Influence Sets Based on Reverse Nearest Neighbor Queries

Flip Korn
AT&T Labs-Research
flip@research.att.com

S. Muthukrishnan
AT&T Labs-Research
muthu@research.att.com

Abstract

Inherent in the operation of many decision support and continuous referral systems is the notion of the "influence" of a data point on the database. This notion arises in examples such as finding the set of customers affected by the opening of a new store outlet location, notifying the subset of subscribers to a digital library who will find a newly added document most relevant, *etc.* Standard approaches to determining the influence set of a data point involve range searching and nearest neighbor queries.

In this paper, we formalize a novel notion of influence based on reverse neighbor queries and its variants. Since the nearest neighbor relation is not symmetric, the set of points that are closest to a query point (*i.e.*, the nearest neighbors) differs from the set of points that have the query point as their nearest neighbor (called the reverse nearest neighbors). Influence sets based on reverse nearest neighbor (RNN) queries seem to capture the intuitive notion of influence from our motivating examples.

We present a general approach for solving RNN queries and an efficient R-tree based method for large data sets, based on this approach. Although the RNN query appears to be natural, it has not been studied previously. RNN queries are of independent interest, and as such should be part of the suite of available queries for processing spatial and multimedia data. In our experiments with real geographical data, the proposed method appears to scale logarithmically, whereas straightforward sequential scan scales linearly. Our experimental study also shows that approaches based on range searching or nearest neighbors are ineffective at finding influence sets of our interest.

1 Introduction

A fundamental task that arises in various marketing and decision support systems is to determine the "influence" of a data point on the database, for example, the influence of a new store outlet or the influence of a new document to a repository. The concept of influence depends on the application at hand and is often difficult to formalize. We first develop an intuitive notion of influence sets through examples to motivate our formalization of it. The following two examples are drawn from spatial domains.

Example 1 (Decision Support Systems): There are many factors that may contribute to a clientele adopting one outlet over another, but a simple premise is to base it on the geographical proximity to the customers. Consider a marketing application in which the issue is to determine the business impact of opening an outlet of Company A at a given location. A simple task is to determine the segment of A's customers who would be likely to use this new facility. Alternatively, one may wish to determine the segment of customers of Company B (say A's competitor) who are likely to find the new facility more convenient than the locations of B. Such segments of customers are loosely what we would like to refer to as *influence sets.* □

Example 2 (Continuous Referral Systems): Consider a referral service wherein a user can specify a street address, and the system returns a list of the five closest FedEx^{TM} drop-off locations.[1] A responsible referral service may wish to give the option (*e.g.*, by clicking a button) to make this a *continuous query*, that is, to request the system to notify the user when this list changes. The referral service will then notify those users whose list changes due to the opening of a closer FedEx drop-off location or the closing of an existing one. When such an event happens, the users who need to be updated correspond to our notion of the influence set of the added or dropped location. □

Both examples above reinforce the notion of the influence set of a data point in terms of geographical proximity. This concept of influence sets is inherent

[1] See `http://www.fedex.com/us/dropoff` for a realization of this.

in many other decision support situations and referral services for which there is no underlying spatial or geographical distance, but for which there is a notion of similarity based on the vector space model (in which "distance" between vectors is taken as a measure of dissimilarity). The following two examples provide illustration.

Example 3 (Profile-based Marketing): A company may wish to keep profiles of its customers' interests so that it can gear a new service towards most customers. For example, suppose AT&T launches a new wireless service. The service may be abstracted a feature vector (*e.g.*, covers New England area, free local calling on weekends, best for \$100-per-month users). The issue is which customers will find this the most suitable plan for their calling patterns; these customers form the influence set of the new service. One approach is to identify such users based on the distance between their profiles and the feature vector representing the new service. □

Example 4 (Maintaining Document Repositories): Consider a repository of technical reports. When a new report is filed, it may be desirable to alert the authors of other TRs who would likely find the document interesting based on similarity to their publications; the set of all such authors corresponds to the notion of influence set we have been developing so far. Here, the influence set is defined based on the similarity between text documents which has been well-explored in the Information Retrieval community. Other similar scenarios abound, such as in a repository of Web pages, precendent legal cases, *etc.* □

Let us now make the notion of an influence set more precise. We start with a data set S, some suitable definition of distance between points in S, and a query point q; the goal is to find the subset of points in S influenced by q. Two suggestions present themselves immediately. The first is to use range queries wherein one specifies a threshold radius ϵ from q, and all points within ϵ are returned. The second is to use the well known concept of *nearest neighbors* (NN), or, more generally, k-nearest neighbors wherein one specifies k, and the k closest points to q are returned.

Both of these suggestions fall short of capturing the intuitive notion of influence we have so far developed. In both cases, parameters have to be engineered to yield an appropriate result size, and it is not obvious how to choose a value without *a priori* knowledge of the local density of points. Range queries may be appropriate for other notions of influence (*e.g.*, the opening of a toxic waste dump on its surrounding population) but not for what is required in the examples given above. NN queries are commonly used in domains which call for searching based on proximity; however, they are not appropriate in this context for similar reasons. Consider Example 1, in which one wants to find potential customers for a new store outlet q. The deciding factor is not how close a customer is to q, but rather if the customer is further from every *other* store than from q. Thus, it may very well be the case that potential customers lie outside a small radius from q, or are further from q than the first few nearest neighbors. Expanding the search radius will not necessarily work around this problem. Although it may encompass more customers who are likely to be influenced by q, it may do so at the trade-off of introducing many customers who are not in the influence set (*i.e.*, customers whose closest store is *not* q). Later, we will make these discussions more concrete and present quantitative measures of comparison (see Section 6).

We address these shortcomings and develop a notion of influence set with broad applications. A fundamental observation which is the basis for our work here is that the nearest neighbor relation is *not symmetric.* For example, if p is the nearest neighbor of q, then q need not be the nearest neighbor of p (see Figure 1).[2] Note that this is the case even though the underlying distance function is Euclidean and, hence, symmetric. Similarly, the k-nearest neighbor relation is not symmetric. It follows that, for a given query point q, the nearest neighbors of q may differ substantially from the set of all points for which q is a nearest neighbor. We call these points the *reverse nearest neighbors* of q.

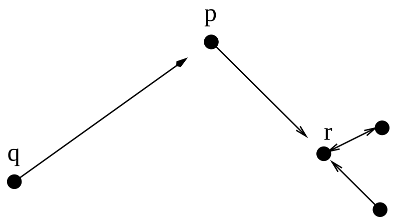


Figure 1: Nearest neighbors need not be symmetric: the NN of q is p, whereas the NN of p is r. (An arrow from point i to point j indicates that j is the nearest neighbor of i.)

We now summarize our contributions:

- We identify a natural and broadly applicable notion for the "influence" of a data point on the database (namely, the *influence set*), and formalize it based on reverse nearest neighbors (RNN) and its variants (such as reverse k-nearest neighbors, reverse furthest neighbor, *etc.*);

[2] That is, provided there are other points in the collection.

- We present a general approach for determining reverse nearest neighbors. Our approach is geometric, reducing the problem to that of testing the enclosure of points in geometric objects; it works for different distance functions and variants of RNNs. Although the RNN query appears to be natural, it has not been studied previously. RNN queries are of independent interest, and as such should be part of the suite of available queries for processing spatial and multimedia data;

- Based on our approach, we propose efficient and scalable R-tree based methods for implementing reverse nearest neighbor queries. We also perform an experimental study of the I/O-efficiency of the proposed R-tree based methods. Using our approach, we show in terms of standard precision and recall measures to assess the output quality, that well known database queries (range and nearest neighbor queries) are *not* effective in finding influence sets.

The structure of the paper is as follows. Section 2 defines RNN queries and describes its relationship to NN queries. Section 3 presents an approach and algorithmic framework for answering RNN queries; we also propose a scalable method for implementating this framework using R-trees in Section 4. Section 5 gives empirical results from experiments for RNN queries. In Section 6, we formalize the basic notion of influence sets based on RNN queries and give results from a qualitative study of the effectiveness of well known queries to substitute for RNN queries. Then we develop the variants of RNN queries needed for generalized notions of influence sets. Section 7 reviews the related work. Section 8 lists the conclusions and gives directions for future work.

2 Reverse Nearest Neighbor Queries

Reverse nearest neighbor (RNN) queries are the basis for influence sets, and are also of independent interest. We define and develop them in this section. We start from the definition of the nearest neighbor (NN) query, a standard query in spatial and multimedia databases and define the RNN query and its variants based on this. We will develop the underlying concepts in two dimensions for simplicity; there will be no difficulty in extending them to higher dimensions. In our discussion, we shall assume the distance between any two points $p = (p_x, p_y)$ and $q = (q_x, q_y)$ is $d(p, q) = (q_x - p_x)^2 + (q_y - p_y)^2$, known as the Euclidean, or L_2, distance.[3]

2.1 Formal Definitions

Suppose we have a collection S of points in the plane. For a nearest neighbor query, we are given a query point q, and the goal is to determine the nearest neighbor set $\mathcal{NN}(q)$ defined as

$$\mathcal{NN}(q) = \{r \in S \mid \forall p \in S : d(q, r) \leq d(q, p)\}.$$

Our focus here is on the inverse relation among the points. Given any query point q, we need to determine the set $\mathcal{RNN}(q)$ of reverse nearest neighbors, defined as

$$\mathcal{RNN}(q) = \{r \in S \mid \forall p \in S : d(r, q) \leq d(r, p)\}.$$

$\mathcal{RNN}(q)$ may be empty, or have one or more elements, and we may wish to return any one of them, or the entire list.

2.2 Variants

There are two variants of this basic scenario that are of interest to us. We will define only the variants for RNN queries, although the corresponding variants of NN queries may also be of interest.

• *Monochromatic vs Bichromatic.* In some applications, the points in S are of two different categories, such as clients and servers; the points may therefore be thought of as being colored *red* or *blue*. The RNN query now consists of a point in one of the categories, say blue, and must determine the red points for which the query point is the closest blue point. Formally, let B denote the set of blue points and R the set of red points. Consider a blue query point q. We have,

$$\mathcal{RNN}(q) = \{r \in R \mid \forall p \in B : d(r, q) \leq d(r, p)\}.$$

We call this the *bichromatic* version; in contrast, the basic scenario above wherein all points were of the same category is the *monochromatic* version. Both versions of the problem are of interest.

At first look, the mono and bichromatic versions of the RNN problem seem very similar. For a blue query point, we consider only the red points and their distance to the closest blue point (vice versa for the red query points). However, at a deeper level, there is a fundamental difference. Let us focus on the L_2 case.

Proposition 1 *For any query point, $\mathcal{RNN}(q)$ may have at most 6 points in the monochromatic case; in the bichromatic case, the size of the set $\mathcal{RNN}(q)$ may be unbounded.*

A proof of this may be found in [17]. From a combinatorial viewpoint, the output of RNN queries is bounded; this in turn affects the efficiency because a RNN query is output-sensitive. This entire phenomenon is not restricted to the plane (*e.g.*, in three dimensions, the

[3] Other L_p distances may also be interest, for example L_1 where $d(p, q) = |q_x - p_x| + |q_y - p_y|$ or L_∞ where $d(p, q) = \max\{|q_x - p_x|, |q_y - p_y|\}$.

$\mathcal{RNN}(q)$ contains at most 12 points under L_2 distance and so on), or the distance function (*e.g.*, in the L_∞ case, the cardinality of $\mathcal{RNN}(q)$ is at most $3^d - 1$ in d dimensions).

• *Static vs Dynamic.* Sometimes we wish to insert or delete points from the set S and still support the RNN query; we refer to this as the *dynamic* case. In contrast, the case when set S is not modified is called the *static* case. The dynamic case is relevant in most applications. The crux here, as in all dynamic problems, is to be able to handle insertions and deletions efficiently without rebuilding the entire data structure.

3 Our Approach to RNN Queries

Our approach for solving the reverse nearest neighbors query problem is quite general, and it applies also to its variants as we shall see.

3.1 Static Case

For exposition, let us consider a basic version of the problem. We are given a set S of points which is not updated, and the distance between any two points is measured using Euclidean distance. Our approach involves two steps.

Step 1. For each point $p \in S$, determine the distance to the nearest neighbor of p in S, denoted $N(p)$. Formally, $N(p) = \min_{q \in S - \{p\}} d(p, q)$. For each $p \in S$, generate a circle $(p, N(p))$ where p is its center and $N(p)$ its radius. (See Figure 2(a) for an illustration.)

Step 2. For any query q, determine all the circles $(p, N(p))$ that contain q and return their centers p.

We have not yet described how to perform the two steps above, but we will first prove that they suffice.

Lemma 1 *Step 2 determines precisely all the reverse nearest neighbors of q.*

Proof. If point p is returned from Step 2, then q falls within the circle $(p, N(p))$. Therefore, the distance $d(p, q)$ is smaller than the radius $N(p)$. In other words, $d(p, q) \leq N(p)$ and hence q is the nearest neighbor of p (equivalently, p is a reverse nearest neighbor of q). Conversely, if p is the reverse nearest neighbor of q, $d(p, q) \leq N(p)$ and, therefore, q lies within the circle $(p, N(p))$. Hence, p will be found in Step 2. □

What our approach has achieved is to reduce the problem of answering the reverse nearest neighbor query to the problem of finding all nearest neighbors (Step 1) and then to what is known in the literature as *point enclosure problems* wherein we need to determine all the objects that contain a query point (Step 2).

Our approach is attractive for two reasons. First, both steps are of independent interest and have been studied in the literature. They have efficient solutions, as we will see later. Second, our approach extends to the variants of our interest as we show below.

Other distance functions. If the distance function is L_∞ rather than L_2, we generate squares $(p, N(p))$ in Step 1 with center p and sides $2N(p)$. (See Figure 2(b) for an illustration.) Similarly, for other L_p distance functions, we will have suitable geometric shapes.

Bichromatic version. Consider only blue query points for now. We perform the two steps above only for the red points in set S. For each red point $p \in S$, we determine $N(p)$, the distance to the nearest blue neighbor. The rest of the description above remains unchanged. We also process for red query points analogously.

3.2 Dynamic Case

Our description above was for the static case only. For the dynamic case, we need to make some modifications. Below we assume the presence of a (dynamically maintained) data structure for answering NN queries. Recall the definition of $N(p)$ for point p from the previous section. Consider an insertion of a point q (as illustrated in Figure 3(a)):

1. Determine the reverse nearest neighbors p of q. For each such point p, we replace circle $(p, N(p))$ with $(p, d(p, q))$, and update $N(p)$ to equal $d(p, q)$;

2. Find $N(q)$, the distance of q from its nearest neighbor, and add $(q, N(q))$ to the collection of circles.

Lemma 2 *The insertion procedure is correct.*

Proof. It suffices to argue that, for each point p, $N(p)$ is the correct distance of p to its nearest neighbor after an insertion. This clearly holds for the inserted point q from Step 2. Among the rest of the points, the only ones which will be affected are those which have q as their nearest neighbor, in other words, the reverse nearest neighbors of q. For all such points p, we update their $N(p)$'s appropriately in Step 1. The remaining points p do not change $N(p)$ as a result of inserting q. Hence, all points p have the correct value of $N(p)$. □

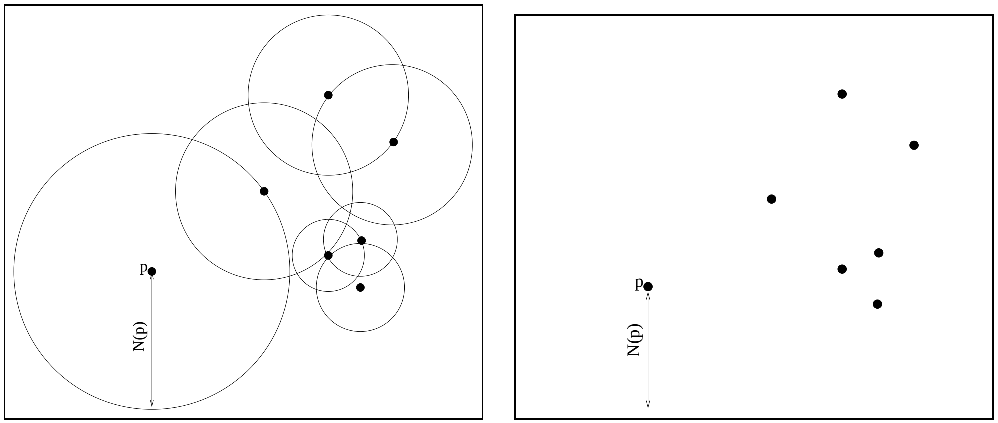
Step 1 is shown in Figure 3(b) where we shrink all circles $(p, N(p))$ for which q is the nearest neighbor of p to $(p, d(p, q))$. Step 2 is shown in Figure 3(c).

Now consider an deletion of a point q (as illustrated in Figure 4(a)):

1. We need to remove the circle $(q, N(q))$ from the collection of circles (see Figure 4(b));

2. Determine all the reverse nearest neighbors p of q. For each such point p, determine its current $N(p)$ and replace its existing circle with $(p, N(p))$.

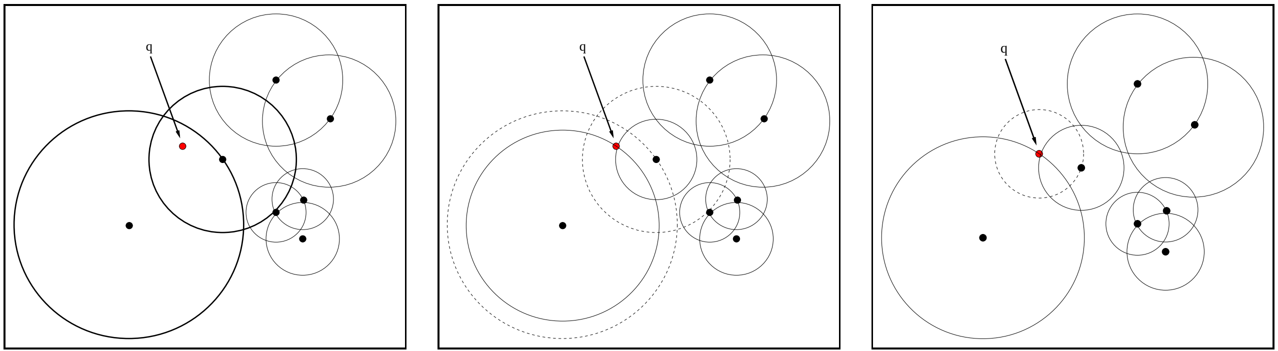
We can argue much as before that the deletion procedure is correct. The crucial observation is that



(a) L_2 case

(b) L_∞ case

Figure 2: A point set and its nearest neighborhoods.



(a) find $\mathcal{RNN}(q)$

(b) shrink circles

(c) find $\mathcal{NN}(q)$

Figure 3: A geometrical illustration of the insertion algorithm.

the only existing circles $(p, N(p))$ that get affected are those that have q on the circumference, that is, those associated with the reverse nearest neighbors of q; their circles get expanded in Step 1 (see Figure 4(c)). The details for how to extend these algorithms to other distance functions and to the bichromatic version are similar to those given in the previous section.

4 Scalable RNN Queries

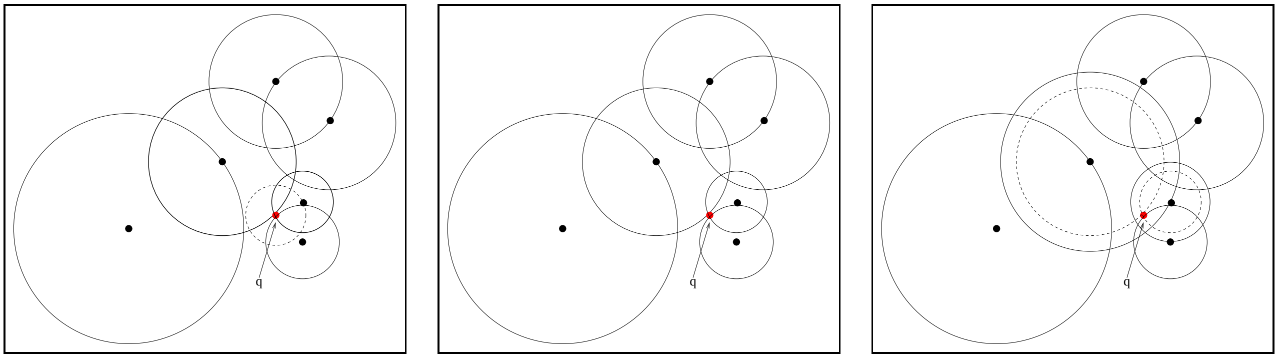
In this section we propose a scalable method for implementing RNN queries on large, out-of-core data sets, based on our approach from Section 3. Like NN queries, RNN queries are I/O-bound (as opposed to, *e.g.*, spatial joins which are CPU-bound), and thus the focus is on I/O performance. Because R-trees [7, 2, 16] have been successfully deployed in spatial databases and because of their generality to support a variety of norms via bounding boxes, we use them in the proposed method. However, note that any spatial access method could be employed (see [6] for a recent survey of spatial access methods). Our deployment of R-trees is standard, but requires some elaboration. First we describe static RNN search; we then present details of the algorithms and data structures for the dynamic case.

4.1 Static Case

The first step in being able to efficiently answer RNN queries is to precompute the nearest neighbor for each and every point. The problem of efficiently computing all-nearest neighbors in large data sets has been studied in [3, 8], and thus we do not investigate it further in this paper.[4]

Given a query point q, a straightforward but naive approach for finding reverse nearest neighbors is to sequentially scan through the entries $(p_i \rightarrow p_j)$ of a precomputed all-NN list in order to determine which points p_i are closer to q than to p_i's current nearest

[4] All-nearest neighbors is a special case of a spatial join.



(a) remove $\mathcal{NN}(q)$ (b) find $\mathcal{RNN}(q)$ (c) expand circles

Figure 4: A geometrical illustration of the deletion algorithm.

neighbor p_j. Ideally, one would like to avoid having to sequentially scan through the data.

Based on the approach in Section 3, a RNN query reduces to a point enclosure query in a database of nearest neighborhood objects (*e.g.*, circles for L_2 distance in the plane); these objects can be obtained from the all-nearest neighbor distances. We propose to store the objects explicitly in an R-tree. Henceforth, we shall refer to this instantiation of an R-tree as an *RNN-tree*. Thus, we can answer RNN queries by a simple search in the R-tree for those objects enclosing q.

4.2 Dynamic Case

As mentioned in Section 4.1, a sequential scan of a precomputed all-NN list can be used to determine the reverse nearest neighbors of a given point query q. Insertion and deletion can be handled similarly. Even if this list were inverted, enabling deletion to be achieved in constant time by looking up the corresponding entry ($p_j \rightarrow \{p_{i_1}, p_{i_2}, \ldots, p_{i_k}\}$), queries and insertions would still require a pass over the data. We would like to avoid having to do this.

We describe how to incrementally maintain the RNN-tree in the presence of insertions and deletions. To do this will require a supporting access method that can find nearest neighbors of points efficiently. At this point, one may wonder if a single R-tree will suffice for finding reverse nearest neighbors as well as nearest neighbors, in other words, if our RNN-tree can be used for this purpose, This turns out to be not the case since geometric objects rather than points are stored in the RNN-tree, and thus the bounding boxes are not optimized for nearest neighbor search performance on points. Therefore, we propose to use a separate R-tree for NN queries, henceforth referred to as an *NN-tree*. Note that the NN-tree is *not* needed for static RNN queries, only for insertions and deletions, and that, in addition to the RNN-tree, it must be dynamically maintained.

```
Algorithm Insert:
Input: point $q$

$\{p_1, p_2, \ldots, p_k\} \leftarrow$ query $q$ in RNN-tree;
for each $p_i$ (with corresponding $R_i$) do
    shrink $R_i$ to $(p_i, d(p_i, q))$;
find $N(q)$ from NN-tree;
insert $q$ in NN-tree;
insert $(q, N(q))$ in RNN-tree;
```

```
Algorithm Delete:
Input: point $q$

delete $q$ from NN-tree;
$\{p_1, p_2, \ldots, p_k\} \leftarrow$ query $q$ in RNN-tree;
delete $(q, N(q))$ from RNN-tree;
for each $p_i$ (with corresponding $R_i$) do
    find $N(p_i)$ from NN-tree;
    grow $R_i$ to $(p_i, N(p_i))$;
```

Figure 5: Proposed Algorithms for Insertion and Deletion.

Figure 5 presents pseudocode for insertion and deletion. The algorithm for insertion retrieves (from the RNN-tree) the reverse nearest neighbors p_i of q, and their corresponding neighborhood objects R_i, without having to scan; each R_i is then reduced in size to $(p_i, d(p_i, q))$. The algorithm for deletion works similarly, using the RNN-tree to find the points p_i affected by the deletion; each corresponding R_i is then expanded to $(p_i, d(p_i, N(p_i))$.

5 Experiments on RNN queries

We designed a set of experiments to test the I/O performance of our proposed method on large data sets. Our goal was to determine the scale-up trend of both static and dynamic queries. We also examined the performance of bichromatic versus monochromatic data. Below we present results from two batches of experiments, for static and dynamic RNN queries.

Methods: We compared the proposed algorithms given in Section 4 to the basic scanning approach. In the static case, the scanning approach precomputes an all-NN list and makes a pass through it to determine the reverse nearest neighbors. In the dynamic case, the scanning approach precomputes and maintains an inverted all-NN list. Each entry in the all-NN list corresponds to a point in the data set, and thus requires storing two items for nearest neighbor information: the point coordinates and nearest neighbor distances. Similarly, the RNN-tree used in the proposed method requires storing each point and its associated nearest neighborhood. Both also use an NN-tree for nearest neighbor search. Thus, the methods require the same storage space.

Data Sets: Our testbed includes two real data sets. The first is mono and the second is bichromatic:

- `cities1` - Centers of 100K cities and small towns in the USA (chosen at random from a larger data set of 132K cities), represented as latitude and longitude coordinates;
- `cities2` - Coordinates of 100K red cities (*i.e.*, clients) and 400 black cities (*i.e.*, servers). The red cities are mutually disjoint from the black cities, and points from both colors were chosen at random from the same source.

Queries: We assume the so-called 'biased' query model, in which queries are more likely to come from dense regions [13]. We chose 500 query points at random (without replacement) from the same source that the data sets were chosen; note that these points are external to the data sets. For dynamic queries, we simulated a mixed workload of insertions by randomly choosing between insertions and deletions. In the case of insertions, one of the 500 query points were inserted; for deletions, an existing point was chosen at random. We report the average I/O per query, that is, the cumulative number of page accesses divided by the number of queries.

Software: The code for our experiments was implemented in `C` on a Sun SparcWorkstation. To implement RNN queries, we extended DR-tree, a disk-resident R*-tree package; to implement NN queries (which were used for the second batch of experiments), we used the DR-tree as is.[5] The page size was set to 4K.

5.1 Static Case

We uniformly sampled the `cities1` data set to get subsets of varying sizes, between 10K and 100K points. Figure 6(a) shows the I/O performance of the proposed method compared to sequential scan. Each query took roughly between 9-28 I/Os for the data sets we tried with our approach; in contrast, the performance of the scanning approach increased from 40 to 400 I/Os with increasing data set size (n). The gap between the two curves clearly widens as n increases, and the proposed method appears to scale logarithmically, whereas the scanning approach scales linearly.

We performed the same experiment for `cities2`. Figure 6(b) plots the I/O performance. It is interesting to note that the performance degrades more with increasing n (from 12-65 I/Os) with bichromatic data; this is primarily because the output size is larger in bichromatic case than in the monochromatic case as remarked earlier. However, this increase again appears to be logarithmic.

5.2 Dynamic Case

Again, we used the `cities1` data set and uniformly sampled it to get subsets of varying sizes, between 10K and 100K points. As shown in Figure 7, the I/O cost for an even workload of insertions and deletions appears to scale logarithmically, whereas the scanning method scales linearly. It is interesting to note that the average I/O is up to four times worse than in the static case, although this factor decreases for larger data sets. We broke down the I/O into four categories – RNN queries, NN queries, insertions and deletions – and found that each took approximately the same number of I/Os. Thus, the maintenance of the NN-tree accounts for the extra I/O compared to the static queries.

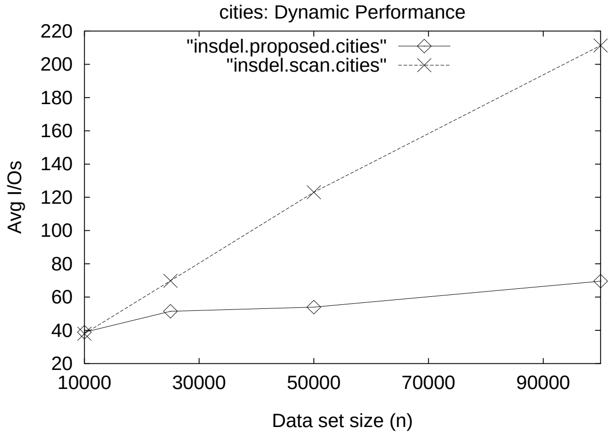
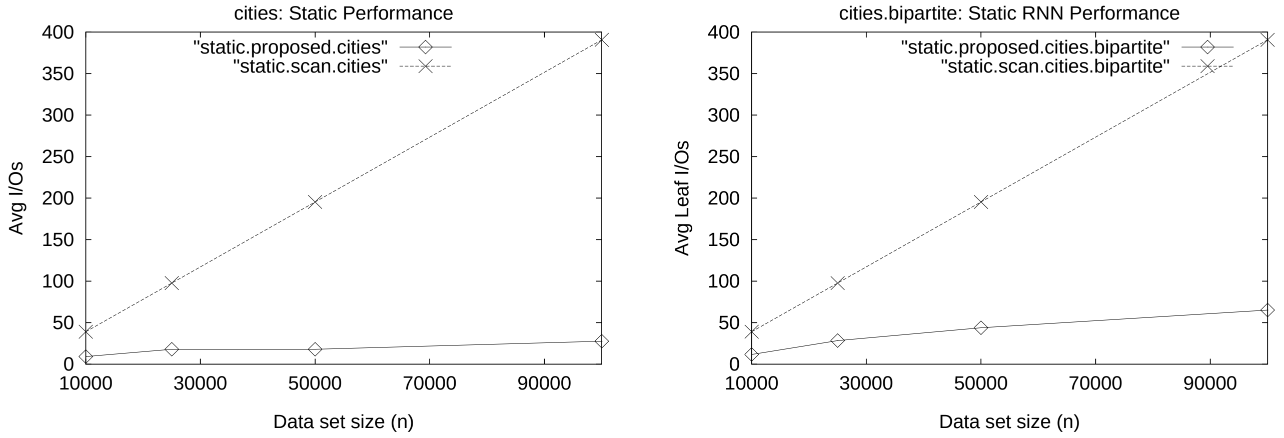


Figure 7: The I/O performance of dynamic RNN queries (proposed method vs. scanning) in the presence of an even mix of insertions and deletions.

[5] available at `ftp://ftp.olympos.umd.edu`.



(a) `cities1` (monochromatic)

(b) `cities2` (bipartite)

Figure 6: The I/O performance of static RNN queries (proposed method vs. scanning) for (a) `cities1` (monochromatic) and (b) `cities2` (bipartite).

6 Influence Sets

6.1 Basic notion and applications

Our first, and most basic, definition of the influence set of a point q is simply that it is the set of all reverse nearest neighbors of q, that is, $\mathcal{RNN}(q)$. This may be mono or bichromatic reverse nearest neighbors, depending on the application.

Before exploring this notion further, let us briefly reexamine the motivating examples from Section 1. In Examples 1 and 2, the influence set of the new location of a store outlet is indeed the set of customers who find the new location the closest amongst all locations of stores. This is an instance of bichromatic RNN. In Example 3, the customers who are influenced by a new service are those whose profiles have the feature vector of the new service closest amongst all service feature vectors. Again, the influence set of the new service corresponds to our basic definition above. In Example 4, the influence set of a new document is the set of all documents in the database that find it the closest under a suitable measure of similarity; here, the definition of an influence set based on monochromatic RNNs applies.

We can think of many other applications where the basic notion of influence set arises. What is perhaps more interesting is that this notion of influence sets *implicitly* arises in many computational tasks.

For example, many problems of interest in Operations Research and Combinatorial Optimization have greedy solutions with good performance. One such example is the *facility location* problem. Here we are given many points and the goal is to designate some as facilities and others as non-facilities. There is a cost to designating a point as a facility, and a cost for non-facilities which equals the cost of accessing the closest facility. This problem is known to be NP-hard, and thus the focus is on designing approximation algorithms for this problem. The method of choice in practice for this problem is the greedy method – it is simple, and is a provably small approximation [14].[6] The greedy algorithm involves repeatedly adding a facility, deleting one, or swapping a facility with a non-facility. In order to implement this algorithm, we need to determine the enhanced cost when a new facility is added which involves looking at precisely those locations whose NN distance is changed when a new facility is added (or deleted, swapped). The set of all such locations is indeed our basic definition of a influence set; these have been implicitly computed in this context for a long time. Another example is that of computing the shortest path from a single point to every other point in the database. When a point is added to a partial solution that greedy algorithms maintain, the distance of remaining points to the partial solution has to be updated and this will again be given by the influence set of the point added to the partial solution. Many other implicit uses of influence sets exist in Combinatorial Optimization.

6.2 Using existing methods

There are two potential problems with the effectiveness of any approach to finding influence sets. One is the *precision* problem wherein a large portion of the retrieved set contains irrelevant points. Conversely, there is the *recall* problem wherein the retrieved set misses some of the relevant points. An effective approach would achieve high precision at high recall

[6] Better approximations exist, but they are based on Linear Programming [10].

(ideally, 100% precision at 100% recall). In this section we present results from an experiment to demonstrate that nearest neighbor queries and range queries are not effective "engineering" substitutes for RNN queries in finding influence sets; we use standard precision and recall metrics from information retrieval to assess their quality.

The first issue that arises in finding influence sets is what region to search in. Two possibilities immediately present themselves: find the closest points (*i.e.*, the k-nearest neighbors) or all points within some radius (*i.e.*, ϵ-range search). Of course, there are many variants of these basic queries, such as searching with weighted distances, searching over polygonal or elliptical regions, *etc.* To demonstrate the ineffectiveness of these approaches, it shall suffice to consider the most basic version. The question then is how to engineer the parameter value (namely k or ϵ) that will contain the desired information. Without *a priori* knowledge of the density of points near the query point q, it is not clear how to choose these values. Regardless, we show that *any* clever strategy to engineer parameter values (be it from histograms, *etc.*) would still fall short.

Figure 8 illustrates this concept. The black points represent servers and the white points represent clients. In this example, we wish to find all the clients for which q is their closest server. The example illustrates that a ϵ-range (alternatively, k-NN) query cannot find the desired information in this case, regardless of which value of ϵ (or k) is chosen. Figure 8(a) shows a 'safe' radius ϵ_l in which all points are reverse nearest neighbors of q; however, there exist reverse nearest neighbors of q outside ϵ_l. Figure 8(b) shows a wider radius ϵ_h that includes all of the reverse nearest neighbors of q but also includes points which are not. In this example, it is possible to achieve 100% precision or 100% recall, but not both simultaneously.

We ran an experiment to investigate how often this trade-off occurs in practice. The experiment was carried out as follows. Suppose we had an oracle to suggest the largest radius ϵ_l admitting no false-positives, *i.e.*, whose neighborhood contains only points in the influence set. For this scenario, we assess the quality of the retrieved set from the number of false-negatives within this radius. More specifically, we measured the recall at 100% precision, that is, the cardinality of the retrieved set divided by that of the influence set. Further suppose we had an oracle to suggest the smallest radius ϵ_h allowing no false-negatives, *i.e.*, whose neighborhood contains the full influence set (equivalently, reverse nearest neighbors). For this scenario, we assess the quality of the retrieved set from the number of false-positives within this radius. More specficially, we measured the precision at 100% recall, that is, the cardinality of the influence set divided by that of the retrieved set.

We used the `cities2` data set in the our experiment and averaged over 100 queries. The results are summarized in Table 1. The quality of the retrieved set at radius ϵ_l is poor, containing a small fraction of the full influence set. The quality of the retrieved set at radius ϵ_h is also poor, containing a lot of 'garbage' in addition to the influenced points.

measure	*radius*	*value*
precision (at 100% recall)	ϵ_h	44.3%
recall (at 100% precision)	ϵ_l	40.2%

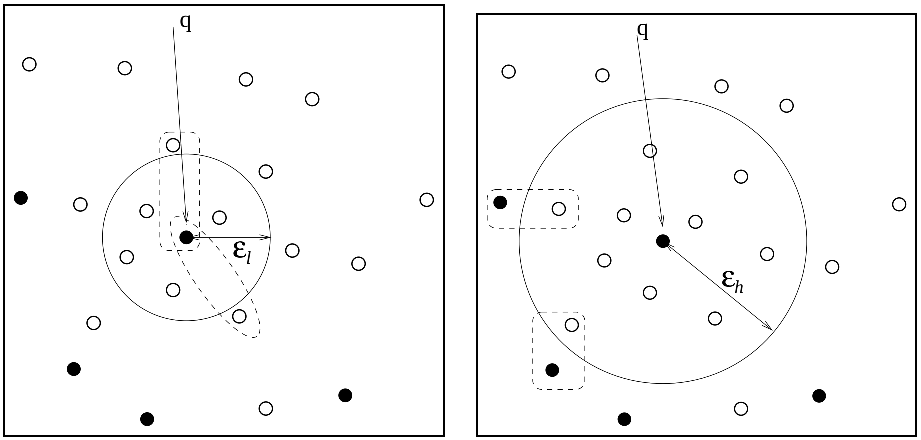
Table 1: The effectiveness of range queries in finding influence sets. Quality is measured by precision at 100% recall and recall at 100% precision.

6.3 Extended notions of influence sets

In this section, we extend the notion of influence sets from the previous section. We do not explore these notions in depth here using experiments; instead we focus on sketching how our approach for finding the basic influence sets can be modified to find these extended influence sets. Some of these modifications will be straightforward, others less so.

Reverse k-nearest neighbors. A rather simple extension of the influence set of point q is to define it to be the set of all points that have q as one of their k nearest neighbors. Here, k is fixed and specified *a priori*. For static queries, the only difference in our solution is that we store the neighborhood of kth neighbor rather than nearest neighbor. (Note that we do not explicitly store the k nearest neighbors.) Each query is an enclosure problem on these objects as in the basic case. For insertions and deletions, we update the neighborhood of the kth nearest neighbor of each affected point as follows. When inserting or deleting q, we first find the set of affected points using the enclosure problem as done for answering queries. For insertion, we perform a range query to determine the k nearest neighbors of each such affected point and do necessary updates. For deletion, the neighborhood radius of the affected points is expanded to the distance of the $(k+1)$th neighbor, which can be found by a modified NN search on R-trees.

Influence sets with predicates. The basic notion of influence sets can be enhanced with predicates. Some examples of predicates involve bounding the search distance (find reverse nearest neighbors within a specified region of interest) and providing multiple facilities (find the reverse nearest neighbors to any, some, or all of multiple points in the set $\{q_1, \ldots, q_m\}$).



(a) range ϵ_l

(b) range ϵ_h

Figure 8: In many cases, any ϵ-range query or k-NN query will be either (a) too small or (b) too big.

For such queries, we can push the predicates inside the R-tree search.

Reverse furthest neighbors. An interesting variation of influence sets is to base it on dissimilarity rather than similarity, in other words, on furthest neighbors rather than nearest neighbors. More formally, define the influence set of a point q to be the set of all points r such that q is farther from r than any other point of the database is from r. This notion of influence has a solution that differs from the basic notion in an interesting way. We sketch the solution here only for the static case, but all modifications to convert this into a dynamic solution are based on ideas we already described before. We will also only describe the solution for the two dimensional case, but extending it to the multi-dimensional case is straightforward.

Say S is the set of points which will be fixed. A query point is denoted q. For simplicity, we will first describe our solution for the L_∞ distance.

Preprocessing: We first determine the furthest point for each point $p \in S$ and denote it as $f(p)$. We will put a square with center p and sides $2d(p, f(p))$ for each p; say this square is R_p.

Query processing: The simple observation is that for any query q, the reverse furthest neighbors r are those for which the R_r does *not* include q. Thus the problem we have is *square non-enclosure* problem. (Recall that, in contrast, the reverse nearest neighbors problem led to square enclosure problem.)

The following observation is the key to solving the square non-enclosure problem.

Lemma 3 *Consider the intervals x_r and y_r obtained by projecting the square R_r on x and y axis respectively. A point $q = (x, y)$ is not contained in R_r if and only if either x_r does not contain x or y_r does not contain y.*

Therefore, if we return all the x_r's that do not contain x as well as those y_r's that do not contain y's, each square r in the output is repeated atmost twice. So the problem can be reduced to a one dimensional problem on intervals without losing much efficiency. Let us restate the one dimensional problem formally: we are given a set of intervals, say N of them. Each query is a one dimensional point, say p, and the goal is to return all interval that do not contain p.

For solving this problem, we maintain two sorted arrays, one of the right endpoints of the intervals and the other of their left endpoints. The following observation is easy:

Proposition 2 *Any interval with its right endpoint to the left of p does not contain p. Likewise, any interval with its left endpoint to the right of p does not contain it.*

Hence, it suffices to perform two binary searches with p in the two arrays, to determine the intervals that do not contain p. Notice that from a practical viewpoint, the only data structure we need is a B-tree to keep these two arrays.

Theorem 1 *There exists an $\Theta(\log_B N + t)$ time algorithm to answer each query in the square non-enclosure problem, where t is the output size, $N = n/B$ is the number of disk blocks, and B is the block size; space used is $\Theta(N)$.*

Although the solution above is simple, we are not aware of any published claim of this result for square non-disclosure problem. As mentioned before, this immediately gives a solution for finding the set of all reverse furthest neighbors for a query point under the L_∞ distance. While a R-tree may be used to solve this problem, our solution above shows that the only data structure we need is to a B-tree. Hence, the solution is very efficient. For other distance functions, we still have non-enclosure problem, but with different shapes (*e.g.*, circles for Euclidean distance). Practical approaches for solving such problems would be to either use bounding boxes to reduce the problem to square non-enclosure with some false positives, or to use R-tree based search.

7 Related Work

There has been a lot of work on nearest neighbor queries [5, 11, 4, 15, 9]. NN queries are useful in many applications: GIS ('*Find the k nearest hospitals from the place of an accident.*'), information retrieval ('*Find the most similar web page to mine.*'), multimedia databases ('*Find the tumor shape that looks the most similar to the query shape.*' [12]), *etc.* Conceptually, a RNN query is different from a NN query; it is the inverse.

There has been work in the area of spatial joins, and more specifically with all-nearest neighbor queries [8]. To the best of our knowledge, none of the previous work has addressed the issue of incremental maintenance. While reverse nearest neighbors is conceptually different, RNN queries provide an efficient means to incrementally maintain all-nearest neighbors.

Both incremental and random incremental Delaunay triangulations could be used for answering RNN queries, as the update step involves identifying points (and their circumcircles) local to the query point whose edges are affected by insertion/deletion, a superset of reverse nearest neighbors.[7] However, these algorithms rely on being able to efficiently locate the simplex containing the query point, a problem for which there is no efficient solution in large data sets. In addition, the algorithms make the general position assumption and do not work well for the bipartite case.

Our approach for RNN queries relied on solving point enclosure problems with different shapes. Point enclosure problems with n rectangles can be solved after $O(n \log^{d-1} n)$ time preprocessing in $O(\log^{d-1} n+t)$ time per query where t is the output size [1]. Such efficient algorithms are not known for other shapes, for dynamic cases, or for external memory datasets. Our R-tree approach is simple and applies to all the variants of RNN queries.

8 Conclusions and Future Work

The "influence" of a point in a database is a useful concept. In this paper, we introduce an intuitive notion of influence based on reverse nearest neighbors, and illustrate through examples that it has broad appeal in many application domains. The basic notion of influence sets depends on *reverse nearest neighbor* queries, which are also of independent interest. We provide the first solution to this problem, and validate its I/O-efficiency through experiments. We also demonstrate using experiments that standard database queries such as range searching and NN queries are ineffective at finding influence sets. Finally, we further extend the notion of influence based on variants of RNN queries and provide efficient solutions for these variants as well.

We have initiated the study of influence sets using reverse nearest neighbors. Many issues remain to be explored, for example, the notion of influence outside of the typical query-response context, such as in data mining. It is often desirable to process the data set to suggest a region in which a query point should lie so as to exert maximal influence. What is the appropriate notion of influence sets in this context? From a technical point of view, efficient solutions for RNN queries are needed in high dimensions. Extensions of our approach to higher dimensions is straightforward; however, in very high dimensions, alternative approaches may be needed, as is the case for high dimensional NN queries. Also, the role of RNN queries can be explored further, such as in other proximity-based problems. It is our belief that the RNN query is a fundamental query, deserving to be a standard tool for data processing.

Acknowledgements

The authors wish to thank Christos Faloutsos, Dimitris Gunopoulos, H.V. Jagadish, Nick Koudas, and Dennis Shasha for their comments.

References

[1] P. Agrawal. Range searching. In E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 575–598. CRC Press, Boca Raton, FL, 1997.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. *ACM SIGMOD*, pages 322–331, May 23-25 1990.

[3] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. In *Proc. of ACM SIGMOD*, pages 237–246, Washington, D.C., May 26-28 1993.

[4] B. Chazelle and L. J. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1:133–162, 1986.

[5] K. Fukunaga and P. M. Narendra. A branch and bound algorithm for computing k-nearest

[7] Alternatively, one could use the dual data structure, Voronoi diagrams.

neighbors. *IEEE Trans. on Computers (TOC)*, C-24(7):750–753, July 1975.

[6] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, June 1998.

[7] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pages 47–57, Boston, Mass, June 1984.

[8] G. R. Hjaltason and H. Samet. Incremental distance join algorithms for spatial databases. *ACM SIGMOD '98*, pages 237–248, June 1998.

[9] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *ACM TODS*, 24(2):265–318, June 1999.

[10] K. Jain and V. Vazirani. Primal-dual approximation algorithms for metric facility location and k-median problems. *Proc. 40th IEEE Foundations of Computer Science (FOCS '99)*, pages 2–13, 1999.

[11] D. G. Kirkpatrick. Optimal search in planar subdivisions. *SIAM J. Comput.*, 12:28–35, 1983.

[12] F. Korn, N. Sidiropoulos, C. Faloutsos, E. Siegel, and Z. Protopapas. Fast nearest-neighbor search in medical image databases. *Conf. on Very Large Data Bases (VLDB)*, pages 215–226, September 1996.

[13] B. Pagel, H. Six, H. Toben, and P. Widmayer. Towards an analysis of range query performance. In *Proc. of ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, pages 214–221, Washington, D.C., May 1993.

[14] R. Rajaraman, M. Korupolu, and G. Plaxton. Analysis of a local search heuristic for facility location problems. *Proceedings of ACM-SIAM Symposium on Discrete Algorithms (SODA '98)*, pages 1–10, 1998.

[15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. of ACM-SIGMOD*, pages 71–79, San Jose, CA, May 1995.

[16] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R+ tree: A dynamic index for multi-dimensional objects. In *Proc. 13th International Conference on VLDB*, pages 507–518, England,, September 1987.

[17] M. Smid. Closest point problems in computational geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry.* Elsevier Science Publishing, 1997.