Lazy Updates: An Efficient Technique to Continuously Monitoring Reverse kNN *

Muhammad Aamir Cheema†, Xuemin Lin†‡, Ying Zhang†, Wei Wang†‡, Wenjie Zhang†‡
†The University of New South Wales, Australia ‡NICTA, Australia
{macheema, lxue, yingz, weiw, zhangw}@cse.unsw.edu.au

ABSTRACT

In this paper, we study the problem of continuous monitoring of reverse k nearest neighbor queries. Existing continuous reverse nearest neighbor monitoring techniques are sensitive towards objects and queries movement. For example, the results of a query are to be recomputed whenever the query changes its location. We present a framework for continuous reverse k nearest neighbor queries by assigning each object and query with a rectangular safe region such that the expensive recomputation is not required as long as the query and objects remain in their respective safe regions. This significantly improves the computation cost. As a by-product, our framework also reduces the communication cost in client-server architectures because an object does not report its location to the server unless it leaves its safe region or the server sends a location update request. We also conduct a rigid cost analysis to guide an effective selection of such rectangular safe regions. The extensive experiments demonstrate that our techniques outperform the existing techniques by an order of magnitude in terms of computation cost and communication cost.

1. INTRODUCTION

Given a query point q, a reverse k nearest neighbor (RkNN) query retrieves all the data points that have q as one of their k nearest neighbors. Throughout this paper, we use RNN queries to refer to RkNN queries for which $k = 1$. We give a more formal definition of the RkNN problem in Section 2. Consider the example of Fig. 1 where the nearest neighbor (the closest object) of q is o_1. However, it is not the RNN of q because the closest point of o_1 is not q. The RNNs of q are o_3 and o_4.

RNN has received considerable attention [10, 16, 1, 15, 12, 17, 19, 26, 25, 21] from database research community based on the applications such as decision support, location based service, resource allocation, profile-based management, etc.

With the availability of inexpensive mobile devices, position locators and cheap wireless networks, location based services are gaining increasing popularity. An example of such location based services is zhiing[1]. Consider that a user needs a taxi and she sends her location to a taxi company's dispatch center. The company notifies to a taxi for which she is the closest passenger (the taxi is RNN of the user).

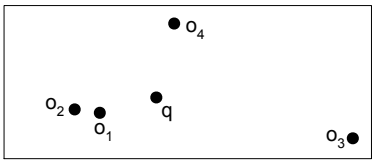


Figure 1: o_3 and o_4 are RNNs of q

Other examples of location based services include location based games, traffic monitoring, location based SMS advertising, enhanced 911 services and army strategic planning etc. These applications may require continuous monitoring of reverse nearest moving objects. For instance, in reality games (e.g., BotFighters, Swordfish), players with mobile devices search for other mobile devices in neighborhood. For example, in the award winning game BotFighters, a player gets points by shooting other nearby players via mobiles. In such an application, some players may want to continuously monitor her reverse nearest neighbors in order to avoid being shot by other players. In the sea-battlefield, a warship may register a continuous RNN query to monitor other warships that might seek assistances from it and then may contact them from time to time.

Driven by such applications, the continuous monitoring of reverse nearest neighbors has been investigated and several techniques have been proposed recently [1, 9, 20, 22] in the light of location-based services. The existing continuous monitoring techniques [1, 9, 20, 22] adopt two frameworks based on different applications. In [1], the velocity of each object is assumed to be explicitly expressed while [9, 20, 22] deal with a general situation where the object velocities may be impossible to be explicitly expressed. In this paper, our research is based on the general situation; that is, object velocities are not explicitly expressible.

The techniques in [9, 20, 22] adopt a two-phase computation. In the *filtering* phase, objects are pruned by using the existing pruning paradigms from [16, 17] and the remaining objects are considered as the candidate objects. In the *verification* phase, every candidate object for which the query is its closest point is reported as the RNN. To update the results, at each time-stamp, if the set of candidate objects is detected to be unchanged then only the verification phase is called to verify the results. Nevertheless, both the filtering and verification phases are required if one of the candidate objects changes its location or other objects

*Xuemin Lin is supported by Australian Research Council Discovery Grants (DP0987557, DP0881035, DP0987273 and DP0666428) and Google Research Award . Wei Wang is supported by ARC Discovery Projects DP0987273 and DP0881779.

VLDB '09, August 24-28, 2009, Lyon, France

[1]http://www.zhiing.com/how.php

move into the candidate region. Similarly, a set of candidate objects is needed to be re-computed (recall filtering) if the query changes its location.

Previous techniques [9, 22, 20] require expensive filtering if a query or any candidate object changes its location. Our initial experiment results show that the cost of verification phase is much lower than the cost of filtering phase. In our technique, we assign each query and object a safe region (a rectangular area). The filtering phase for a query is not required as long as the query and its candidate objects remain in their corresponding safe regions. This significantly reduces the computation time of continuously monitoring RkNN queries.

As a by-product, our proposed framework also significantly reduces the communication cost in a client-server architecture. In the existing techniques, every object reports its location to the server at every time-stamp regardless whether query results will be affected or not. Consequently, such a computation model requires transmission of a large number of location updates; doing this has a direct impact on the wireless communication cost and power consumption - the most precious resources in mobile environment [7]. In our framework, each moving object reports its location update only when it leaves the region. This significantly saves the communication costs. Since we choose a simple safe region shape (a rectangle), our framework can be easily integrated with the work in [7] to develop a safe region based system that supports RkNN, range and kNN queries

Contributions. Below, we summarize our contributions:

1. We present a framework for continuously monitoring RNN together with a novel set of effective pruning and efficient increment computation techniques. It not only reduces the total computation cost of the system but also reduces the communication cost.

2. We extend our algorithm for the continuous monitoring of RkNN. Our algorithm can be used to monitor both *mono-chromatic* and *bichromatic* RkNN (to be formally defined in Section 2.1).

3. We provide a rigid analysis of the relationship between the computation/communication costs and the safe regions. This also guides us to effectively select the safe regions.

4. Our extensive experiments demonstrate that the developed techniques outperform the previous algorithms by an order of magnitude in terms of computation cost and communication cost.

The remaining of the paper is organized as follows. In Section 2, we give the problem statement, related work and motivation. Section 3 presents the framework of our techniques and a set of novel pruning techniques. Section 4 presents our techniques for continuously monitoring RNN queries, as well as a rigid cost analysis. Section 5 gives the extension of our techniques to multidimensional space, to RkNN, and to Bichromatic RkNN. The experiment results are reported in Section 6. Section 7 concludes the paper.

2. BACKGROUND INFORMATION

2.1 Problem Definition

There are two types of RkNN queries [10] namely, *monochromatic* and *bichromatic* RkNN queries. Below we define both.

Monochromatic RkNN query: Given a set of multidimensional points P and a point $q \in P$, a monochromatic RkNN query retrieves points $p \in P$, $dist(p,q) \leq dist(p,p_k)$ where $dist$ is a distance metric that is assumed to be Euclidean distance in this paper, and p_k is the kth nearest point to p in $P - \{q\}$.

Note that, in such queries, both the data objects and the query objects belong to the same class of objects. Consider an example of the reality game BotFighters, where a player issues a query to find other players for whom she is the closest person.

Bichromatic RkNN query: Given two sets O and P each containing different types of objects, a bichromatic RkNN query for a point $q \in O$ is to retrieve every object $p \in P$ such that $dist(p,q) \leq dist(p,o_k)$ where o_k is the kth nearest point of p in $O - \{q\}$.

In contrast to monochromatic queries, the query and data objects belong to two different classes. Consider the example of a battlefield where a medical unit might issue a bichromatic RNN query to find the wounded soldiers for whom it is the closest medical unit.

2.2 Related Work

First, we present pruning techniques for *snapshot* RNN queries. Snapshot RNN queries report the results only once and do not require continuous monitoring.

Snapshot RNN Queries: Korn *et al.* [10] were first to study RNN queries. They answer RNN query by precalculating a circle for each data object p such that the nearest neighbor of p lies on the perimeter of the circle. RNN of a query q are the points that contain q in its circle. Techniques to improve their work were proposed in [24, 12].

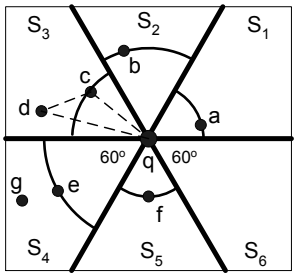


Figure 2: Six-regions pruning

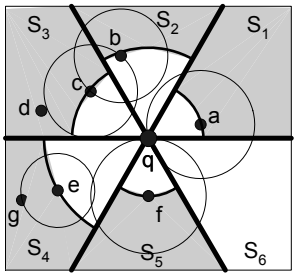


Figure 3: Filtering and verification

First work that does not need any pre-computation was presented by Stanoi *et al.* [16]. They solve RNN queries by partitioning the whole space centred at the query q into six equal regions of 60° each (S_1 to S_6 in Fig. 2). It can be proved that only the nearest point to q in each partition can possibly be the RNN. This also means that, in two-dimensional space, there are at most six possible RNNs of a query. Consider the region S_3 where c is the nearest object to q and d cannot be the RNN because its distance to c is smaller than its distance to q. This can be proved by the triangle Δqcd where $\angle dqc \leq 60°$ and $\angle dcq \geq 60°$, hence $dist(d,c) \leq dist(d,q)$. Fig. 3 shows the area (shown shaded) that cannot contain RNN of q.

In *filtering* phase, the candidate RNN objects (a, b, c, e and f in our example) are selected by issuing nearest neighbor queries in each region. In *verification* phase, any candidate object for which q is its nearest neighbor is reported as RNN (a and f). In this paper, we call this approach *six-regions pruning* approach.

Tao *et al.* [17] use the property of perpendicular bisectors to answer RkNN queries. Consider the example of Fig. 4, where a bisector between q and c is shown that divides the

space into two half-spaces (the shaded half-space and the white half-space). Any point that lies in the shaded half-space $H_{c:q}$ is always closer to c than to q and cannot be the RNN for this reason. Their algorithm prunes the space by the half-spaces drawn between q and its neighbors in the unpruned region. Fig. 5 shows the example where half-spaces between q and a, c and f ($H_{a:q}$, $H_{c:q}$ and $H_{f:q}$, respectively) are shown and the shaded area is pruned. Then, the candidate objects (a, c and f) are verified as RNNs if q is their closest object. We call this approach *half-space pruning* approach. It is shown in [17] that the half-space pruning is more powerful than the six-regions pruning and it prunes larger area (compare the shaded areas of Fig. 3 and Fig. 5).

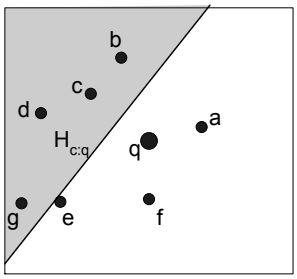


Figure 4: Half-space pruning

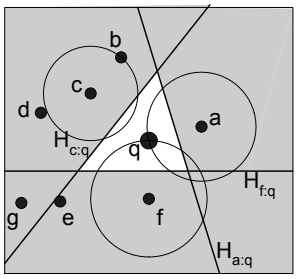


Figure 5: Filtering and verification

Wu *et. al* [21] propose an algorithm for RkNN queries in 2d-space. Instead of using bisectors to prune the objects, they use a convex polygon obtained from the intersection of the bisectors. Any object that lies outside the polygon can be pruned.

Continuous RNN Queries: Computation-efficient monitoring of continuous range queries [5, 11], nearest neighbor queries [13, 27, 23, 8, 18] and reverse nearest neighbor queries [1, 22, 9, 20] has received significant attention. Although there exists work on communication-efficient monitoring of range queries [7] and nearest neighbor queries [7, 14], there is no prior work that reduces the communication cost for continuous RNN queries. Below, we briefly describe the RNN monitoring algorithms that improve the computation cost.

Benetis *et al.* [1] presented the first continuous RNN monitoring algorithm. However, they assume that velocities of the objects are known. First work that does not assume any knowledge of objects' motion patterns was presented by Xia *et al.* [22]. Their proposed solution is based on the six-regions approach. Kang *et al.* [9] proposed a continuous monitoring RNN algorithm based on the half-space pruning approach. Consider the examples of Fig. 3 and Fig. 5, the results of the RNN query may change in any of the following three scenarios:

1. the query or one of the candidate objects changes its location
2. the nearest neighbor of a candidate object is changed (an object enters or leaves the circles shown in Fig. 3 and Fig. 5)
3. an object moves into the unpruned region (the areas shown in white in Fig. 3 and Fig. 5)

Xia *et al.* [22] use this observation and propose a solution for continuous RNN queries based on the six-regions approach. They answer RNN queries by monitoring six pie-regions (the white areas in Fig. 3) and the circles around the candidate objects that cover their nearest neighbors. Kang *et al.* [9] use the same observation and propose a solution based on the half-space pruning approach. They continuously monitor RNN queries by monitoring the unpruned region (white area in Fig. 5) and the circles around the candidate objects that cover their nearest neighbors. Both the approaches use a grid structure to store the locations of the objects and queries. They mark the cells of the grid that lie or overlap with the area to be monitored. Any object movement in these cells triggers the update of the results.

To the best of our knowledge, there exists only one solution for continuous monitoring of RkNN queries [20] which is similar to the six-regions based RNN monitoring presented in [22]. Wu *et al.* [20] issue k nearest neighbor (kNN) queries in each region instead of single nearest neighbor queries. The kNNs in each region are the candidate objects and they are verified if q is one of their k closest objects. To monitor the results, for each candidate object, they continuously monitor the circle around it that contains k nearest neighbors.

It is important to note that the problem of RkNN queries is different from all-nearest neighbor queries [4] where nearest neighbors of *every* object in a given dataset is to be found from another dataset.

2.3 Motivation

Both the six-regions and the half-space based solutions have two major limitations.

1. As illustrated in the three scenarios presented in the previous section, the existing techniques are sensitive to object movement. If a query or any of its candidate objects changes its location, filtering phase is called again which is computationally expensive. For example, if a query is continuously moving, at each timestamp both of the approaches will have to compute the results from scratch. For example, in the half-space based approach, the half-spaces between q and its previous candidates are redrawn and the pruning area is adjusted. In our initial experiments, we find that the cost of redrawing the half-spaces (and marking and unmarking the relevant cells) is computationally almost as expensive as the initial computation of the results.

2. The previous techniques require every object to report its exact location to the server at every timestamp regardless whether it affects the query result or not. This has a direct impact on the two most precious resources in mobile environment, wireless communication cost and power consumption. Ideally, only the objects that affect the query results should report their locations. For example, in Fig. 5, as long as objects d, e and g do not enter into the white region or the three circles, they do not affect the results of the query.

Motivated by these, we present a framework that provides a computation and communication efficient solution. Note that, in some applications, the clients may have to periodically report their locations to the server for other types of queries. In this case, saving the communication cost is not possible. Nevertheless, our framework significantly reduces the computation costs for such applications[2].

3. FRAMEWORK

Each moving object and query is assigned a safe region of a rectangular shape. Although other simple shapes (e.g.,

[2]In rest of the paper, we present our technique assuming that the clients send their locations only for the RkNN query. For the case when the clients periodically send their locations for other types of queries, our techniques can be easily applied. The only change is that the safe regions are stored on the server which ignores the location updates from the objects that are still in their safe regions. Experiments show superiority of our approach for both of the cases.

circles) could be used as safe regions, we choose the safe region of a rectangular shape mainly because defining effective pruning rules is easier for the rectangular safe regions. The clients may use their motion patterns to assign themselves better safe regions. However, we assume that such information is not utilized by the clients or the server because we do not assume any knowledge about the motion pattern of the objects. In our framework, the server recommends the side lengths of the safe regions (a system parameter) to the clients. A client assigns itself a new safe region such that it lies at the center of the safe region.

An object reports its location to the server only when it moves out of its safe region. Such updates issued by the clients (objects) are called *source-initiated* updates [7]. In order to update the results, the server might need to know the exact location of an object that is still in its safe region. The server sends a request to such object and updates the results after receiving its exact location. Such updates are called *server-initiated* updates [7].

If an object stops moving (e.g., a car is parked), it notifies the server and the server reduces its safe region to a point until it starts moving again.

In the previous approaches [22, 9], the pruned area becomes invalid if the query point changes its location. On the other hand, in our framework, the query is also assigned with a safe region and the pruned area remains valid as long as the query and its candidate objects remain in their respective safe regions and no other object enters in the unpruned region. Although the query is also assigned with a safe region, it reports its location at every timestamp. This is because its location is important to compute the exact results and a server-initiated update would be required (in most of the cases) if it does not report its location itself. Moreover, the number of queries in the system is usually much smaller than the number of objects. Hence, the location updates by the queries do not have significant effect on the total communication cost.

Table 1 defines the notations used throughout this paper.

Notation	Definition
$B_{x:q}$	a perpendicular bisector between point x and q
$H_{x:q}$	a half-space defined by $B_{x:q}$ containing point x
$H_{q:x}$	a half-space defined by $B_{x:q}$ containing point q
$H_{a:b} \cap H_{c:d}$	intersection of the two half-spaces
$A[i]$	value of a point A in the i^{th} dimension
$maxdist(x, y)$	maximum distance between x and y (each of x and y is either a point or a rectangle)
$mindist(x, y)$	minimum distance between x and y (each of x and y is either a point or a rectangle)
R_{fil}, R_{cnd}, R_q	rectangular region of the filtering object, candidate object and query, respectively
$R_H[i]$	highest coordinate value of a rectangle R in i^{th} dimension
$R_L[i]$	lowest coordinate value of a rectangle R in i^{th} dimension

Table 1: Notations

Like existing work on continuous spatial queries [13, 9, 22], we assume that the errors due to the measuring equipments are insignificant and can be ignored. Our continuous monitoring algorithm consists of the following two phases.

Initial computation: When a new query is issued, the server first computes the set of candidate objects by applying pruning rules presented in Section 3.1. This phase is called *filtering* phase. Then, for each candidate object, the server verifies it as RkNN if the query is one of its k closest points. This phase is called *verification* phase.

Continuous monitoring: The server maintains the set of candidate objects throughout the life of a query. Upon receiving location updates, the server updates the candidate set if it is affected by some location updates. Otherwise, the server calls verification module to verify the candidate objects and reports the results.

3.1 Pruning Rules

To the best of our knowledge, we are first to present novel pruning rules for RNN queries that can be applied when locations of the objects are unknown within their rectangular regions. These pruning rules can also be applied on the minimum bounding rectangles of the spatial objects that have irregular shapes (in contrast to the assumption that the spatial objects are points). In Section 5, we extend the pruning rules for RkNN queries.

In this section, an object that is used for pruning other objects is called a *filtering* object and the object that is being considered for pruning is called a *candidate* object.

3.1.1 Half-space Pruning

First, we present the challenges in defining this pruning rule by giving an example of a simpler case where the exact location of a filtering object p is known but the exact location of q is not known on a line MN (shown in Fig. 6). Any object x cannot be the RNN of q if $mindist(x, MN) \geq dist(x, p)$ where $mindist(x, MN)$ is the minimum distance of x from the line MN. Hence, the boundary that defines the pruned area consists of every point x that satisfies $mindist(x, MN) = dist(x, p)$. Note that for any point x in the space on the right side of the line L_N, $mindist(x, MN) = dist(x, N)$. Hence, in the space on the right side of the line L_N, the bisector between p and the point N satisfies the equation of the boundary (because for any point x on this bisector $dist(x, N) = dist(x, p)$).

Similarly, on the left side of L_M, the bisector between p and M satisfies the condition. In the area between L_M and L_N, a parabola (shown in Fig. 6) satisfies the equation of the boundary. Hence the shaded area defined by the two half-spaces and the parabola can be pruned. Note that the intersection of half-spaces $H_{p:N}$ and $H_{p:M}$ does not define the area correctly. As shown in Fig. 6 , a point p' lying in this area may be closer to q than to the point p.

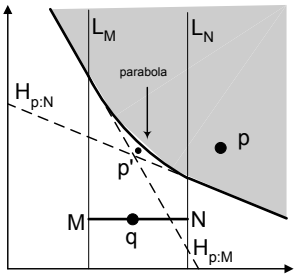


Figure 6: The exact location of the point q on line MN is not known

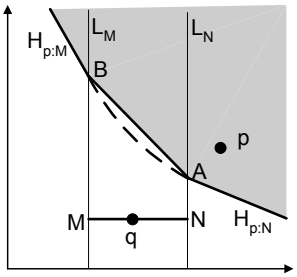


Figure 7: Approximation of parabola by a line

Unfortunately, the pruning of the shaded area may be expensive due to presence of the parabola. One solution is to approximate the parabola by a line AB where A is the intersection of $H_{p:N}$ and L_N and B is the intersection of $H_{p:M}$ and L_M. Fig. 7 shows the line AB and the pruned area is shown shaded.

Another solution is to move the half-spaces $H_{p:M}$ and $H_{p:N}$ such that both pass through a point c that satisfies $mindist(c, MN) \geq dist(c, p)$ (e.g., any point lying in the shaded area of Fig. 6). This approximation of the pruning area is tighter if the point c lies on the boundary. Fig. 8 shows the half-spaces $H_{p:M}$ and $H_{p:N}$ moved to such point c.

A half-space that is moved is called *normalized* half-space and a half-space $H_{p:M}$ that is moved is denoted as $H'_{p:M}$. Fig. 8 shows the normalized half-spaces $H'_{p:M}$ and $H'_{p:N}$ and their intersection can be pruned (the shaded area).

Among the two possible solutions discussed above, we choose normalized half-spaces in developing our pruning rules for the following reason. In our relatively simple example, the number of half-spaces required to prune the area by using the normalized half-spaces is two (in contrast to three lines for the other solution). The difference between this number becomes significant when both the query and the filtering object are represented by rectangles especially in multidimensional space. This makes the pruning by normalized half-spaces a less expensive choice.

Now, we present our pruning rule that defines the pruned area by using two half spaces in two dimensional space and 2^d half-spaces for d-dimensional space when $d > 2$. This pruning rule uses the normalized half-spaces between 2^d selected corners of the two rectangles to prune the space. Below, we give a formal description of our pruning rule in d dimensional space. Then, we briefly describe the reason of its correctness in two dimensional space. First, we define the following concepts:

Antipodal Corners Let C be a corner of rectangle $R1$ and C' be a corner in $R2$, the two corners are called *antipodal corners*[3] if for every dimension i where $C[i] = R1_L[i]$ then $C'[i] = R2_H[i]$ and for every dimension j where $C[j] = R1_H[j]$ then $C'[j] = R2_L[j]$. For example, in two dimensional space, a lower-left corner of $R1$ is the antipodal corner of the upper-right corner of $R2$. Fig. 9 shows two rectangles $R1$ and $R2$. The corners B and M are two antipodal corners. Similarly, other pairs of antipodal corners are (D, O), (C, N) and (A, P).

Antipodal Half-Space A half-space that is defined by the bisector between two antipodal corners is called *antipodal half-space*. Fig. 9 shows two antipodal half-spaces $H_{M:B}$ and $H_{O:D}$.

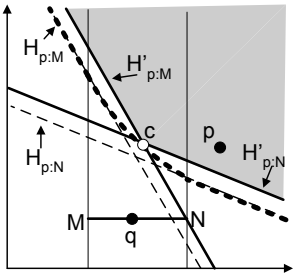


Figure 8: Defining pruned region by moving half-spaces

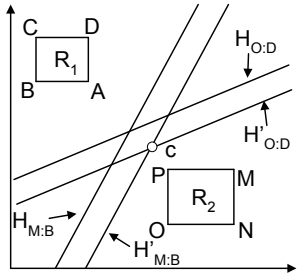


Figure 9: Antipodal corners and normalized half-spaces

Normalized Half-Space Let B and M be two points in hyper-rectangles $R1$ and $R2$, respectively. The normalized half-space $H'_{M:B}$ is a space defined by the bisector between M and B that passes through a point c such that $c[i] = (R1_L[i]+R2_L[i])/2$ for all dimensions i for which $B[i] > M[i]$ and $c[j] = (R1_H[i]+R2_H[j])/2$ for all dimensions j for which $B[j] \leq M[j]$. Fig. 9 shows two normalized (antipodal) half-spaces $H'_{M:B}$ and $H'_{O:D}$. The point c for the two half-space is also shown. The inequalities (1) and (2) define the half-space $H_{M:B}$ and its normalized half-space $H'_{M:B}$, respectively.

$$\sum_{i=1}^{d}(B[i] - M[i]) \cdot x[i] < \sum_{i=1}^{d} \frac{(B[i] - M[i])(B[i] + M[i])}{2} \quad (1)$$

[3] $R_L[i]$ ($R_H[i]$) is the lowest (highest) coordinate of a hyper-rectangle R in i^{th} dimension

$$\sum_{i=1}^{d}(B[i] - M[i]) \cdot x[i] < \sum_{i=1}^{d}(B[i] - M[i]) \times \begin{cases} \frac{(R1_L[i] + R2_L[i])}{2}, & \text{if } B[i] > M[i] \\ \frac{(R1_H[i] + R2_H[i])}{2}, & \text{otherwise} \end{cases} \quad (2)$$

Note that the right hand side of the Equation (1) cannot be smaller than the right hand side of Equation (2). For this reason $H'_{MB} \subseteq H_{MB}$. Now, we present our pruning rule.

PRUNING RULE 1 : Let R_q and R_{fil} be the rectangular regions of the query q and a filtering object p, respectively. For any point p' that lies in $\bigcap_{i=1}^{2^d} H'_{C_i:C'_i}$, $mindist(p', R_q) > maxdist(p', R_{fil})$ where $H'_{C_i:C'_i}$ is normalized half-space between C_i (the i^{th} corner of the rectangle R_{fil}) and its antipodal corner C'_i in R_q. Hence p' can be pruned.

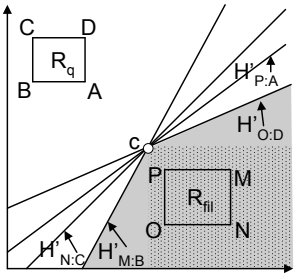


Figure 10: Pruning area of half-space pruning and dominance pruning

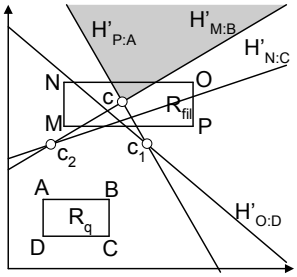


Figure 11: Any point in shaded area cannot be RNN of q

Fig. 10 shows an example of the half-space pruning where the four normalized antipodal half-spaces define the pruned region (the area shown shaded). The proof of correctness is non-trivial and is given in our technical report (Lemma 5) [3]. Below, we present the intuitive justification of the proof.

Intuitively (as in example of Fig. 8), if we draw all possible half-spaces between all points of R_q and R_{fil} and move them to a point c for which $mindist(c, R_q) \geq maxdist(c, R_{fil})$, then the intersection of these half-spaces correctly approximates the pruned region. Also note that in two dimensional space, at most two normalized spaces define such area. Consider the example of Fig. 10, where only $H'_{O:D}$ and $H'_{M:B}$ define the pruned region (the reason is that these two have largest and smallest slopes among all other possible half-spaces). In fact, the antipodal corners are defined such that the half-spaces having largest and smallest slopes are among the four antipodal half-spaces. Moreover, the point c shown in Fig. 10 satisfies $mindist(c, R_q) = maxdist(c, R_{fil})$ because normalized half-spaces are defined such that c lies at the middle of the line that joins the corners A and N. Hence the four normalized antipodal half-spaces correctly approximate the pruned region.

For ease of explanation, in Fig. 10, we have shown an example where the two rectangles R_q and R_{fil} do not overlap each other in any dimension. If the two rectangles overlap each other in any dimension (as in Fig. 11), the four half-spaces do not meet at the same point. In Fig. 11, $H'_{O:D}$ and $H'_{P:A}$ are moved to c_1 and $H'_{N:C}$ and $H'_{M:B}$ are moved to point c_2. However, it can be verified by calculating the intersection that the half-spaces that define the pruned region ($H'_{M:B}$ and $H'_{P:A}$) meet at a point c that satisfies $mindist(c, R_q) \geq maxdist(c, R_{fil})$.

3.1.2 Dominance Pruning

We first give the intuition behind this pruning rule. Consider the example of Fig. 10 again. The normalized half-spaces are defined such that if R_{fil} and R_q do not overlap each other in any dimension then all the normalized antipodal half-spaces meet at same point c. We also observe that the angle between the half-spaces that define the pruned area (shown in grey) is always greater than 90°. Based on these observations, it can be verified that the space dominated by c (the dotted-shaded area) can be pruned. Formal proof is given in our technical report (Lemma 6) [3].

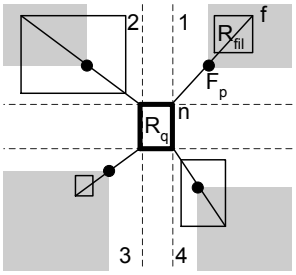


Figure 12: Shaded areas can be pruned

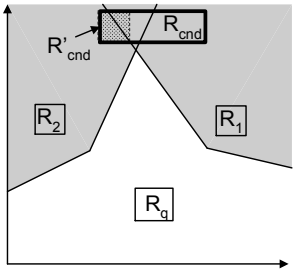


Figure 13: R_{cnd} can be pruned by R_1 and R_2

Let R_q be the rectangular region of q. We can obtain the 2^d regions as shown in Fig. 12. Let R_{fil} be the rectangular region of a filtering object that lies completely in one of the 2^d regions. Let f be the furthest corner of R_{fil} from R_q and n be the nearest corner of R_q from f (as shown in region 1 of Fig. 12). A point F_p that lies at the centre of the line joining f and n is called a *frontier point*.

PRUNING RULE 2 : Any candidate object p' that is dominated by the frontier point F_p of a filtering object cannot be RNN of q.

Fig. 12 shows four examples of dominance pruning (one in each region). In each partition, the shaded area is dominated by the frontier point of that partition and can be pruned. Note that if R_{fil} overlaps R_q in any dimension, we cannot use this pruning rule because the normalized antipodal half-spaces in this case do not meet at the same point. For example, the four normalized antipodal half-spaces intersect at two points in Fig. 11. In general, the pruning power of this rule is less than that of the half-space pruning. Fig. 10 shows the area pruned by the half-space pruning (the shaded area) and dominance pruning (the dotted area). The main advantage of this pruning rule is that the pruning procedure is computationally more efficient than the half-space pruning, as checking the dominance relationship is easier.

3.1.3 Metric Based Pruning

PRUNING RULE 3 : A candidate object can be pruned if $maxdist(R_{cnd}, R_{fil}) < mindist(R_{cnd}, R_q)$ where R_{cnd} is the rectangular region of the candidate object.

This pruning approach is the least expensive because it requires a simple distance comparison. Recall that the half-space (or the dominance) pruning defines a region such that any point p' that lies in it is always closer to the filtering object than to q. Metric based pruning checks this by a simple distance comparison. However, this does not mean that the metric based pruning has at least as much pruning power as half-space or dominance pruning. This is because the half-space and dominance pruning can trim the rectangular region of a candidate object that lies in the pruned region. It may lead to pruning of a candidate object when more than one filtering objects are considered.

Consider the example of Fig. 13, where two rectangles R_1 and R_2 of two filtering objects are shown. The rectangle R_{cnd} cannot be pruned when half-space pruning is applied on R_1 or R_2 alone. However, the rectangle R_{cnd} can be pruned when both R_1 and R_2 are considered. As in [17], we use loose trimming of the rectangle by using trimming algorithm [6]. The trimming algorithm trims a part of the rectangle that cannot be pruned. First, R_{cnd} is pruned by the half-spaces of R_1 and the trimming algorithm trims the rectangle that lies in the pruned region. The unpruned rectangle R'_{cnd} (shown with dotted shaded area) is returned. This remaining rectangle completely lies in the area pruned by R_2 so the candidate object is pruned. Note that metric based pruning cannot prune R_{cnd}.

Also note that if the exact location of a candidate object is known (R_{cnd} is a point) and metric based pruning fails to prune the object then half-space pruning and dominance pruning also fail to prune the object. Hence, half-space pruning and dominance pruning are applied only when the exact location of a candidate object is not known.

3.1.4 Pruning if exact location of query is known

If the exact location of the query or a filtering object is known, previous pruning rules can be applied by reducing the rectangles to points. However, a tighter pruning is possible if the exact location of the query is known. Below, we present a tighter pruning rule for such case.

PRUNING RULE 4 : Let R_{fil} be a hyper-rectangle and q be a query point. For any point p that lies in $\bigcap_{i=1}^{2^d} H_{C_i:q}$ (C_i is the i^{th} corner of R_{fil}), $dist(p,q) > maxdist(p, R_{fil})$ and thus p cannot be the RNN of q.

PROOF. Maximum distance between a rectangle R_{fil} and any point p is the maximum of distances between p and the four corners, i.e., $maxdist(p, R_{fil}) = max(dist(p, C_i))$ where C_i is the i^{th} corner of R_{fil}. Any point p that lies in a half-space $H_{C_i:q}$ satisfies $dist(p,q) > dist(p, C_i)$ for the corner C_i of R_{fil}. Hence a point p lying in $\bigcap_{i=1}^{2^d} H_{C_i:q}$, satisfies $dist(p,q) > maxdist(p, R_{fil})$. □

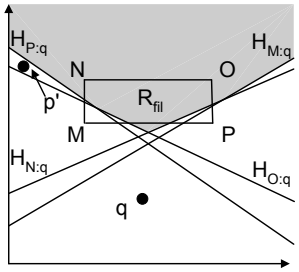


Figure 14: Half-space pruning when exact location of query is known

Consider the example of Fig. 14 that shows the half-spaces between q and the corners of R_{fil}. Any point that lies in the shaded area is closer to every point in rectangle R_{fil} than to q.

It is easy to prove that the pruned area is tight. In other words, any point p' that lies outside the shaded area may possibly be the RNN of q. Fig. 14 shows such point p'. Since it does not lie in $H_{P:q}$ it is closer to q than to the corner P. Hence it may be the RNN of q if the exact location of the filtering object is at corner P.

3.1.5 Integrating the pruning rules

Algorithm 1 is the implementation of all the pruning rules. Specifically, we apply pruning rules in increasing order of their computational costs (i.e., metric based pruning, dominance pruning and then half-space pruning). While simple pruning rules are not as restricting as more expensive ones, they can quickly discard many non-promising candidate objects and save the overall computational time.

Algorithm 1 : Prune(R_q, S_{fil}, R_{cnd})

Input: R_q: rectangular region of q ; S_{fil}: a set of filtering objects ; R_{cnd}: the rectangular region of candidate object
Output: returns true if R_{cnd} is pruned; otherwise, returns false
Description:

```
1: for each R_fil in S_fil do
2:    if maxdist(R_cnd, R_fil) < mindist(R_q, R_cnd) then
      // Pruning rule 3
3:       return true
4:    if mindist(R_cnd, R_fil) > maxdist(R_q, R_cnd) then
5:       S_fil = S_fil − R_fil  // R_fil cannot prune R_cnd
6: if exact location of cnd is known then
7:    return false  // the object cannot be pruned
8: for each R_fil in S_fil do
9:    if R_fil is fully dominated by R_q in a partition P then
      // Pruning rule 2
10:      trim the part of R_cnd that is dominated by F_p
11:   return true if R_cnd is pruned
12: return
13: for each R_fil in S_fil do
14:   Trim using half-space pruning  // Pruning rule 1
15:   return true if R_cnd is pruned
16: return false
```

Three subtle optimizations in the algorithm are:

1. As stated in Section 3.1.3, if the exact location of the candidate object is known then only metric based pruning is required. So, we do not consider dominance and half-space pruning for such candidates (line 7).

2. If $mindist(R_{cnd}, R_{fil}) > maxdist(R_q, R_{cnd})$ for a given MBR R_{fil}, then R_{fil} cannot prune any part of R_{cnd}. Hence such R_{fil} is not considered for dominance and half-space pruning (lines 4-5).

3. If the frontier point F_{p_1} of a filtering object R_{fil_1} is dominated by the frontier point F_{p_2} of another filtering object R_{fil_2}, then F_{p_1} can be removed from S_{fil} because the area pruned by F_{p_1} can also be pruned by F_{p_2}. However, note that a frontier point cannot be used to prune its own rectangle. Therefore, before deleting F_{p_1}, we use it to prune the rectangle belonging to F_{p_2}. This optimization reduces the cost of dominance pruning.

4. CONTINUOUS RNN MONITORING

In this section, we present our RNN monitoring algorithm called SAC (**S**wift **A**nd **C**heap) due to its computational efficiency and communication cost saving.

4.1 Data Structure

Our system has an object table and a query table. Object table (query table) stores the id and the rectangular region for each object (query). In addition, the query table stores a set of candidate objects S_{cnd} for each query.

Main-memory computation is the main paradigm in online/real-time query processing [13, 9, 22]. Grid structure is preferred when updates are intensive [13] because complex data structures (e.g., R-tree, Quad-tree) are expensive to update. For this reason, we choose grid-based data structure to store the locations and rectangular regions of moving objects and queries. Each cell contains two lists: 1) *object list*; 2) *influence list*. Object list of a cell c contains object id of every object whose rectangular region overlaps the cell c. This list is used to identify the objects that may be located in this cell. Influence list of a cell c contains query ids of all queries for which this cell lies in (or overlaps with) the unpruned region. The intuition is that if an object moves into this cell, we know that the queries in the influence list of this cell are affected.

Range queries and constrained NN queries (nearest neighbors in constrained region) are issued to compute RNNs of a query (e.g., six constrained nearest neighbor queries are issued in the six-regions based approach). In our algorithm, we also need an algorithm to search the nearby objects in a constrained area (the unpruned region). Several continuous nearest neighbors algorithms [27, 13, 23] based on grid-based index have been proposed. However, the extension of these grid-access methods for queries on constrained area becomes inefficient. i.e., the cells around queries are retrieved even if they lie in the pruned region. To efficiently search nearest neighbors in a constrained area, we propose a grid-based access method where the grid is treated as a conceptual tree.

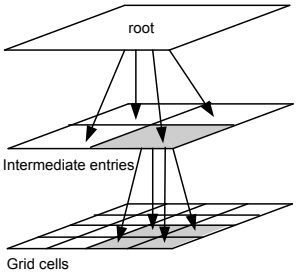


Figure 15: Conceptual grid-tree of a 4×4 grid

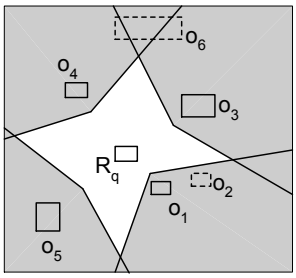


Figure 16: Illustration of filtering phase

Fig. 15 shows an example of the conceptual grid-tree of a 4×4 grid. For a grid-based structure containing $2^n \times 2^n$ cells where $n \geq 0$, the root of our conceptual grid-tree is a rectangle that contains all $2^n \times 2^n$ cells. Each entry at l-th level of this grid-tree contains $2^{(n-l)} \times 2^{(n-l)}$ cells (root being at level 0). An entry at l-th level is divided into four equal non-overlapping rectangles such that each such rectangle contains $2^{(n-l-1)} \times 2^{(n-l-1)}$ cells. Any n-th level entry of the tree corresponds to one cell of the grid structure. Fig. 15 shows root entry, intermediate entries and the cells of grid. Note that the grid-tree does not exist physically, it is just a conceptual visualisation of the grid.

The spatial queries algorithms that can be applied on R-tree can easily be applied on the conceptual grid tree. The advantage of using this grid-tree over previously used grid-based access methods is that if an intermediate entry of the tree lies in the pruned region, none of the cells inside it are accessed.

4.2 Initial Computation

The initial computation consists of two phases namely *filtering* and *verification*. Below we discuss them in detail.

4.2.1 Filtering

In this phase (Algorithm 2), the grid-tree is traversed to select the candidate objects and these objects are stored in S_{cnd}. These candidate objects are also used to prune other objects. Initially, root entry of the grid-tree is inserted in a min-heap H. We try to prune every de-heaped entry e (line 6) by using the pruning rules presented in the previous section. If e is a cell and cannot be pruned, we insert the objects into heap that are in its object list. Otherwise, if e

is an intermediate entry of the grid-tree, we insert its four children into the heap H with key $mindist(c, R_q)$. If e is an object and is not pruned, we insert it into S_{cnd}. The algorithm stops when the heap becomes empty.

Algorithm 2 : Filtering

```
1: for each query q do
2:    S_cnd = φ
3:    Initialize a min-heap H with root entry of Grid-Tree
4:    while H is not empty do
5:       de-heap an entry e
6:       if (not Pruned(R_q, S_cnd, e)) then  // Algorithm 1
7:          if e is a cell in Grid then
8:             for each object o in object list of e do
9:                insert o into H if not already inserted
10:         else if e is an intermediate entry of grid-tree
            then
11:            for each of its four children c do
12:               insert c into H with key mindist(c, R_q)
13:         else if e is an object then
14:            S_cnd = S_cnd ∪ {e}
```

Fig. 16 shows an example of the filtering phase. For better illustration, the grid is not shown. Objects are numbered in order of their proximity to q. Algorithm iteratively finds the nearest objects and prunes the space accordingly. In the example of Fig. 16, the algorithm first finds o_1 and prunes the space. Since the next closest object o_2 lies in the pruned space, it is not considered and o_3 is selected instead. The algorithm continues and retrieves o_4 and o_5 and the shaded area is pruned. The algorithm stops because there is no other object in the unpruned area (the white area). The rectangles of the pruned objects are shown in broken lines.

One important note is that in this phase, the call to pruning algorithm at line 6 does not consider the exact locations of any object or query for pruning even if the exact location is known. This is because we want to find a set of candidate objects S_{cnd} such that as long as all of them remain in their rectangular regions and no other object enters in the unpruned area, the set of candidate objects is not affected. For example, the set of candidate objects $\{o_1, o_3, o_4, o_5\}$ will not change unless q or any candidate object moves out of its rectangular region or any of the remaining objects (o_2 and o_6) moves in the unpruned area (the white area).

Marking the cells in unpruned area: To quickly identify that an object has moved into the unpruned area of a query q, each cell that lies in the unpruned area is marked. More specifically, q is added in the influence list of such cell. We mark these cells in a hierarchical way by using the grid-tree. For example, if an entry completely lies in the unpruned region, all the cells contained by it are marked. The cells are unmarked similarly.

4.2.2 Verification

At this stage, we have a set of candidate objects S_{cnd} for each query. Now, we proceed to verify the objects. Since every query q reports its location to the server at every timestamp, we can use its location to further refine its S_{cnd}. More specifically, any object $o \in S_{cnd}$ cannot be the RNN of q for which $mindist(o, q) \geq maxdist(o, o')$ for any other $o' \in S_{cnd}$. If the object cannot be pruned by this distance based pruning, we try to prune it by using pruning rule 4. For every query q, its candidate objects that cannot be pruned are stored in a list S_{global}.

The server sends messages to every object in S_{global} for which the exact location is not known. The objects send their exact locations in response. For each query q, the list of candidate objects is further refined by using these exact locations. As noted in [16], at this stage, the number of candidate objects for a query cannot be greater than six in two dimensional space. We verify these candidate objects as follows.

Algorithm 3 : Verification

```
1: Refine S_cnd using the exact location of q
2: Request objects in S_cnd to send their exact locations
3: Select candidate objects based on exact location of the
   objects
4: Verify candidate objects (at most six) by issuing boolean
   range queries
```

For a candidate object o, we issue a *boolean range query* [15] centered at o with range $dist(o, q)$. In contrast to the conventional range queries, a boolean range query does not return all the objects in the range. It returns true if an object is found within the range, otherwise it returns false. Fig. 17 shows an example, where candidate objects are o_1 to o_4. The object o_3 cannot be the RNN because o_6 (for which we know the exact location) is found within the range. Similarly, o_4 cannot be the RNN because the rectangular region of o_6 completely lies within the range. The object o_2 is confirmed as RNN because no object is found within the range. The only candidate object for which the results is undecided is o_1 because we do not know the exact location of object o_5 which may or may not lie within the range. The server needs its exact location in order to verify o_1. For each query q, the server collects all such objects. Then, it sends messages to all these objects and verifies all undecided candidate objects upon receiving the exact locations.

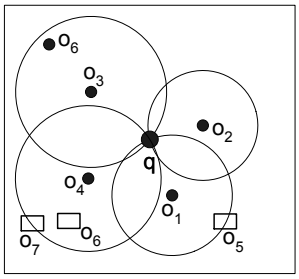


Figure 17: Illustration of verification phase

Note that, to compute the results of all queries, the server issues at most two request streams and receives at most two response streams.

4.3 Continuous Monitoring

The set of candidate objects S_{cnd} of a query changes only when the query or one of the candidate objects leaves its rectangular region or when any other object enters into the unpruned region. If S_{cnd} is not affected, we simply call the verification phase to update the results. Otherwise, we have to update S_{cnd}.

One approach to update S_{cnd} is to prune the area using current rectangular regions of q and its candidate objects. Any object that is found in the unpruned region is included in S_{cnd}. The cells that correspond to the old unpruned regions are unmarked and the cells that lie in or overlap with the new region are marked. In our experiments, we found that this update of S_{cnd} and grid cells is almost as expensive as computing the S_{cnd} from scratch. Below, we show that if we choose to compute S_{cnd} from scratch, we may save computation cost in next timestamps.

When a query or one of its candidate objects leaves its rectangular region, other candidate objects may also have moved and are likely to leave their regions in next few timestamps which will trigger the expensive filtering phase again.

Since we have to communicate with these candidate objects in verification phase anyway, we ask them to not only send their exact locations but also their new rectangles. After receiving these new rectangular regions, we compute the results of queries as in initial computation. Now all the candidate objects have new safe regions and the set of candidate objects is expected to remain unchanged for longer.

Suppose that an object o is candidate for two queries q_1 and q_2 and S_{cnd} of q_1 is affected by a location update of any other object o'. We cannot ask o to update its rectangular region because it will affect S_{cnd} of query q_2 as well. Hence, the server only asks an object to update its rectangular region if it does not affect other queries.

4.4 Cost Analysis

In this section, we analyse the computation and communication cost for our proposed solution. First, we present a pruning rule based on six-regions approach and compute the communication cost. Then, we show that the pruning rules used in our technique are superior. Hence the communication cost gives an upper bound. Then, we analyse the computation cost.

Assumptions: We assume that the system contains N objects in a unit space (extent of the space on both dimensions is from 0 to 1). Each rectangular region is a square and width of each side is w. The centers of all rectangular regions are uniformly distributed.

Communication cost: Consider the example of Fig. 18 where a 60° region bounded by the angle $\angle EqC$ is shown in thick lines. Suppose that we find a filtering object whose rectangular region R_{fil} is fully contained in the region. Any object o' can be pruned if $dist(o', q) \geq maxdist(R_{fil}, q)$. In other words, the possible candidates may lie only in the space defined by qEC where EC is an arc and $qC = qE = maxdist(R_{fil}, q)$.

Let r be the distance between q and the center of R_{fil}. Then, $maxdist(R_{fil}, q) \leq r + w/\sqrt{2}$ where $w/\sqrt{2}$ is the half of the diagonal length of R_{fil}. Since, all objects are represented by rectangular regions, any object is possible RNN candidate that has its centre at a distance not greater than $w/\sqrt{2}$ from the region qEC. So, the range becomes $(r + \sqrt{2}w)$. Total number of candidates that overlap or lie within the region qEC is

$$\frac{\pi(r + \sqrt{2}w)^2 N}{6}$$

Let R be the maximum of r of all six regions, the total number of candidate objects is bounded by

$$| S_{cnd} | = \pi(R + \sqrt{2}w)^2 N \quad (3)$$

The server sends request to all these candidate objects and receives their exact locations. So the total number of messages M_1 at this stage is bounded by

$$M_1 = 2\pi(R + \sqrt{2}w)^2 N \quad (4)$$

After receiving the updates, the server eliminates the candidate objects that cannot be the RNN (based on their exact locations). As proved in [16], the number of candidate objects cannot be greater than six. Hence, the server needs to verify those six candidate objects. In order to verify a candidate object o, the server issues a range query of distance $dist(o, q)$ centered at o. In worst case, all the objects that lie within this range must report their exact locations. Total number of objects that overlap or lie within the range is

$$\pi(dist(o, q) + w/\sqrt{2})^2 N$$

Since these candidate objects belong to the nearest neighbors in each region, $dist(o, q)$ corresponds to the distance of closest object in the region. For all six regions, the maximum of $dist(o, q)$ is the distance of sixth nearest neighbor from q (assuming uniform distribution). So the maximum range is the radius of a circle around q that contains six objects. As we assume a unit space, the radius of such circle that contains six objects is $\sqrt{\frac{6}{N\pi}}$. So the maximum number of messages M_2 required to verify all six candidate objects is

$$M_2 = 6 \times 2\pi(\sqrt{\frac{6}{N\pi}} + w/\sqrt{2})^2 N$$

$M_1 + M_2$ are the messages required to retrieve the *server-initiated* updates. Let M_3 be the number of *source-initiated* updates (the objects that leave their rectangular regions). Let v be the average speed of objects. An object starting at center of the square of width w and moving with speed v will take at least $w/2v$ time to leave the region. So, total number of updates M_3 at each timestamp is

$$M_3 = N \times min(\frac{2v}{w}, 1)$$

Note that the equation bounds the number of source-initiated updates by N. The total communication cost per timestamp $(M_1 + M_2 + M_3 + 1)$ where 1 denotes the location update of the query. Note that if w is small, the number of source-initiated updates M_3 increases and if w is large, the number of server-initiated updates $(M_1 + M_2)$ increases.

Now, we find R. Note that to use the pruning of Fig. 18, we had assumed that R_{fil} completely lies in the 60 degree region EqC. Hence r in Equation (3) corresponds to the distance of the closest object in each region that completely lies in it. Similarly, R is the maximum of r of each region.

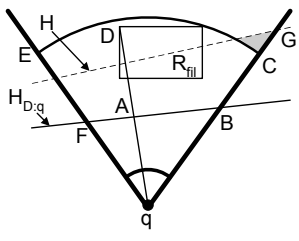


Figure 18: Half-space pruning vs six-regions based pruning

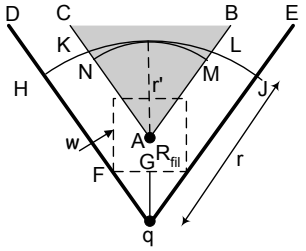


Figure 19: An object completely lying in the 60 degree region

Fig. 19 shows a region DqE and a rectangular region R_{fil} of a filtering object (shown in broken line). Note that any rectangular region of side length w with center lying in ABC (the shaded area) will completely lie in the region DqE. In other words, r corresponds to the closest object of q in the region that has center lying in ABC.

Let $r = qH = qJ$ as shown in Fig. 19. Let the radius belonging to area AMN be r'. The radius r' can be computed as $r' = r - qA$ where $qA = qG + GA = qG + w/2$. The length of $qG = 0.866w$ which can be found by the triangle FGq where $FG = w/2$ and $\angle GFq = 60°$. Hence $r' = r - 1.366w$.

It can be verified that when $r = \sqrt{\frac{6}{N\pi}} + 1.366w$, then $\pi(r')^2 N = 6$. In other words when radius is r, one object in each region will be found such that it completely lies in the region. So M_1 can be rewritten as

$$M_1 = 2\pi(\sqrt{\frac{6}{N\pi}} + 2.78w)^2 N$$

The cost (M_1+M_2+M_3+1) is the cost for one RNN query. The cost of multiple RNN queries is $\mid Q \mid \cdot(M_1+M_2+1)+M_3$ where $\mid Q \mid$ is the number of queries.

Now, we show that the area pruned by our proposed approach (pruning rule 4) contains the area pruned by previously described six regions based approach. Consider the example of Fig. 18 where R_{fil} completely lies in the region. The area pruned by six-regions approach is the area of region outside qCE where CE is an arc and $qC = maxdist(R_{fil}, q)$. Our pruning approach prunes the area defined by the intersection of the four half-spaces between q and the corners of R_{fil}. Fig. 18 shows a half-space H (shown in broken line) that crosses the region at a point G such that $qG > qC$. This half-space fails to prune some area pruned by the six region based approach (the six region based approach prunes the shaded area which this half-space H fails to prune).

In order to prove that our pruning approach always contains the area pruned by the six-region based approach, we need to show that all four half-spaces between q and the corners of R_{fil} cross the region at a point B such that $qB \leq qC$. Fig. 18 shows a half-space $H_{D:q}$ between corner D and q. Consider the right triangle qAB where $\angle BqA \leq 60°$. The length of qB is $\frac{qA}{cos(\angle BqA)}$. The maximum possible value of qB is $2 \times qA$ when $\angle BqA$ is 60°. Since $2 \times qA = qD$ and $qD \leq qC = maxdist(R_{fil}, q)$, so $qB \leq qC$. Similarly, it can be proved that $qF \leq qE$. Hence all the four half-spaces contain the area pruned by the region based approach.

Computation cost: Let C_{fil} and C_{ver} be the costs of the filtering phase and the verification phase, respectively. The computation cost at each timestamp is $\rho \times C_{fil} + C_{ver}$ where ρ is the probability that at a given timestamp at least one of the following two events happens: i) the query or any of the candidate objects leaves its safe region; ii) any other object enters in the unpruned region of the query.

The verification cost includes using the exact locations of M_1 objects to further refine the set of candidate objects and using boolean range queries to verify the remaining candidate objects (at most six). Let the cost of refining an object be C_{ref} and the cost of a boolean range query be C_{br}, the verification cost is $C_{ver} = M_1 \times C_{ref} + \mid S_{cnd} \mid \times C_{br}$ where $\mid S_{cnd} \mid \leq 6$.

5. EXTENSIONS

Since our proposed pruning rules can be applied in multidimensional space, the extension of our algorithm to arbitrary dimensionality is straightforward. Below, we present extension of our algorithm to RkNN monitoring.

RkNN Pruning: An object cannot be RkNN of a query if it is pruned by at least k filtering objects. We initialize a counter to zero and trim R_{cnd} by each filtering object. When the whole rectangle is trimmed, the counter is incremented and the original rectangle is restored. We continue this process by trimming with remaining filtering objects. If the counter becomes equal to k, the object is pruned.

Suppose k is 2 and consider the example of Fig. 20 where R_{cnd} and three filtering objects R_1, R_2 and R_3 are shown. Filtering objects are considered in order R_1, R_2 and R_3. R_{cnd} is trimmed to R'_{cnd} when R_1 is used for pruning. R'_{cnd} is completely pruned by R_2. The counter is incremented to one and the original rectangle R_{cnd} is restored. Now, R_{cnd} is trimmed by R_3 and the counter is incremented to two because whole rectangle is trimmed. The algorithm prunes R_{cnd} because it has been pruned two times.

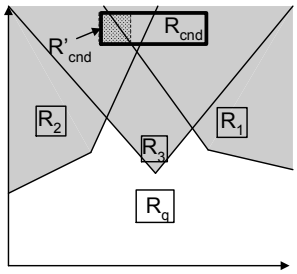


Figure 20: RkNN Pruning

Note that if the filtering objects are processed in order R_1, R_3 and R_2, the candidate object cannot be pruned. Finding the optimal order is difficult and trying all possible orders is computationally expensive. This will make filtering of this candidate object more expensive than its verification. Hence, if a candidate object is not pruned by the above mentioned pruning, we consider it for verification.

RkNN Verification: An object o cannot be RkNN if the range query centered at o with range $dist(o, q)$ contains greater than or equal to k objects. Otherwise, the object is reported as RkNN. Suppose k is 2 and consider the example of Fig. 17 again. The candidate objects o_2 and o_3 are confirmed as R2NNs because there are less than 2 objects within their ranges. The object o_1 is also confirmed because at most one object (o_5) lies within the range. The result for o_4 is undecided, so the location of o_7 is requested. Note that we do not need to request the exact location of o_6.

Bichromatic Queries: Now, we briefly present the extension of our proposed solution to bichromatic queries. Let there be two sets of objects O and P and query q belongs to O. The area is pruned by iteratively finding nearby filtering objects that belong to O and lie in the unpruned region. The pruning of area is stopped when there is no filtering object in the unpruned region. The objects of type P that lie in the unpruned region are the candidate objects. The server asks these candidate objects to report their exact locations. Upon receiving the exact locations, any candidate object p is reported as RNN if there does not lie an object of type O within a circle with radius $dist(p, q)$ centered at p. If the result is undecided, type O objects that have rectangles overlapping with the circles are requested to send their locations. Based on these received locations, the result is computed and reported to the client.

6. EXPERIMENT RESULTS

All the experiments were conducted on Intel Xeon 2.4 GHz dual CPU with 4 GBytes memory. For RNN queries ($k = 1$), we compare our algorithm with state-of-the-art algorithm (IGERN) [9] which has been shown superior in [9] to other RNN monitoring algorithms [22, 20]. For RkNN queries ($k > 1$), we compare our algorithm with CRkNN [20] which is the only available RkNN monitoring algorithm. In accordance with work in [9] and [20], we choose 64×64 grid structure for IGERN and 100×100 grid structure for CRkNN. For our algorithm, the grid cardinality is 64×64.

Similar to previous work, we simulated moving cars by using the spatio-temporal data generator [2]. Input to the generator is road map of Texas[4] and output is a set of cars (objects and queries) moving on the roads. The size of data universe is 1000 $Km \times 1000$ Km. The parameters of datasets are shown in Table 2 and default values are shown in bold.

The server reports the results continuously after every one

[4]http://www.census.gov/geo/www/tiger/

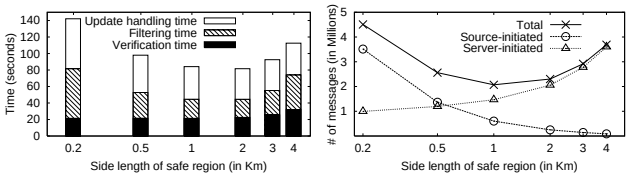
Parameter	Range
Number of objects (×1000)	40, 60, 80, **100**, 120
Number of queries	100, 300, **500**, 700, 1000
Average speed (in Km/hr)	40, 60, **80**, 100, 120
Side length of safe region (in Km)	0.2, 0.5, **1**, 2, 3, 4
Mobility (%)	5, 20, 40, 60, **80**, 100

Table 2: System Parameters

second (i.e., the timestamp length is 1 sec). Both the objects and queries are cars moving on roads, so they have similar properties (e.g., average speed, mobility). Mobility refers to the percentage of objects/queries that are moving at any timestamp (percentage of objects/queries that change their locations between two consecutive timestamps). All queries are continuously monitored for five minutes (300 timestamps) and the results shown correspond to total CPU time and communication cost. Communication cost is the total number of messages sent between clients and server.

As discussed in Section 2.3, there may be some applications where the objects have to report their locations to the server for other types of queries like range queries, nearest neighbor queries etc. In such case, the server is responsible for checking whether an object lies in the safe region or not. In order to show the superiority of our technique in all kinds of applications, the computation costs shown in the experiments include the cost of checking whether each object lies in its safe region or not. Obviously, the computation cost would be lesser for the case when the clients report their locations only when they leave their safe regions.

Fig. 21(a) shows the effect of the safe region size on computation time. The computation cost consists of update handling cost, filtering cost and verification cost. The update handling cost includes the cost of checking whether an object/query is in its safe region or not and updating the underlying grid structure if the object/query leaves the safe region. If the safe region is too small, the set of candidate objects is affected frequently and the filtering is required more often. Hence, the cost of the filtering phase increases. On the other hand, if the safe region is too large, the number of candidate objects increases and the verification of these candidates consumes more computation time. Also, the cost of filtering phase increases because lesser space can be pruned if the safe region is large. The update handling cost is larger for smaller safe regions because the objects and queries leave the safe regions more frequently.

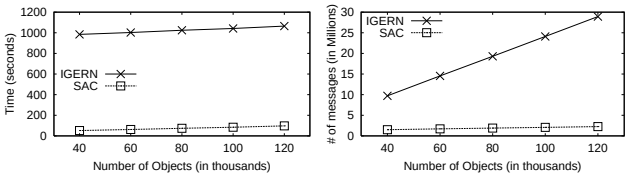


(a) Computation time (b) Communication cost

Figure 21: Effect of safe region size

Fig. 21(b) studies the effect of safe region size on communication cost. As studied in Section 4.4, the number of source-initiated updates increases if the side length of the safe region is small. On the other hand, if the safe region is large, the number of server-initiated updates increases. Fig. 21(b) verifies this. In current experiment settings, our algorithm performs best when the side length of the safe region is 1Km so we choose this value for the remaining experiments.

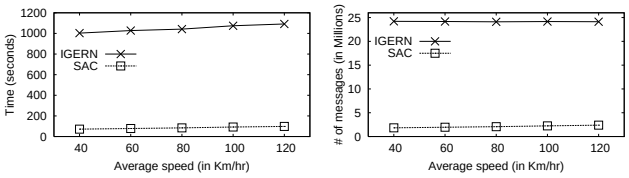
Fig. 22 shows the effect of the number of objects. Our algorithm not only outperforms IGERN but also scales better. The composition of CPU time is not shown due to the huge



(a) Computation time (b) Communication cost

Figure 22: Effect of datasize

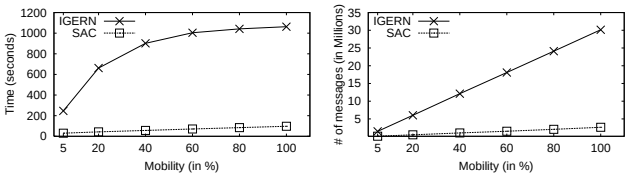
difference in the performance of both algorithms. However, the composition of CPU time is similar to Fig. 21(a) for our algorithm. For IGERN, the filtering phase takes 95% to 99% of the total cost in all experiments. This is because the expensive filtering phase is called frequently.



(a) Computation time (b) Communication cost

Figure 23: Effect of Speed

Fig. 23 studies the effect of the average speed of queries and objects. Fig. 23(a) shows that the computation time increases for both of the approaches as the speed increases. For our approach, the time increases because the objects and queries leave their respective safe regions more frequently and the filtering phase is called more often. Fig. 23(b) shows that IGERN requires an order of magnitude more messages than our approach. The communication cost for our approach increases due to the larger number of source-initiated updates as the speed increases.



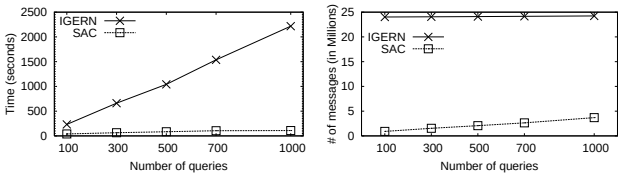
(a) Computation time (b) Communication cost

Figure 24: Effect of data mobility

Fig. 24(a) compares the computation time for increasing data mobility. As expected, IGERN performs good when the object mobility is low (e.g., 5%). However, its computation cost increases significantly as the object mobility increases. Our algorithm performs better for all cases and scales decently. Fig. 24(b) studies the effect of objects and queries mobility on the communication cost. Since only the moving objects report their locations, the number of messages increase with the increase in mobility. However, our algorithm consistently gives improvement of more than an order of magnitude compared to IGERN.

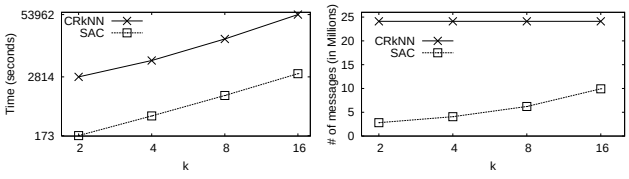
Fig. 25 studies the effect of number of queries. Fig. 25(a) shows that our algorithm gives more than an order of magnitude improvement over IGERN in terms of CPU time and scales better. In accordance with the analysis in Section 4.4, Fig. 25(b) show that the communication cost of our approach increases with the number of queries.

Fig. 26 studies the effect of k on communication and computation time. Fig. 26(a) compares our approach with [20] referred as CRkNN. Computation cost of both approaches increases with increase in k. However, our algorithm scales better (note the log scale). CRkNN continuously monitors $6k$ range queries to verify the candidate objects. To monitor



(a) Computation time (b) Communication cost

Figure 25: Effect of number of queries



(a) Computation time (b) Communication cost

Figure 26: Effect of k

these queries, it keeps a counter for the number of objects leaving and entering within the range. However, this information becomes useless when the candidate object or query changes its location. As shown in Fig. 26(b), communication cost for our approach increases for larger values of k. This is mainly because the number of candidate objects that require verification increases with k. Communication cost of our algorithm reaches to 24 million when $k = 64$ (CPU time $23,000$ sec). We were unable to run CRkNN for $k > 16$ due to its large main-memory requirement.

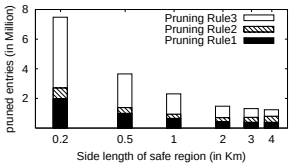


Figure 27: Effectiveness of pruning rules

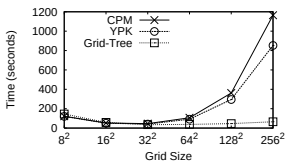


Figure 28: Effectiveness of grid-tree

Fig. 27 shows the effectiveness of pruning rules for different safe region sizes. Pruning rules are applied in the same order as in Algorithm 1. If a pruning rule fails to prune an entry (an object or a node of the grid-tree), the next pruning rule is used to prune it. Fig. 27 shows that a greater number of entries are pruned if the safe region size is small. Majority of the entries are pruned by the metric based pruning (pruning rule 3) when the safe regions are small. The average time to prune an entry by metric based pruning, dominance pruning and half-space pruning is 1.1, 2.3 and 10.5 micro seconds, respectively.

Now, we show the effectiveness of grid-tree over previous proposed grid access methods CPM [13] and YPK [27]. Fig. 28 shows the total CPU time for our RNN monitoring algorithm when the underlying constrained nearest neighbor algorithm (and marking and unmarking of cells) use CPM, YPK and grid-tree. We change the grid size from 8×8 to 256×256. Grid-tree based RNN monitoring algorithm scales much better with increase in number of cells.

7. CONCLUSION

In this paper, we studied the problem of continuous reverse k nearest neighbor monitoring. Our proposed approach not only significantly improves the computation cost but also reduces the communication cost for client-server architectures. We also present a thorough theoretical analysis. Experiment results demonstrate an order of magnitude improvement in terms of both the computation time and the communication cost.

8. REFERENCES

[1] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *IDEAS*, pages 44–53, 2002.

[2] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[3] M. A. Cheema, X. Lin, Y. Zhang, and W. Wang. Lazy updates: An efficient technique to continuously monitoring reverse knn. In *UNSW Technical Report, 2009. Available at ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0905.pdf.*

[4] Y. Chen and J. M. Patel. Efficient evaluation of all-nearest-neighbor queries. In *ICDE*, 2007.

[5] B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *EDBT*, pages 67–87, 2004.

[6] J. Goldstein, R. Ramakrishnan, U. Shaft, and J.-B. Yu. Processing queries by linear constraints. In *PODS*, 1997.

[7] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD Conference*, pages 479–490, 2005.

[8] G. S. Iwerks, H. Samet, and K. P. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *VLDB*, pages 512–523, 2003.

[9] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, 2007.

[10] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbor queries. In *SIGMOD*, 2000.

[11] I. Lazaridis, K. Porkaew, and S. Mehrotra. Dynamic queries over mobile objects. In *EDBT*, pages 269–286, 2002.

[12] K.-I. Lin, M. Nolen, and C. Yang. Applying bulk insertion techniques for dynamic reverse nearest neighbor problems. *ideas*, 00:290, 2003.

[13] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *SIGMOD*, 2005.

[14] K. Mouratidis, D. Papadias, S. Bakiras, and Y. Tao. A threshold-based algorithm for continuous monitoring of k nearest neighbors. *TKDE*, pages 1451–1464, 2005.

[15] A. Singh, H. Ferhatosmanoglu, and A. S. Tosun. High dimensional reverse nearest neighbor queries. In *CIKM*, 2003.

[16] I. Stanoi, D. Agrawal, and A. E. Abbadi. Reverse nearest neighbor queries for dynamic databases. In *ACM SIGMOD Workshop on Research Issues in Data Mining and Knowledge Discovery*, pages 44–53, 2000.

[17] Y. Tao, D. Papadias, and X. Lian. Reverse knn search in arbitrary dimensionality. In *VLDB*, 2004.

[18] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *VLDB*, pages 287–298, 2002.

[19] Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse nearest neighbor search in metric spaces. *TKDE*, 18(9), 2006.

[20] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Continuous reverse k-nearest-neighbor monitoring. In *MDM*, 2008.

[21] W. Wu, F. Yang, C. Y. Chan, and K.-L. Tan. Finch: Evaluating reverse k-nearest-neighbor queries on location data. In *VLDB*, 2008.

[22] T. Xia and D. Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE*, page 77, 2006.

[23] X. Xiong, M. F. Mokbel, and W. G. Aref. Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.

[24] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, 2001.

[25] M. L. Yiu and N. Mamoulis. Reverse nearest neighbors search in ad hoc subspaces. *TKDE*, 19(3):412–426, 2007.

[26] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. In *ICDE*, 2005.

[27] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, 2005.