ISSN 2320 - 2602
Volume 2, No.4, April 2013
International Journal of Advances in Computer Science and Technology
Available Online at http://warse.org/pdfs/2013/ijacst01242013.pdf

Efficient Algorithm to Monitor Continuous kNN Queries

Mahady Hasan
School of Engineering and Computer Science, Independent University, Bangladesh, Bangladesh,
mahadyh@yahoo.com

ABSTRACT

Continuous monitoring of k nearest neighbor (kNN) queries has attracted significant research attention in the past few years. A safe region is an area such that as long as a kNN query remains in it, the set of its k nearest neighbors does not change. Hence, the server does not need to update the query results unless the query moves out of its safe region. Previous work uses time-parameterized kNN (TPkNN) queries to construct the safe region. In this paper, we present an efficient technique to construct safe regions by using much cheaper RangeNN queries. Moreover, unlike TPkNN queries, the cost of a RangeNN query is not significantly affected as the value of k increases. Hence, our proposed algorithm scales better with the increase in the value of k. We also present a technique to efficiently update the safe regions when the underlying dataset is dynamic (i.e., objects appear or disappear from the dataset). Extensive experimental results show that the proposed algorithm provides an order of magnitude improvement over existing approach on both the static and dynamic datasets.

Key words: Continuous kNN Queries, Nearest Neighbor Query, Safe Region, Spatial Database.

1. INTRODUCTION

With the availability of inexpensive mobile devices, position locators and cheap wireless networks, location based services are gaining increasing popularity. Examples of location based services include location based games, geo-social networking, traffic monitoring, location based SMS advertising, enhanced 911 services and army strategic planning etc. Due to the popularity of these location based services, continuous monitoring of spatial queries has gained significant attention. The continuous monitoring of range queries [4, 12, 2], k nearest neighbor (kNN) queries [13, 22, 20, 9, 18] and reverse nearest neighbor queries [10, 19, 3] has been widely studied in recent past.

In this paper, we study the problem of moving kNN queries over static data objects, i.e., the queries are constantly moving whereas the data objects do not change their locations. We also consider the case where the data objects may appear or disappear from the dataset. The moving kNN queries have many applications. Consider the example of a car driver who is interested in five nearest gas stations. He may issue a kNN queries to continuously monitor the nearby gas stations. As another example, a ship sailing through an ocean may continuously monitor nearest icebergs to avoid accidents. A fighter plane may also issue a kNN query to continuously monitor the nearest enemy bases to attack.

Recently, several safe region based approaches have been developed to answer various spatial queries. Safe region is an area such that the expensive computation to update the results is not required as long as the moving object remains inside the safe region. In our earlier work, we developed the safe region based approaches to continuously monitor range queries [2] and reverse k nearest neighbor queries [3]. In this paper, we propose a safe region based approach to continuously monitor kNN queries. Therefore, the results of the kNN query are not required to be updated unless the query leaves the safe region. We next discuss two computational models [2] to monitor spatial queries and show that our safe region based approach is suitable for both models.

Safe region based approach to monitor spatial queries has gained significant attention. The safe region is an area such that as long as the query stays in it, the set of its results remain same. We have studied the safe region concept in case of range query in [2] and reverse nearest neighbors query in [3]. In this paper, we propose a safe region based approach to continuously monitor the moving kNN queries.

Client-server model. In this model, a central server computes the results for all the queries issued by the clients. Several existing techniques [13, 22, 20] assume that the server maintains the data objects and related information in the main memory. In contrast, our safe region based approach does not require the server to maintain any information in the main memory. Once a query arrives, the server computes the safe region and sends it to the client. When the client leaves the safe region, it sends its new location and old safe region to the server. The server uses this information to compute a new safe region. An advantage of this approach is that the server can provide on-demand service, i.e., the server can go to sleep mode if there is no query in the system or if there is no request from a client to get a new safe region.

Local computation model. In this model, the client (e.g., a

@ 2012, IJACST All Rights Reserved

GPS device) stores the data objects in its memory card and processes the query using its own computational power. Due to limited main memory and less computational power of the clients, it is challenging to compute the result whenever the query moves. Our technique ensures that the results are not required to be updated as long as the query remains in the safe region. Also, we do not require the objects to be stored in the main-memory. These features enable our technique to be used by the clients that have less main-memory and computational capacity. Below, we provide some of the advantages of our safe region based approach.

The safe regions based approach reduces the overall computational cost. This is because the results are updated only when a query leaves its safe region.

The safe region is a polygon that consists of around six edges [23] on average. The time complexity to check whether a client is inside its safe region or not is linear in number of edges. Hence, even the clients with low computational powers can efficiently check if they are within their safe regions or not.

As mentioned earlier, the server can provide the service on-demand because we do not need to maintain any information in the main memory. Moreover, the clients are required to contact the server only when they leave the safe regions. In contrast, the techniques that do not use safe regions require the clients to report their exact locations at every timestamp (i.e., after every t time units). Hence, our approach may reduce the communication cost between clients and server assuming that the clients contact the server only for the kNN queries.

It is important to note that the safe region corresponds to the Voronoi cell of the query [15]. For a kNN query, an order k Voronoi diagram can be constructed and the order k Voronoi cell can be treated as safe region. However, the Voronoi diagram based solution has several major limitations as mentioned in [23]. For example, the value of k is usually not known in advance and pre-computing several order k Voronoi diagrams for different values of k is computationally expensive and incurs high space requirement.

To address the above mentioned problems, [23] use time parameterized k nearest neighbor (TPkNN) [17] queries to create the Voronoi cell on the fly. However, a TPkNN query is expensive and its cost increases as the value of k increases. Another problem is that this solution does not handle dynamic dataset (i.e., where the objects may appear or disappear from the dataset). To address these issues, we present an efficient technique to construct and update the safe region. Below, we summarize our contributions:

We devise an efficient safe region construction approach that uses RangeNN [1] queries to construct the safe region in contrast to relatively much expensive[2] TPkNN queries used in the previous approach [23]. Moreover, unlike the TPkNN queries, the value of k does not have a significant effect on the cost of RangeNN queries. This leads to a substantial improvement in computation time for larger values of k.

- We extend our approach to efficiently update the safe regions of the queries for dynamic datasets where the objects may appear or disappear.
- Extensive experimental study demonstrates more than an order of magnitude improvement for both the static and dynamic datasets.
- The rest of the paper is organized as follows. Section 2 discusses the related work. We present our technique in Section 3. In Section 4, we present the experimental results followed by conclusion in Section 5.

2. RELATED WORK

Continuous monitoring of the kNN queries has gained significant research attention [18, 8, 9, 21]. Based on the problem setting and framework used for continuous monitoring of kNN queries, we divide this section into two parts. First, we briefly describe the related works that use the timestamp model and then, we discuss the works that use safe region concept.

2.1 Timestamp Model

In timestamp model, the server receives exact locations of all the moving objects and queries at each timestamp (e.g., after every t time units) and updates the results accordingly. If the length of the timestamp is large then the result of the moving query may become invalid between two timestamps. On the other hand, if the timestamp length is smaller then the computation cost increases because the results are to be updated more frequently.

YPK-CNN [22], SEA-CNN [20] and CPM [13] are some of the notable algorithms for continuous monitoring of the kNN queries using the timestamp model. These algorithms index data with a grid and the initial results are retrieved by searching the cells around the query point. SEA-CNN proposed a shared execution algorithm which improves the performance for large number of queries. CPM finds the result of a kNN query by traversing the cells around query point. The algorithm processes only the cells that intersect the circle centered at the query point q with radius equal to the distance between q and the kth NN.

[1] RangeNN query finds the nearest object of the query q with in a given radius from a specific point p.

[2] Our experimental study demonstrates that a RangeNN query is more than an order of magnitude faster than a TPkNN query.

@ 2012, IJACST All Rights Reserved

These algorithms are specifically designed for moving objects and are sensitive to query movement (i.e., if the query moves, the algorithms compute almost everything from scratch). On the other hand, our algorithm is specifically designed for moving queries. Moreover, these algorithms follow the timestamp model (i.e., the results are updated after every t time units). If the timestamp length is large the results are less accurate and if the timestamp length is small the computation and communication cost increases. On the other hand, in our approach the results remain correct as long as the query is in the safe region.

2.2 Safe Region Model

Voronoi diagram based approach. A Voronoi Diagram is constructed by drawing perpendicular bisectors between the objects of the underlying dataset. In a Voronoi Diagram, each object of the dataset lies within a cell called its voronoi cell. The voronoi cell of an object o has a property that any point that lies in it is always closer to o than any other object in the dataset. Figure 1 demonstrates the Voronoi diagram. The voronoi cell of the object o_2 is shown shaded in Figure 2. The NN of the query q is o_2 as long as q resides in the voronoi cell of o_2. Hence, the safe region of this query is the voronoi cell of o_2. The result of the query changes only when it moves out of this cell. In Figure 2, q moves to a new location q_0 and the safe region of q is now the voronoi cell of the object o_5. For a kNN query, a k order Voronoi Diagram can be constructed and k order voronoi cells can be treated as safe regions.

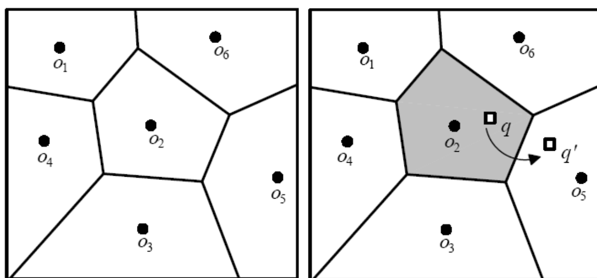
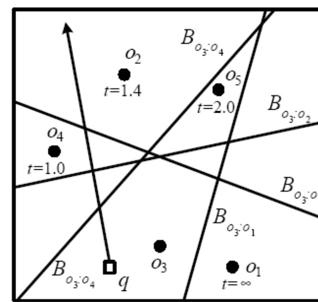


Figure 1: Voronoi Diagram **Figure 2:** Voronoi Cell

LBSQ. In [23], the authors use time-parameterized k nearest neighbor (TPkNN) queries to create the Voronoi cell on the fly. Before we present the LBSQ, we first discuss the time-parameterized query. Tao et al. [17] define time parameterized kNN (TPkNN) queries. Given the velocity vector of the query q, the TPkNN query finds the set of current kNNs and the time at which the set of kNNs of the query is changed. A TPkNN query also returns the object that causes the change to the set of kNNs. This object is known as the influence object and the time at which the object affects the result is known as the influence time.

Figure 3 shows an example of a TPkNN query (k=1). The trajectory of the query q is also shown. The NN of q is o_3. To find, the earliest time at which the NN of the query q changes, we may find the influence time for each object o_i such that at that time o_i becomes closer to q than o_3. The smallest of these influence times is the result influence time and the related object is the influence object. In Figure 3, the perpendicular bisectors between o_3 and other objects are drawn ($B_{o3:oi}$ is the bisector between the objects o_3 and o_i). By the property of a perpendicular bisector, when the query crosses a bisector $B_{o3:oi}$ it becomes closer to o_i than it is to o_3. Hence, the influence time of any object o_i is the time at which the query crosses its bisector. Figure 3 shows influence time of each object. The influence time of o_1 is infinity because its bisector $B_{o3:o1}$ does not intersect the query trajectory. The object o_4 has the smallest influence time, hence the answer is the object o_4, which influences the result at time t = 1.0. A tree structure can be used to answer TPkNN queries by applying any kNN algorithm where the distance metric is the influence time of the entry. For details, please see [17].



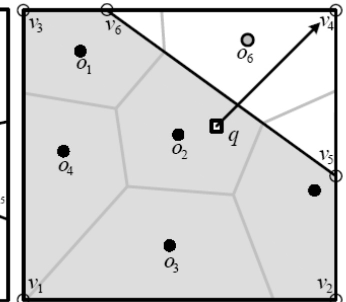


Figure 3: TP query **Figure 4:** LBSQ (1st Step)

The most relevant work to our approach is [23]. Based on TPkNN queries, the authors present an algorithm (called LBSQ in this paper) that constructs the safe region of a kNN query on the fly. Initially, the safe region is the whole space bounded by the vertices of the data space (v_1, v_2, v_3 and v_4 in Figure 4). The NN of the query q is o_2. The algorithm randomly chooses a vertex (v_4 in the example) and issues a TP query towards it which returns o_6 as answer. The bisector between o_2 and o_6 is drawn and the safe region is updated (the shaded region in Figure 4). The algorithm continues by selecting a random vertex v and issuing a TP query towards it. For any returned object o_i, the algorithm updates the safe region by drawing the bisector between o_i and the NN (o_2).

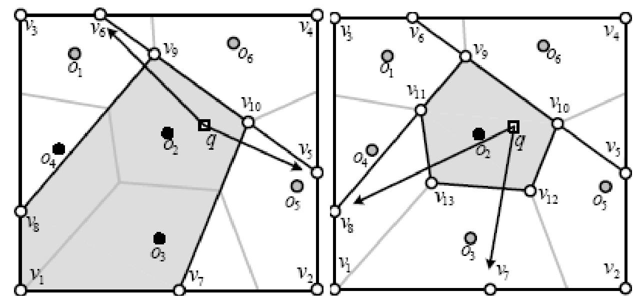


Figure 5: LBSQ (2nd Step) **Figure 6:** LBSQ (Final Step)

A vertex is marked confirmed if either the TP query does not

@ 2012, IJACST All Rights Reserved

return any object or it returns an object for which the bisector has already been drawn. The algorithm stops when all the vertices are confirmed. Figure 5 shows the TP queries issued towards vertices v_5 and v_6 and the shaded area is the safe region after the bisectors between o_2 and the returned objects (o_1 and o_5) have been drawn. Figure 6 shows the final safe region where all the vertices of the safe region (v_9 to v_{13}) have been confirmed. The problem with this algorithm is the TP queries. Since, these queries are very expensive and the computation time increases substantially by increasing the value in k.

Incremental Rank Updates. [11] present an algorithm called incremental rank updates (IRU) that uses (n-1) bisectors to maintain the order of n objects according to their distances from q. Figure 7 shows an example where the dataset consists of three objects and their ranking is <o_1, o_2, o_3> based on their respective distances from q (e.g., o_1 is the closest and o_3 is the furthest object). The bisectors between rank adjacent objects are drawn (see the bisectors $B_{o1:o2}$ and $B_{o2:o3}$). If q crosses any bisector $B_{oi:oi+1}$, the ranks of the objects o_i and o_{i+1} are swapped. For example, if the query crosses the bisector $B_{o1:o2}$ (as shown in the figure), the ranks are swapped and o_2 becomes the closest object and o_1 becomes the second closest object. The problem with this approach is that it needs to access all n data objects and checks (n-1) bisectors every time the query moves.

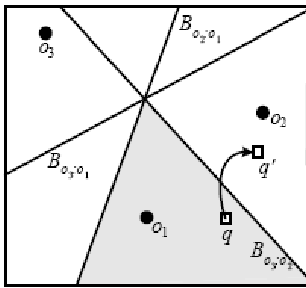


Figure 7: LRU Algorithm

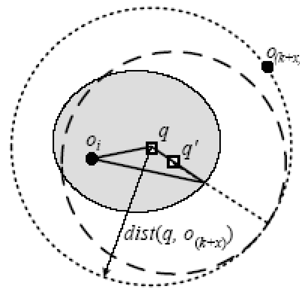


Figure 8: V*-Diagram

SR. In order to monitor a kNN query, Song and Roussopoulos (2001) present a technique that computes and sends (k+x) NNs to the client. Let $dist(q, o_k)$ and $dist(q, o_{(k+x)})$ be the distances of the kth NN and (k+x)th NN of q. It can be proved that if q moves to a new location q_0, the new kNNs of q are among the (k+x) objects provided that $2*dist(q, q_0)+dist(q, o_k) \leq dist(q, o_{(k+x)})$.

V*-diagram. Similar to SR [16], in V*-Diagram, [14] compute and send (k+x)NNs to the client. Let o_i be one of the (k+x)NNs of q, they prove that o_i remains one of the (k+x)NNs if q moves to a position q_0 such that $dist(q_0, o_i) \leq dist(q, o_{(k+x)}) - dist(q, q_0)$. Figure 8 shows the example where object o_i remains one of the (k+x)NNs as long as q remains in the shaded area. These k+x objects are sent to the client and the client uses IRU [11] to maintain the ranks of these k+x objects.

SR and V* diagram have two major problems: 1) It is difficult to estimate the proper value of x. A high value increases the network overhead, storage requirements at clients and computation power consumption of the client objects. On the other hand, a low value may not be useful; 2)The client objects have to continuously compute kNNs from the k+x objects and hence the algorithm somehow shifts the computation task to the clients. This has direct impact on power consumption of the clients. In contrast, our proposed approach does not heavily rely on the computation power of the clients. In our approach, the safe region that is sent to the client consists of around 6 edges [23] and it is easy for a client to check whether the client is inside the safe region or not.

3. TECHNIQUE

In Section 3.1, we formally define the problem and then study the problem characteristics. Then, we present our algorithm for static datasets in Section 3.2. In Section 3.3, we present the techniques for the dynamic datasets where objects may appear or disappear from the dataset. Discussion is presented in Section 3.5.

3.1 Definitions and Problem Characteristics

Given a set of objects, a query point q, and a positive integer k, the kNN query is to report the k closest objects to q. We formally define the problem below.

Let O be a set of objects and q be a query. A set of k nearest neighbors (kNNs) of a query $N=\{n_1, \ldots, n_k\}$ is a set of objects from O such that for every $n_i \in N$ and every $o_j \in O-N$, $dist(q, n_i) \leq dist(q, o_j)$ where dist is a distance metric that is assumed to be Euclidean distance in this paper.

Definition 1: Safe Region S is a region such that as long as a kNN query q remains in it, the set of its k nearest neighbors N does not change.

If a client (that issued query q) is aware of its safe region, it does not need to contact the server to update its set of kNNs as long as q resides in the safe region. This saves the computation cost. Before we formally define the safe region, we introduce the notion of perpendicular bisectors and half-spaces.

A perpendicular bisector $B_{n:o}$ between two points n and o divides the space into two half-spaces. Let $H_{n:o}$ be the half-space containing n and $H_{o:n}$ be the half-space containing o. By definition of a perpendicular bisector, every point p in $H_{n:o}$ is always closer to n than it is to o (i.e., $dist(p, n) < dist(p, o)$). Figure 9 shows a bisector $B_{n:o2}$ between two points n and o_2 and the two corresponding half-spaces $H_{n:o2}$ and $H_{o2:n}$. The query point q is closer to n then it is to o because q lies in $H_{n:o2}$.

@ 2012, IJACST All Rights Reserved

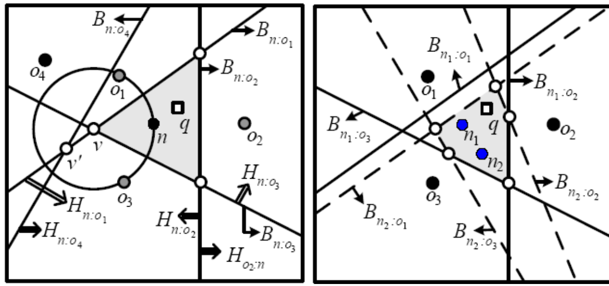


Figure 9: Safe region for 1NN **Figure 10:** Safe region for 2NN

Intuitively, if a point p lies in every half space $H_{n:oj}$ for every object o_j then $dist(p, n) \leq dist(p, o_j)$ for every object o_j . In other words, n would be the closest object of such point p. In Figure 9, the bisectors between n and four objects (o_1 to o_4) have been drawn. The shaded area corresponds to the intersection of the half spaces $H_{n:oj}$ for $j=\{1, 2, 3, 4\}$. Hence, every point p in the shaded area lies in every half space $H_{n:oj}$ for $j=\{1, 2, 3, 4\}$. For this reason, n is the closest object for any point in the shaded area. In other words, the object n is the nearest neighbor of a query q as long as the query remains in the shaded area (i.e., the shaded area is the safe region of q if k=1). Lemma 1 formalizes and generalizes this observation for arbitrary values of k.

Lemma 1: Let $N=\{n_1, \dots , n_k\}$ be the set of kNNs of a query q. The intersection of all half-spaces $H_{ni:oj}$ for every $n_i \in N$ and every $o_j \in O-N$ defines a region such that as long as the query resides in it, the set of its kNNs N is unchanged.

Proof 1: We prove this by contradiction. Assume that q resides in its safe region and $o_j \in O-N$ is an object such that $dist(q, o_j) < dist(q, n_i)$ for any $n_i \in N$. Since safe region is the intersection of all half-spaces $H_{ni:oj}$, a query q that resides in it satisfies $dist(q,n_i) < dist(q,o_j)$ which contradicts the assumption.

Figure 10 shows an example of the safe region for a 2NN query where the two NNs are n_1 and n_2. The bisectors between the two NNs and the objects o_1 to o_3 are drawn. For clarity, the bisectors between n_1 and the objects are shown by solid lines and the bisectors between n_2 and the objects are shown by dashed lines. The shaded area is the safe region and corresponds to the intersection of every half space $H_{ni:oj}$ for i={1, 2} and j = {1, 2, 3}.

A straight forward approach to compute the safe region is to consider every bisector $B_{ni:oj}$ and take the intersection of each half space $H_{ni:oj}$. However, this approach requires computing the bisectors of all objects with each of the kNNs. In Lemma 3, we show that we do not need to consider all the bisectors in order to create the safe region. Before we present the details, we define few terms and notations.

A bisector $B_{ni:oj}$ that forms an edge of the safe region is called a representative bisector. Note that not all the bisectors contribute in defining the safe region. The bisector $B_{n:o2}$ in Figure 9 is a representative bisector and the bisector $B_{n:o4}$ is not a representative bisector.

The object that is associated with the representative bisector is called an influence object. An object o is called a visited object if its bisector with all kNNs have been considered for constructing the safe region. The objects o_1, o_2 and o_3 in Figure 9 are the influence objects whereas o_4 is not an influence object. The bisectors for all the objects shown in Figure 9 and Figure 10 have been considered so they are visited objects.

A vertex is the intersection of two bisectors $B_{ni:oj}$ and $B_{nx:oy}$. A confirmed vertex is the vertex of the safe region (i.e., it is an intersection of two representative bisectors). Vertex v in Figure 9 is a confirmed vertex whereas the vertex v_0 is not a confirmed vertex. Please note that a confirmed vertex lies at the boundary of the safe region. Now, we present lemmas that can be used to see if a vertex is a confirmed vertex or not (i.e., if the vertex is lies at the boundary of the safe region or not). First, we present the lemma for k = 1 and then we extend it for arbitrary values of k.

Lemma 2: Let n be the NN of a query q and v be a vertex. The vertex v is a confirmed vertex if no object lies in the circle of radius R centered at v where $R = dist(v, n)$.

Proof 2: Assume that the circle does not contain any object and o_4 (as shown in Figure 9) is any object that lies outside the circle. If the vertex v does not lie in the safe region then there must be a half-space $H_{o4:n}$ such that v lies in $H_{o4:v}$ (i.e., v is closer to o_4 than it is to n). Any point p that lies in the half-space $H_{o4:n}$ satisfies $dist(p, o_4) < dist(p, n)$. However, for vertex v, $dist(v, o_4) > dist(v, n)$ because the object o_4 lies outside the circle. Hence there is no such half-space $H_{o4:n}$ that contains v. Therefore, the vertex v lies in the safe region and is a confirmed vertex.

Lemma 3: Let $N = \{n_1,\dots , n_k\}$ be the set of kNNs of query q and v be any vertex. The vertex v is a confirmed vertex if no object $o \in O-N$ lies in the circle centered at v with radius $R = maxdist(v,N)$ where $maxdist(v,N)$ is $max(dist(v, n_i))$ for every $n_i \in N$.

Proof 3: Figure 11 shows a vertex v and the circle with radius $R = maxdist(v,N)$. Assume that the circle does not contain any object and o_4 is any object that lies outside the circle (as shown in Figure 11). The vertex v satisfies $dist(v, n_i) < dist(v, o_4)$ for every $n_i \in N$, hence v lies in every $H_{ni:o4}$. For this reason, the vertex v lies in the safe region and is a confirmed vertex.

@ 2012, IJACST All Rights Reserved

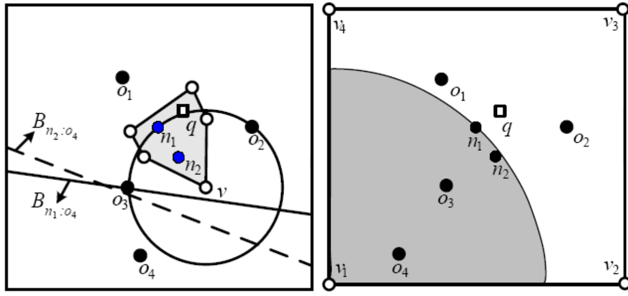


Figure 11: Lemma 3 illustration **Figure 12:** First RangeNN

Table 1: Notations

Notation	Defination
$B_{x:q}$	a perpendicular bisector between point x and q
$H_{x:q}$	a half-space defined by $B_{x:q}$ containing the point x
$H_{q:x}$	a half-space defined by $B_{x:q}$ containing the point q
$H_{a:b} \cap H_{c:d}$	the intersection of the two half-spaces
$Dist(x, y)$	the distance between two points x and y
$v <B_{ni:oj} \cap B_{nx:oy}>$	a vertex v formed by the intersection of the two bisectors
N	the set of k nearest neighbors $\{n_1, \ldots, n_k\}$
$maxdist(v,N)$	max(dist(v, n_i)) for every $n_i \in N$ where v is a vertex

3.2 Algorithm for Static Datasets

Before we present the details of the algorithm, we describe the main idea. Initially, the whole data space is assumed to be the safe region. Then, the objects are retrieved iteratively (in a certain order) and their bisectors are considered to update the safe region. At any stage, if all the vertices of the safe region are the confirmed vertices, the algorithm stops and reports the safe region to the client.

Algorithm 1 presents the details. The algorithm maintains a set of vertices V (initialized to four vertices of the universal data space). All data objects are indexed by R-tree [5]. First, the set N containing k nearest neighbors of the query q is computed by using best-first search algorithm [7] on Rtree. Then, the algorithm selects an unconfirmed vertex v from V that has minimum range (e.g., its $R = maxdist(v,N)$ is minimum among all $v \in V$) and checks whether it is a confirmed vertex or not by using Lemma 3. More specifically, the algorithm checks whether there is any object in the circle of range $R = maxdist(v,N)$ centered at v. If there is no object in the circle, the algorithm marks the vertex as confirmed. The intuition behind selecting a vertex with minimum range is that such vertex has higher chances to be a confirmed vertex (i.e., it has a smaller circle and has lower chances to contain any object).

Algorithm 1: **Construct Safe Region (*q*)**

Output: Returns V and I (associated with safe region)
1: V = {Vertices of the data space}
2: compute kNNs of q and store in N
3: **while** there is an unconfirmed vertex in V **do**
4: select a vertex $v <B_{ni:oj} \cap B_{nl:om}>$ that has minimum $maxdist(v,N)$
5: $R = maxdist(v,N) = dist(v, oj)$ /* Lemma 4 */
6: $o = RangeNN(q, v, R)$
7: **if** $o \neq$ NULL **then**
8: update V using bisectors between o and each $n_i \in N$
9: **else**
10: confirm v

If there are more than one objects in the circle, the algorithm selects the nearest object to the query q. This operation can be regarded as finding the nearest object o of q from the objects lying within the range R of a vertex v. Hence, we call it RangeNN query. We present the implementation of RangeNN query later. The safe region is updated by considering the bisectors between kNNs of q and the object o. For a given bisector $B_{ni:o}$, the safe region is updated by removing the vertices from V that lie in $H_{o:ni}$ and adding the intersection points of $B_{ni:o}$ and the safe region. The algorithm stops when all the vertices are confirmed. To show the correctness of the algorithm, we need to show that the algorithm finds all the vertices of the safe region and does not include any unconfirmed vertex. The proof of correctness is similar to Lemma 3.1 in [23] and is omitted.

Algorithm 2: RangeNN(*q*, *v*, *R*)

Output: Returns the nearest neighbor of q from the objects that lie within distance R from v
1: Initialize a min-heap H with root entry of the tree
2: **while** H is not empty **do**
3: deheap an entry e
4: **if** e is an intermediate or leaf node **then**
5: **for each** of its children c **do**
6: **if** $mindist(c, v) < R$ **then**
7: insert c into H with key $mindist(c, q)$
8: **else if** $e \in O$ **and** e **not in** kNNs of q **then**
9: **return** e
10: **return** ø

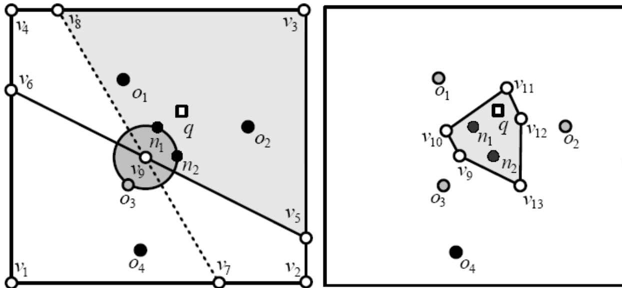
Algorithm 2 represents detailed implementation of the RangeNN[3] query. The algorithm assumes that the objects are indexed by a tree structure (e.g., R-tree). A min-heap is initialized with the root of the tree and the algorithm starts deheaping the entries iteratively. If a deheaped entry e is an intermediate or leaf node, the algorithm inserts all of its children that lie within the range (line 6) into the heap. The key of each inserted children is its minimum distance from the query. If the deheaped entry is an object which is not one of the kNNs, the algorithm returns it. If no such object is found, the algorithm returns null.

Example 1: Figure 12 illustrates our algorithm for a 2NN query where n_1 and n_2 are the NNs of q. Initial safe region is

[3] Note that RangeNN query does not access all the objects within the range. It accesses the entries in ascending order of their minimum distances from q and stops when the NN is found.

@ 2012, IJACST All Rights Reserved

the data space bounded by four vertices v_1 to v_4. First, a RangeNN query is issued on vertex v_1 with range $R=dist(v, n_1)$ which returns the object o_3. Then, the bisectors between o_3 and the NNs are drawn. In Figure 13, the bisector between o_3 and n_1 is shown in solid line and the bisector between o_3 and n_2 is shown by dashed line. These bisectors update the set of vertices V and the new safe region (the shaded area) now contains vertices v_3, v_5, v_9 and v_8. Then, a RangeNN query is issued on vertex v_9 with range $R=dist(v_9, n_1)$ and it is marked confirmed because no object is found within the range. The algorithm continues in this way until all the vertices are confirmed. The final safe region (shown shaded in Figure 14) contains the vertices v_9 to v_{13}. The objects o_1 to o_3 are retrieved during the construction of the safe region (i.e., the objects o_1 to o_3 are the influence objects).



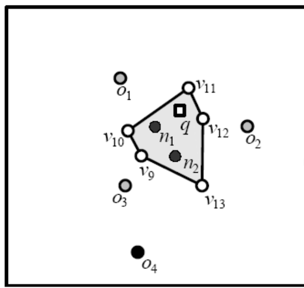


Figure 13: Safe region step1 **Figure 14:** Final safe region

Finally, we show that to compute the range $R=maxdist(v,N)$ of a vertex v (at lines 4 and 5 of Algorithm1), we do not need to compute distance of v from every nearest neighbors $n_i \in N$. More specifically, we show that the range $R=dist(v, o_j)$ where the vertex v is the intersection of two bisectors $B_{ni:oj}$ and $B_{nx:oy}$.

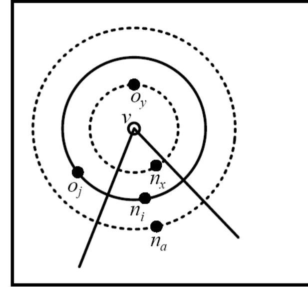
Lemma 4: Range for a vertex $v <B_{ni:oj} \cap B_{nx:oy}>$ is $R = maxdist(v,N) = dist(v, n_i) = dist(v, o_j) = dist(v, n_x) = dist(v, o_y)$ where both o_j and o_y are visited objects.

Proof 4: By definition of the perpendicular bisector, $dist(v, n_i) = dist(v, o_j)$ and $dist(v, n_x) = dist(v, o_y)$. First, we show that $dist(v, n_i) = dist(v, n_x)$ and then we will show that $dist(v, n_i) = maxdist(v,N)$.

Assume $dist(v, n_i) > dist(v, n_x)$. Then, the circle of radius $R = dist(v, n_i)$ centered at v contains the object o_y (Figure 15). So the bisector $B_{ni:oy}$ removes the vertex v from V. The object o_y is a visited object (its bisector with all nearest neighbors have been considered to update V) and v is still present in V. Hence, the assumption does not hold. Similarly, the assumption that $dist(v, n_i) < dist(v, n_x)$ also does not hold. Hence, $dist(v, n_i) = dist(v, n_x)$.

Now, we show that $maxdist(v,N) = dist(v, n_i)$. Assume there is an object $n_a \in N$ such that $dist(v, n_a) > dist(v, n_i)$. Then, the circle centered at v with radius $dist(v, n_a)$ contains the objects o_j and o_y (see Figure 15). Bisectors $B_{na:oj}$ and $B_{na:oy}$ would remove the vertex v from V because o_j and o_y have been visited. Hence, the assumption does not hold and $maxdist(v,N) = dist(v, n_i)$.

Note that the set of vertices V is initialized to the vertices of the data space. For such vertices, the observation does not hold and we compute $maxdist(v,N)$ for such vertices.



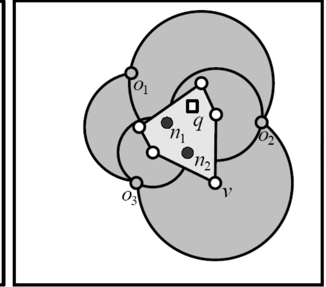
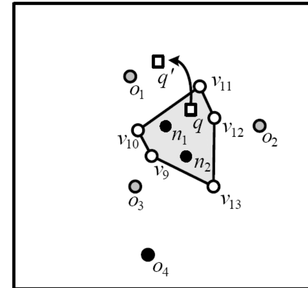


Figure 15: Lemma 4 **Figure 16:** Impact Region

3.3 Algorithm for Static Datasets

First, we define impact region. The impact region is an area such that as long as a query remains in its safe region and no object appears (disappears) in (from) the impact region, the safe region of the query is unchanged. It is easy to prove that the impact region consists of circles around vertices with radius set to their corresponding nearest neighbors. In Figure 16, the impact region is shown shaded. Below, we formally define the impact region.



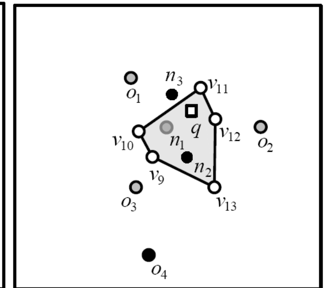


Figure 17: Query moves **Figure 18:** Change in NN

Definition 2: Let V be a set of vertices of a safe region. Let $Circ_v$ be a circle centered at a vertex $v <B_{ni:oj} \cap B_{nx:oy}>$ with radius $R_v = dist(v, o_j)$. The impact region is the area covered by all circles $Circ_{vi}$ for each $v_i \in V$.

Object updates in the impact region can change the results of a query in following two ways;

1. The set of its kNNs is changed.
2. The set of influence objects of the query is changed.

Below, we present our approach to handle each case.

Case 1 The set of kNNs may be changed if at least one or more of the following three happens: 1) the query moves out of the safe region (as in Figure 17); 2) one or more of the kNNs disappear (n_1 disappears in Figure 18); 3) a new object appears and becomes one of the kNNs (n_3 in Figure 18). To handle the case when the kNNs of a query change, we first

@ 2012, IJACST All Rights Reserved

compute new kNNs of the query. Then, we construct an initial safe region by drawing the bisectors between the new kNNs and the existing influence objects. Finally, we issue RangeNN queries to confirm its vertices as in Algorithm 1. The algorithm stops when all the vertices are confirmed.

Example 2: Figure 18 shows an example of 2NN query. The NN n_1 disappears and a new object n_3 appears. First, we compute the new NNs (n_2 and n_3). Then, we draw perpendicular bisectors between the NNs (n_2 and n_3) and the existing influence objects (o_1, o_2 and o_3). The intersection of these bisectors forms an initial safe region with vertices v_{11}, v_{14}, v_{15} and v_{16} (Figure 19). Finally, we issue RangeNN queries to confirm these vertices.

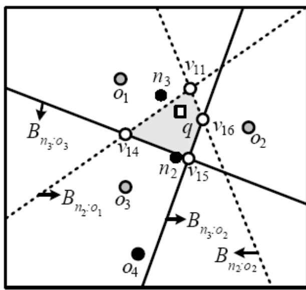


Figure 19: Handling Case 1

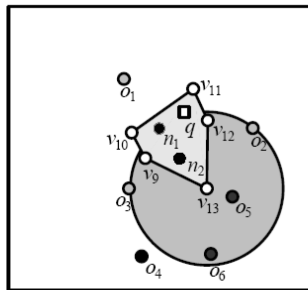


Figure 20: Objects appear

Case 2 If one or more of the influence objects disappear, we draw the bisectors between kNNs and the remaining influence objects to construct an initial safe region. Then, all the vertices are confirmed by issuing RangeNN queries. If an object o appears in the impact region and it does not affect the set of kNNs of the query, we update the safe region as follows; Every vertex v_i is marked unconfirmed that contains the object o in its circle $Circ_{vi}$. The RangeNN queries are issued to confirm all the vertices that have been marked unconfirmed.

Example 3: Consider the example of Figure 20 where two new objects o_5 and o_6 appear. The vertex v_{13} contains o_5 and o_6 in its circle hence v_{13} is marked unconfirmed. The RangeNN query is issued to confirm v_{13} which returns the object o_5. The bisectors between o_5 and the NNs (n_1 and n_2) are drawn, which change the safe region by adding two new vertices v_{14} and v_{15} (see Figure 21). These two vertices are confirmed by issuing RangeNN queries. The final safe region is shown shaded.

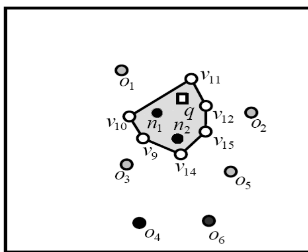


Figure 21: Handling Case 2

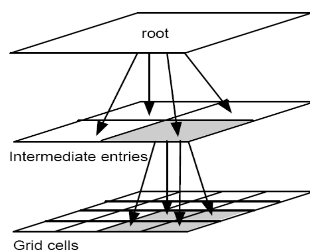


Figure 22: Conceptual grid tree

3.4 Data structure

First, we describe the data structure that is common for both the static and dynamic datasets. The system stores a query table that contains the information about the queries. More specifically, for each query it maintains its id, the set of its kNNs, the vertices of its safe region and the set of its influence objects. Now, we present the data structure that is different for the static dataset and the dynamic dataset.

For the static dataset, we index the objects with an R-tree. For the dynamic dataset, we use grid based structure because a more complex structure (like R-tree) would be very expensive to maintain dynamically [13]. To efficiently determine whether an object update lies in the impact region or not, we maintain *impact list* for each cell of the grid. The impact list of a cell c contains the id of every query that has its impact region overlapping with the cell. In addition, each cell c contains a list of objects that lie within this cell.

To efficiently answer RangeNN queries, we use conceptual grid-tree (introduced in [3] and further studied in [6]). Figure 22 shows an example of the Conceptual Grid-Tree (CGT) 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 lth 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 structures. Figure 22 shows root entry, intermediate entries and the cells of grid. Note that the grid-tree does not exist physically, it is just a conceptual visualization of the grid. Algorithm 2 can be directly applied on the grid-tree.

3.5 Discussion

First, we compare our algorithm with LBSQ [23] for static datasets.

1. LBSQ [23] issues a TPkNN query to confirm a vertex whereas we issue a RangeNN query to confirm the vertices. The advantage of using TPkNN query is that it guarantees that the returned object is an influence object. However, to find such an object the TPkNN query has to do expensive computation of influence time for the objects. On the other hand, although the RangeNN query does not provide such guarantee, it is significantly less expensive than the TPkNN query. As expected, experimental results show that although the number of RangeNN queries issued is more than the number of TPkNN queries (4% to 5% more), our algorithm gives an order of magnitude improvement in computation time because a RangeNN query is around 40-50 times less expensive than a TPkNN query.

2. The cost of RangeNN query is not significantly affected as the value of k increases. Although the increase of k value

@ 2012, IJACST All Rights Reserved

results in a larger range of the RangeNN query, the cost of RangeNN is not significantly affected because the RangeNN query traverses the R-tree in best-first order and stops (regardless the value of k) as soon as one closest object is found. Our experiments (see Figure 23) confirm that the cost of a RangeNN query is not significantly affected by the value of k. On the other hand, the cost of TPkNN increases with increase in the value of k. For this reason, our algorithm scales better as the value of k increases.

In addition to the above mentioned advantages, our approach has following advantages over LBSQ [23] on the dynamic datasets.

1. Our algorithm uses impact region and as long as there is no update in the impact region of a query, its results are unaffected. In other words, on receiving object updates, our algorithm updates the results of only the affected queries. In contrast, LBSQ cannot determine whether an update has affected the results of a query or not. Hence, upon receiving the object updates it needs to re-compute/verify the safe region of all the queries.

2. Even if an object update affects the results of a query, we may update the results by verifying only the vertices that have been affected (please see Example 3). In contrast, LBSQ needs to verify all the vertices by issuing TPkNN queries.

3. In case of an object update, LBSQ needs the exact location of all the queries to check if kNNs of any query have been changed by this update. On the other hand, our algorithm does not retrieve the location of any query that has no object update in its impact region. This reduces the communication cost.

4. EXPERIMENTS

In this section, we present the results of our experiments. We compare our algorithm with LBSQ [23] because other algorithms use timestamp model [22, 20, 13], assume known query trajectory path [17, 18] or assume that clients have sufficient computation resources to maintain kNNs from given (k+x) or more NNs [11, 16, 14] (see Section 2). First, we discuss the experimental setup in Section 4.1. Then, we present results for static and dynamic datasets in Section 4.2 and Section 4.3, respectively.

4.1 Setup

We use real dataset[4] as well as normal and uniform datasets. The real dataset contains 128,700 unique data points in a data space of 350km×350km. In dynamic datasets, each object reports a status update (the object appears or disappears) at a timestamp with a probability ρ. The timestamp length is one second. The moving queries were simulated using the spatio-temporal data generator [1]. The results of each query are monitored for hundred seconds. Table 2 shows the parameters used in our experiments and the default values are shown in bold.

All the experiments were conducted on Intel Xeon 2.4 GHz dual CPU with 4 GBytes memory. Our algorithm is called RSR (RangeNN based Safe Region) and the previous work is referred as LBSQ. The algorithms are compared in terms of total CPU time and the number of nodes accessed.

Table 2: System parameters

Parameter	Range
Number of objects (£1000)	20, 40, 60, 80, 100 ,**128**
Datasets	Uniform, Normal, **Real**
Number of queries	100, 200, **500**, 1000, 2500, 5000
Value of k	1, 3, 10, **30**, 100
Query Speed (km/hr)	25, 50, **75**, 100, 125
Object update probability	0.01, 0.02, **0.03**, 0.05, 0.1, 0.3

4.2 Static Dataset

First, we show that although the number of RangeNN queries issued by our algorithm is slightly higher than the number of TPkNN queries issued by LBSQ, our algorithm outperforms LBSQ because the average cost of a RangeNN query is much lower than that of a TPkNN query.

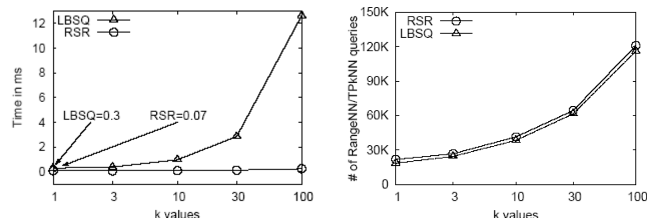


Figure 23: RangeNN/TPkNN cost **Figure 24:** # of RangeNN/TPkNN

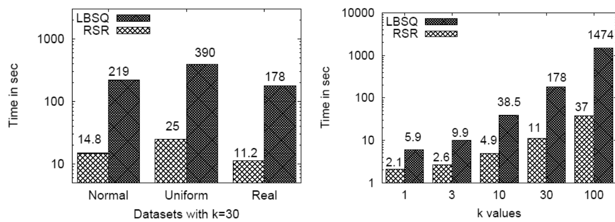
Figure 23 shows the average time taken by a RangeNN query and a TPkNN query for the increasing value of k. The cost of RangeNN query is less affected by the increase in k. Figure 24 shows that the difference in the number of RangeNN queries and TPkNN queries issued by both the algorithms is negligible. The number of the RangeNN queries and TPkNN queries increase for larger value of k mainly because the size of safe region becomes smaller. Hence, the query moves out of the safe region more frequently and a new safe region is computed more often. The costs of RangeNN queries and TPkNN queries are the most considerable costs for both of the algorithms (e.g., RangeNN queries takes 70% to 80% of the total time in our approach and TPkNN queries take 95% to 99% of the total time in LBSQ).

4.2.1 Effect of data distribution

In Figure 25, we show the performance of our algorithm for different data distributions. Our algorithm gives an order of magnitude improvement for all data distributions. Next, we focus only on the real dataset.

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

@ 2012, IJACST All Rights Reserved



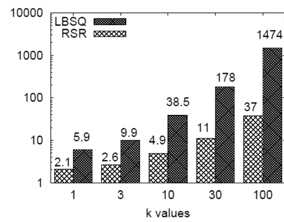
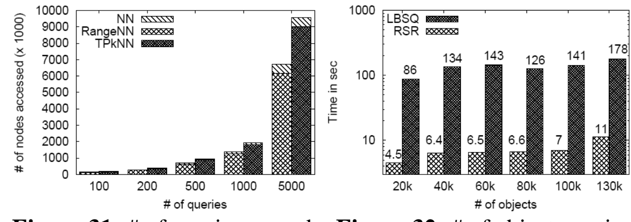


Figure 25: data distribution **Figure 26:** k Vs time



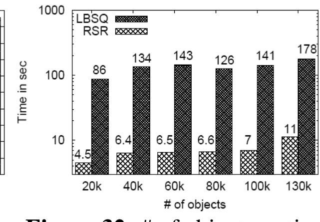
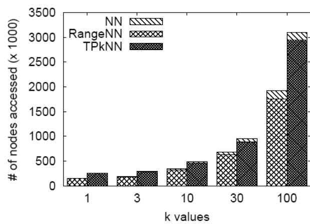


Figure 31: # of queries vs node **Figure 32:** # of objects vs time

4.2.2. Effect of k

Figure 26 studies the effect of k on the computation times of both algorithms. Our algorithm not only outperforms LBSQ but also scales better. Figure 27 shows the number of nodes accessed by both the algorithms for the kNN queries, RangeNN queries and TPkNN queries. The results show that the number of nodes accessed by the kNN queries is much lower as compared to that of the RangeNN queries and the TPkNN queries.



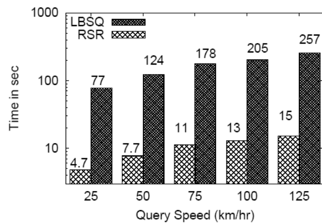
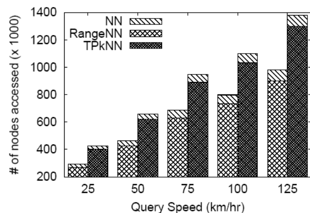


Figure 27: k vs # of nodes **Figure 28:** query speed vs time

4.2.3. Effect of query speed

Figure 28 shows the effect of query speed on both the algorithms. The computation costs of both the algorithms rise with the increase in speed. This is because the query leaves its safe region more frequently and new safe region is computed more often. Our algorithm performs around 15 times better than LBSQ. Figure 29 shows the number of nodes accessed by both algorithms. Our RangeNN queries based algorithm accesses lesser nodes than the TPkNN queries based approach.



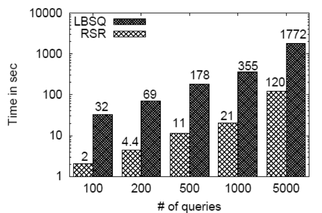


Figure 29: query speed vs # of node **Figure 30:** # of query vs time

4.2.4. Effect of number of queries

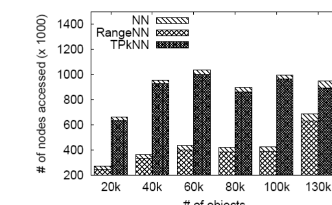
Figure 30 and Figure 31 demonstrate the performance of the algorithms for different number of queries. Our algorithm provides an order of magnitude improvement over LBSQ in terms of CPU time and scales better (note the log scale). Moreover, our algorithm accesses lesser number of nodes.

4.2.5. Effect of number of objects

Our experiments show that both of the algorithms are not significantly affected by the number of objects. The reason is that the number of objects does not have substantial effect on the costs of the TPkNN queries and the RangeNN queries which are the most influential costs in both the algorithms. Nevertheless, our algorithm performs better in terms of both CPU time and the number of nodes accessed. Figure 32 shows that our algorithm performs around 20 times better than LBSQ in terms of computation time. Figure 33 shows that LBSQ accesses around 2.5 times more nodes than our algorithm.

4.3. Dynamic Dataset

Now, we show the performance of both the algorithms on dynamic datasets where the objects may appear or disappear. We use grid structure for our algorithm and refer to it as GSR. For LBSQ, we load R-tree into main memory before evaluating the queries. For fair evaluation, we do not consider time taken in updating the R-tree for their algorithm. We extended LBSQ for dynamic datasets as follows; At each timestamp, the algorithm marks every query for which the set of its kNNs has been affected. For such queries, the safe region is computed from scratch. For any other query, all vertices of its safe region are marked unconfirmed and TPkNN queries are issued to confirm these vertices. Note that if the vertices are not confirmed, a new influence object affecting the safe region may be missed.



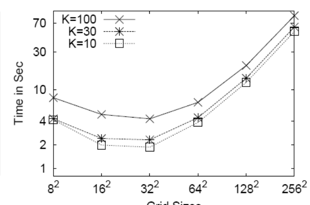


Figure 33: # of objects vs nodes time **Figure 34:** grid cardinality vs time

4.3.1. Effect of grid size

Since we use grid structure, we first study the effect of grid cardinality. Figure 34 shows the performance of our algorithm on different grid sizes for different values of *k*. In accordance with previous works that use grid based approaches, the performance degrades if the grid size is too small or too large. More specifically, if the grid has too low

@ 2012, IJACST All Rights Reserved

cardinality the cost of RangeNN and kNN queries increase because each cell contains larger number of objects. On the other hand, if the grid cardinality is too high then many of the cells are empty and it affects the performance of the RangeNN and kNN queries. We find that our algorithm performs better on 32×32 grid and we use this grid size for the rest of the experiments.

4.3.2. Effect of object update probability

Figure 35 demonstrates the effect of object update probability on both of the algorithms. The costs of both algorithms rise as the object update probability increases. However, our algorithm performs much better than LBSQ. This is because LBSQ needs to confirm all vertices of the safe region at every timestamp whereas our algorithm confirms only the affected vertices of the queries affected by the object updates. The cost of LBSQ increases with the higher number of updates because the set of kNNs of the queries change more frequently and the safe regions are required to be computed from scratch.

Figure 36 shows the number of TPkNN queries issued by LBSQ algorithm and the number of RangeNN queries issued by our algorithm. As expected, the number of RangeNN queries is much less than the number of TPkNN queries because our algorithm only recomputes the safe regions of the queries that are affected by the object updates. We note that the average costs of the RangeNN query and TPkNN query are not affected by change in the object update probability so we do not include the results.

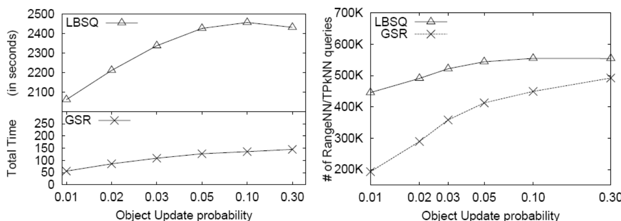


Figure 35: update probability vs time

Figure 36: update probability vs # of RangeNN/TTkNN

4.3.3. Effect of k

Figure 37 studies the effect of k on both of the algorithms for dynamic datasets. Clearly, our algorithm outperforms LBSQ and scales better because the average cost of the RangeNN query is not substantially affected by k.

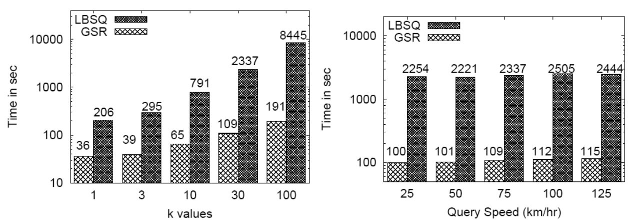


Figure 37: k vs time **Figure 38:** query speed vs time

4.3.4. Effect of query speed

Figure 38 shows the effect of query speed for both of the algorithms. The results show that the effect of query speed is not as significant as was noted in static datasets. The reason is that for dynamic datasets the total time of both algorithms increases because safe regions are recomputed more often due to the object updates. So, the effect of recomputation of safe regions for the query movement becomes less significant.

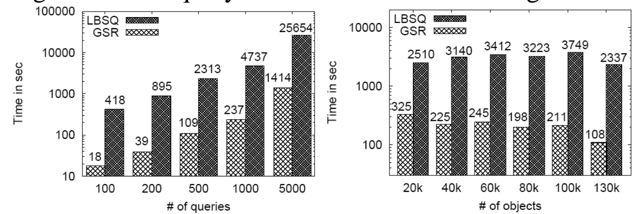


Figure 39: query size vs time **Figure 40:** object size vs time

4.3.5. Effect of number of queries and objects

Figure 39 studies the effect of number of queries on both algorithms. Our algorithm provides an order of magnitude improvement over the previous algorithm. Figure 40 shows that both the algorithms are not significantly affected by the number of objects (as for static datasets). The reason is already stated in Section 4.2.5.

5. CONCLUSION

Previous algorithm uses TPkNN queries to compute the safe region of a kNN query. In this paper, we present an efficient algorithm to construct the safe region by using much cheaper RangeNN queries. Moreover, RangeNN queries are not substantially affected by the value of k. We also present an efficient technique to update the safe region for the dynamic datasets. Experimental results show an order of magnitude improvement for both the static and dynamic datasets.

REFERENCES

1. T. Brinkhoff. **A framework for generating network-based moving objects**. GeoInformatica 6.2, pp 153–180, 2002.
2. M.A. Cheema, L. Brankovic, X. Lin, W. Zhang and W. Wang. **Multi-guarded safe zone: An effective technique to monitor moving circular range queries**. ICDE, pp189-200, 2010.
3. M.A. Cheema, X. Lin, Y. Zhang, W. Wang and W. Zhang. **Lazy updates: An efficient technique to continuously monitoring reverse knn**. VLDB 2, pp 1138–1149, 2009.
4. B. Gedik and L. Liu, **Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system**, EDBT, pp. 67–87, 2004.
5. A. Guttman, **R-trees: A dynamic index structure for spatial searching**, SIGMOD, pp. 47–57, 1984.

@ 2012, IJACST All Rights Reserved

6. M. Hasan, M.A. Cheema, W. Qu and X. Lin, **Efficient algorithms to monitor continuous constrained nearest neighbor queries**, DASFAA (1), pp. 233–249, 2010.
7. G. R. Hjaltason and H. Samet. **Distance browsing in spatial databases**. ACM Trans. Database Syst. 24, 265–318, 1999.
8. H. Hu, J. Xu and D.L. Lee. **A generic framework for monitoring continuous spatial queries over moving objects**, SIGMOD, pp. 479–490, 2005..
9. G.S. Iwerks, H. Samet and K.P. Smith. **Continuous k-nearest neighbor queries for continuously moving points with updates**, VLDB, pp. 512–523, 2003.
10. J.M. Kang, M.F. Mokbel, S. Shekhar, T. Xia and D. Zhang. **Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors**, ICDE, pp. 806–815, 2007.
11. L. Kulik and E. Tanin. **Incremental rank updates for moving query points**, GIScience, pp. 251–268, 2006.
12. I. Lazaridis, K. Porkaew and S. Mehrotra. **Dynamic queries over mobile objects**, EDBT, pp. 269–286, 2002.
13. K. Mouratidis, M. Hadjieleftheriou and D. Papadias. **Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring**, SIGMOD, pp. 634–645, 2005.
14. S. Nutanong, R. Zhang, E. Tanin and L. Kulik. **The v*-diagram: a query-dependent approach to moving knn queries**, VLDB 1, pp 1095–1106, 2008.
15. A. Okabe, B. Boots and K. Sugihara. **Spatial tessellations: concepts and applications of Voronoi diagrams**, John Wiley and Sons Inc, 1992.
16. Z. Song and N. Roussopoulos. **K-nearest neighbor search for moving query point**, SSTD, pp. 79–96, 2001.
17. Y. Tao and, D. Papadias. **Time-parameterized queries in spatiotemporal databases**, SIGMOD, pp. 334–345, 2002.
18. Y. Tao, D. Papadias and Q. Shen. **Continuous nearest neighbor search**, VLDB, pp. 287–298, 2002.
19. T. Xia and D. Zhang. **Continuous reverse nearest neighbor monitoring**, ICDE, pp. 77, 2006.
20. X. Xiong, M.F. Mokbel and W.G. Aref. **Sea-cnn: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases**, ICDE, pp. 643–654, 2005.
21. X. Xiong, M.F. Mokbel, W.G. Aref, S.E. Hambrusch and S. Prabhakar. **Scalable spatio-temporal continuous query processing for locationaware services**, SSDBM, pp. 317–326, 2004.
22. X. Yu, K.Q. Pu and N. Koudas. **Monitoring k-nearest neighbor queries over moving objects**, ICDE, pp. 631–642, 2005.
23. J. Zhang, M. Zhu, D. Papadias, Y. Tao and D.L. Lee. **Locationbased spatial queries**, SIGMOD, pp. 443–454, 2003.

@ 2012, IJACST All Rights Reserved