Front.Comput.Sci.
DOI

RESEARCH ARTICLE

Answering range – based reverse kNN and why – not reverse kNN queries

Zhefan ZHONG[1], **Xin LIN** (✉)[1], **Liang HE**[1]

1 Shanghai Key Laboratory of Multidimensional Information Processing, East China Normal University, Shanghai 200062, China

Abstract

Given a point q, a reverse k nearest neighbor (RkNN) query retrieves all the data points that have q as one of their k nearest neighbors. Despite significant progress on this problem, there is a research gap in finding RkNNs not just for an object, but for a given range, which is a natural extension of the problem. Moreover, in the last several decades, the database systems have been greatly developed. As explained applications become more practical, it is especially able to explain its query result, which is called "why-not" questions, and is the focus of concern. For example, users often use RkNN queries to investigate the surrounding circumstances. Nevertheless, they often feel disappointed that they only have a little RkNNs in the query result and they want more. Motivated by this, we develop algorithms for *exact* processing of range-based RkNN with *arbitrary* values of k on *dynamic* datasets, which retrieve all the data points that have any position in the given query range R as one of their k nearest neighbors. Afterwards, we propose an algorithm based on range-based RkNN processing, using divide-and-conquer method with flooding approach to solve the "why-not" query. The experimental results demonstrate the efficiency and the accuracy of our proposed optimizations and algorithms.

Keywords Range-based RkNN Queries, Why-not Queries, Location-based Services.

Received; accepted

E-mail: *zhfzhong@ica.stc.sh.cn*, {*xlin*, *lhe*}*@cs.ecnu.edu.cn*

1 Introduction

Given a dataset D and a point q, a reverse nearest neighbor (RNN) query retrieves all the points $p \in D$ that have q as their nearest neighbor. Although the RNN problem was first proposed in [1], it still has received considerable attention due to its importance in several applications involving decision support, resource allocation, profile-based marketing, etc.

Despite significant progress on this problem, there is a research gap in finding RNNs not just for an object, but for a given range, which is a natural extension of the problem. In this paper, we proposed a range-based reverse k nearest neighbors (RRkNN) query, it retrieves all the points $p \in D$ that have any position in the query range R as their kNN. It is useful in our daily life. Imagine this, a corporation advertise their new product via mobile phone. In order to lower the cost, they only push the popularize advertisement to the passengers who are their reverse kNN, according to the selling department which is a range area. In this paper, We assume that the shape of range R is rectangle.

Moreover, in the last several decades, the database systems have been greatly developed. As explained applications become more practical, these systems should be more user-friendly, interactive and cooperative. That is, users are not satisfied with only receiving the query answer, but also want to know why the system returns the current set. If the database system can offer a good explanation for the query answer set, it would be very helpful for users to understand the information needed and thus refine her initial query [2] [3]. For example, taxi-hailing applications are very popular

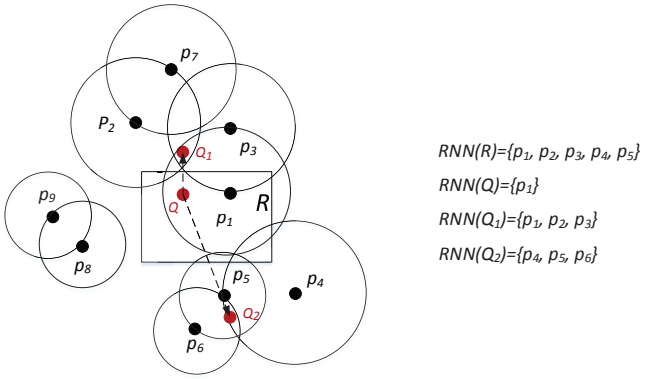


Fig. 1 Range-based RNN and why-not RNN examples

nowadays. When a cab driver using a taxi-hailing app, he wants to know how to be as more potential customers' kNN as possible (i.e. have more RkNNs). It will be denoted as why-not RkNN query in the rest of this paper.

Fig. 1 shows a range R and nine 2D points, where each point p is associated with a circle covering its nearest neighbor. For example, the NN of p_4 (e.g. p_5) is in the circle centered at p_4. Some of these circles (such as circle of p_1) intersect with the range R. Accordingly, $p_1 \in$ RNN(R) (see Definition 2 in Section 3.1). In this case, we can easily get the RNN(R)=$(p_1, p_2, p_3, p_4, p_5)$. Suppose a user present an RNN query at a position Q, he will get the point p_1 as the only result. If he wants to get more RNNs, he can move to the position Q_1 and Q_2, the RNNs of Q_1 are (p_1, p_2, p_3) and the RNNs of Q_2 are (p_4, p_5, p_6).

As discussed in Section 2, all the previous methods for RNN search cannot handle a range-based RNN query, not to mention offering an explanation for the query answer set. Motivated by this, focusing on monochromatic reverse nearest neighbor problems, we develop algorithms to handle range-based reverse k nearest neighbor (RRkNN) queries, which retrieve all the points $p \in D$ that have any position in the query range R as one of their k nearest neighbors. Specifically, we follow the filter-refinement framework, in which the filter step retrieves a set of candidate results that is guaranteed to include all the actual reverse nearest neighbors and the refinement step eliminates the false hits. Afterwards, we propose an algorithm based on range-based RkNN processing, and use a divide-and-conquer method with flooding approach to solve the why-not RkNN query. None of the existing techniques can effectively answer why-not RkNN query accurately. In fact, answering why-not RkNN query is very useful since RkNN query is very common in our daily life.

The rest of the paper is organized as follows. Section 2 surveys related work on RNN and why-not search. In Section 3, we formulate the problem and illustrate that it is computationally expensive for existing algorithms. In Section 4, we present some interesting problem characteristics, and propose a new algorithm with demonstrations to solve the range-based RkNN problem efficiently. Section 5 proposes our method to solve why-not RkNN query. In Section 6, we report experimental results and we conclude the paper in Section 7.

2 Related Work

2.1 RkNN Query

There exist various versions of RNN problem include (1) continuous RNN [4], in which the database contains linearly moving objects with fixed velocities, and the goal is to retrieve all RNNs of q for a future interval; (2) bichromatic RNN [5], given a set Q of queries, the goal is to find the objects $p \in D$ that are closer to some $q \in Q$ than any other point of Q; (3) stream RNN [6], where data arrives in the form of streams, and the goal is to report aggregated results over the RNNs of a set of query points.

Algorithms for RNN processing can be classified into two categories depending on whether they require preprocessing, or not. The original RNN method [1] pre-computes for each data point p its nearest neighbor NN(p). Then using the RNN-tree, the reverse nearest neighbors of q can be efficiently retrieved by a point location query, which returns all circles that contain q. Similarly, Yang and Lin [7] combine the R-tree and RNN-tree in the RdNN-tree. Another solution based on pre-computation is proposed in [8]. All techniques that rely on pre-processing cannot deal efficiently with updates because each insertion or deletion may affect the vicinity circles of several points. Stanoi et al. [9] solve the problem by utilizing some interesting properties of RNN retrieval, adopts a two-step processing method(filter-refinement framework). Singh et al. [10] finds the kNNs of the query q, eliminates the candidates that are closer to some other candidate than q and then applies boolean range queries on the remaining candidates to determine the actual RNNs. [11] develop algorithms for exact processing of RkNN with arbitrary values of k on dynamic multi-dimensional datasets. Their methods utilize a conventional data-partitioning index on the dataset and do not require any pre-computation. However, all these previous methods for RNN search can not handle a range-based RkNN query.

2.2 Why-Not Query

Recently, why-not questions have been explored on several fields, such as SQL queries [12], reverse skyline queries [13], reverse top-k queries [14] [15], range-based skyline queries in road networks [16], top-k geo-social keyword queries in road networks [17], metric probabilistic range queries [18]. And [19] [20] attempt to answer user queries when query results are created via processes and datasets that are opaque to the user.

There exist various solutions to answer why-not questions. For example, Chapman et al. [12] try to find the causes why the expected data point does not appear in the query output. In addition, another approach is modifying the data point so that the expected objects could return to query result, proposed by Huang et al. [21] for SPJ (Select-Project-Join) and [22] for SPJUA (SPJ-Union-Aggregation) queries, are telling the users how the data should be modified, such as adding a object. Then, some people modified the original query so that the non-answer point appears in the refined query answer set, which was proposed by Tran and Chan [2] for SPJA (SPJ-Aggregation) queries. Chen et al. propose an answering engine YASK for spatial keyword query services [23] and two query refinement models, namely preference adjustment [24] and keyword adaption [25]. He et al. [3] also propose a query refinement approach for answering why-not questions in top-k queries, while [13] studies the problem of answering why-not questions in reverse skyline queries in light of the above aspects. But why-not query is query-dependent, none of the existing techniques can effectively answer why-not RkNN queries accurately.

3 Problem Statement And Analysis

In this section, we first give preliminaries of range-based RkNN query and the why-not RkNN query. Afterwards, we formally define the problem of why-not RkNN query. Finally we introduce some concepts used for solving this problem.

3.1 Preliminaries

Given a spatial dataset D of n objects, each object p with 2 attribute values can be represented as a point $p = (p[1], p[2])$ in a 2-dimensional data space. For simplicity, we assume that all attribute values are numeric. A range-based RkNN query is composed of a query range R and retrieves all the points $p \in D$ that have any position in the query range R as one of their kNNs. In this paper, we consider the Euclidean distance as the metric. The query results would then be a set of objects whose kNN vicinity circle [1] overlaps query range R. A why-not RkNN query is based on an initial RkNN query, in which a user is not satisfied with the RkNNs she gets, and is willing to move her position to have more RkNNs.

Definition 1 (Reverse k Nearest Neighbor Query(RkNNQ))**.** *Given a dataset D and a query point q(k), the query returns all the points $p \in D$ that have q as one of their k nearest neighbors, denote as RkNN(q).*

Definition 2 (Range-Based Reverse k Nearest Neighbor Query(RRkNNQ))**.** *Given a dataset D and a query range R, the query returns an answer set denote as RkNN(R), $\forall\ o \in D$, $\forall\ p \in R$, if $o \in RkNN(p)$, then $o \in RkNN(R)$. That is, $RkNN(p) \subseteq RkNN(R)$.*

3.2 Problem Definition

Initially, a user specifies an RkNN query $q(k)$. After she gets the result points counted S, she may pose a why-not question with a hope that the system returns her a refined RkNN query $q'(k)$ such that she gets more RkNNs. We use Δq as the *penalty*, ΔS as the $benefit$ to measure the quality of the refined query, where $\Delta q = \|q - q'\|$, $\Delta S = \|S - S'\|$.

In order to capture a user's preference to the changes of S and q on her original query $q(k)$, a basic scoring model that sets the arguments λ_S and λ_q to ΔS and Δq, respectively, where $\lambda_S + \lambda_q = 1$, is as follows:

$$Penalty(q) = \lambda_q \cdot \Delta q. \quad Benefit(S) = \lambda_S \cdot \Delta S.$$

$$Score(S, q) = Benefit(S) + (-Penalty(q)) = \lambda_S \cdot \Delta S + (-\lambda_q \cdot \Delta q).$$

Note that the basic scoring model has its discrimination. One possible way to mitigate this discrimination is to normalize ΔS and Δq respectively. To do so, we normalize Δq using DIS, the maximum range of the dataset D. Similarly, we normalize ΔS using n, the sum of objects in the dataset D. A new modified scoring function is as follows:

$$Score(S, q) = \frac{\lambda_S \cdot \Delta S}{n} + (-\frac{\lambda_q \cdot \Delta q}{DIS}). \tag{1}$$

Definition 3 (Why-Not RkNN Query)**.** *Given a RkNN query q, the query figures out a refined query q' that maximizing the RkNNs with the minimum change of q, that is, maximizing $Score(S, q)$ (as shown in Fig. 1: Q, Q_1, Q_2).*

Nevertheless, our solution indeed works for all kinds of monotonic (with respect to both ΔS and Δq) scoring functions.

3.3 Problem Analysis

As we can see, in why-not RkNN queries, the refined position can be located in every location of the map, then all the objects in the dataset should be taken into account, a new position means a new round of RkNN process. We can precompute kNNs for each data point p, but it can not handle arbitrary values of k. To solve the why-not queries, we propose a divide-and-conquer method with a flooding approach, based on the range-based RkNN processing. Compare to existing point-based RkNN query, a range-based RkNN query needs great amount of calculation. One of the straightforward methods is to precompute the kNNs for all the data points, however, it's unreasonable since it is costly to extends to arbitrary values of k.

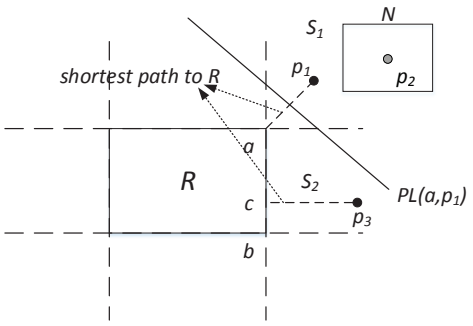
4 Range-Based RkNN Query

In this section, we present our method to solve range-based RkNN queries. Section 4.1 illustrates some problem characteristics that permit the development of efficient algorithms presented in Section 4.2. Section 4.3 presents properties that permit pruning of the search space for arbitrary values of k, then extends our methods for range-based RkNN queries. We assume that dataset D is indexed by R-tree.

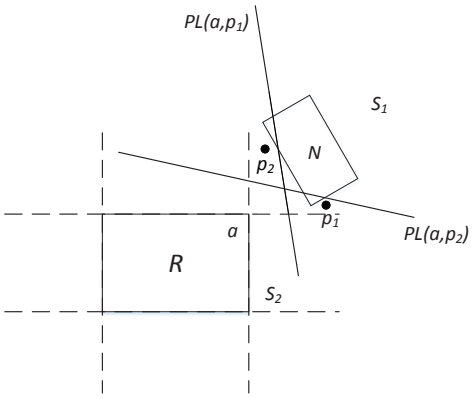
4.1 Problem Characteristics

Consider single range-based RNN processing first. As shown in Fig. 2a, if we divide the space into nine subspaces according to the edge of the query range R, we can obtain the candidate results of RNNs in every subspace separately. There are two kinds of subspaces except the query range itself: (i) the ones adjacent to a vertex (e.g., a) of query range, be composed of extension line of two edges of R like S_1, in which a point p_1's shortest path to R is $|p_1a|$; (ii) the other ones adjacent to an edge (e.g., ab) of query range, be composed of an edge and extension line of two edges of R like S_2, in which a point p_3's shortest path to R is $|p_3c|$ ($p_3c \perp ab$).

Consider the perpendicular bisector $\perp (a, p_1)$ between the vertex of query range a and an arbitrary data point p_1 as shown in Fig. 2a. The bisector divides S_1 into two half-planes: $PL_a(a, p_1)$ that contains a, and $PL_{p_1}(a, p_1)$ that contains p_1. Any point (e.g., p_2) in the $PL_{p_1}(a, p_1)$ cannot be



(a) Pruning with one point



(b) Pruning with two points

Fig. 2 Half-plane pruning strategy for subspaces like S_1

a RNN of R because it is closer to p_1 than a. Similarly, a node MBR (e.g., N) that falls completely in $PL_{p_1}(a, p_1)$ cannot contain any candidate. In some cases, the pruning of an MBR requires multiple half-planes. For example, in Fig. 2b, although N does not fall completely in $PL_{p_1}(a, p_1)$ or $PL_{p_2}(a, p_2)$, it can still be pruned since it lies entirely in the *union* of the two half-planes. In general, if p_1, p_2, ...p_n are n data points, then any node whose MBR falls inside $\bigcup_{i=1\sim n} PL_{p_i}(a, p_i)$ cannot contain any RNN result.

Similarly, as shown in Fig. 3, in the subspaces like S_2, consider the parabola $Par(L, p_1)$ having L as its directrix and p_1 as its focus. The parabola divides S_2 into two half-planes: $Par_L(L, p_1)$ that contains L, and $Par_{p_1}(L, p_1)$ that contains p_1. Any point (e.g., p_2) in $Par_{p_1}(L, p_1)$ cannot be a RNN of R because it is closer to p_1 than L. And a node MBR (e.g., N) that completely falls in $Par_{p_1}(L, p_1)$ cannot contain any candidate.

As such, the pruning approach shown in Algorithm 1 can safely eliminate node MBRs (data points) who can not contain (be) candidates, we can use it to obtain the RNN candidates. While for the data points in query range R, they are born to be RNNs.

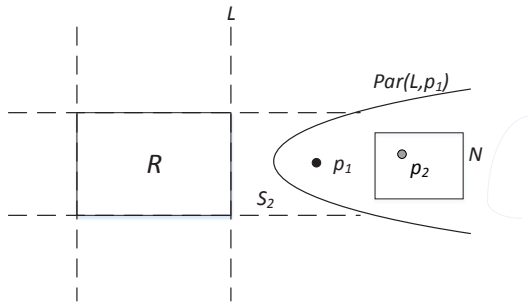
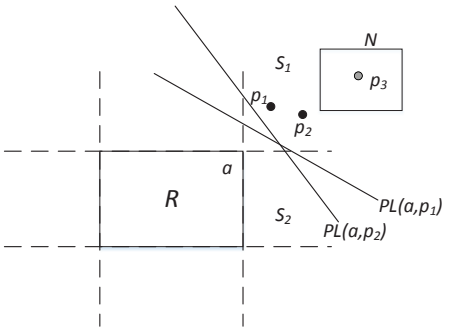
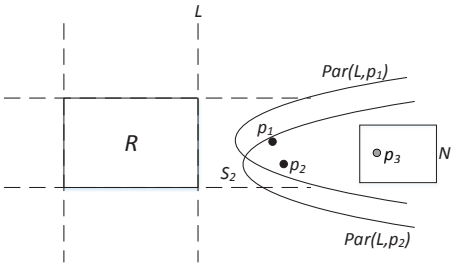


Fig. 3 Half-plane pruning strategy for subspaces like S_2 (pruning with one point)



(a) Range 2-pruning algorithm in subspaces like S_1



(b) Range 2-pruning algorithm in subspaces like S_2

Fig. 4 Examples of R2NN queries

Algorithm 1 Range pruning algorithm

INPUT: query range R, $(p_1, p_2, ...p_n)$, point p, node N //$p_1, p_2, ...p_n$ are arbitrary data points. Take S_1 for example
OUTPUT: $dist(p, R)$, $mindist(N, R)$

1: **for** i = 1 to n
2: **if** p in $PL_{p_i}(a, p_i)$
3: **then** $dist(p, R) = \infty$ //p can be pruned
4: **if** N falls completely in $\bigcup_{i=1\sim n} PL_{p_i}(a, p_i)$
5: **then** $mindist(N, R) = \infty$ //N can be pruned
6: **if** p cannot be pruned **then** return $dist(p, R)$ //minimum distance from p to R
7: **if** N cannot be pruned **then** return $mindist(N, R)$ //minimum distance from N to R

4.2 The TPR Algorithm

Based on the above discussions, we adopt a two-step framework that retrieves a set of candidate RNNs (filtering step) and then removes the false hits (refinement step). Different from [9] and [10], our algorithm (hereafter, called TPR) traverses the R-tree in a best-first manner, retrieving potential candidates in ascending order of their distance to the query range R in each subspace respectively, because RNNs are likely to be near R. The concept of half-plane is used to prune node MBRs (data points) that cannot contain (be) candidates. In the refinement step, we applies boolean range queries on the remaining candidates to determine the actual RNNs. Algorithm 2 shows the details of TPR algorithm.

4.3 Range-Based RkNN Processing

This section presents properties that help pruning of the search space for arbitrary values of k and extends our TPR algorithm for range-based RkNN query.

Fig. 4 shows an example with $k = 2$. In Fig. 4a, p_3 is not a R2NN of R, since p_3 is in the *intersection* of $PL_{p_1}(a, p_1)$ and $PL_{p_2}(a, p_2)$. In Fig. 4b, p_3 is not a R2NN of R, since p_3 is in the *intersection* of $Par_{p_1}(L, p_1)$ and $Par_{p_2}(L, p_2)$. Both p_1 and p_2 are closer to p_3 than R. Similarly, a node MBR N can not contain any candidates (i.e., N can be pruned at the filter step). In some cases, several half-planes' *intersections* are needed to prune a node. As shown in Algorithm 3.

5 Why-Not RkNN Query

In this section, we present our method to solve why-not RkNN query. Section 5.1 shows a brief introduction of divide-and-conquer method for this problem. Details will be

Algorithm 2 TPR algorithm

INPUT: dataset D indexed in R-Tree $\mathcal{T}$, query range R
OUTPUT: S_{rnn}

Initialize sets S_{cnd} //RNN candidates
Divide the space into nine parts around query range R
In the subspaces that adjacent to a vertex (edge) of query range like $S_1(S_2)$ **do**
  Initialize a min-heap H accepting entries of the form (e, key)
  Insert $(\mathcal{T}, 0)$ to H
  while H is not empty
  (e, key)=de-heap H
    if e can be pruned **then** goto 6
    else //entry may be or contain a candidate
      if e is data point p **then** S_{cnd}=$S_{cnd} \cup p$
      else if e points to a leaf node N
        for each point p in N (sorted on $dist(p, R)$)
          if p cannot be pruned **then** insert $(p, dist(p, R))$ in H
      else //e points to an intermediate node N
        for each entry N_i in N
          if N_i cannot be pruned **then** insert $(N_i, mindist(N_i, R))$ in H
for each point p *in* S_{cnd} apply a boolean range query to determine the S_{rnn}
insert all the points p in query range R into S_{rnn}
return S_{rnn}

discussed in Section 5.2. Section 5.3 shows the error analysis of our method presented.

5.1 Divide And Conquer Method

As discussed in Section 3, a naive solution for why-not RkNN query needs plenty of precomputing and can not handle arbitrary values of k. The divide-and-conquer method is suitable in this situation. As shown in Fig. 5, divide the space into ranges sized R (the size of R will be discussed in Section 6), q is the original RkNN query located in one of these ranges. Consider an arbitrary point p in range R, only the RkNNs of R have a chance to be RkNN of p. Obviously, the kNN vicinity circles [1] of RkNNs intersect with each other, as shown in Fig. 5a, the subrange which overlaps most vicinity circles may contain the desired refined q'. We first figure out the best subrange in range R that fits the user's preference (more RkNNs and with the smallest penalty). Then we apply a flooding procedure, exploring the ranges around R step by step until the farther position can not be superior to the former optimal position, in terms of the scoring function, as shown in Fig. 5b.

Algorithm 3 Range k-pruning algorithm

INPUT: query range R, $(p_1, p_2, ...p_n)$, point p, node N, k //$p_1, p_2, ...p_n$ are candidate data points, $n \geq k$. Take S_1 for example
OUTPUT: $dist(p, R)$, $mindist(N, R)$

for i = 1 to C_n^k //C_n^k subset
    //Assume a subset is $(\sigma_1, \sigma_2, ...\sigma_k)$
  if p in $\bigcap_{i=1\sim k} PL_{\sigma_i}(a, \sigma_i)$
    then $dist(p, R) = \infty$ //p can be pruned
if N falls completely in $\bigcup_{j=1\sim C_n^k}(\bigcap_{i=1\sim k} PL_{\sigma_i}(a, \sigma_i))$
  then $mindist(N, R) = \infty$ //N can be pruned
if p cannot be pruned **then** return $dist(p, R)$ //minimum distance from p to R
if N cannot be pruned **then** return $mindist(N, R)$ //minimum distance from N to R

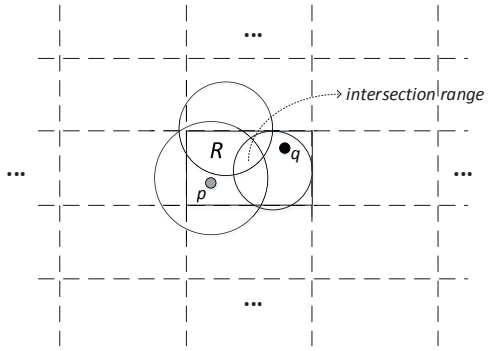
5.2 Why-Not Processing

The best subrange in R must be a irregular convex hull, which is formed by the RkNNs' vicinity circles intersect with each other (see Fig. 5a). It is costly to find the exact convex hull, for simplicity, we'll use the approximation method to approaching the exact solution and the error analysis will be discussed in Section 5.3.

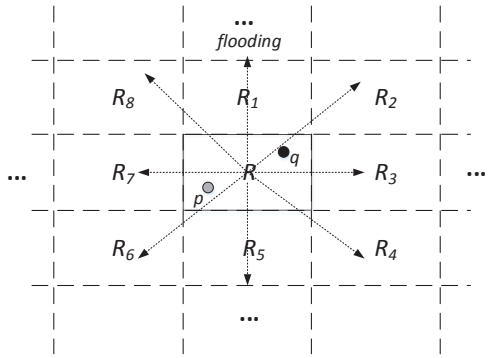
As shown in Fig. 6, consider a why-not RNN query. Initially, we get the RNN of range R: $p_1, p_2, p_3, p_4, p_5, p_6, p_7$ and draw their NN vicinity circle. To find (one of) the best subrange(s) in a range R, we quartering R iteratively until the sub-rectangles are with a small enough error margin (see Section 5.3): $d \leq \varepsilon$. To obtain the RNNs of a sub-rectangle in R, we check the RNNs of R that whose vicinity circle overlaps with the sub-rectangle. It is clear that the exact convex hull that has most RNNs in R must intersect with at least one sub-rectangle. Then decide the best sub-rectangle: overlaps with most RNNs' vicinity circles and with the smallest penalty, in this case the bolded sub-rectangle R_1 has most RNNs. p_1, p_2, p_3, p_4, p_5 are the RNN of R_1. In the flooding step, see Fig. 5b. We conduct the outer ranges $R_1 \sim R_8$ around initial R using the quartering method, and then the more peripheral ranges. Maintaining an optimal sub-rectangle during all the procedures above. As soon as the farther position cannot exceed previous best subrange, stop flooding. The details are shown in Algorithm 4.

5.3 Error Analysis

Initially, we set the error margin threshold value ε. The quartering procedure will not be stopped until $d_r \leq \varepsilon$, where d_r is the length of the final sub-rectangle's length. And the quartering times $t \geq log_2^{d_R/\varepsilon}$, where d_R is the length of ini-



(a) Divided space



(b) Flooding procedure

Fig. 5 Divide and conquer method

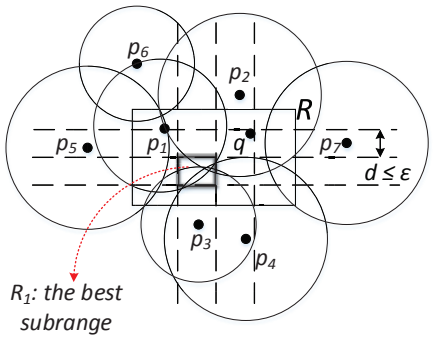


Fig. 6 Quartering approach

Algorithm 4 Quartering-Flooding range algorithm

INPUT: RkNN query $q(k)$
OUTPUT: refined $q'(k)$

1: Divide the space into ranges sized R
2: Determine the RkNNs of R //the range that contains q
3: Quartering the range R iteratively, until $d_r \leq \varepsilon$ //where d_r is the length of the final subrectangle's length and ε is the error margin
4: Determine the best rectangle *rec* in R
//Flooding
5: **for** each ranges around R, **do** step 2-4, continuing hold a optimal rectangle *rec*
6: **stop flooding** while the farther position can not be superior to *rec*
7: return $q'(k)$ //nearest position from *rec* to q

tial R. Therefore, the precision of our approximation method depends on the margin threshold value ε.

6 Performance Evaluation

In this section, we evaluate the performance of our proposed algorithms through experiments. We first present the experimental setup in Section 6.1, followed by the results for range-based RkNN queries in Section 6.2 and why-not RkNN query in Section 6.3 respectively.

6.1 Experimental Setup

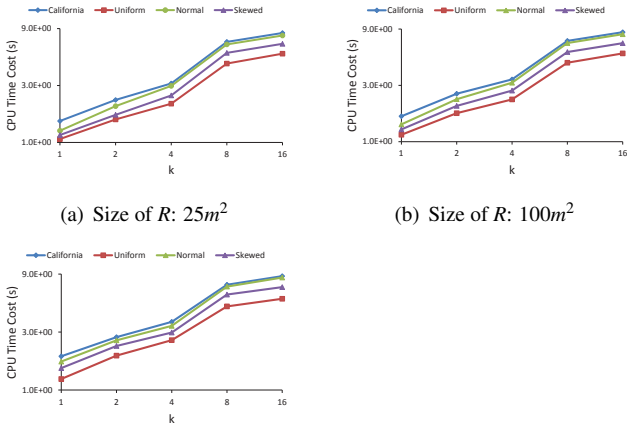
6.1.1 Implemented Algorithm

As mentioned above, all the previous methods cannot handle a range-based RkNN query or a why-not RkNN query accurately. The naive approach for each query, which are based on precomputing, need great amount of calculations and they can not extend to arbitrary values of k. Therefore, for comparison, we implement the TPL algorithm proposed in [11], using average sampling approach to approximate handle these two queries. The more sampling numbers there are, the more accurate the result, and comes with more cost at the same time.

For fairness, we implement all these algorithms in each experiment to demonstrate the effects of our proposed optimizations.

6.1.2 Datasets

The experiments are conducted on five datasets: CaliforniaDB (CD), a spatial data in California; NA dataset contains spatial data corresponding to geometric locations in the



(a) Size of R: $25m^2$ (b) Size of R: $100m^2$

(c) Size of R: $400m^2$

Fig. 7 Varying data distribution, size of R and k (for RkNN)

North America; three synthesized datasets, a normal distribution dataset, an uniform distribution dataset and a skewed dataset. The CD dataset and the skewed dataset have about 200K objects in total and the NA dataset has 569k objects, the normal, uniform synthesized datasets have 1024K objects in total. Each dataset is indexed by an R-tree [26].

6.1.3 System Setup And Metrics

We execute our experiments on a PC (Inter(R) Core(TM)2 E7500, 2.93 GHz CPU) running Windows 7 operating system. The simulation codes are written in Java (JDK 1.6). All objects are assumed to be located in a 100,000m*100,000m space.

The experiments investigate the effect of the following parameters: (i)data size and distribution, (ii)size of query range R, (iii)k (for RkNN) and (iv)user's preference λ_S and λ_q. The query locations or query ranges are randomly selected in the space. Each experimental result is the average result over 200 queries. The cost includes both the I/O overhead and CPU time. For performance metrics, we measure the CPU time and the accuracy (actual RkNN numbers and error distance) in the experiment.

6.2 Experiments For Range-Based RkNN Queries

6.2.1 Effect Of Data Distribution And Size Of R

In the first experiment, we evaluate the effect of varying the data distribution and the size of query range R. As shown in Fig. 7, the difference of CPU time in all datasets with various size of query range R is not that big. That's because our TPR algorithm returns the RkNNs of R within one-time traverse of the R-tree. This observation is confirmed by all experiments (including the real data) despite the different settings.

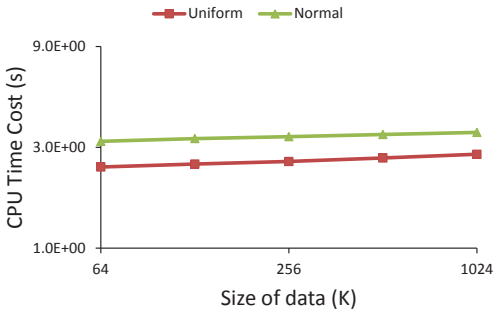


Fig. 8 Varying size of datasets: uniform, normal (k=4, size of R: $400m^2$)

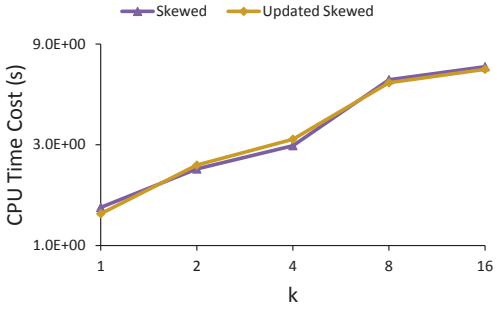


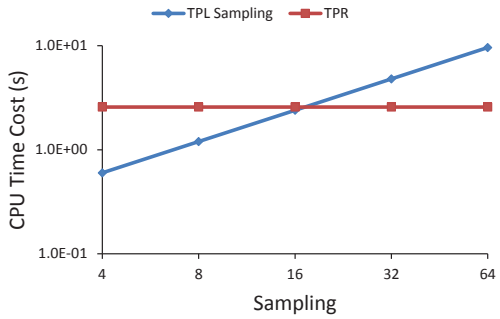
Fig. 9 Addition, deletion and modification on skewed dataset (size of R: $400m^2$)

6.2.2 Effect Of k (for RkNN)

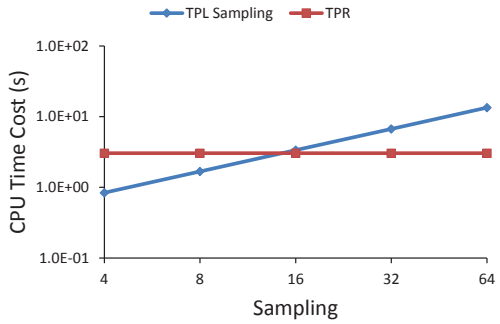
As shown in Fig. 7, we set the size of query range R to $25m^2$, $100m^2$, $400m^2$, respectively. As expected, the overhead of TPR algorithm grows with k, due to the significant increase in CPU time. This is because a larger k degrade the pruning effect of the points or R-tree nodes. Note that the average number of candidates retrieved increases almost linearly with k in the filter step and thus need more computations in the refinement step.

6.2.3 Effect Of Dynamic Dataset

In Fig. 8, we varing the size of datasets: uniform, normal, set $k = 4$, size of R: $400m^2$. In Fig. 9, the skewed dataset has been updated. These experiments are illustrate that neither the size of dataset nor the update of dataset have a substantial effect on the proposed TPR algorithm. That's because the TPR algorithm doesn't need a pre-compute processing, it follows the filter-refinement framework, in which the filter step retrieves a set of candidate results that is guaranteed to include all the actual reverse nearest neighbors and the refinement step eliminates the false hits. The TPR algorithm



(a) Uniform, k=4, size of R: $400m^2$



(b) NA, k=4, size of R: $400m^2$

Fig. 10 TPR vs. TPL Sampling: CPU Time(s)

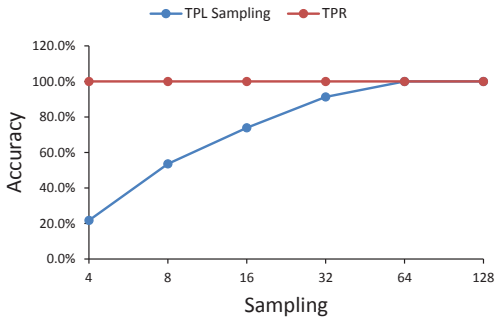


Fig. 11 TPR vs. TPL Sampling: Accuracy (CD, k=4, size of R: $400m^2$)

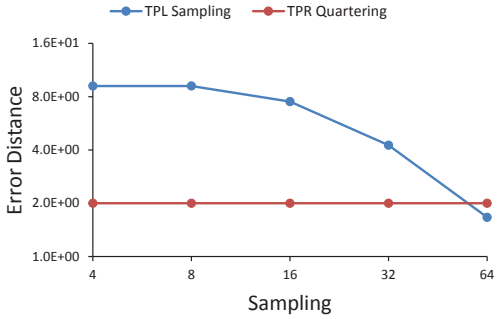


Fig. 12 TPR Quartering vs. TPL Sampling: Error Distance (CD, k=4, size of R: $256m^2$, error margin $\varepsilon = 2m$)

returns the RkNNs of R within one-time traverse of the R-tree.

6.2.4 TPR vs. TPL Sampling

In this experiment, we compare our TPR algorithm to the T-PL Sampling algorithm. Original TPL algorithm is intended to solve the point-based RkNN query, for comparison, we implement the TPL algorithm using average sampling approach to approximate handle the range-based RkNN queries. The more samplings there are, the more accurate the result, but comes with more cost at the same time. Fig. 10 shows the total cost of TPR and TPL Sampling as a function of the sampling number for uniform distribution dataset and NA dataset, set k=4 and the size of R=$400m^2$. In this case, the CPU time of TPL Sampling algorithm increase linearly to the sampling number and the TPR algorithm doesn't need any sampling. When the sampling number come up to about 16, the CPU time of TPR and TPL Sampling are almost the same, but the TPL Sampling can not ensure a correct result. Fig. 11 illustrates the accuracy of TPR and TPL Sampling, fixes k=4 and the size of R=$400m^2$ using CD dataset. Our TPR returns the exact RkNNs for the query range and the TPL Sampling needs as many of sampling as possible. As expected, in our experiment the sampling number come up to about 64 can T-PL Sampling computes the correct result. Therefore, the TPR algorithm performs better with respect to the accuracy.

6.3 Experiments For Why-Not RkNN Queries

6.3.1 TPR Quartering vs. TPL Sampling

Fig. 12 compares the performance of TPR Quartering and TPL Sampling in finding the best subrange (position) in a single range R, using real dataset CD, fixs k=4, the size of R=$256m^2$ and the error margin $\varepsilon = 2m$. As expected, for TPL Sampling, more samplings makes the result more accurate. In this case if we sampling 64 points can the TPL exceed our TPR Quartering in terms of accuracy (with a higher CPU Time cost however).

6.3.2 TPR Quartering-Flooding vs. TPL Sampling

Next we explore the performance of TPR Quartering-Flooding and TPL Sampling using uniform distribution dataset, fix k=4, the size of R=$400m^2$, λ_S=λ_q=0.5 and the error margin $\varepsilon = 5m$. As shown in Fig.13, our TPR Quartering-Flooding method doesn't need any sampling and the CPU time of TPL Sampling algorithm increase linearly to the sampling number. Note that a why-not RkNN query needs to ex-

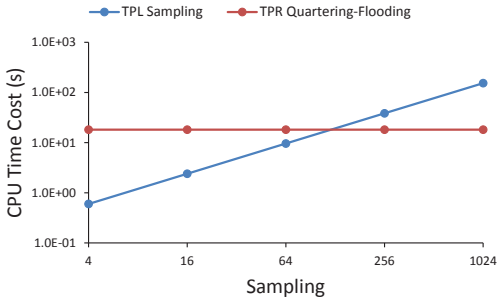


Fig. 13 TPR Quartering-Flooding vs. TPL Sampling: CPU Time(s) (Uniform, k=4, size of R: $400m^2$, λ_q=0.5, error margin $\varepsilon = 5m$)

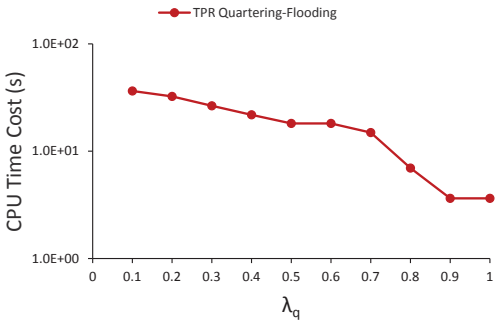
plore a broadness range instead of the original R, our TPR Quartering-Flooding algorithm is far more efficiency.

6.3.3 Effect Of User Preference: λ_S And λ_q

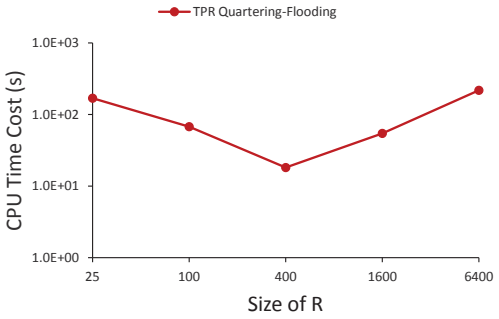
Fig.14a illustrates the performance of TPR Quartering-Flooding algorithm as a function of λ_S, λ_q for uniform distribution dataset. We set $\lambda_S = 1 - \lambda_q$, fix k=4, size of R=$400m^2$ and the error margin $\varepsilon = 5m$. As λ_q increasing, the CPU Time of our proposed algorithm tend to decrease. This is because a bigger λ_q indicates the user's unwillingness to change her position and thus shrink our explored range and result in the lower cost.

6.3.4 Effect Of Size Of R

Fig.14b illustrates the performance of TPR Quartering-Flooding algorithm as a function of original size of R for CD dataset. In this experiment we fix k=4, λ_q=λ_S=0.5 and the error margin $\varepsilon = 5m$. As expected, either a big or a small R are all lead to higher CPU Time cost. A small range R leads to endless flooding: explore the ranges around R and operate the TPR algorithm to compute their RkNNs, over and over. A big range R leads to endless quartering: divide the range R and then compute the RkNNs of these subranges, over and over. As discussed in Section 6.2, the size of a range R does not have a significant effect on the TPR performance, so it's a balance between quartering and flooding to determine the size of initial R in the why-not RkNN query.



(a) Uniform, k=4, size of R: $400m^2$, error margin $\varepsilon = 5m$



(b) CD, k=4, λ_q=0.5, error margin $\varepsilon = 5m$

Fig. 14 Varying λ and size of R

7 Conclusions And Future Work

In this paper we have discussed the problem of range-based RkNN queries and why-not RkNN queries. We have proposed algorithms for *exact* processing of range-based RkNN with *arbitrary* values of k on *dynamic* datasets, based on this, we proposed an algorithm to solve the why-not RkNN query. In particular, we extensively conduct experiments and our experimental results demonstrate that our proposed methods outperform the straightforward method in all aspects, and are superior to exist methods in terms of the efficiency and the accuracy. In the future, we intend to extend this work to the range-based RkNN query with *irregular* range, which retrieves all the points that have any position in the given irregular range as one of their k nearest neighbors.

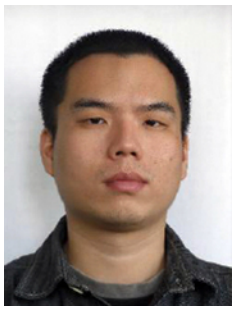
Acknowledgements This research is funded by the National Natural Science Foundation of China (No.61773167), the Natural Science Foundation of Shanghai (No.17ZR1444900) and the Science and Technology Commission of Shanghai Municipality (No.17511102702). The computation is performed in the Supercomputer Center of ECNU.

References

1. Korn F, Muthukrishnan S. Influence sets based on reverse nearest neighbor queries. Journal of Special Interest Group on Management Of Data, 2000, 29(2):201-212.
2. Tran Q T, Chan C Y. How to conquer why-not questions. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 2010, 15-26.

3. He Z, Lo E. Answering why-not questions on top-k queries. In: Proceedings of International Conference on Data Engineering. 2012, 750-761.
4. Benetis R, Jensen C, Karciauskas G, Saltenis S. Nearest neighbor and reverse nearest neighbor queries for moving objects. Journal of International Database Engineering Applications Symposium, 2002, 15(3):44-53.
5. Stanoi I, Riedewald M, Agrawal D, Abbadi A. Discovery of influence sets in frequently updated databases. In: Proceedings of Very Large Data Bases. 2001, 99-108.
6. Korn F, Muthukrishnan S, Srivastava D. Reverse nearest neighbor aggregates over data streams. In: Proceedings of Very Large Data Bases. 2002, 814-825.
7. Yang C, Lin K. An index structure for efficient reverse nearest neighbor queries. In: Proceedings of International Conference on Data Engineering. 2001, 485-492.
8. Maheshwari A, Vahrenhold J, Zeh N. On reverse nearest neighbor queries. Journal of Canadian Conference on Computational Geometry, 2002, 17(1):63-95.
9. Stanoi I, Agrawal D, Abbadi A. Reverse nearest neighbor queries for dynamic databases. Journal of Special Interest Group on Management Of Data, 2000, 29(5):44-53.
10. Singh A, Ferhatosmanoglu H, Tosun A. High dimensional reverse nearest neighbor queries. In: Proceedings of Conference on Information Knowledge Management. 2003, 91-98.
11. Tao Y, Papadias D, Lian X. Reverse kNN search in arbitrary dimensionality. In: Proceedings of Very Large Data Bases. 2004, 744-755.
12. Chapman A, Jagadish H V. Why not? In: Proceedings of the ACM SIGMOD International Conference on Management of Data, 2009, 523-534.
13. Islam M S, Zhou R, Liu C. On answering why-not questions in reverse skyline queries. In: Proceedings of International Conference on Data Engineering, 2013, 973-984.
14. Gao Y, Liu Q, Chen G, Zheng B, Zhou L. Answering why-not questions on reverse top-k queries. In: Proceedings of Very Large Data Bases. 2015, 738-749.
15. Liu Q, Gao Y, Chen G, Zheng B, Zhou L. Answering why-not and why questions on reverse top-k queries. Journal of Very Large Data Bases, 2016, 25(6):867-892.
16. Miao X, Gao Y, Guo S, Chen G. On efficiently answering why-not range-based skyline queries in road networks. To appear in TKDE, 2018.
17. Zhao J, Gao Y, Chen G, Chen R. Why-not questions on top-k geo-social keyword queries in road networks. In: Proceedings of International Conference on Data Engineering. 2018, 965-976.
18. Chen L, Gao Y, Wang K, Jensen C S, Chen G. Answering why-not questions on metric probabilistic range queries. In: Proceedings of International Conference on Data Engineering. 2016, 767-778.
19. Andrew J K, Brad A M. Designing the whyline: a debugging interface for asking questions about program behavior. In: Proceedings of Sigchi Conference on Human Factors in Computing Systems. 2004, 151-158.
20. Brad A M, David A W, Andrew J K, Duen H C. Answering why and why not questions in user interfaces. In: Proceedings of Sigchi Conference on Human Factors in Computing Systems. 2006, 397-406.
21. Huang J, Chen T, Doan A, Naughton J F. On the provenance of non-answers to queries over extracted data. Journal of Very Large Data Bases, 2008, 1(1):736-747.
22. Herschel M, Hernandez M A. Explaining missing answers to spjua queries. Journal of Very Large Data Bases, 2010, 3(1):185-196.
23. Chen L, Xu J, Jensen C S, Li Y. YASK: A why-not question answering engine for spatial keyword query services. Journal of Very Large Data Bases, 2016, 9(13):1501-1504.
24. Chen L, Lin X, Hu H, Jensen C S, Xu J. Answering why-not questions on spatial keyword top-k queries. In: Proceedings of International Conference on Data Engineering. 2015, 279-290.
25. Chen L, Xu J, Lin X, Jensen C S, Hu H. Answering why-not spatial keyword top-k queries via keyword adaption. In: Proceedings of International Conference on Data Engineering. 2016, 697-708.
26. Beckmann N, Kriegel H P, Schneider R, Seeger B. The R*-tree: An effcient and robust access method for points and rectangles. In: Proceedings of the ACM SIGMOD International Conference on Management of Data. 1990, 322-331.

Zhefan Zhong received the BE degree in 2011 and is currently a PhD candidate of computer science in East China Normal University, China. His research interests include location-based services, spatial databases, especially the optimization of query processing in sophisticated databases.



Xin Lin received the PhD degree in computer science and engineering from Zhejiang University, China. He is currently a professor in the Department of Computer Science, East China Normal University. His research interests include location-based services, spatial databases, and privacy-aware compute.

Liang He is a Professor in the Department of Computer Science, East China Normal University. His current research interests mainly lie in big data analysis and knowledge processing. In the past years, he has published more than 70 research papers in international conferences and journals.