Continuous Reverse Nearest Neighbor Monitoring

Tian Xia Donghui Zhang*
College of Computer and Information Science
Northeastern University
360 Huntington Avenue, Boston, MA 02115
{*tianxia, donghui*}@*ccs.neu.edu*

Abstract

Continuous spatio-temporal queries have recently received increasing attention due to the abundance of location-aware applications. This paper addresses the Continuous Reverse Nearest Neighbor (CRNN) Query. *Given a set of objects O and a query set Q, the CRNN query monitors the* exact *reverse nearest neighbors of each query point, under the model that both the objects and the query points may move unpredictably. Existing methods for the reverse nearest neighbor (RNN) query either are static or assume a priori knowledge of the trajectory information, and thus do not apply. Related recent work on continuous range query and continuous nearest neighbor query relies on the fact that a simple* monitoring region *exists. Due to the unique features of the RNN problem, it is non-trivial to even define a monitoring region for the CRNN query. This paper defines the monitoring region for the CRNN query, discusses how to perform initial computation, and then focuses on incremental CRNN monitoring upon updates. The monitoring region according to one query point consists of two types of regions. We argue that the two types should be handled separately. In continuous monitoring, two optimization techniques are proposed. Experimental results prove that our proposed approach is both efficient and scalable.*

1. Introduction

Mobile devices and widespread wireless networks have brought a proliferation of location-aware applications. Examples include traffic monitoring, enhanced 911 service and mixed-reality games (e.g. BotFighters [3]). Such applications involve objects and query points that move unpredictably, where their locations are updated frequently. In this highly dynamic environment, results of traditional spatial queries are no longer static, but instead are constantly changing. A very recent trend in spatio-temporal database research is to continuously monitor such query results. Mokel *et al.* [8] proposed a framework called SINA which continuously monitors range queries over moving data. Iwerks *et al.* [4] presented a continuous fuzzy sets approach to maintain the spatial semijoin queries over linearly moving objects. Other work [9, 14, 16] addressed the problem of continuous nearest neighbor (CNN) monitoring.

This paper addresses the *continuous reverse nearest neighbor* (CRNN) query. Given a set of objects O and a query set Q, all being static or moving, the CRNN query monitors the *exact* reverse nearest neighbors of each query point over time. In this paper, we consider the monochromatic case of the RNN query where, an object o is considered as a query point q's reverse nearest neighbor, if there does not exist another object o' such that $dist(o, o') < dist(o, q)$.

An example of the CRNN query in the location-aware environment is the BotFighter mixed-reality game. In this game, players find and "shoot" other players using their mobile phones to gain points. Only nearby players can be shot. In order to avoid other players who will potentially shoot him, a player may register a monitoring query at the server to continuously monitor his RNNs. Another application of the CRNN query is in the battlefield, where a soldier registers a CRNN query to monitor the soldiers who might seek help from him.

Previous work on finding RNNs focused either on the static query [10, 15, 7, 12], or the predictive query [1]. The predictive query is based on the assumption of knowing the trajectory information. However, these techniques become inefficient or inapplicable in the CRNN monitoring problem. Unlike the static RNN query, the CRNN query requires updating the result set efficiently to reflect the current motion of objects and queries. To recompute the RNN query for every time instance is extremely inefficient, and usually the results will become immediately outdated in a highly dynamic environment. On the other hand, the movements of objects in real applications are usually unknown

*Partially supported by NSF CAREER Award IIS-0347600.

and the motion patterns are constantly changing [11]. Thus, the trajectory-based TPR-tree [13] will be too expensive to maintain for the CRNN query and its derived methods will be inefficient. To the best our knowledge, this paper is the first work that provides an *incremental* and *scalable* solution to the CRNN query.

In our paper, we do not make any assumption of the movement of objects. Instead, the objects periodically send updates of their locations to the central server. Multiple long running queries from geographically distributed clients are registered in the server. We use a grid index for the moving objects and query points, since more complicated index structures are expensive to maintain in a highly dynamic environment [16, 9]. The server usually processes (i.e. computes and updates) the queries in main memory to deal with the high rate of (object or query) location updates.

We call a region that may affect a query the *monitoring region* of that query. Our approach defines the monitoring region of a CRNN query point, and provides efficient and incremental algorithms to maintain the monitoring region. The monitoring region of a CRNN query point is intrinsically more difficult than that of a continuous range query or a CNN query, as it depends on the *object-object* relation (i.e. the distance between objects may change the results). As we will see, there is a need to differentiate two different shapes of regions for a CRNN query point: pie-shaped regions and circle-shaped regions. We provide a new algorithm to compute the initial results of a query point in a *filter-refinement* fashion, combining the method in [10] (denoted as *SAE*) and the conceptual space partitioning of [9].

After the initial computation, we maintain the monitoring regions (i.e. the pie-regions and circ-regions) incrementally. We monitor the pie-regions by using the conventional book-keeping information in the index and the query table. To maintain the circ-region, however, the conventional method will incur frequent cell updates even when the pie-regions are unchanged, and will perform unnecessary NN search operation to tightly bound the circ-regions. In our approach, we store the circ-regions separately using an in-memory FUR-tree (Frequent Updated R-tree [6]) and a hash table, so that the circ-regions do not need to be kept tight all the time. New algorithms are proposed to handle the updates happening inside the circ-regions. Two optimization techniques, *lazy-update* and *partial-insert* are also proposed to avoid unnecessary NN searches and reduce the updates on the FUR-tree.

The key contributions of this paper are summarized as follows.

1. We define the monitoring region of a CRNN query to consist of two parts: the pie-regions and the circ-regions. The defined monitoring region reduces a CRNN query to the monitoring of several (constrained) CNNs.

2. We provide an efficient extension of the SAE method [10] to compute the initial CRNN query result. The filter step performs NN searches simultaneously for up to six candidates and visits the optimal number of processed cells. The refinement step is partially integrated with the filter step.

3. Updates of the pie-regions and circ-regions are handled incrementally and efficiently. In particular, a new monitoring scheme is proposed to process the updates of the circ-regions, using an in-memory FUR-tree and a hash table. Two important optimizations, *lazy-update* and *partial-insert*, are also proposed.

The rest of this paper is organized as follows. Section 2 reviews existing work on RNN queries and continuous queries, focusing on the CNN query. Section 3 defines the monitoring region of a CRNN query. While Section 4 extends the SAE method to compute the initial results of the CRNN queries, Section 5 provides incremental algorithms to process the updates happening in the monitoring regions. Section 6 presents the experimental results. Finally, Section 7 concludes this paper.

2. Related Work

The RNN query was first studied by Korn and Muthukrishnan [5]. In their method, a circle is pre-computed for each object, and it is centered at the object and having its NN on the perimeter. A separate R-tree is used to index the set of such circles. Given a query point q, the RNN problem is then reduced to finding the circles containing q. Yang and Lin [15] improved the above method by *virtually* storing a circle along each object in the original R-tree. Every point in the leaf is augmented with the distance to its nearest neighbor (or *dnn*), and every index entry stores the max *dnn* of all points in the sub-tree. The augmented R-tree (named *Rdnn-tree*) not only reduces the storage, but also provides additional pruning during the RNN search.

The methods based on pre-computation incur extra update cost of maintaining the correct *dnn* for each object or index entry, especially in the dynamic environment, where objects may move. Stanoi *et al.* [10] proposed an approach (denoted as SAE) without pre-computation. SAE divides the space centered at the query q into six equal partitions of $60°$, from S_0 to S_5 shown in Figure 1. It can be proved that the only candidates of the RNNs are the six NNs of q in each partition. For instance, in Figure 1, the candidates of q's RNNs are o_1 in S_0, o_2 in S_1, o_3 in S_2, o_4 in S_3 and o_6 in S_4 (note that there are no objects in S_5). Since o_1 and o_4 are false positives whose NNs are o_0 and o_5, respectively, the RNNs of q are o_2, o_3 and o_6. SAE follows the filter-refinement framework, such that (1) it first finds six constrained NNs in each region as the candidates, and

(2) for each candidate, it performs an NN search to see if the candidate really considers q as NN. Later, Tao *et al.* introduced another efficient method (denoted as TPL) for the static RNN search. Given a query point q, TPL recursively prunes the space using the bisector between q and its NN in the unpruned space, until there is no object left. Then in the refinement step, TPL removes false positives by re-using the pruned MBRs.

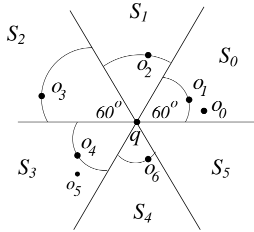
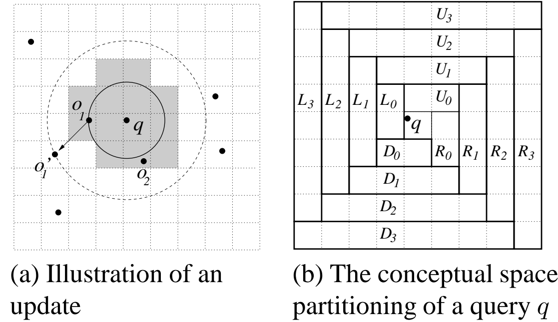


Figure 1. Illustration of SAE.

The CNN query was recently studied in [14, 16, 9]. and three methods (denoted as SEA-CNN, YPK-CNN, CPM-CNN, respectively) were proposed. All three methods share the same framework in handling the updates, by keeping a monitoring region for each query point. During the updates, the monitoring regions are adjusted when objects move in and out of the region. For example, in Figure 2(a), the initial NN of q is o_1. And q is associated with the shaded cells (approximation of the circle). When o_1 moves to o_1', a new NN search is performed, bounded by the enlarged circle with radius $d(q, o_1')$.

CPM-CNN proposed the idea of the conceptual space partitioning around each query. It improves the straightforward NN search on the grid by organizing the cells into conceptual rectangles for each query point. An example of the conceptual rectangles are shown in Figure 2(b). The rectangles are denoted by the direction (**U**p, **D**own, **L**eft or **R**ight) and the level (i.e. the number of rectangles between q and itself). To perform an NN search using CPM-CNN, we push the conceptual rectangles into the heap sorted by the mindist to the query. When a rectangle is examined, the cells within it is then pushed into the heap.

Recently Lee *et al.* [6] proposed the *Frequent Update R-tree* (FUR-tree), which incorporates localized bottom-up update strategies into the R-tree. An example is shown in Figure 3. The FUR-tree uses a secondary index (the hash table in Figure 3) for access to leaf nodes, as well as a direct access table for quick access to a node's parent. When locality is present in updates, the FUR-tree determines whether to enlarge the leaf MBR or insert the new object in a sibling leaf node. Otherwise, the standard R-tree insertion will be applied.



(a) Illustration of an update

(b) The conceptual space partitioning of a query q

Figure 2. An example of the CNN query.

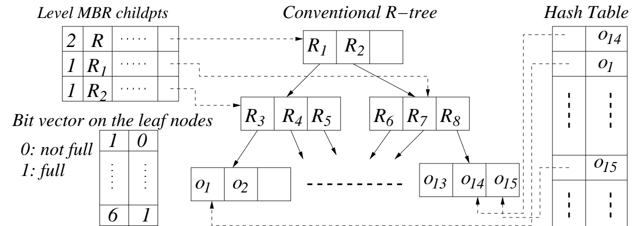


Figure 3. Illustration of the FUR-tree.

3. The Monitoring Region

This section starts with reviewing the existing monitoring regions for the continuous range query and the CNN query. Then the challenges for designing a monitoring region for the CRNN query are identified. Finally we present our CRNN monitoring region.

The *monitoring region* of a continuous query enables the possibility of incremental processing. It is a region around a query point, such that it guarantees the query results are unaffected as long as no update happens inside the region. The monitoring region should be as small as possible, since defining the whole space as the monitoring region is useless in query processing. Ideally, it should be the smallest region, in which the updates will affect the query result. Without ambiguity, we use both *monitoring region* and *region* interleavingly for the same meaning.

We approximate monitoring regions by collections of cells as shown in Figure 4. It is easy to observe that the monitoring region of a *continuous range query* is the query range (usually a rectangle or circle), and that of a *CNN query* is a circle centered at the query point and having the NN on the perimeter. Both of them have the following properties.

1. The region usually has a regular shape (e.g. a rectangle or a circle). In particular, the region of a continuous range query is fixed.

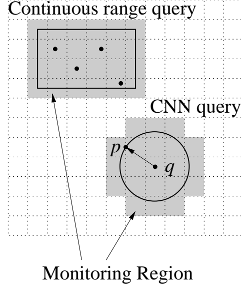


Figure 4. The monitoring regions of a continuous range query and a CNN query.

2. The region only contains the result objects. For example, in the continuous range query, any object entering the region will become a result.
3. The region does not rely on the distances between objects. This is because the continuous range and CNN queries only consider *object-query* (distance) relation.

However, defining the monitoring region of a CRNN query is intrinsically more difficult, as it does not satisfy any of the above properties. A straightforward proposal might be to consider the union of every circle, whose center is some RNN object and whose radius is its distance to the query point q. This is illustrated in Figure 5. The rationale is: whenever an object moves into such a circle of some RNN object O, we know O will not be an RNN anymore. Unfortunately, this approach is incorrect. In Figure 5, if o_5 moves to o_5', since the change happens outside the shadowed region, this approach will fail to detect the new RNN object o_4. Therefore, we should monitor some objects even though they are not RNNs.

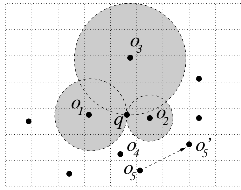


Figure 5. An example which shows the region that only covers the RNN objects is not a correct monitoring region for a CRNN query.

This example also illustrates that to monitor a CRNN query, it is necessary to consider the *object-object* relationship as well as the object-query relationship. While the distance between o_4 and q is unchanged, o_4 may become an RNN of q due to the change of distance between o_4 and o_5. This implies that, besides the RNNs, the monitoring region should also include additional objects, among which the change of distances will affect the query results.

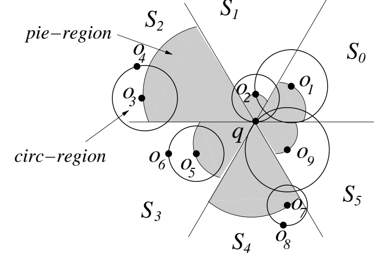


Figure 6. The monitoring region of a CRNN.

We define the monitoring region of a CRNN query point to consist of two parts: the ***pie-regions*** and the ***circ-regions***. As shown in Figure 6, the shadowed areas are the six pie-regions and the circles are six circ-regions. The pie-regions and circ-regions are further described as follows. Given a query point q, the space is divided into six partitions of $60°$ centered at q. As proved by [10], the candidates of the RNNs of q are the six constrained NN in the six partitions. A pie-region in a partition S_i is a pie centered at q and having the constrained NN in S_i on the perimeter. Intuitively, the pie-regions monitor the updates that will change the candidate set, and potentially affect the results. Furthermore, since the object-object relation is important in the CRNN query, the candidates in each partition and their nearby objects should also be monitored. A circ-region in a partition S_i is a circle centered at the candidate in S_i and having either q or an object nearer than q on the perimeter. The circ-regions monitor the updates that will make a previous result be a false positive, or make a previous false positive be an RNN. If a partition does not contain any object, there does not exist a circ-region in this partition, and the pie-region is extended to the border of the space.

One way of implementing the circ-region is to make its radius equal to the distance between the candidate and its NN. As we will see, such an implementation will incur expensive maintenance of the NN circles, and will perform unnecessary NN searches. In our method, we allow the radius of a circ-region to be the distance between the candidate and an arbitrary object that is nearer to the candidate than q, if the candidate is a false positive. To ensure the effectiveness of the circ-region, we propose two important optimizations, *lazy-update* and *partial-insert*, which will be presented in Section 5.

By introducing the monitoring region of a CRNN query, we reduce the problem to monitoring of several (constrained) CNNs. The following theorem proves the correctness of the monitoring region. Due to space limitations, the proofs of theorems in this paper are omitted.

Theorem 1 *The monitoring region of a CRNN query, consisting of pie-regions and circ-regions as shown in Figure 6, guarantees no update outside the monitoring region will affect the query results.*

However, not all methods computing the static RNN query will lead to the monitoring region of a CRNN query point. Although TPL [9] is the state-of-the-art method for the static RNN query, it is not suitable for the CRNN query. In TPL, the space is divided by the bisectors between the query point and objects, which depend on positions of objects. Such space partitioning is *object-dependent*, and is not fixed for a given query point. On the other hand, the space partitioning in SAE is *query-dependent*. The benefit of the *query-dependent* partitioning is that the partitions are fixed with a query and not affected by moving objects. This property enables the possibility of incremental process of the updates, and inspires the definition of the monitoring region of a CRNN query point.

Based on the monitoring region of a CRNN query point, in the next section, we present how to extend SAE to compute the initial results and initialize the monitoring regions of the CRNN query. In Section 5, we show how to maintain the pie-regions and circ-regions incrementally and efficiently, especially for the circ-regions.

4. CRNN Query Initialization

In this section, we extend SAE to efficiently compute the initial results and initialize the monitoring region of a query point. Each query point q is stored in a query table (QT), and is associated with six partitions (denoted from S_0 to S_5 as in SAE). For each such partition S_i, QT stores (1) the constrained NN $cand_i$ of q and the distance $d(q, cand_i)$; (2) an object nn_cand_i nearer to $cand_i$ than q and the distance $d(nn_cand_i, cand_i)$. If S_i does not contain any object, $cand_i$ and nn_cand_i are all $null$ and the corresponding distances are infinity. If q is the NN of $cand_i$, nn_cand_i is set to $null$ and the distance is set to $d(q, cand_i)$.

Figure 7 presents the algorithm to compute the initial results and the pie-regions/circ-regions for each query point. We combine SAE with the conceptual rectangles [9] in the filter step to avoid unnecessary visits of cells. In the following, we use *rectangle* referring to the conceptual rectangle without any ambiguity. Six constrained NN searches are performed simultaneously and cells are visited only once and when necessary. We illustrate the filter step with an example in Figure 8. Similar to the CPM NN search, in Step 2, the cell containing q and four rectangles around q (thick-lined in Figure 8a) is pushed into the heap. Partitions in which the constrained NN has been found are marked as *"finished"*. In order to ensure concurrent and optimal search for the six partitions, there are three cases where our method is different from the CPM NN search:

Algorithm *initCRNN(q, G)*
Input: Query q and the grid index G
Output: RNNs of q

1. *For each partition S_i of q, set $cand_i = null$, $nn_cand_i = null$, $d(q, cand_i) = \infty$, $d(nn_cand_i, cand_i) = \infty$, and a boolean $fin_i = false$*
2. *Initialize the heap H by pushing the cell containing q and four level zero conceptual rectangles into the heap.*
3. ***Repeat***
   - 3.1 *Get the next entry e from H.*
   - 3.2 *For each S_i intersecting with e and $fin_i = false$, if e's key is larger than $d(nn_cand_i, cand_i)$, set $fin_i = true$. If all such S_i's have $fin_i = true$,* ***continue****.*
   - 3.3 *If e's mindist expires, re-set the mindist and re-insert e into H,* ***continue****.*
   - 3.4 *If e is a rectangle, push the next level rectangle of same direction into H, and for each cell c in e,*
     - (1) *If c intersects with at least one partition with $fin_i = false$, compute the mindist of c and insert c into H.*
   - 3.5 *Else e is a cell, add q into the query list of e. For each object o in e,*
     - (1) *Compare with all $cand_i \neq null$ and update nn_cand_i and $d(nn_cand_i, cand_i)$ if necessary. Let $cand_j$ be the nearest candidate to o.*
     - (2) *Let o in S_k, update $cand_k$ and $d(q, cand_k)$ if necessary. If $cand_k = o$, update nn_cand_k and $d(nn_cand_k, cand_k)$, using q or $cand_j$, whichever is nearer.*
4. ***Until*** *all partitions S_i have $fin_i = true$ or H is empty.*
5. *For each S_i s.t. $nn_cand_i = q$, perform NN search on $cand_i$. Update nn_cand_i and $d(nn_cand_i, cand_i)$.*
6. *Output $cand_i$ if $cand_nn_i = q$.*

Figure 7. The CRNN initialization.

- C1. Throughout Step 3, the *mindist* of a cell/rectangle is the distance between the query point and the part outside the finished partitions. It preserves the tightest lower bound for the unpruned objects.
- C2. In Step 3.2, if a cell/rectangle is fully contained in the finished partitions, it is not pushed into the heap.
- C3. In Step 3.3, when a cell/rectangle is de-heaped, its mindist is checked whether it is *expired*. That is, the mindist no longer represents the part outside finished partitions (i.e. the actual mindist increases). A new mindist is computed and the cell/rectangle is reinserted into the heap.

An example of Case 2 (C2) is shown in Figure 8(a). When rectangle L_1 is popped from the heap, partition S_2 is marked as *finished*, as the mindist between L_1 and q is larger than $d(q, o_1)$. Cell $c_{2,5}$ (the shadowed cell) in L_1 is

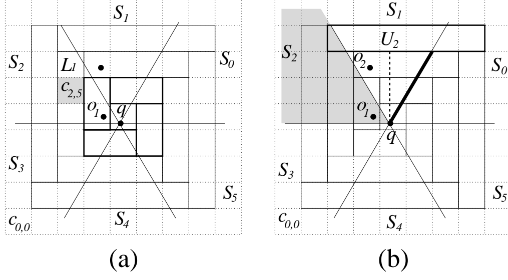


Figure 8. An example of the filter step of a CRNN query initialization.

not pushed into the heap. Figure 8(b) shows an example of Case 3 (C3). Partition S_2 is shadowed to indicate its *finished* status. When rectangle U_2 is popped from the heap, similar to S_2, S_1 is marked as *finished*. The mindist associated with U_2 (the thick dotted line) is expired as it does not represent the tightest lower bound for the unpruned the part of U_2 in partition S_0. A new mindist (the thick line) replaces the old one and U_2 is re-inserted into the heap without expansion. The rationale is that we may not need to examine the content of the cell/rectangle with the new mindist.

The following theorem proves the correctness of our filter step of the initialization.

Theorem 2 *The filter step of the CRNN query initialization returns correct results (up to six constrained NNs), and examines the minimum number of cells (i.e. it is optimal).*

To eliminate the false positives found in the filter step, we partially integrate the refinement step with the filter step in Step 3.5. An object examined in the filter step is used to invalidate any existing candidate if it is a false positive. Then in Step 5, we only need to perform NN searches for the candidates that haven't been invalidated before. The gain is to reduce the number of NN searches, since a false positive may be identified without performing an NN search.

5. Incremental CRNN Monitoring

In this section, we discuss how to store and maintain the pie-regions and circ-regions. Upon updates of objects, we incrementally modify the pie-regions and circ-regions associated with affected queries. When an existing query q moves to a new location, we treat the update as deleting q with the old location and re-compute q with the new location. We handle the query update in this way because re-computing a moving query is more efficient than updating from the old query result, as shown in [16, 9].

While pie-regions are stored using the traditional book-keeping technique, circ-regions are stored separately to ensure efficient maintenance. In the following, we first differentiate various cases of updating the pie-regions in Section 5.1. Next, Section 5.2 presents a new scheme to handle the updates in circ-regions, and proposes two optimizations, the *lazy-update* and the *partial-insert*.

5.1. Handling Updates in Pie-regions

Similar to previous work on continuous queries, we also use the book-keeping technique to store pie-regions in the grid index. Pie-regions are initialized in Step 3.5 of the algorithm *initCRNN* in Figure 7, when the cells intersecting with pie-regions are visited. When an object update happens in a pie-region, there are three cases according to the movement of the object, shown in Figure 9: (1) some object (o_4 or o_5) enters a pie-region; (2) a candidate (o_2 or o_5) leaves a pie-region; (3) a candidate (o_7) moves in the same pie-region. For clarity, we omit the circ-regions in Figure 9.

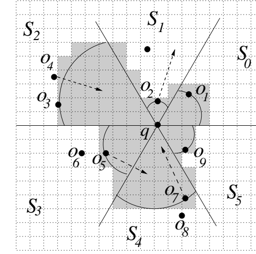


Figure 9. Three cases of updates in pie-regions.

Figure 10 describes the process of handling three cases upon an object update. The algorithm updates all the pie-regions affected by the old location and new location of an object. In the meantime, due to the change of candidates, expired circ-regions are updated by the newly computed circ-regions using Algorithm *updateCand* (to be presented in Section 5.2). Step 3.1 handles the first case, where a pie-region shrinks by setting the new candidate. Step 3.2 handles the second case, where a pie-region needs to be re-computed by constrained NN searches. Finally step 3.3 handles the third case, where the radius of a pie-region is re-computed. In all cases, a new circ-region may be computed. Again, similar to the optimization in the initialization process, we determine the circ-region centered at o by first searching in existing candidates. If no candidate can prove o as a false positive, a NN search is then performed.

Furthermore, Updating the pie-regions upon each object update may be inefficient when multiple updates happens in the same pie-region. Therefore, the algorithm *updatePie* could be extended to handle the updates of multiple objects, as follows. For each affected query point, the updates of objects are grouped by the six partitions, and for each affected

Algorithm updatePie (U, QT, G)
Input: An object update U, query table QT and the grid index G.

1. *Initialize an empty set C storing the changed circ-regions.*
2. *Suppose o issues U. $o.loc_{old} \in c_i$ and $o.loc_{new} \in c_j$*
3. *For each query q in either c_i's or c_j's query list,*

    / Case 1 */*

    3.1 *If o enters a pie-region in S_i, set $cand_i = o$ and $dnn_i = dist(o, q)$. Determine the new circ-region. Invoke* **updateCand** *to update the circ-region.*

    / Case 2 */*

    3.2 *If o is a candidate and leaves a pie-region in S_j, perform a constrained NN search in S_j to determine the new pie-region. If $cand_j \neq null$, determine the new circ-region. Invoke* **updateCand** *to update the circ-region.*

    / Case 3 */*

    3.3 *If o is a candidate and moves in a same pie-region in S_k, update dnn_k. Determine the new circ-region. Invoke* **updateCand** *to update the circ-region.*

Figure 10. Algorithm *updatePIE*.

partition, modify the pie region only once according to the nearest updated object.

5.2. Handling Updates in Circ-regions

Straightforwardly, we could store circ-regions by associating every cell that intersects with them. However, this method is expensive for circ-regions, for the following reasons. First, given a query q, unlike pie-regions, a circ-region is not always changed incrementally. That is, besides shrinking and enlarging of a circ-region, it may be changed to a completely different circle. An example is shown Figure 11. The circ-region around o_1 is changed to a different circle when o_1 moves to o'_1 along the perimeter of the pie-region. For clarity, we omit the pie-regions and some other circ-regions. Second, a circ-region may change frequently, even if the pie-region in the same partition is unchanged. For example, both circ-regions in S_1 and S_3 is changed while the corresponding pie-regions remain the same. Third and most importantly, in order to ensure the minimum number of cells associated with a circ-region, the straightforward method requires frequent computation of NN queries for candidates, even when the query results are unchanged. For example, also in Figure 11, although the update of o_6 does not change the query results, an NN search is still performed because of the enlargement of the circ-region.

Since the circ-regions do not need to be kept tight all the time, we propose to store the circ-regions separately, to eliminate the need for updating cells and unnecessary NN searches. Each query has up to six candidates around it.

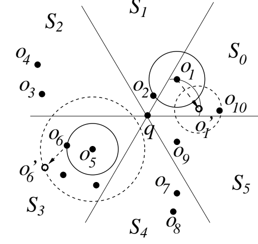


Figure 11. Updates in circ-regions.

Since candidates are always the constrained NNs in a query, the updates of candidates show strong locality, which could be handled efficiently by the FUR-tree. Therefore, the set of all candidate objects in the space is indexed globally by an in-memory FUR-tree, and the circ-regions corresponding to each candidate are virtually stored in the FUR-tree. That is, similar to the Rdnn-tree [15], for every candidate in the leaf, it stores the radius of the circ-region (i.e. $cand_dnn$), and for every index entry, it stores the max radius for all candidates in the sub-tree. Besides the radius, each candidate will also store the queries it belongs to. Also, as we mentioned in Section 3, given a candidate $cand$, a circ-region have either a query point q or any object nearer to $cand$ than q on the perimeter. We denote such object as nn_cand. We use a hash table, called *NN-Hash* to store the set of nn_cand's and the pointers to their corresponding candidates in the leaf. Intuitively, if a circ-region is affected by the old location of an object, it could be quickly accessed by following the pointer in NN-Hash.

Algorithm updateCand (o, o_nn, o_dnn, o', o_nn', o_dnn', q)
Input: Old candidate o, its NN o_nn and the radius o_dnn. New candidate o', its NN o_nn' and the radius o_dnn'. The query q.
Action: Update the old circ-region of q to the new one.

1. *Update the global FUR-tree using the modifi ed bottom-up strategies.*
2. *If the radius of the circ-region is changed, update and propagate the max radius stored in the index levels if necessary.*
3. *If o is deleted, update the o_nn in the NN-Hash. Add o_nn' to the NN-Hash if $o_nn' \neq null$.*

Figure 12. Algorithm *updateCand*.

Figure 12 presents the algorithm to update a circ-region in a partition of a query point. In step 1, the update is performed as if the old candidate o moves to the location of the new candidate o'. FUR-tree's bottom-up strategies are applied with two modifications: (1) o' is inserted into the FUR-tree's hash table, and (2) o is not deleted from the tree if o is associated with other queries. In the case $o' = null$, the modified bottom-up strategies are only applied to o. In

Algorithm updateCirc (U)
Input: An update U of object o moving from o_{old} to o_{new}.
Action: Update existing circ-regions if necessary.

1. *If o is in NN-Hash, locate the candidates $cand_i$ which consider o_{old} as their NN. For each $cand_i$,*
   1.1 *If $dist(o_{new}, cand_i) < dnn_i$, update $cand_dnn_i$ and propagate the radius to the root of the FUR-tree if necessary.*
   1.2 *Else perform NN search for $cand_i$ and radius in the FUR-tree.*
2. *Perform a query on the FUR-tree using o_{new} to locate all candidate $cand_i$, where $cand_dnn > dist(o_{new}, cand_i)$. For each $cand_i$, update $cand_nn$ and $cand_dnn$ accordingly. Propagate the new radius to the root of the FUR-tree if necessary.*

Figure 13. Algorithm *updateCirc*.

step 2 we may also need to update the max radius stored in the index entries, if the radius of the circ-region is changed. In step 3, if o is deleted, the pointer from o_nn to o should be deleted. We update o_nn in the NN-Hash. If o_nn does not point to any candidate, we remove it from the NN-Hash. Algorithm *updateCand* provides an efficient method to replace the old circ-region with the new one. Due to the high locality nature of the candidates, in most cases, the algorithm *updateCand* only accesses leaf nodes and propagate the radius upwards in logarithmic time.

5.2.1. The Lazy-Update Optimization

To update the existing circ-regions, we apply the *lazy-update* optimization to avoid unnecessary NN searches for candidates. Figure 13 presents the algorithm when an update affects the radii of circ-regions. Step 1 updates the circ-regions that affected by the old location of o. As long as the circ-region does not cover the query, it is guarantee that the corresponding candidate is still a false positive. Therefore, in this case, we only update the radius. An NN search is performed only when the enlarged circ-region covers q. As in the previous example in Figure 11, the unnecessary NN search for o_5 will not be performed. Step 2 performs a containment query on FUR-tree to retrieve the circ-regions which contain the new location of o. In this case, circ-regions are shrunk.

Algorithm *updateCirc* shows an efficient approach for changing the existing circ-regions. Especially, the lazy-update optimization ensures that NN searches are performed only when needed, and the effectiveness of circ-regions is still preserved. This is because the radius of a circ-region is *progressively* reduced by the updates issued inside the circ-region.

5.2.2. The Partial-Insert Optimization

To further reduce the number of radius updates, we propose another optimization called *partial-insert*. Instead of storing all the candidates and the radii of circ-regions in the FUR-tree, we only store the candidates whose circ-regions' radii are larger than a threshold, say, 80% of the distance to the query. Other candidates $cand$ and the corresponding nn_cand are stored in a hash table. As long as the distance between $cand$ and nn_cand is smaller than the threshold, we do nothing. Once a circ-region is shrunk below the threshold, its candidate is removed from the FUR-tree and inserted into the hash table. When a circ-region is enlarged above the threshold, it is inserted into the FUR-tree.

There are several gains using the partial-insert optimization. First, the size of the FUR-tree is reduced due to storing only the large circ-regions. Second, the containment query of a new location will retrieves less number of affected circ-regions and thus less number of circ-regions will be updated. Third, the insertion of a new candidate with a small circ-region is simple and efficient, which updates only the hash table.

6. Performance

In this section, we evaluate the performance of our approach to the CRNN query. We first compare our method with a straightforward extension of the R-tree based TPL method [12], which computes the results of all query points at every time stamp. In order to be fair in comparison, we perform TPL on the FUR-tree [6]. Since our method is in nature better than the straightforward approach, we then focus on comparing three variations of our methods, showing the effectiveness of the two optimizations, namely the *lazy-update* and the *partial-insert*.

6.1. Experimental Settings

All algorithms were implemented in Java, running on a PC with 2.66-GHz Pentium 4 processor and 1GB Memory. Our datasets were created with the *Network based Generator of Moving Objects* [2]. The input of the generator is the road map of Oldenburg (a city in Germany). The output is the set of moving objects on the road network. We generated the moving queries in the same way as the objects. The parameters of the datasets we used are listed in Table 1. The values that are in bold and italic face are the default values in our experiments. The object mobility is represented by the percentage of object location updates. The query point mobility is defined similarly.

In accordance with the real world monitoring systems, which usually assumes main-memory processing to handle the intensive updates [9, 16], all our experiments were per-

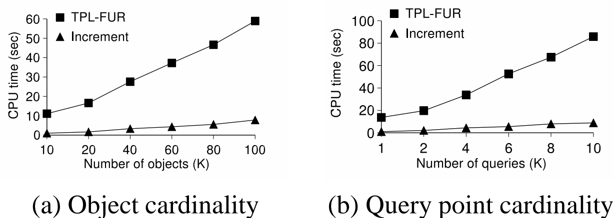
Parameter	Values
# of objects (K)	10, 20, 40, 60, ***80***, 100
# of query points (K)	1, 2, 4, ***6***, 8, 10
Object mobility (%)	1, 5, ***10***, 15, 20
Query point mobility (%)	1, ***5***, 10, 15, 20

Table 1. Dataset Parameters.

formed in memory and the CPU time (in seconds) are reported. The queries were evaluated at every time stamp. We simulated 30 time stamp and report the average of the CPU time of updating (excluding the initialization). In our method, the fan-out of the in-memory FUR-tree containing the circ-regions are 20 entries. The threshold in the *partial-insert* optimization is 80%. We use a grid index of 128×128 cells. In the straightforward solution, we utilize the FUR-tree with the fan-out of 50 entries for the TPL method.

6.2. Comparison with TPL

We compare our approach (using both lazy-update and partial-insert optimizations) with the straightforward solution using the TPL method. The TPL method is current best approach for computing RNNs in the static case, and the FUR-tree is the optimized for frequent updates of objects. To answer the CRNN query in the straightforward solution, we index the objects using an FUR-tree and compute the RNNs of every query point at each time stamp using TPL. The comparison are shown in Figure 14. **TPL-FUR** represents the straightforward solution, and **Increment** represents our method. Figure 14(a) and Figure 14(b) shows the update time when varying the number of objects and the query points, respectively. As expected, our method outperforms the straightforward approach more than an order of magnitude. Our method is well scalable with increase of the number of objects and query point.



(a) Object cardinality (b) Query point cardinality

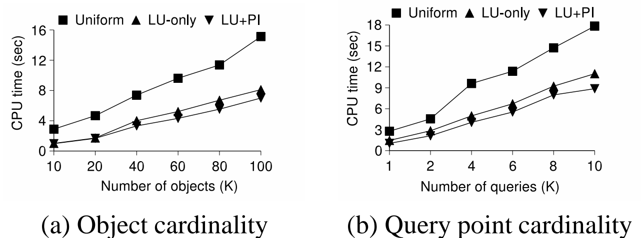
Figure 14. Comparison with the straightforward solution.

In the next section, we focus on our own approach, studying the effect of handling the circ-regions separately, using two optimizations, namely, the lazy-update and the partial-insert.

6.3. Effect of the Optimizations

Three variations of of our approach are compared. The names and their descriptions are listed below.

1. **Uniform**: stores the pie-regions and the circ-regions uniformly, by associating with query points the cells which intersect with either pie-regions and circ-regions. When a circ-region changes, it performs an NN search to keep the region smallest.
2. **LU-only**: stores the circ-regions separately while only applies the *lazy-update* optimization.
3. **LU+PI**: is our complete approach, applying both *lazy-update* and *partial-insert* optimizations.



(a) Object cardinality (b) Query point cardinality

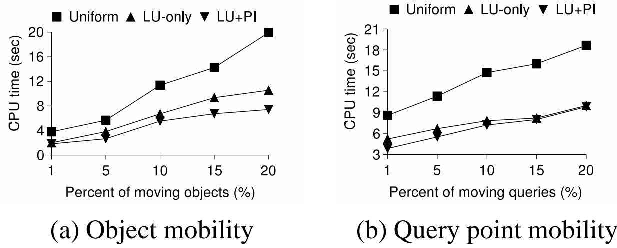
Figure 15. Varying the data size.

Figure 15(a) shows the performance of three methods when varying the number of objects and query points and fixing the percentage of updates at each time stamp. The update cost increases as the number of objects increases. When the number of objects are small, the difference between *LU-only* and *LU+PI* is negligible. Since *LU+PI* improves the already efficient method *LU-only*, only large amount of updates on circ-regions will show the performance difference. As shown in the figure, *LU+PI* visibly outperforms *LU-only* when the number of objects is larger than 40K.

Figure 15(b) shows the performance when the number of query points changes. With the increase of query points, the number of pie-regions and circ-regions increases as well. We see more improvement of *LU+PI* over *LU-only* in Figure 15(b) than in Figure 15 (a).

In both figures, *LU-only* and *LU+PI* outperforms *Uniform*, as *Uniform* handles circ-regions expensively. With the increase of the objects or query points, the update cost of *Uniform* increases faster.

In the next set of experiments, we study the effect of data mobility. Figure 16(a) compares three methods by varying the mobility of objects. With the increase of object updates, the number of affected pie-regions and circ-regions increases. Recall that an pie-region update will trigger updating the candidate and updating the corresponding circ-regions. When the number of object update is large (e.g.



(a) Object mobility (b) Query point mobility

Figure 16. Varying the percentage of moving data per time stamp.

20%), the cost of updating circ-regions dominate the total cost and *LU+PI* outperforms *LU-only* as large as 30%. Consequently, *Uniform* shows a large increase of CPU time at the 20 mark on the x-axis.

Figure 16(b) gives the performance of three methods when the percentage of mobile queries is varied from 1% to 20%. In this case, the number of updates of objects is fixed. When few query points issue updates, the total cost is dominated by updating circ-regions, caused by the moving objects. We see 25% improvement of *LU+PI* over *LU-only* at the 1 mark on the x-axis. With the increase of query point updates, the total cost become dominated by recomputing the new location of a query point. Therefore, while the CPU time of *LU+PI* and that of *LU-only* increase with the increase of the query mobility, the difference between them decreases. Again, as expected in Section 5.2, *Uniform* performs much less efficiently than other two methods, for it incurs many unnecessary NN searches for candidates.

7. Conclusions

Motivated by location-aware applications, this paper addressed the problem of Continuous Reverse Nearest Neighbor (CRNN) monitoring. Objects are indexed by a regular grid. We presented an incremental and scalable approach based on the proposed concept of CRNN monitoring region. For one query point, the CRNN monitoring region consists of up to 6 pie-regions and 6 circ-regions. Initial computation of the monitoring region and query result is done by integrating an existing RNN technique for static objects with the space partitioning model used in other continuous monitoring papers. For incremental maintenance, only updates of locations that fall into the monitoring region may affect the previously maintained query result. We store and maintain the pie-regions and circ-regions separately. The pie-regions are stored using a book-keeping technique. The circ-regions are stored using an in-memory FUR-tree and a hash table. Two optimizations (*lazy-update* and *partial-insert*) have been proposed to efficiently handle the updates of the circ-regions. Experimental results proved that our approach is over an order of magnitude more efficient than straightforward methods.

References

[1] R. Benetis, C. S. Jensen, G. Karčiauskas, and S. Šaltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proc. of Int. Database Engineering & Applications Symposium (IDEAS)*, pages 44–53, 2002.

[2] T. Brinkhoff. A Framework for Generating Network-Based Moving Objects. *GeoInformatica*, 6(2), 2002.

[3] J. Elliott. Text Messages Turn Towns into Giant Computer Games. *Sunday Times*, April 29, 2001.

[4] G. S. Iwerks, H. Samet, and K. Smith. Maintenance of Spatial Semijoin Queries on Moving Points. In *VLDB*, pages 828–839, 2004.

[5] F. Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *SIGMOD*, pages 201–212, 2000.

[6] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting Frequent Updates in R-Trees: A Bottom-Up Approach. In *VLDB*, pages 608–619, 2003.

[7] A. Maheshwari, J. Vahrenhold, and N. Zeh. On Reverse Nearest Neighbor Queries. In *Proc. of Canadian Conf. on Computational Geometry (CCCG)*, pages 128–132, 2002.

[8] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, pages 623–634, 2004.

[9] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD*, pages 634–645, 2005.

[10] I. Stanoi, D. Agrawal, and A. El Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *ACM/SIGMOD Int. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 44–53, 2000.

[11] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present, and the Future in Spatio-Temporal. In *ICDE*, pages 202–213, 2004.

[12] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, pages 744–755, 2004.

[13] S. Šaltenis, C. S. Jensen, S. Leutenegger, and M. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD*, pages 331–342, 2000.

[14] 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.

[15] C. Yang and K.-I. Lin. An index structure for efficient reverse nearest neighbor queries. In *ICDE*, pages 485–492, 2001.

[16] X. Yu, K. Q. Pu, and N. Koudas. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *ICDE*, pages 631–642, 2005.