperrectangle $\Omega \subset \mathcal{R}^d$ (boundary inclusive), denoted as $RNN(\Omega)$, is defined as the set of the nearest neighbors (NN) for every point in Ω. That is,*

$$RNN(\Omega) = \{NN(p) | p \in \Omega\}. \quad (1)$$

$NN(p)$ denotes point p's nearest neighbor. Ω is called the *query range*. By definition, it is a d-dimensional hyperrectangle. When it degenerates to *zero-dimension* and *one-dimension*, RNN degenerates to PNN and CNN, respectively.

Definition 1 can be extended for k-range-nearest-neighbor (kRNN) queries. $kRNN(\Omega)$ is the union of kNNs for every point in Ω, that is, $kRNN(\Omega) = \{kNN(p) | p \in \Omega\}$.

We use *Euclidean* distance as the distance metric. This implies that each object is a point or can be represented by a point for distance measurement. As such, in the sequel, we restrict the data set to a point data set.

4 In-Memory RNN Processing Techniques

In this section, we study the in-memory RNN query processing algorithms in two-dimensional and d-dimensional spaces. The problem is a computational geometry one: Given the point data set P and the query range Ω, we need to find all RNNs in P.

According to Definition 1, any object inside Ω is an RNN of Ω since it is the NN for the same point it occupies. We call it an internal RNN. Since internal RNNs can be found by a range query, we then focus on finding external RNNs; that is, the RNNs outside Ω. The following lemma indicates that the external RNNs for Ω are exactly the same as the NNs for all points on Ω's boundary.

Lemma 1. *The necessary and sufficient condition for an object p to be an external RNN for Ω is that p is not in Ω but is the NN for at least one point on Ω's boundary.*

Proof. The necessary condition is inherent in the RNN definition. We prove the sufficient condition by contradiction. Assume p is not the NN for any point on the boundary of Ω. Since p is an RNN, p is at least the NN for one point i inside Ω. See Fig. 2a for a 2D illustration. Let i' denote the intersection point of segment $\overline{pi}$ and the boundary of Ω. Since p is not the NN of i', there must be another object p' such that $\overline{i'p'}$ is shorter than $\overline{i'p}$; that is, $|\overline{i'p'}| < |\overline{i'p}|$. Adding $|\overline{ii'}|$ on both sides of the inequality, we have $|\overline{i'p'}| + |\overline{ii'}| < |\overline{pi}|$. Since $|\overline{i'p'}| + |\overline{ii'}| \geq |\overline{p'i}|$, we get $|\overline{p'i}| < |\overline{pi}|$, which contradicts our assumption that p is the NN of i. Therefore, the sufficient condition must hold. □

4.1 The 2D Case

Lemma 1 tells us that finding the external RNNs of Ω is equivalent to finding the NNs for every point on Ω's boundary. In a 2D space, this boundary comprises four line segments, so the problem is converted to "finding NNs for all points on a line segment L. These NNs are called L's line-nearest-neighbors (LNNs).

4.1.1 LNN Search and Its Incremental Version

First, we show by the following lemma that L has only a finite number of LNNs.

Lemma 2. *Line L can be divided into a finite number of subsegments; every point in a subsegment has the same NN.*

Proof. Suppose there is a point i on L whose NN is p and p' is the second nearest neighbor of i (refer to Fig. 2b). Let s denote $|\overline{pi}|$ and t denote $|\overline{p'i}|$. Then, there is a subsegment $[i - \frac{t-s}{2}, i + \frac{t-s}{2}]$ on L such that all points on it have p as

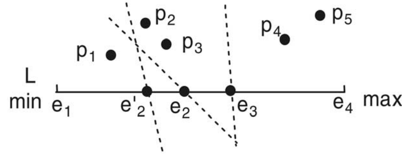


Fig. 3. LNN Search Example.

their NN. This is because their distances to p are always less than $s + \frac{t-s}{2} = \frac{t+s}{2}$, and their distances to p' or other objects are always greater than $t - \frac{t-s}{2} = \frac{t+s}{2}$. Since the length of L is finite and the length of this subsegment is $t - s$, which has a lower bound, the number of such subsegments is finite. □

Next, we show that an object can be the NN for at most one such subsegment.

Lemma 3. *If the number of subsegments is minimized; that is, if any two adjacent subsegments have different NNs, then all subsegments have different NNs.*

Proof. See Fig. 2c. Assume p is the NN for the points in subsegment $\overline{e_1e_2}$, which is the leftmost subsegment with p as its NN. Since the number of subsegments is minimized, an object p' is as close to e_2 as p is. In other words, e_2 is the intersection point of L and the perpendicular bisector of $\overline{pp'}$. The perpendicular bisector divides L into two parts: Points to the left of e_2 are nearer to p than to p', and points to the right of e_2 are nearer to p' than to p. This means that p cannot be the NN for any point to the right of e_2. Combining the assumption that p cannot be the NN for any point to the left of e_1, p is the NN only for subsegment $\overline{e_1e_2}$. □

Finally, we show that the NNs for all the subsegments are sorted by their projected values on L.

Lemma 4. *Let $e_1, e_2, \ldots$ denote the end points of all the subsegments from left to right, and $p_1, p_2, \ldots$ denote the NNs for each subsegment. Then, $\forall i < j$, $p_i.x < p_j.x$, where "$p.x$" denotes the projected value of object p on the axis parallel to L.*

Proof. We prove this lemma by contradiction. If this is not true, at least two adjacent subsegments violate this rule; without loss of generality, let us assume that they are $\overline{e_{k-1}e_k}$ and $\overline{e_ke_{k+1}}$. In other words, $p_{k-1}.x > p_k.x$ (cf. Fig. 2d). Therefore, p_k is to the left of the perpendicular bisector of $\overline{p_kp_{k-1}}$ while p_{k-1} is to the right. Then, points on L and to the left of e_k are nearer to p_{k-1} than to p_k, and vice versa. This contradicts the assumption that p_{k-1} is the NN for $\overline{e_{k-1}e_k}$ and p_k is the NN for $\overline{e_ke_{k+1}}$. □

Based on these lemmas, we then show how the LNN algorithm works through an example (see Fig. 3). The algorithm first sorts all objects by their projected values on L and names them p_1, p_2, p_3, p_4, p_5.[1] Then, it scans the objects in this order and computes the subsegment for each object. If an object has no such subsegment, it means that the object is not an LNN. The endpoints of the subsegments are obtained from the intersections of L and the perpendicular bisectors (pb) of every pair of consecutive objects. In the example, initially, $e_1 = L.min$ is the left-hand side endpoint of the subsegment for p_1. When p_2 is scanned, e_2 is obtained as the left-hand side endpoint for p_2 (and also the right-hand side endpoint for p_1). However, when p_3 is scanned, the pb of $\overline{p_2p_3}$ intersects L at a point somewhere to the left of e_2. This means that p_2 has no corresponding subsegment because either p_1 or p_3 is nearer than p_2 to any point on L. Thus, p_2 is not an LNN and is therefore removed. Now that p_1 and p_3 are consecutive objects, the left-hand side endpoint for the subsegment of p_3, namely, e'_2 is recomputed. The algorithm continues to scan p_4 and obtain e_3 as the left-hand side endpoint for the subsegment of p_4. Then, it scans p_5, but the pb of $\overline{p_4p_5}$ does not intersect L. This means that p_5 has no subsegment and as such is not an LNN. Since p_4 is the rightmost object remaining, the right-hand side endpoint of its subsegment, namely, e_4, is extended to $L.max$. In the final result, p_1, p_3, and p_4 are LNNs, with corresponding subsegments $\overline{e_1e'_2}$, $\overline{e'_2e_3}$, and $\overline{e_3e_4}$.

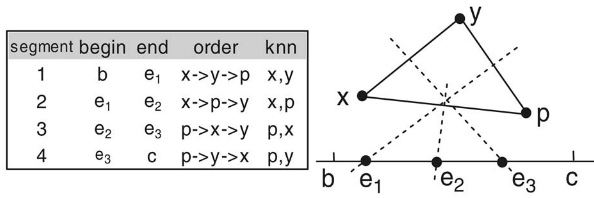
Theorem 5. *The aforementioned algorithm finds and only finds the LNNs for L.*

Proof. First, we prove that any removed object is not an LNN since when an object is removed, there must be some objects on either its immediate left or its immediate right that are closer to a point on L than the object is. Second, we prove that if an object p is not an LNN, it is always removed. If this is not the case, p must have a subsegment according to the algorithm. However, the object to the immediate left or right of p cannot be the NN for any point on this subsegment because the two pbs that determine this subsegment ensure that p is nearer to any point on this subsegment than these two objects are. No remaining objects can be the NN either. Otherwise, it would contradict Lemma 3. Therefore, we finally arrive at the fallacy that no object can be the NN for any point on this subsegment. Therefore, p must have already been removed by the algorithm. □

The LNN algorithm is required to sort the objects, which takes $O(nlogn)$ time. After that, the sequential scan of the objects takes $O(n)$ time. In addition, the algorithm recomputes previous endpoints at most n times because each recomputation corresponds to one object removal. As a result, the time complexity of the LNN algorithm is $O(nlogn)$.

A more practical use of the LNN algorithm is when a left index node is visited and the objects are added to the candidate RNN set and, thus, we need to compute the new LNNs from the existing LNNs. As such, we devise the following **incremental** version of the LNN algorithm. It first locates the new object p in the current LNN list according to p's projected value on L. It then sequentially scans p and the objects to the right of p in the list, using the standard LNN algorithm. The scan stops if the most recently scanned object has the same subsegment as before because this means that all remaining LNN objects are not affected by the insertion of p. The time complexity of the incremental LNN algorithm is $O(n)$ because it scans at most n objects and recomputes at most n endpoints.

1. In the case where two or more objects have the same projected values, the algorithm only keeps the object that is nearest to L.



segment	begin	end	order	knn
1	b	e_1	x->y->p	x,y
2	e_1	e_2	x->p->y	x,p
3	e_2	e_3	p->x->y	p,x
4	e_3	c	p->y->x	p,y

Fig. 4. p Splitting Segment bc in 2LNN.

4.1.2 Extension to kRNN

When $k > 1$, Lemmas 1 still holds. Thus, we can reduce the $kRNN$ problem to the $kLNN$ problem. Lemmas 2 and 3 are still valid, except that each k-object set, instead of a single object, has a subsegment. When an object is scanned, it affects the previous subsegments in a more complex way than merely removing the objects. Fig. 4 illustrates a 2LNN example. Before p is scanned, x and y are the 2LNN for $\overline{bc}$. When p is scanned, the perpendicular bisectors of $\overline{xy}$, $\overline{yp}$, and $\overline{px}$ are drawn and the intersection points e_1, e_2, e_3 are computed. These points divide $\overline{bc}$ into four subsegments: $\overline{be_1}$, $\overline{e_1e_2}$, $\overline{e_2e_3}$, and $\overline{e_3c}$. Each subsegment has a unique order of x, y, and p based on their distances to any point on the subsegment. For example, $\overline{be_1}$ has the order $x \rightarrow y \rightarrow p$. The order for the next subsegment $\overline{e_1e_2}$ is $x \rightarrow p \rightarrow y$, which is obtained by flipping y and p in $x \rightarrow y \rightarrow p$, the order of $\overline{be_1}$. The rationale is that the two subsegments are separated by e_1, which is the intersection point of L and the perpendicular bisector of $\overline{yp}$. Thereafter, the order for $\overline{e_2e_3}$ is $p \rightarrow x \rightarrow y$, which is obtained by flipping x and p in $x \rightarrow p \rightarrow y$, the order of $\overline{e_1e_2}$, and so on. The 2LNN set for each subsegment is simply the first two objects in the order. As a result, p splits the original subsegment $\overline{bc}$ (whose 2LNNs are $\{x, y\}$) into four subsegments: $\overline{be_1}$ ($2LNN = \{x, y\}$), $\overline{e_1e_2}$ ($2LNN = \{x, p\}$), $\overline{e_2e_3}$ ($2LNN = \{p, x\}$), and $\overline{e_3c}$ ($2LNN = \{p, y\}$). Sometimes such a split causes two or more consecutive subsegments to have the same set of kLNNs because the flipping does not occur at the top-k objects. As such, a postprocessing step is necessary to linearly scan and merge such subsegments.

During the scan of each object p, the $kLNN$ algorithm checks if p splits a preceding subsegment as above, starting from the rightmost subsegment and then leftwards. When p no longer splits a preceding subsegment, that is, when all objects in the kLNN set of this subsegment are nearer than p to any point on this subsegment, the check terminates and the algorithm continues to scan the next object. The following lemma guarantees the correctness of this termination criterion.

Lemma 6. *If p is not a kLNN for subsegment $\overline{ab}$, then p is not a kLNN for any subsegment $\overline{xy}$, where $y \leq a$.*

Proof. By mathematical induction, we only need to prove that p is not a kLNN for the subsegment to the immediate left of $\overline{ab}$, denoted as $\overline{xa}$. Suppose this is not true. Since the object order for $\overline{ab}$ is obtained by flipping the positions of two objects in the order for $\overline{xa}$ and since p cannot be the object to be flipped to a lower position in the order (because p is the rightmost object scanned so far), p must still be a kLNN for $\overline{ab}$, which contradicts our assumption. Therefore, p is not a kLNN for $\overline{xa}$. □

Each split check for p with a preceding subsegment needs to compute $k(k+1)/2$ perpendicular bisectors, and the maximum number of subsegments after splitting is also $k(k+1)/2$. Let N denote the number of subsegments in the final $kLNN$ result. Scanning p takes at most $O(k^2N)$ time. Therefore, the total time complexity of the kLNN algorithm is $O(nk^2N)$. Similar to the LNN algorithm, the incremental version of the kLNN algorithm scans the new object p and the objects to the right of p until no subsegment changes. The time complexity is $O(mk^2N)$, where m is the number of objects scanned.

4.2 The d-Dimension Case

In d-dimensional spaces ($d \geq 3$), RNNs cannot be computed using the same techniques as in Section 4.1 because the boundary of Ω is no longer composed of four line segments. Rather, it is composed of $2d$ hyperplanes of $d-1$ dimension. Since other geometric approaches such as building d-dimensional Voronoi diagrams (or even k-order Voronoi diagrams for kRNN) are costly, we propose in this section a linear programming approach to solve the d-dimensional RNN problem.

Let variable x denote a point in the d-dimensional space, and x_i denote its coordinate in the ith dimension. As such, Ω is denoted as $a_i \leq x_i \leq b_i, \forall 1 \leq i \leq d$, where a_i, b_i are the bounds. Given data set $\mathcal{D} = \{p^{(1)}, p^{(2)}, \ldots, p^{(n)}\}$, object $p^* \in \mathcal{D}$ is an RNN if and only if the following set of inequalities with regard to x can be satisfied simultaneously:

$$\begin{array}{ll} dist^2(p^*,x) - dist^2(p_1,x) \leq 0 & \sum_{i=1}^{d}[2(p_i^{(1)} - p_i^*)x_i + (p_i^*)^2 - (p_i^{(1)})^2] \leq 0 \\ dist^2(p^*,x) - dist^2(p_2,x) \leq 0 & \sum_{i=1}^{d}[2(p_i^{(2)} - p_i^*)x_i + (p_i^*)^2 - (p_i^{(2)})^2] \leq 0 \\ \ldots \Longrightarrow \ldots & \\ dist^2(p^*,x) - dist^2(p_n,x) \leq 0 & \sum_{i=1}^{d}[2(p_i^{(n)} - p_i^*)x_i + (p_i^*)^2 - (p_i^{(n)})^2] \leq 0 \\ a_i \leq x_i \leq b_i, \forall 1 \leq i \leq d & a_i \leq x_i \leq b_i, \forall 1 \leq i \leq d. \end{array} \quad (2)$$

The right-hand side derivation shows that these inequalities are linear with respect to x_i, $1 \leq i \leq d$. As such, they can be regarded as the set of linear constraints in a linear programming (LP) problem regarding the variable set x_i. We build an LP problem $\mathcal{P}$ with the objective function $C = 0$ and the same linear constraints as above. Determining whether these inequalities can be satisfied simultaneously is then equivalent to testing whether $\mathcal{P}$ has a feasible solution.

We use the classic *Simplex* or *Ellipsoid* method to solve $\mathcal{P}$. If the solver shows that the problem is feasible, p^* is an RNN, otherwise, it is not. Using an LP solver to test the feasibility does not degrade the performance since it was shown that the cost of a feasibility test for $\mathcal{P}$ is already half the cost of finding the optimal solution for $\mathcal{P}$ [19]. This approach can also be extended to kRNN: We choose $n-k$ objects from $\mathcal{D}$ (excluding p^*) and form an LP problem $\mathcal{P}$. There are $\binom{n-1}{k-1}$ possible $\mathcal{P}$s. As long as one of them is feasible, p^* is a kRNN.

Although many LP solvers such as the Simplex and Ellipsoid methods practically run in polynomial time, their performance still depends heavily on d and n (the cardinality of $\mathcal{D}$) [19]. Since d cannot be changed, we propose some heuristics to efficiently decrease n. The idea is to choose a set of "seed" objects from $\mathcal{D}$, denoted as $\mathcal{D}'$, and for each object $p^{(i)} \in \mathcal{D}$, we use $p^{(i)}$ and $\mathcal{D}'$ to form a smaller-scale LP problem $\mathcal{P}'$. If $\mathcal{P}'$ has no feasible solution, it means

that $p^{(i)}$ is not even the (k)RNN in a smaller data set, let alone the (k)RNN for $\mathcal{D}$. Therefore, we can prune $p^{(i)}$ from $\mathcal{D}$. After pruning all such $p^{(i)}$s from $\mathcal{D}$, we can generate the LP problem $\mathcal{P}(s)$ for each remaining object in $\mathcal{D}$ and test the feasibility. As the cardinality of $\mathcal{D}$ is much smaller after pruning, the computational cost is significantly reduced.

If the seed objects are the actual (k)RNNs, the preceding algorithm can prune the most number of the objects. However, this optimal seed set is unknown at the pruning stage. As such, we propose the following two heuristics to approximate it.

Heuristic 7. *MINDIST Seed Heuristic: Choose the $d + k$ nearest objects to Ω in $\mathcal{D}$. The distance metric is MINDIST.*

Heuristic 8. *Face Seed Heuristic: Choose a random point on each face of Ω and take its (k)NN as seeds.*

The first heuristic takes advantage of the fact that objects in $\mathcal{D}$ are bounded by an MBR. If the MBR does not overlap Ω, objects that are closer to Ω are more likely to be the (k)RNNs. The second heuristic finds some true (k)RNNs along the boundary. Since all $2d$ faces are considered, the seed set can include as many (k)RNNs as possible.

5 Secondary Memory RNN Pruning Techniques

In this and the next sections, we turn our focus to the database aspect. More specifically, we presume that a range query has been issued in advance to find all internal RNNs. Thus, we investigate techniques for processing external RNNs on the R-tree index and show how unnecessary index traversal can be avoided to save I/O and CPU costs. The traversal still adopts the DFS or BFS paradigms: When it accesses a leaf R-tree node, it calls the in-memory RNN algorithms (i.e., the incremental (k)LNN-based algorithm for 2D or the LP-based method for d-dimension). In this section, we first present pruning heuristics for DFS or BFS and then present a CNN-based algorithm specialized for 2D cases. In the next section, we further propose an auxiliary index called EXO-tree to speed up the processing of RNN as well as any other types of NN queries.

5.1 Pruning Techniques for DFS and BFS RNN Search

We define the *minimum distance* between two sets of points S and R, denoted as $\delta(S, R)$, as the minimum distance between any point in S and any point in R:

Definition 2. $\delta(S, R) = \min_{s,r} dist(s, r)$, *where* $s \in S$ *and* $r \in R$.

We also define the *maximum distance* between two sets of points S and R, denoted as $\Delta(S, R)$, as the maximum distance between any point in S and any point in R:

Definition 3. $\Delta(S, R) = \max_{s,r} dist(s, r)$, *where* $s \in S$ *and* $r \in R$.

We also define $\rho(S, C)$ as the *maximum distance* between any point in S and its NN in the current RNN candidate set C:

Definition 4. $\rho(S, C) = \max_s dist(s, NN(s))$, *where* $s \in S$.

Both S and R can be finite or infinite (for example, a spatial range). Under these distance metrics, we propose the following three pruning heuristics.

Heuristic 9. *An R-tree node n (n also denotes its MBR) with $\Delta(C, \Omega) < \delta(n, \Omega)$ should not be visited.*

Heuristic 10. *An R-tree node n that satisfies "for each face of Ω's boundary, $\mathcal{F}$, $\rho(C, \mathcal{F}) < \delta(n, \mathcal{F})$" should not be visited.*

Heuristic 11. *Nodes are visited in the ascending order of their minimum distances to Ω, that is, $\delta(n, \Omega)$.*

The first two heuristics essentially identify conditions under which no object in node n can be closer to any point in Ω than the current RNN candidates. The last heuristic is based on the fact that the closer an object is to Ω, the more likely it is an external RNN of Ω. In essence, the three heuristics are generalizations of the three heuristics proposed in [5]: When Ω degenerates to L, $\delta(n, \Omega)$ becomes MINDIST(n,L) and $\Delta(C, \Omega)$ becomes SL_{MAXD}. Thus, Heuristics 9 and 11 degenerate, respectively, to the first and third heuristics in [5]. Nonetheless, the second heuristics are different: The face-based pruning Heuristic 10 does not degenerate to the second heuristic in [5] which is a split-point-based pruning. Although such pruning is more fine-grained, it is only feasible for line segments. By contrast, the face-based pruning works for two or high-dimensional query inputs where the NN distribution on a face is more complicated than a set of line segments divided by split points. For the same reason, the computation of $\rho(C, \mathcal{F})$ needs to be approximated in d-dimensional spaces.[2] We can bound it by any RNN candidate p's maximum distance to $\mathcal{F}$; that is, $\Delta(p, \mathcal{F})$. To make the pruning most effective, however, we should use the lowest bound; that is, the minimum of $\Delta(p, \mathcal{F}$ values among all $p \in C$.

As for the time complexity, computing $\delta(n, \Omega)$ needs $O(d)$ time. Thus, the first heuristic runs in $O(d)$ time, assuming that $\Delta(C, \Omega)$ is always maintained whenever C is updated. Similarly, the second heuristic runs in $O(d^2)$ time because it must check all of the $2d$ faces. The third heuristic also takes $O(d)$ time to compute $\delta(n, \Omega)$. In addition, we need to count the time spent on maintaining $\Delta(C, \Omega)$ and $\rho(C, \mathcal{F})$.

Generalizing these heuristics for kRNN queries is no more complicated than literally replacing every "NN" with "kNN." For example, distance metric $\rho(C, \mathcal{F})$ becomes $\rho_k(C, \mathcal{F})$, which denotes the largest distance of any point on $\mathcal{F}$ from its kth NN in C, and to bound it in d-dimensional spaces, we need to use the "kth minimum," instead of the "minimum" $\Delta(p, \mathcal{F})$ value.

Algorithm 1 shows the complete pseudocode for the RNN algorithm in the BFS paradigm that applies these heuristics.

Algorithm 1 The Complete RNN Search Algorithm in BFS Paradigm
Input: Ω, the query range
Output: C, the set of RNNs
Procedure:

2. It is still easy in 2D spaces because $\mathcal{F}$ is simply a line segment. After the preceding in-memory LNN algorithm is executed, $\rho(C, \mathcal{F})$ is set to the maximum distance between any endpoint and its LNN.

1: Initialize C and the priority queue Q for the BFS paradigm
2: Enqueue the root node to Q
3: **while** Q is not empty **do**
4: Dequeue node n from Q
5: **if** $\Delta(C, \Omega) \geq \delta(n, \Omega)$ **then**
6: **if** $\exists$ face $\mathcal{F}$ of Ω, $\rho(C, \mathcal{F}) \geq \delta(n, \mathcal{F})$ **then**
7: **if** n is an object **then**
8: $C = C \cup n$
9: Call (the incremental version of) the in-memory RNN algorithm on C
10: **else**
11: **for** each child entry v of n **do**
12: Enqueue node v
13: return C

5.2 CNN-Based Search

Lemma 1 shows that external RNNs are exactly the set of NNs for every point on Ω's boundary. In 2D spaces, the boundary is composed of four line segments, which inspires us to exploit the continuous nearest-neighbor (CNN) algorithm in [5] to process 2D RNN queries. However, invoking the CNN algorithm four times is inefficient because it might retrieve the same R-tree nodes more than once. Such redundant accesses occur more frequently as Ω becomes smaller. Therefore, we revise the original CNN algorithm so that it traverses the R-tree only once to retrieve all the NNs for the four line segments. The changes are as follows: 1) While pruning in the original CNN algorithm is based on $MINDIST(n, l)$, where l represents one line segment, the enhanced algorithm considers the four lines segments as a whole by replacing $MINDIST(n, l)$ with $\delta(n, \Omega)$. 2) To prune more nodes, the internal RNNs are set as the initial external RNN candidate set, C. As CNN has been generalized to kCNN in [5], this approach can directly answer kRNN queries.

6 EXO-Tree

In this section, we first investigate the drawbacks of the traditional DFS and BFS paradigms. To address the problems, we propose an solution-based auxiliary index, called EXO-tree, to augment the primary index (for example, an R-tree). We then show how DFS and BFS can use it to speed up any type of NN (PNN, CNN, and especially RNN) query processing.

6.1 "Fringe Effect"

For PNN queries, DFS and BFS are efficient if the query point q resides deep inside the MBR of a leaf R-tree node n because: 1) n is the first leaf node explored. 2) When it is explored, the current NN candidate is updated, and since all of the other nodes are probably farther from q than the candidate, the algorithm can stop immediately. However, if q is close to the boundary of n or not inside any leaf node, all nodes in the vicinity might need to be visited even after n is visited because they may contain objects closer to q than the current candidate. Fig. 5 shows such an example: For q_1, only leaf node n_1 is accessed. By comparison, p_2 is the actual NN for q_2, but DFS must visit n_1, then n_3, and then n_2.

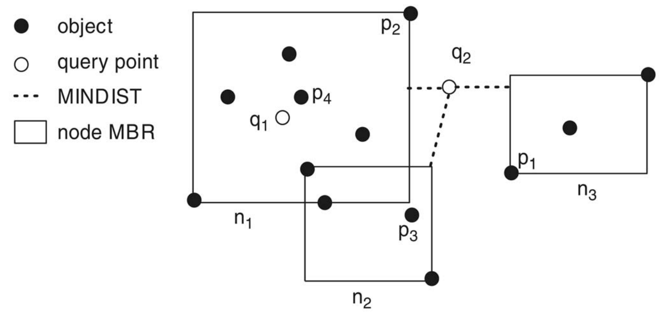


Fig. 5. Performance problems for DFS and BFS.

Although it finds p_2 when visiting n_1, it cannot avoid the subsequent visits of n_2 and n_3. BFS has the same problem, since n_2 and n_3's MINDIST to q_2 are both less than that of p_2, they precede p_2 in the priority queue and thus are accessed before p_2.

We call such a performance downgrade the "fringe effect" because it occurs when q resides in the fringe area of a node or outside any node of the same level in the R-tree. The higher the level (closer to the R-tree root) in which this problem occurs, the more dramatic the fringe effect since many more intermediate and leaf nodes are unnecessarily visited. RNN queries intensify this effect because a query point is now enlarged to a range, which increases the possibility that the neighboring nodes have NN candidates.

6.2 Basic Idea

The fringe effect is inevitable for R-tree or any partition-based spatial index. Therefore, we propose an auxiliary solution-based index to complement the primary index and reduce this effect. A key observation from Fig. 5 is that, for q_2, even though n_1, n_2, and n_3 all need to be visited, not all object entries in them need to be accessed. In general, if a query point is outside node n, only some objects in n could possibly be its NN. Based on this observation, we divide the objects in each index node n into two categories: those that can never be the NN for any query point outside n and those that can. We call them internally influential objects and externally influential objects, or **internal objects** (INOs) and **external objects** (EXOs) for short. For example, in Fig. 5, p_2 in n_1 is an EXO, but p_4 is an INO because it is not the NN for any point outside n_1 (we show in the next section why this is true). Intuitively, EXOs are located close to the boundary, and they form a (small) subset of all the objects in n.

Note that an object might be an EXO for node n but an INO for n's ancestors. Nonetheless, an EXO of n must be an EXO of any of n's descendants that contain this object. Thus, each object is associated with the highest level node in which this object is an EXO. In this way, all EXOs form a hierarchy superimposed on the primary index. Such a hierarchy on EXOs, called an EXO-tree, is stored as an auxiliary index. With this index, if n does not overlap q but still needs to be accessed without further traversing the subtree rooted at n, we only need to retrieve n's EXOs and search for q's NN among them.

6.3 Computing External Objects

To compute the EXOs for node n, the straight-forward approach is to compute the Voronoi cells for all objects in n: Those objects whose Voronoi cells are not totally inside n are EXOs. However, computing the exact Voronoi cells requires knowledge of the entire data set, so this approach is costly. We thus propose a local algorithm that only needs a linear scan of the objects in n to determine the EXOs. Efficient as it is, such localization has its own cost: The resultant EXOs are just a superset of the actual EXOs because the algorithm ignores all objects outside n.

In the same spirit of Lemma 1, the following lemma shows that to find EXOs in n, it is only necessary to find NNs for all points on n's boundary. In 2D spaces, this is the same as the LNN problem in Section 4.1.1, except that all objects are on one side of L. Therefore, the LNN algorithm and its incremental version can still be applied.

Lemma 12. *The sufficient and necessary condition for an object p to be an EXO in node n is that p must be the NN for at least one boundary point of n.*

Proof. The proof is the same as that of Lemma 1 except that now p resides inside the boundary rather than outside it as seen in Lemma 1. □

In d-dimensional spaces, to determine if object p_i is an EXO, we build an LP problem that is the same as (2), except that the last constraint is "$x_i \geq b_i$ or $x_i \leq a_i$, rather than "$a_i \leq x_i \leq b_i$. If the LP problem is feasible, p_i is an EXO. Otherwise, it is not.

If n is not a leaf node, the EXOs do not have to be computed from scratch. Instead, n's EXOs can be computed from the EXOs of n's child nodes since EXOs of higher level nodes must also be EXOs of lower level nodes.

6.4 EXO-Tree Index

The auxiliary index, called "external objects tree" (EXO-tree), is an index of the EXOs for every node in the primary index (for example, R-tree). Each EXO-tree node corresponds to a node in the primary index. However, the two nodes store different information:

1. A leaf EXO-tree node only stores the entries of external objects that belong to it, while a leaf primary index node stores the entries of all objects that belong to it.
2. Each external object entry in a leaf EXO-tree node has an additional attribute "level" that denotes the level of the top-most primary index node in which this object is still an external object.
3. In an intermediate EXO-tree node, each entry also has a "level" attribute that designates the highest "level" value of all EXOs contained in this entry.

By this design, the EXO-tree need not store redundant EXOs for any intermediate node because they must also be the EXOs of some leaf nodes. Fig. 6b is an example of an EXO-tree. A, B, a, b, c, d are R-tree nodes and $1, \ldots, 10$ are EXOs. The second digit in each entry is the level attribute for the entry. Thus, $(b, 1)$ means the top-most level of EXOs in node b is 1 (object 5). A leaf entry corresponds to an EXO. Thus, $(4, 0)$ means object 4 is an EXO only for leaf node b, and $(1, 2)$ means 1 is an EXO for the leaf node a, level 1 node A and the root node.

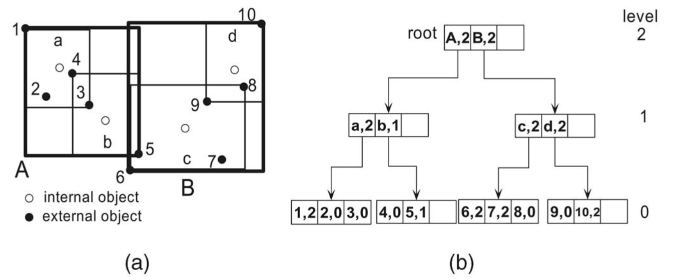


Fig. 6. Example of EXO-tree. (a) Objects placement and (b) corresponding EXO-tree.

To build the EXO-tree from the primary index, we compute all nodes' EXOs in a bottom-up fashion: the leaf nodes first, then their parents, ..., and, finally, the root node. Note that for an EXO p, its $level$ is not determined until p is no longer an EXO for some ancestor node. When the values of the $level$ for all EXOs are determined, the values of the $level$ for all intermediate node entries can be determined.

6.5 Integration with R-Tree: EXR-Tree

In this section, we consider R-tree as the primary index. The EXO-tree can be stored separately or integrated into the R-tree to avoid storing common entry information (for example, MBR and node id) twice. The integration is achieved by augmenting each node entry with a $level$ attribute required by the EXO-tree. In addition, since the leaf nodes of the EXO-tree and the R-tree are different, they are stored in separate disk pages. Such an augmented R-tree is called an *EXR-tree*. Three storage issues arise in the EXR-tree:

1. An EXO-tree leaf node contains only a small portion of the objects in the corresponding R-tree leaf node. As such, we group nearby leaf EXO-tree nodes to fit in one disk page. To minimize the I/O cost, these nodes are grouped by their Hilbert curve [20] order to reserve spatial locality.
2. For intermediate nodes, although the $level$ attribute needs only one byte, augmenting the entries of an existing R-tree might make the nodes unable to fit into disk pages. In these cases, we store the $level$ attributes of an EXR-tree node separately on disk or memory.[3]
3. So far, only the EXOs for leaf nodes are directly stored. To retrieve the EXOs for an intermediate node, we have to visit all its descendant nodes and choose EXOs whose level values are not lower than the level of this node. This is inefficient, particularly because the EXOs for higher level nodes are more frequently accessed than those for the lower level nodes. Therefore, we buffer all high level EXOs to avoid any disk access. The threshold level that

3. To illustrate how small this part of the data is, imagine a data set with 10 million objects and an R-tree with 4k-byte page size. The number of intermediate nodes is less than 1,000. As such, even if all nodes have the maximum number of entries (about 200), the $level$ data only take up 200K bytes in total.

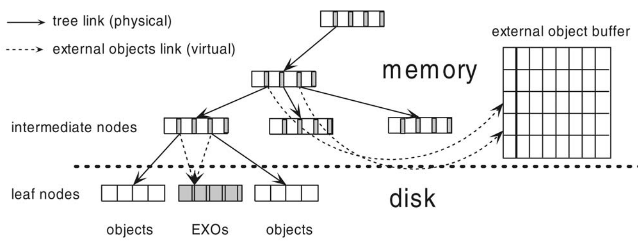


Fig. 7. EXR-tree: R-tree with EXO-tree integration.

determines "high" or "low" adapts to the available memory size.

A sketch of an EXR-tree is given in Fig. 7. All gray areas are the additional storage required for the EXR-tree besides the R-tree: The slim gray area in each entry designates the *level* attribute. Although depicted inside the entry, it can be stored outside the entry. At the leaf node level, both the R-tree nodes (in white) and the EXO-tree nodes (in gray) are stored separately. For each entry, a solid arrow denotes a link to the corresponding tree node, whereas a dotted arrow denotes a link to the external objects. In the figure, all external objects at level 1 or higher are buffered, so every entry that corresponds to a node at level 1 or higher has the dotted arrow pointing to the buffer and every remaining entry has a dotted arrow pointing to the disk page. The dotted arrows are implemented virtually and stored outside the entries by a hashing function. In contrast, the solid arrows are stored physically with the entry. The advantage of such storage organization is that the original R-tree remains unchanged: The EXR-tree is just the R-tree plus some auxiliary structures.

6.6 Processing NN Query on EXO-Trees

The EXO-tree can speed up any NN query (PNN, CNN, or RNN) for any "branch and bound" search paradigm (DFS or BFS). The improvement lies in that, if query q (which could be a point, a line, or a range) does not overlap the MBR of an entry n, even if no heuristics can prune n,[4] it is only necessary to access n's external objects, which are only a small portion of the entire set of objects in n and are probably already buffered in the memory.

More specifically, in the DFS paradigm, before the recursive call to process q on entry n, we check if q overlaps n. If it is negative, the external objects of n are retrieved instead of carrying out the recursive call. In the BFS paradigm, when n is dequeued, besides checking whether n is an object or an intermediate node, we also check if q overlaps n. If it is negative, the external objects of n, instead of n's child entries, are enqueued into the priority queue.

6.7 Handling Updates

In this section, we show how the EXO-tree handles updates; that is, object insertions, object deletions, and index node splits. Insertion or deletion of an object in a leaf node n might change the EXO set of n. We thus need to recompute the set. If the set actually changes, the update procedure is propagated to its parent. The parent then gathers all the EXOs of its child entries, recomputes its own EXO set, and propagates the update to its parent, until the EXO set ceases to change.

Splitting index node n into n_1 and n_2, on the other hand, does not invalidate the EXO set of any lower level node. The EXO set of a new node n_1 is computed from the EXO sets of the child entries in n that now belong to n_1, and so is the EXO set of n_2. Similarly, since n_1 and n_2 are the new child entries of n's parent, the parent must recompute its own EXO set. Similar to insertion/deletion, if the EXO set actually changes, the update procedure must be propagated to its parent.

7 Performance Evaluation

In this section, we evaluate the performance of all proposed RNN processing techniques. We also evaluate the performance gain of EXR-tree for PNN, CNN, and RNN queries.

7.1 Experimental Setup

The simulation testbed runs on the Win32 platform with Pentium 4 1.8G CPU, 512MB memory. The testbed consists of two 2D real data sets: "NE" contains 123,593 postal addresses in New York, Philadelphia, and Boston; "US" contains the centroids of 569,120 major road segments throughout the United States [data set]. The testbed also consists of a series of synthetic data sets "UN-xD" with increasing dimensions: UN-2D, UN-4D, UN-6D, and UN-8D. Each data set contains 100,000 objects that are uniformly and randomly distributed. All six data sets are normalized to unit (hyper)rectangles and indexed by R*-trees [10].

There are two tunable parameters in the experiments: len, the side length of Ω and mem, the available memory. We set the same length for each dimension of Ω, making it a (hyper)cube. Thus, len designates the RNN query's workload. The centroid of Ω is randomly and uniformly generated. The memory is used to cache R-tree or EXR-tree index nodes, or buffer high-level external objects of the EXR-tree.[5] mem is expressed as the percentage of the size of the R-tree. To be fair, both the R-tree and the EXR-tree apply LRU (least-recently-used) as their cache replacement policy.

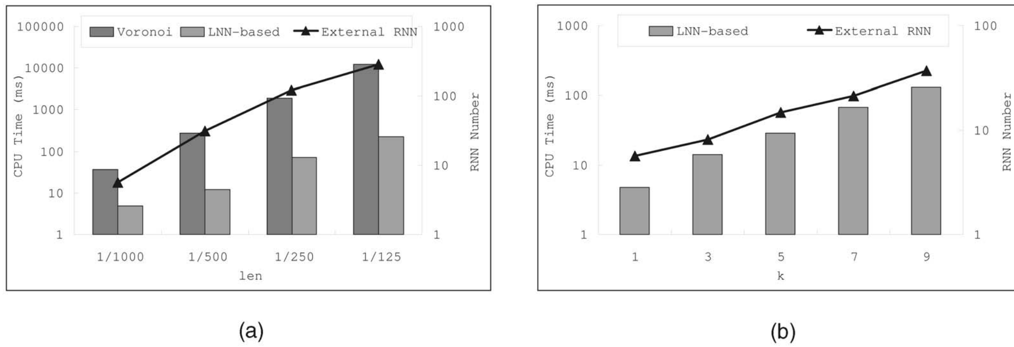
The performance metrics are CPU time and the number of accessed disk pages per query. If we need to evaluate the overall cost in terms of time, we charge 10ms for each page access, as in [5]. To obtain stable statistics, each measurement is obtained by averaging the results from 1,000 independent queries, after 200 (20 percent) queries have been run as warm-up queries.

7.2 In-Memory RNN Processing

In this section, we evaluate the performance of the LNN-based (for 2D) and LP-based (for d-dimension) algorithms. To compare with the LNN-based algorithm, we also implement the Voronoi-diagram-based algorithm using Fortune's sweepline technique [22]. We measure the CPU time dedicated to in-memory processing, which is plotted in Fig. 8a. The figure also shows the average number of external RNNs for different Ω sizes. Although both the

4. That is, the $MINDIST$ (for PNN) or $\delta(n, \Omega)$ (for RNN) distance metric is not high enough.

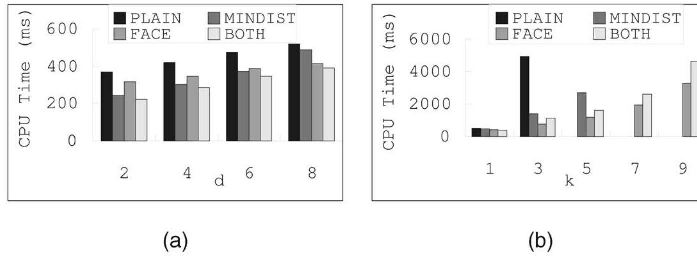
5. We do not consider the runtime memory for the algorithms, for example, the stack for DFS recursive calls and the priority queue for BFS because they are implementation-dependent.



(a)

(b)

Fig. 8. CPU Time for 2D RNN Queries (NE data set). (a) LNN-based algorithm versus Voronoi diagram and (b) LNN algorithm with different k values.



(a)

(b)

Fig. 9. CPU time for d-dimensional RNN queries (UN_xD data sets). (a) CPU time versus d. (b) CPU time versus k ($d = 8$).

LNN-based and the Voronoi-diagram-based algorithms have $O(nlogn)$ worst-time complexity, the figure shows that the former always outperforms the latter by at least an order of magnitude. We also observe that the LNN-based algorithm runs almost linearly to the number of external RNNs while the Voronoi algorithm runs exponentially. This is because the LNN algorithm can be incremental and only costs $O(n)$ time whereas the Voronoi algorithm cannot: It has to recompute the entire diagram whenever a leaf R-tree node is visited.

In the next experiment, we fix len at $1/1,000$ and measure the CPU time for the kLNN-based algorithm. Fig. 8b shows its trend with respect to k: The CPU time increases as the number of subsegments increases drastically. Nevertheless, even when $k = 9$, the CPU time is about 130ms, a reasonable cost for desktops. The experiments on len=1/500,1/250,1/125 show similar results. Therefore, we conclude that the (k)LNN-based algorithm is efficient for 2D RNN in-memory processing.

Next, we evaluate the LP-based algorithms on the UN-xD data sets ($x = 2, 4, 6, 8$). Since two heuristics (MINDIST and Face Seed) are proposed in Section 4.2, we evaluate all four combinations: PLAIN (without heuristics), MINDIST (the MINDIST heuristic), FACE (the FACE heuristic), and BOTH (both heuristics). Fig. 9a shows the CPU time for the four data sets. In general, none of these algorithms are sensitive to d since the size of the LP problem depends more on the number of objects n than d. Nonetheless, the performance does degrade due to the curse of dimensionality [23]. For the same reason, MINDIST is the most sensitive to d: As d increases, the top $d + k$ objects are less likely to become the RNNs. In contrast, FACE is less sensitive since it always chooses true RNNs as seeds. As such, BOTH achieves the best performance for all d values.

In the next experiment, we fix $d = 8$ and evaluate under different k values. k significantly affects the CPU time because it affects the number of LP problems to solve. From Fig. 9b, we observe that PLAIN is only feasible for $k \leq 3$ and that even MINDIST becomes extremely slow when $k \geq 7$. This is again due to the fact that the top $d + k$ objects are less likely to be RNNs as k increases. Thus, we do not count in PLAIN and MINDIST beyond their feasible ranges. FACE is the best algorithm here because it is the least sensitive to k. In contrast, BOTH hardly outperforms FACE, which means that the overhead of executing MINDIST outweighs its additional performance gain added to FACE. As such, we conclude that BOTH is the best for small k and FACE is the best for medium and large k.

7.3 Secondary Memory Pruning

In this section, we evaluate the performance of the secondary-memory pruning heuristics given various parameter settings. The heuristics are implemented in a BFS paradigm. In the 2D case, we also compare them with the proposed CNN-based algorithm in terms of the number of R-tree page accesses.

Figs. 10a and 10b show the results for the two real data sets. The BFS and CNN-based algorithms show similar performance, although the latter is slightly better than the former by 5 percent to 15 percent. This is expected as Tao et al. already proved that the CNN algorithm is nearly I/O optimal for NN queries on line segments [5]. As such, the close performance of BFS indicates that our pruning heuristics are still competitive even in 2D. The performance gap is even less when len increases because a larger proportion of page accesses is due to the internal RNN query.

In the next experiment, we compare the performance of kRNN queries using the same data sets (see Figs. 11 and 11b). We find that, when k increases, the gap between BFS

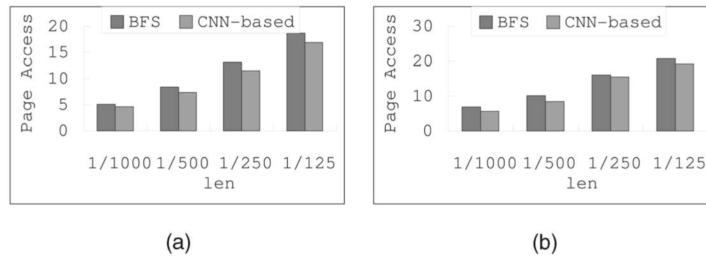


Fig. 10. Page accesses versus len for 2D. (a) NE and (b) US.

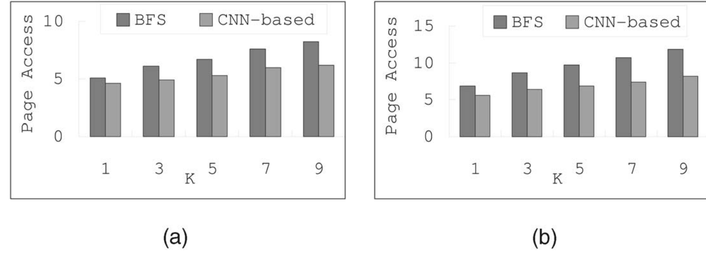


Fig. 11. Page accesses versus k for 2D. (a) NE and (b) US.

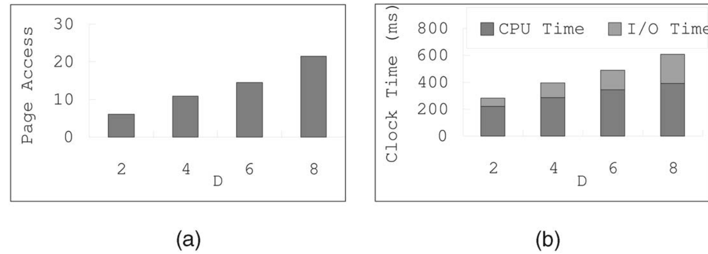


Fig. 12. BFS performance under UN-xD data sets. (a) Number of page accesses and (b) clock time.

and CNN-based algorithms increases accordingly. This is attributed to the increasing performance gap between the face-based and the split-point-based pruning methods, as the increasing number of split points makes a split-point-based pruning method still competitive for large k. Data set US reveals a larger gap than NE as the objects in US are more skewed, making the face-based pruning method even worse. But even so, CNN-based algorithm never outperforms BFS by more than 30 percent even when $k = 9$. Given the fact that BFS works for arbitrary dimensions, the performance gap in 2D is acceptable.

In the next experiment, we evaluate BFS in different dimensions using the UN-xD data set. Fig. 12a shows the number of page accesses, which increases steadily when d increases. However, it does not deteriorate drastically because the increase is attributed more to the dimensionality curse than to the pruning heuristics. Fig. 12b integrates the CPU time from Fig. 9a (using BOTH) to show the overall clock time to process an RNN query. Like in Fig. 12a, we find that the clock time increases steadily. One interesting observation is that the overall time becomes more I/O bound (less CPU bound) when d increases. This can be interpreted as follows: Although both the in-memory and secondary memory processing are cursed by the dimensionality, the former is less cursed as the FACE heuristic is independent of d.

7.4 The Effect of EXR-Tree

In this section, we evaluate the performance gain (in terms of page accesses) of EXR-tree over R-tree for all types of NN queries: PNN, CNN, and RNN. For each query type, we measure the performance of BFS with EXR-tree (denoted as BFS-EXR), with R-tree (denoted as BFS-R) and with SR-tree (denoted as BFS-SR). Since the EXR-tree takes up some memory to buffer its high-level EXOs, to make the comparison fair, we add the same amount of memory to the R-tree and SR-tree cache.

7.4.1 *EXR-Tree Construction and Maintenance*

We first evaluate the cost of constructing and maintaining an EXR-tree from an R-tree. The maintenance cost is measured under a workload consisting of 1,000 insertions followed by 1,000 deletions. Tables 1 and 2 show the space and time cost for all six data sets. From the tables, we make the following observations:

1. The storage overhead of EXR-trees is small, adding only 13 percent to 17 percent to the original R-tree.
2. The size of level 1^+ EXOs is small (tens of kilobytes for any data set), thus buffering them incurs only a small amount of memory;
3. The time cost to compute the EXR-tree is small, especially given the fact that it is a one time cost.
4. The maintenance of EXR-tree takes up 40 percent to 60 percent of the total maintenance cost, which is

TABLE 1
EXR-Tree Size

	NE	US	UN-2D	UN-4D	UN-6D	UN-8D
number of objects	123,593	569,120	100,000	100,000	100,000	100,000
number of leaf EXOs	39,800	232,507	33,921	35,681	38,522	41,453
number of other EXOs	2,783	15,036	2,323	2,497	2,656	2,910
R-tree size(MB)	3.8	18.5	3.2	3.9	4.5	5.2
EXR-tree size(MB)	4.36	21.75	3.63	4.45	5.20	6.06
EXR-tree percentage	14.4%	17.6%	13.4%	14.3%	15.5%	16.5%
1^+ level EXO size(KB)	43	282	40	76	101	129

TABLE 2
EXR-Tree Construction and Maintenance Cost

	NE	US	UN-2D	UN-4D	UN-6D	UN-8D
construction time	16	52	247	302	416	529
EXR-tree maintenance time(sec)	0.37	1.13	5.08	5.72	7.09	9.41
R-tree maintenance time(sec)	0.2	0.62	2.13	2.28	2.69	3.34
EXR-tree percentage	45.9%	45.1%	58.1%	60.1%	62.1%	64.5%

reasonable as EXR-tree redundantly stores external objects.

5. As d increases, the number of EXOs increases, a result of the "dimensionality curse."

7.4.2 PNN and CNN Results

In the first experiment, we evaluate the EXR-tree on PNN queries using all six data sets. Fig. 13a shows the improvement of BFS-EXR over BFS-R, which ranges from 13 percent to 32 percent. BFS-SR, on the other hand, is comparable to BFS-EXR only for the two 2D real data sets. This is due to their different mechanisms of speeding up NN search: While SR-tree refines the bounding area for a more accurate $MINDIST$ value, EXR-tree directly stores the external objects that lead to more node accesses. As such, EXR-tree is less vulnerable to dimensionality and skewness than SR-tree. Nonetheless, a skew data set increases the gain of EXR-tree because it is more likely for the query point to locate outside all the child nodes, which BFS-EXR avoids visiting. Low dimensionality also increases the gain of EXR-tree since there are fewer external objects as d decreases, making the EXR-tree pruning more efficient.

In the next experiment, we evaluate the BFS-EXR and BFS-R algorithms on CNN queries. We vary the query length [5] and plot the number of page accesses in Fig. 13b.[6] The overall performance gain ranges from 10 percent to 40 percent, which is significant. However, the gain shrinks as the query length increases because a longer line segment is more likely to intersect a node, making the EXR-tree pruning less effective. Furthermore, the gain in NE is larger than that in UN-8d due to higher skewness and smaller dimensionality.

7.4.3 RNN Results

In this section, we compare BFS-EXR, BFS-R, and BFS-SR on RNN queries. Three experiments were run, each varying len, k, and mem (the cache size) individually. In the first experiment, len varies from 1/1000 to 1/125 and we can see from Fig. 14a an improvement of EXR-tree over R-tree of 20 percent to 50 percent. This improvement is more evident than CNN given the same len setting because the query input is a multidimensional range rather than a one-dimensional line segment. On the other hand, SR-tree is not suitable for RNN query as its improvement over R-tree is only visible for the 2D data set and small len; it is even worse than R-tree in cases of high dimensions and large len. This is attributed to the fact that SR-tree is less accurate than R-tree when it comes to the computation of $MINDIST$ between the bounding area and the query range. Therefore, we do not compare BFS-SR in the subsequent experiments.

Fig. 14b shows the results of the second experiment where k varies from 1 to 9. Although EXR-tree can still handle $k > 1$, large k degrades the performance more than large len does. This is because the EXR-tree becomes less efficient for larger k values as the number of EXOs increases. Nonetheless, even in the extremely adverse case, that is, uniform data set with a high dimension and high k values, BFS-EXR still outperforms BFS-R by 5 percent to 10 percent, which shows that EXR-tree is a robust index for RNN queries. In the third experiment, we vary mem, the cache memory for the BFS-EXR algorithm or the BFS-R algorithm. Fig. 15 shows that for both data sets, EXR-tree enlarges its performance gain over R-tree when mem

6. To simplify the charts, in the sequel only the results for data sets NE and UN-8d are shown.

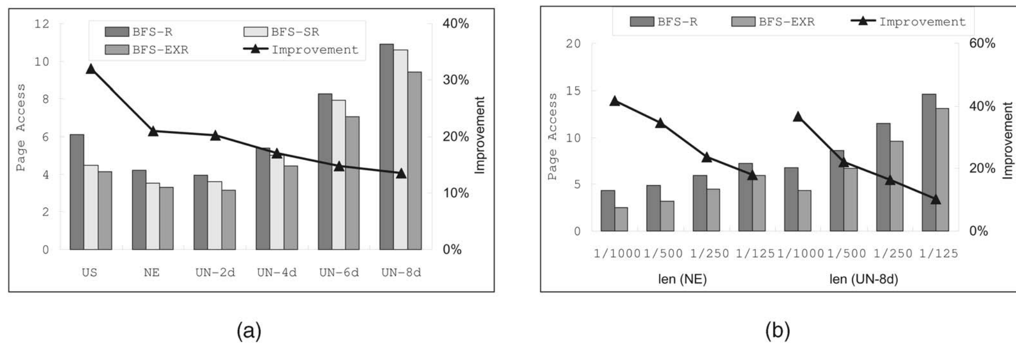


Fig. 13. (a) PNN and (b) CNN performance gain under all data sets.

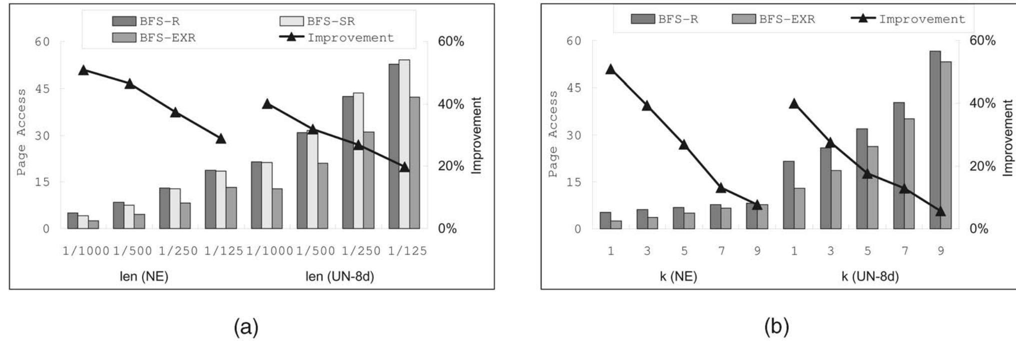


Fig. 14. RNN performance gain under NE and UN-8d data sets. (a) Various len. (b) Various k.

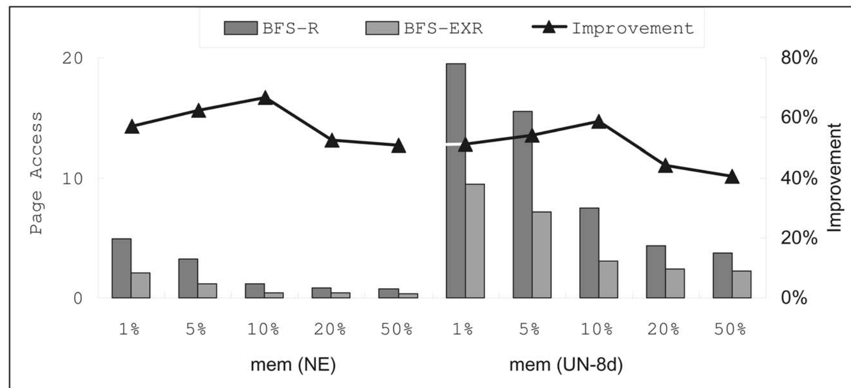


Fig. 15. RNN performance gain under various cache sizes.

increases from 1 percent to 10 percent, but shrinks when mem increases from 20 percent to 50 percent. This is because EXR-tree manages the memory more efficiently: The LRU cache strikes a balance between caching R-tree nodes and caching leaf EXO-tree nodes. In other words, compared to R-tree, EXR-tree caches "better" nodes that can save more I/O cost. As a side effect, the performance of EXR-tree also saturates earlier than that of R-tree, which explains why the improvement shrinks when $mem > 20\%$.

7.4.4 EXR-Tree Summary

We summarize the characteristics of EXR-tree as follows:

1. It achieves a better performance on skew data set than on uniform data set.
2. It achieves a higher performance gain when the dimension of the query input increases; that is, the performance gain on RNN queries is higher than on CNN queries, which is still higher than on PNN queries.
3. It achieves a better performance in lower dimensions.
4. It can handle kNN queries, but for each k, a separate EXR-tree must be built unless there is a maximum value of k for which an EXR-tree can be built; furthermore, the EXR-tree becomes less beneficial as k increases and more objects become EXOs.
5. The EXR-tree uses cache memory more efficiently than the R-tree.

8 Conclusion and Future Work

In this paper, we proposed the range nearest neighbor (RNN) query as an extension to point nearest neighbor (PNN) and continuous nearest-neighbor (CNN) queries. We then devised efficient in-memory processing algorithms and secondary memory pruning heuristics for both 2D and

d-dimensional spaces. All these techniques were generalized to handle kRNN queries. We also proposed an innovative solution-based auxiliary index, EXO-tree, which is dedicated to speeding up the processing of all types of NN queries. Empirical results showed that our proposed techniques are efficient and robust given various data sets, query range sizes, numbers of nearest neighbors returned (k), dimensions, and cache sizes. We also performed extensive experiments on EXR-tree, an R-tree augmented by an EXO-tree, and showed that it significantly improves the NN search performance. It achieves the best performance for skewed data sets, small k values, small to medium dimensions, and RNN queries type. Since EXO-tree is orthogonal to other NN processing techniques, it can be incorporated into any existing NN searching algorithms.

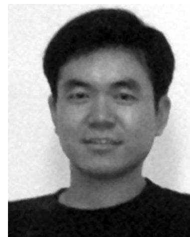
In future work, we will extend the query shape from a hyperrectangle to an arbitrary shape that has a close-form mathematical expression. We are also interested in handling a moving object data set rather than a stationary spatial data set. Regarding the EXO-tree, since it is independent of the primary index that it is attached to, it would be meaningful to investigate its performance on other spatial indexes such as kd-trees (integrated as an EXkd-tree). In addition, since the performance of EXO-tree degrades as k increases, we will seek alternative ways to index objects for kNN (including kPNN, kCNN, and kRNN) queries when k is large.

ACKNOWLEDGMENTS

This work was supported by Research Grants Council, Hong Kong SAR, China, under grants HKUST6179/03E and HKUST6277/04E.

REFERENCES

[1] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas, "Fast Nearest Neighbor Search in Medical Image Databases," *Proc. 22nd Int'l Conf. Very Large Data Bases,* pp. 215-226, 1996.
[2] T. Seidl and H.-P. Kriegel, "Optimal Multi-Step K-Nearest Neighbor Search," *Proc. 1998 ACM SIGMOD Int'l Conf. Management of Data,* pp. 154-165, 1998.
[3] R. Weber, H.-J. Schek, and S. Blott, "A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces," *Proc. 24th Int'l Conf. Very Large Data Bases,* pp. 194-205, 1998.
[4] B. Cui, B.C. Ooi, J. Su, and K.-L. Tan, "Contorting High Dimensional Data for Efficient Main Memory KNN Processing," *Proc. 2003 ACM SIGMOD Int'l Conf. Management of Data,* pp. 479-490, 2003.
[5] Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," *Proc. Very Large Data Bases Conf.,* pp. 287-298, 2002.
[6] N. Roussopoulos, S. Kelley, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 71-79, 1995.
[7] G.R. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems (TODS),* vol. 24, no. 2, pp. 265-318, 1999.
[8] A. Guttman, "R-Trees: A Dynamic Index Structure for Spatial Searching," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 47-57, 1984.
[9] T. Sellis, N. Roussopoulos, and C. Faloutsos, "The R^+-Tree: A Dynamic Index for Multidimensional Objects," *Proc. Very Large Data Bases Conf.,* pp. 3-11, 1987.
[10] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger, "The R^+-Tree: An Efficient and Robust Access Method for Points and Rectangles," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 322-331, 1990.
[11] Z. Song and N. Roussopoulos, "K-Nearest Neighbor Search for Moving Query Point," *Proc. Symp. Spatial and Temporal Databases,* pp. 79-96, 2001.
[12] D.A. White and R. Jain, "Similarity Indexing with the SS-Tree," *Proc. Int'l Conf. Data Eng. (ICDE),* pp. 516-523, 1996.
[13] N. Katayama and S. Satoh, "The SR-Tree: An Index Structure for High-Dimensional Nearest Neighbor Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data,* pp. 369-380, 1997.
[14] S.-W. Kim, C.C. Aggarwal, and P.S. Yu, "Effective Nearest Neighbor Indexing with the Euclidean Metric," *Proc. Conf. Information and Knowledge Management,* pp. 9-16, 2001.
[15] C. Yu, B.C. Ooi, K.-L. Tan, and H.V. Jagadish, "Indexing the Distance: An Efficient Method to KNN Processing," *Proc. Very Large Data Bases Conf.,* pp. 421-430, 2001.
[16] S. Berchtold, B. Ertl, D.A. Keim, H.-P. Kriegel, and T. Seidl, "Fast Nearest Neighbor Search in High-Dimensional Spaces," *Proc. Int'l Conf. Data Eng.,* pp. 209-218, 1998.
[17] S. Berchtold, D.A. Keim, H.-P. Kriegel, and T. Seidl, "Indexing the Solution Space: A New Technique for Nearest Neighbor Search in High-Dimensional Space," *IEEE Trans. Knowledge and Data Eng.,* vol. 12, no. 1, pp. 45-57, Jan./Feb. 2000.
[18] B. Zheng, J. Xu, W.-C. Lee, and D.L. Lee, "Grid-Partition Index: A Hybrid Approach to Nearest-Neighbor Queries in Wireless Location-Based Services," *VLDB J.,* 2005.
[19] M.J. Panik, *Linear Programming: Mathematics, Theory and Algorithms.* Kluwer Academic, 1996.
[20] B. Moon, H. Jagadish, C. Faloutsos, and J.H. Saltz, "Analysis of the Clustering Properties of the Hilbert Space-Filling Curve," *IEEE Trans. Knowledge and Data Eng.,* vol. 13, no. 1, pp. 124-141, Jan./Feb. 2001.
[21] Y. Theodoridis, "Spatial Datasets: An Unofficial Collection," http://dke.cti.gr/People/ytheod/research/datasets/ spatial.html, 2005.
[22] S.J. Fortune, "A Sweepline Algorithm for Voronoi Diagrams," *Algorithmica,* vol. 2, pp. 153-174, 1987.
[23] A. Hinneburg, C.C. Aggarwal, and D.A. Keim, "What is the Nearest Neighbor in High Dimensional Spaces?" *Proc. 26th Int'l Conf. Very Large Data Bases,* pp. 506-515, 2000.



Haibo Hu (http://www.cs.ust.hk/~haibo) received the BEng degree in computer science and engineering from Shanghai Jiaotong University, China, in 2001. He is currently a PhD candidate in the Department of Computer Science, Hong Kong University of Science and Technology. His research interests include wireless data dissemination, location-based services, mobile and pervasive computing, and spatiotemporal databases.

Dik Lun Lee (http://www.cs.ust.hk/~dlee) received the PhD degree in computer science from the University of Toronto in 1985. He is a professor in the Department of Computer Science at the Hong Kong University of Science and Technology. Prior to joining HKUST, he was an associate professor in the Department of Computer and Information Science at the Ohio State University, Columbus. He served as a guest editor for several special issues and chaired/co-chaired many international conferences. He was the founding conference chair for the International Conference on Mobile Data Management. His research interests include document retrieval and management, discovery, management and integration of information resources on the Internet, and mobile and pervasive computing. He was the chairman of the ACM Hong Kong Chapter.