Continuous Reverse *k*-Nearest-Neighbor Monitoring

Wei Wu
Singapore-MIT Alliance
National University of Singapore
wuw@nus.edu.sg

Fei Yang Chee-Yong Chan Kian-Lee Tan
School of Computing
National University of Singapore
{yangfei,chancy,tankl}@comp.nus.edu.sg

Abstract

The processing of a Continuous Reverse k-Nearest-Neighbor (CRkNN) query on moving objects can be divided into two sub tasks: continuous filter, and continuous refinement. The algorithms for the two tasks can be completely independent. Existing CRkNN solutions employ Continuous k-Nearest-Neighbor (CkNN) queries for both continuous filter and continuous refinement. We analyze the CkNN based solution and point out that when $k > 1$ the refinement cost becomes the system bottleneck. We propose a new continuous refinement method called CRange-k. In CRange-k, we transform the continuous verification problem into a Continuous Range-k query, which is also defined in this paper, and process it efficiently. Experimental study shows that the CRkNN solution based on our CRange-k refinement method is more efficient and scalable than the state-of-the-art CRkNN solution.

1. Introduction

A Reverse k-Nearest-Neighbors (RkNN) query issued from object (or location) q returns the objects whose k nearest neighbors include q. Formally, $RkNN(q) = \{o | q \in kNN(o)\}$, where $kNN(o)$ is object o's k nearest neighbors [7]. Reverse k-Nearest-Neighbors queries are gaining research interests due to its applications in decision making (discovery of the influence sets) [6, 10], location based services, and computer (and mobile) games.

In this paper, we investigate the problem of processing Continuous Reverse k-Nearest-Neighbor (CRkNN) queries on moving objects. Specifically, we look at the in-memory processing of CRkNN queries based on moving objects' location updates (and location uncertainty is not taken into account). A CRkNN query's result needs to be kept up-to-date when moving objects update their locations. Evaluating such queries is challenging because of two reasons: 1) both *object-query* distance and *object-object* distance play an important role in CRkNN processing, and therefore an object's location update may cause several changes to a CRkNN query's result; 2) moving objects have high location update rates.

A filter-refinement framework is effective in processing RkNN queries, as shown in both snapshot and continuous RkNN algorithms [9, 11, 13, 5]. In the filter phase, most objects that are not query results are pruned using some techniques (please refer to Section 2.1 for a review of RkNN pruning methods), and this phase leaves a relatively small number of objects called *candidate*s. In the refinement phase, each candidate is checked to see whether its kNNs include the RkNN query object (i.e. $q \in kNN(o)$).

In the processing of a continuous RkNN query, both filter and refinement need to be done continuously because object movement may change both filter result and refinement result. In this paper we call them continuous filter and continuous refinement.

It is desirable to have a CRkNN solution that makes use of existing spatio-temporal system components. Existing CRkNN solutions [13, 2] use variants of Continuous k-Nearest-Neighbor (CkNN) queries for both filter and refinement.

We analyze the CkNN based solution and point out: continuous refinement dominates the CRkNN processing cost when $k > 1$, and this solution does not scale with k, because the number of candidates increases with k and maintaining each candidate's kNNs continuously is expensive.

We present a continuous refinement method called CRange-k. An object o's kNN set includes q if and only if there are fewer than k objects that are nearer to o than q is. We verify a candidate o of a CRkNN query q[1] as follows: we continuously check whether the number of objects whose distances to o are shorter than $dist(o, q)$ is fewer than k. Here $dist(o, q)$ is the distance between o and q.

We formalize this verification method as a new kind of query called Range-k query. A Range-k query is specified as $< o, r, k >$ where o is an object, r is a distance, and k is a threshold value. Its result is the value of the expression

[1] In this paper, "a CRkNN query q" means a CRkNN query issued from object q.

$|\{p|dist(o,p) < r\}| < k$, i.e. a Boolean value (True/False) indicating whether there are fewer than k objects within the specified distance r to the object o. A continuous Range-k query is cheap to process because we only need to maintain a count, and both r and k can be used to minimize the query's monitoring region. We present algorithms for the efficient evaluation of continuous Range-k queries.

We did extensive experiments to study the performance of our CRange-k continuous refinement method. Experimental results show that the CRkNN solution based on CRange-k performs much better than the existing CkNN based solution.

The rest of the paper is organized as follows. Section 2 surveys related works. In section 3 we analyze CkNN based CRkNN solution and identify its bottleneck. Then we present our CRange-k verification method in Section 4. Experimental study is shown in Section 5. Finally, Section 6 concludes this paper.

2. Related Works

Various algorithms have been proposed for processing snapshot Reverse k-Nearest-Neighbor (RkNN) queries (including Bi-chromatic RNN) in different settings [9, 10, 8, 11, 1, 14, 12]. Recently researchers have begun to work on continuous RNN and RkNN queries [13, 5, 12].

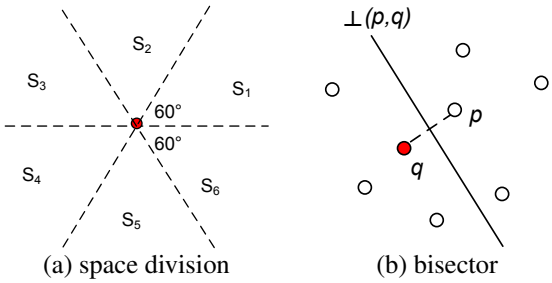
Our work focuses on the exact answer of continuous RkNN query in Euclidian space (w.r.t the moving objects' reported positions in the database). In the following we review the works that are closely relevant to ours. All these works employ a filter-refinement framework.

2.1. Snapshot RkNN

The emphasis of algorithms for snapshot RkNN queries has been to develop efficient filter methods that can prune as many objects as possible. Two interesting filter methods have been developed for RkNN processing. In this paper, we call them 60-degree-pruning and TPL-pruning.

The 60-degree-pruning method is developed by Ioana Stanoi et al. in [9] for processing RNN queries. Its idea is illustrated in Figure 1(a): the space around a RNN query q can be divided into six equal-size regions $(S_1, S_2, \ldots, S_6)$, then (thanks to the special property of 60 degree angle) in each region only the nearest neighbor of q can possibly be a reverse nearest neighbor of q. By this, q's RNN candidates are restricted to q's nearest neighbor in each subspace. Based on this idea, a RNN query is processed as follows: six constrained nearest neighbor queries [4] are used to find the candidates, then a nearest neighbor query is used to verify each candidate.

This 60-degree-pruning method can be extended easily to the general case where $k \geq 1$: in each sub-space, only the RkNN query point's k nearest neighbors are result candidates. An example is shown in Figure 2(a). This filtering method results in $6 * k$ candidates for each RkNN query.



(a) space division (b) bisector

Figure 1. Pruning techniques.

The TPL-pruning method is proposed by Yufei Tao, Dimitris Papadias, and Xiang Lian in [11]. The basic idea of TPL is illustrated in Figure 1(b): the perpendicular bisector between the query point q and an arbitrary object point p divides the space into two half planes, i.e. the $PL_q(p,q)$ that contains q and the $PL_p(p,q)$ that contains p, then the points in $PL_p(p,q)$ cannot be a RNN of q because p is closer to them than q is. Similarly, for RkNN, an object can be pruned if the object is in at least k $PL_p(p,q)$ planes. Based on this finding, the TPL algorithm processes a RkNN query as follows. In the filter step, objects near query point q are used to (draw perpendicular bisectors to) do filtering; these objects and the objects that are not pruned are the candidates. In the refinement step, candidates are verified by checking whether q is one of their k nearest neighbors. Experiments in [11] show that the number of candidates after TPL-pruning is normally between $2 * k$ and $3 * k$.

2.2. Continuous RkNN

The emphasis of existing Continuous RkNN (CRkNN) query processing algorithms is on defining the monitoring region[2] of a CRkNN query and updating the query result based on moving objects' location updates.

Tian Xia and Donghui Zhang [13] investigated the processing of Continuous RNN (CRNN, i.e. $k = 1$) queries. Their method is based on the 60-degree-pruning technique. The monitoring region of a CRNN query is defined as six pie-regions (determined by the query point and the six candidates) and six cir-regions (determined by the six candidates and their nearest neighbors). Then for a CRNN query: in each sub-space a *continuous constrained nearest neighbor* query is used to monitor the candidate in that sub-space (continuous filter). For each candidate, a *continuous nearest neighbor* query is used to do continuous refinement.

[2] A continuous query's monitoring region is the region where object location updates will trigger the query's continuous processing.

In [2], while mainly focusing on processing predictive kNN and RkNN queries with the TPR-tree, Rimantas Benetis, Christian S. Jensen et al. also discussed the problem of processing CRNN and CRkNN queries. Their CRkNN processing solution is based the extension of the 60-degree-pruning technique. In each sub-space, the set of candidates is continuously maintained, and each candidate's kNNs are also continuously maintained to do continuous refinement.

In [5], James M. Kang, Mohamed F. Mokbel et al. presented an CRNN (i.e. $k = 1$) processing algorithm called *IGERN*. *IGERN* utilizes the TPL-pruning method. As such, it monitors fewer candidates, and is more efficient in CRNN monitoring than the 60-degree-pruning based solution [13].

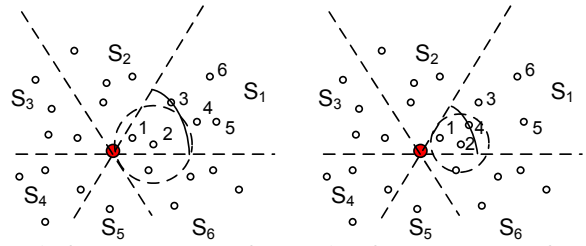
Since TPL-pruning results in fewer candidates, it will be appealing to apply TPL-pruning in general CRkNN monitoring where $k \geq 1$. Unfortunately, when $k > 1$, defining the continuous monitoring region for TPL-pruning and designing an incremental TPL-pruning algorithm are non-trivial, and no one has proposed a solution. The monitoring region defined in *IGERN* [5] only applies to CRNN ($k = 1$). Also for this reason, the *IGERN* algorithm cannot be extended easily to handle CRKNN queries where $k > 1$.

Our work in this paper differentiates itself from the existing works by focusing on continuous verification rather than continuous filter. In the next section, we analyze CRkNN processing and show why continuous refinement deserves our attention.

3. Analysis of CkNN based CRkNN Processing

The processing of a Continuous Reverse k-Nearest-Neighbor (CRkNN) query can be divided into two tasks: continuous filter, and continuous refinement. Continuous filter maintains the query's result candidates (the objects that are not pruned with the applied pruning technique). The job of continuous refinement is to continuously verify whether each candidate is a query result. Here "continuous" means both the candidates set and each candidate's state (whether it is result object) are maintained up-to-date. Note an object's movement may change both candidate set and candidates' states.

Figure 2 depicts an example of the changes an object's movement can cause to a CR3NN ($k = 3$) query. In this example, the 60-degree-pruning method is used to do filtering and 3NN query is used to do refinement. For clarity, we only show the pie-region (monitoring region for continuous filter) in S_1 and the circ-region (monitoring region for continuous refinement) of object 2. In Figure 2(a), i.e. Before the movement of object 4, the candidates in sub-space S_1 are object 1, 2, and 3 because they are the query point's 3NNs in S_1, and the object 2 is a query result because its 3NN includes the query point. After the movement of object 4, in Figure 2(b), the candidates in S_1 become object 1, 2 and 4, and object 2 is not a result object anymore because its 3NN now does not include the query point.



(a) before movement of o_4 (b) after movement of o_4

Figure 2. Example of impact of object movement on RkNN.

In CRkNN monitoring, the interaction between the two tasks (filter and refinement) are very simple: when a new candidate is identified in continuous filtering, the system starts the candidate's continuous refinement; when an object is not a candidate anymore, the system ends the candidate's continuous refinement. Therefore, the algorithm for continuous filter and the algorithm for continuous refinement are independent[3].

Up to now, in the existing CRkNN ($k \geq 1$) solutions (please refer to Section 2 for a review), both the continuous filter method and the continuous refinement method are based on Continuous kNN (CkNN) queries. The continuous filter method applies the 60-degree-pruning technique: from the CRkNN query point, the space is divided into six 60-degree sub-spaces, and in each sub-space the query point's kNNs are the candidates (see Figure 2 for an example); then six Continuous Constrained k-Nearest-Neighbor queries are used to do continuous filter (i.e. monitor the query point's six kNNs in the six sub-spaces). The continuous refinement method is to maintain each candidate's kNNs with a CkNN query (see Figure 2 for an example).

3.1. Bottleneck of CkNN based Solution

Here we compare the costs of continuous filter and continuous refinement, with the objective to determine the bottleneck of the existing CkNN based CRkNN solution.

In continuous filter, six Continuous Constrained k-Nearest-Neighbor (CCkNN) queries are used to maintain a CRkNN query's candidates, the cost is therefore $6 * Cost_{cccknn}$ where $Cost_{cccknn}$ is the cost of processing a CCkNN query. CCkNN query can be processed by adapting existing continuous kNN (CkNN) query processing techniques, i.e. considering only the objects in the specific

[3] As a result, the continuous refinement method proposed in this paper can integrate with any continuous filter method to construct a CRkNN solution.

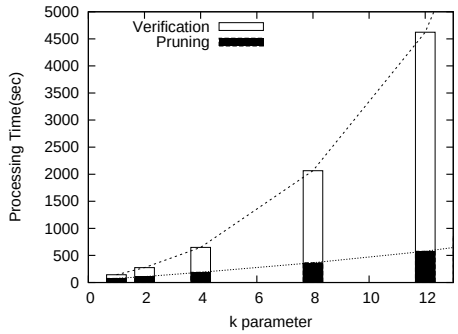


Figure 3. Filter vs. refinement

region. Adding a filter condition to the existing CkNN query processing algorithm will not add much additional cost, therefore we can write the cost of a CCkNN query as $a * Cost_{cknn}$ where a is a constant little larger than 1. Then the cost of continuous candidate monitoring of a CRkNN query is $6 * a * Cost_{cknn}$ where $Cost_{cknn}$ is the cost for processing a CkNN query.

For the continuous refinement task, a CkNN query is used for each candidate, hence the cost of continuous refinement of a CRkNN query is $\sharp Can * Cost_{cknn}$ where $\sharp Can$ is the number of candidates of the CRkNN query. As we mentioned, the 60-degree-pruning method results in $6 * k$ candidates[4]. Therefore the cost of continuous refinement is $6 * k * Cost_{cknn}$.

The ratio of continuous refinement cost ($Cost_{cr}$) to the continuous filter cost ($Cost_{cf}$) is

$$\frac{Cost_{cr}}{Cost_{cf}} = \frac{6 * k * Cost_{cknn}}{6 * a * Cost_{cknn}} = \frac{k}{a} \quad (1)$$

Since $1 < a < 2$, the cost of continuous refinement dominates the total cost of CRkNN processing when $k > 1$. More importantly, when k increases, the cost of continuous refinement increases much faster than the cost of continuous filter.

Our experiments verify this analysis (please refer to Section 5 for the setup of this experiment). Figure 3 shows the breakdown of CRkNN cost (when CkNN is used for continuous refinement) with the increase of k. It is clear that the continuous refinement time dominates the total processing time, and the ratio of continuous refinement cost to the continuous filter cost increases with k.

This result indicates that in CRkNN, when $k > 1$, the continuous refinement will become the bottleneck of the CRkNN monitoring system. In the following sections, we propose our efficient method for doing continuous refinement.

[4]Even if TPL-pruning can be applied to CRkNN, the result of the analysis still holds, because in TPL-pruning the number of candidates also increases with k.

4. CRange-k Continuous Refinement

In this section, we propose a new continuous refinement method called CRange-k.

Given a CRkNN query q's candidate o: object q is one of object o's kNNs if and only if the number of objects, whose distances to o are shorter than $dist(o, q)$, is smaller than k, where $dist(o, q)$ is the distance between o and q. A mathematical representation of the observation is:

$$q \in kNN(o) \Leftrightarrow |\{p : dist(p, o) < dist(q, o)\}| < k \quad (2)$$

Verifying whether o is a result object can be transformed to such a question: is the number of objects whose distances to o are shorter than $dist(q, o)$ smaller than k? To facilitate discussion, we formalize this question as a new kind of query called *Range-k Query*. A Range-k query is specified as $< o, r, k >$ where o is an object, r is a distance, and k is a threshold value, and the query's result is the boolean value of the following expression: $|\{p : dist(p, o) < r\}| < k$.

Notice the result of the Range-k query $< o, dist(q, o), k >$ tells whether o is a result object q. The continuous refinement can be done with a corresponding continuous Range-k query.

It is worth noting that Range-k query is closely related to kNN query and Range query. A continuous Range-k query can even be evaluated with any continuous kNN query algorithm or continuous Range query algorithm. However, an algorithm specifically designed for Range-k query can be much more efficient. We will discuss this in detail in Section 4.3.

In the remainder of this section, we present our CRange-k continuous verification algorithm by describing it as an continuous Range-k query processing algorithm.

In Range-k query, we are only interested in the relationship between k and the number of objects within the range. Therefore, the processing of a Range-k query can terminate as soon as their relationship is clear. Our algorithm makes use of this property.

In the algorithm, we use a grid to index both moving objects and Range-k queries. The objects are mapped to the grid cells based on their locations. A Range-k query is indexed in a minimal set of cells such that by monitoring the object movements in these cells we can keep the query result refresh.

The algorithm consists of two parts: a *Search* part and a *Maintenance* part. *Search*'s functionality is to compute the query's result by looking for objects that are within the given range, and to index the query into proper cells. An object's location update triggers the Maintenance of the queries indexed in the corresponding cell (or cells when the object moves from one to another cell). *Maintenance*'s

main functionality is to update the query's result upon object location changes. *Maintenance* uses *Search* to find more objects if necessary. The two parts share and maintain the following information for each Range-k query:

• count: the number of objects that are within the circle found in visited cells;

• V: set of cells checked so far;

• U: a FIFO queue of cells that we need to check.

In the above, "circle" means $circle(o, r)$, i.e. the circle centered at o with radius r.

A continuous Range-k query is first processed with *Search*, and then continuously maintained by *Maintenance*. We present *Search* and *Maintenance* by looking at the initial process and continuous maintenance of a Range-k query.

4.1. Initial Processing

When a Range-k query $< o, r, k >$ is issued, the information we are going to maintain for this query is initialized as follows: $count \leftarrow 0$; $V \leftarrow \emptyset$; $U \leftarrow cell(o)$. Here $cell(o)$ means the cell that contains object o. Then the *Search* routine is used to do the initial processing of the query.

Here is an overview of how *Search* processes a new Range-k query: starting from the cell that contains object o, the cells that intersect with $circle(o, r)$ are checked until one of the following two conditions is satisfied: 1) $count \geq k$, which means at least k objects are within the given range; 2) all the cells intersecting with $circle(o, r)$ have been checked and the count is smaller than k. In the first case, False is returned; in the second case, True is returned. During the course, the query is indexed into the visited cells.

Figure 4 lists the pseudo-code of the *Search* routine.

This *Search* routine returns false if $count \geq k$ after a cell is checked. This is to visit as few cells as possible and to index the query in a minimal number of cells (object movement in these cells will trigger the query's *Maintenance*).

One detail of the algorithm is that we distinguish the cells totally covered by the circle from the cells partially intersected with the circle. We avoid checking objects in totally covered cells. For a covered cell, all objects in it are in the circle so we just increase the count directly by the cell's number of objects (lines 7-8). For a partially intersected cell, we check each object in it to determine how many are in the circle and increase the count accordingly (lines 9-13).

Another detail of the algorithm is that we look at the neighbors of an intersecting cell to find other cells that intersect with the circle. This is based on the following observation: the cells that intersect with a given circle are connected. For a new Range-k query, U is initialized with $cell(o)$ (which must intersect with the circle). Then when checking a cell that intersects with the circle, its un-checked neighbors are added to U. By this, we visit the cells that intersect with the circle layer by layer, and the cells totally covered by the circle are likely to be visited early.

```
Range-k Search (o, r, k, count, V, U)
// Inputs
o: a Range-k query point; r: the range; k: the k value;
count, V, U: see Section 4 for description.
// Output: result of Range-k query < o, r, k >.
1.  WHILE U ≠ ∅
2.    e = deque(U)
3.    IF e intersects with circle(o, r)
4.      put e into V
5.      put neighbours(e) − V − U into U
6.      add reference of this query into cell e
7.      IF e is totally covered by circle(o, r)
8.        count = count + e.♯objects
9.      ELSE
10.       FOR EACH object o in e
11.         IF o is in circle(o, r)
12.           THEN count = count + 1
13.       END FOR
14.     END IF
15.   END IF
16.   IF count ≥ k THEN RETURN false
17. END WHILE
18. RETURN true
```

Figure 4. The Search routine of Range-k

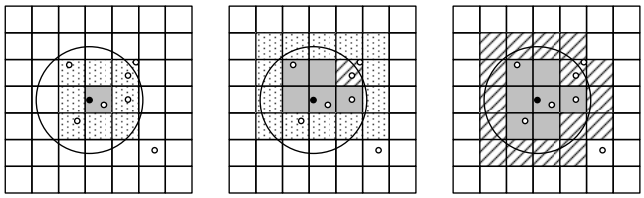


Figure 5. *Search*'**s Sample States**

Figure 5 shows some sample states a *Search* can be in. In the figures, gray cells are the covered cells, shaded cells are the partially covered cells, and dotted cells are the cells in the to-be-visited queue.

4.2. Continuous Maintenance

Maintenance does an incremental processing of the query and keeps both the data structures and the query's result up-to-date.

In *Search*, we put a reference of the Range-k query to the visited cells that intersect with $circle(o, r)$. For example, they are the grey cells and line shaded cells in Figure 5.

With the reference of the Range-k query in these cell, the incremental maintenance for the query will and only will be triggered by the location updates that may change the query's result. An object's movement triggers the *Maintenance* of the queries indexed in its starting and ending cells (they could be the same cell).

Maintenance (Figure 6) does an incremental processing of a query when triggered by an object's movement. *Maintenance* examines the impact of this movement on the count (the number of objects that are within range and in the visited cells). If the object moved from a visited cell to a visited cell, we check both its starting and ending location to see if count needs to be updated (case 1, lines 3-8). If the object moved from a visited cell to an un-visited cell, we only need to check its starting location to see if count needs to be decreased (case 2, lines 9-12). If the object moved from an un-visited cell to a visited cell, we only need to check its ending location to see if count needs to be increased (case 3, lines 13-16). Note if an object moved from an un-visited cell to another un-visited cell, it will not trigger this query. And notice that the invariant of the algorithm is that count is the number of objects that are *in the visited cells* and in the circle.

Range-k Maintenance (o, r, k, $count$, V, U, p)
// Inputs
o, r, k, $count$, V, U: same as in *Search*;
p: a moving object;
// Output: result of Range-k query $< o, r, k >$.

1. $sc \leftarrow$ the cell p was in before move
2. $ec \leftarrow$ the cell p is in after move
3. IF $sc \in V$ AND $ec \in V$ //case 1
4. IF p moves into $circle(o, r)$ from outer
5. THEN $count = count + 1$
6. IF p moves out of $circle(o, r)$ from inner
7. THEN $count = count - 1$
8. END IF
9. IF $sc \in V$ AND $ec \notin V$ //case 2
10. IF p was in $circle(o, r)$ before move
11. THEN $count = count - 1$
12. END IF
13. IF $sc \notin V$ AND $ec \in V$ //case 3
14. IF p was in $circle(o, r)$ after move
15. THEN $count = count + 1$
16. END IF
17. IF $count \geq k$
18. RETURN false
19. ELSE
20. RETURN ***Search***(o, r, k, $count$, V, U)
21. END IF

Figure 6. Incremental *Maintenance* of Range-k

After updating the count, if $count \geq k$ then query result false is immediately returned (lines 17-18), otherwise *Search* will be invoked to compute the result of the query (lines 19-20). Note *Search* was used to do initial processing of the query, but here it is used to do incremental processing. This time *Search* does not start from scratch; it does an incremental search based on current states of $count$, V, and U. *Search* either returns false if $count \geq k$ after visiting other cells intersecting with the circle; or returns true if all cells intersecting with the circle have been visited (U is empty) and $count < k$.

With the algorithm, the processing of a continuous Range-k query (therefore the continuous verification method based on it) is efficient because: (1) *Search* indexes the query in a minimum number of cells, so that the *Maintenance* will not be triggered unless necessary; (2) for most cases ($count > k$), the algorithm can return in constant time; (3) if *Search* is called because $count \leq k$, when queue V is already exhausted, *Search* will return in constant time.

4.3. Discussion

In this section, we briefly discuss 1) other possible applications of Range-k queries, 2) the relationship between Range-k query and kNN query and Range query, and 3) why CRange-k verification is more efficient than kNN query or Range query based verification in CRkNN processing.

Although we define Range-k query for the purpose of using it in CRkNN processing, this kind of query is generally meaningful. It can be used in certain applications to help making decisions. For example, in battlefield a soldier may want to know whether there are more than k accompanying soldiers within a specific range. If so she/he is safe, otherwise she/he may need support. As another example, a mobile object that provides service to other objects via Bluetooth may need to know whether there are more than k (e.g., its service capacity) objects within a distance (e.g., the Bluetooth transmission range). If so, more service-provider objects are needed, otherwise the object itself is able to service its nearby customer objects.

Range-k query is closely related to Range query and kNN query because it combines the *range* and *k* parameters from them. kNN query algorithm and Range query algorithm can even be used to process a Range-k query $< o, r, k >$. To use kNN query algorithm, we first find o's kNN, and then compare r with the distance between o the its k-th nearest neighbor. To use Range query algorithm, we first find all the objects in range, and then compare its size with k.

Because of the relationship shown above, any continuous kNN query algorithm and any continuous Range query algorithm can be used to do continuous verification in CRkNN. But both of them will be an overkill and incur

high maintenance cost. Continuously maintaining kNN is expensive, and if kNN based algorithm is used then candidates near the CRkNN query point will have an unnecessarily large monitoring region. If Range query is used, then the candidates that are farther from the CRkNN query point can have a very big monitoring region. In this way the query is unnecessarily triggered by too many location updates. What makes a continuous Range-k efficient to process is that it does not ask for precise information and both the range and the k value can be used to make its monitoring region as small as possible. For candidates near to the CRkNN query point, range limits the monitoring region for this candidate; for candidates far from the CRkNN query point, the k value limits the monitoring region for this candidate (notice that our Range-k algorithm terminates either $count \geq k$ or cells intersecting with the circle have all been visited). In Section 5 we experimentally study their performance when applied in CRkNN monitoring.

5. Experimental Study

We implemented one continuous filter module and two continuous refinement modules. The continuous filter module is based on the 60-degree-pruning (TPL-pruning is not used simply because how to use it in continuous filter when $k > 1$ is still open) and uses six continuous constrained kNN queries to maintain a CRkNN query's candidates. One of the two continuous refinement modules uses our CRange-k refinement method, and the other employs Continuous kNN (CkNN) query based refinement method as in existing CRkNN solutions. In the CkNN based refinement module, we implemented the state-of-the-art CkNN algorithm [7].

The one continuous filter and two continuous refinement modules form two CRkNN solutions (recall that a CRkNN solution consists of one filter method and one refinement method). For simplicity, in the experimental result analysis, we use RF to refer to the one containing our CRange-k refinement method, and use CK to refer to the one containing the CkNN based refinement method.

The metric used in this performance study is processing time. The processing is done in memory. Moving objects and CRkNN queries datasets are generated using Network-based Generator [3] with the Oldenburg map. We study the effect of the following parameters: grid index granularity, value of k, number of moving objects, number of CRkNN queries, moving objects' speed. These parameters are summarized in Table 1. All the experiments are done on a workstation with 2 Intel Xeon 3.0 CPUs and 2 GB memory, running Red Hat Linux Enterprise 3 with Java 5.

Effect of Grid Index Granularity. Grid index granularity has a big impact on the performance of CRkNN solutions. Its effect is shown in Figure 7. We can observe

Table 1. System Parameters

Parameter	Default	Range
Number of objects (*no*)	20K	10,20,50,70,100(K)
Number of queries (*nq*)	2K	1,2,5,7,10 (K)
Value of k (*k*)	8	1,2,4,6,8,12,16
Object/Query speed (*v*)	middle	slow, middle, fast
No. of grid cells (nc)	100^2	$32^2, 64^2, 100^2,$ $128^2, 150^2, 256^2$

that RK outperforms CK at any grid index granularity. This is because RK issues a continuous Range-k query for each candidate and CK issues a continuous kNN query for each candidate. With the same k value, a Range-k query is always cheaper than a kNN query. The figure also shows that both a too coarse and a too fine granularity will hurt the performance. This is a common phenomenon in grid index based systems.

Effect of *k*. A comparison of processing time with respect to parameter k is depicted in Figure 8. The processing time of both RK and CK increases with k, but the performance of RK is always better than CK, and their difference increases rapidly with k. RK's processing time is almost linear with k, while CK's is almost quadratic with k. This is because: a CRkNN query's number of candidates increases linearly with k; a continuous Range-k query's maintenance cost is almost constant; a continuous kNN query's maintenance cost also increases with k.

Figure 9 shows the breakdown of processing times with respect to the k parameter. The contrast is obvious: as the k becomes large, the refinement time of CK grows rapidly and thus becomes the bottleneck of the whole system. For example, when k=12, 87.5% of the processing time is spent on refinement. For RK, the refinement time increases significantly slower. Furthermore, the ratio of refinement time is almost a constant (around 50%) and it is no longer the dominant factor for the total processing time.

Effect of Object Number. When the population of objects gets bigger, both CR and CK's processing time become longer (Figure 10). This is within expectation as more update messages are processed when the number of objects increases. When the number of objects increases, object density is higher, then object movement is more likely to trigger a CRkNN query's filter and refinement. CR is more scalable than CK because it maintains a smaller monitoring region for candidates near the CRkNN query point.

Effect of Query Number. Figure 11 shows that the processing time of RK and CK grow almost linearly when increasing the number of CRkNN queries. This is natural since the total processing time is the sum of all queries' processing times.

Effect of Moving Speed. The effect of objects' moving speed on CRkNN query's processing time is shown in Fig-

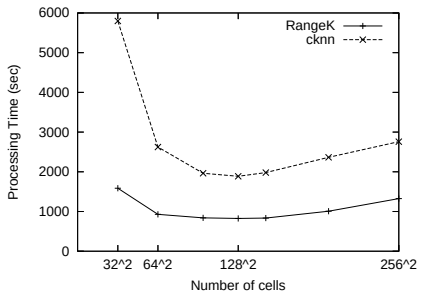


Figure 7. Effect of nc

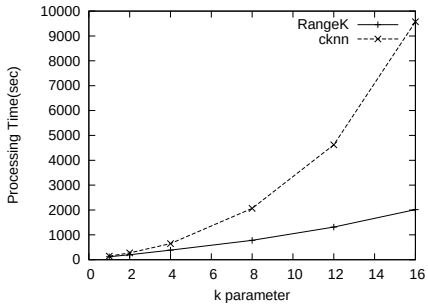


Figure 8. Effect of k

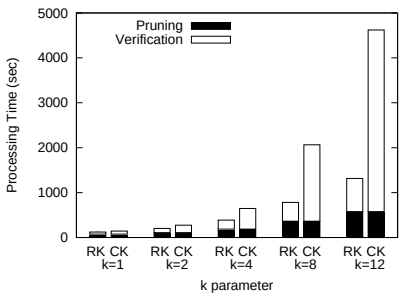


Figure 9. Breakdown

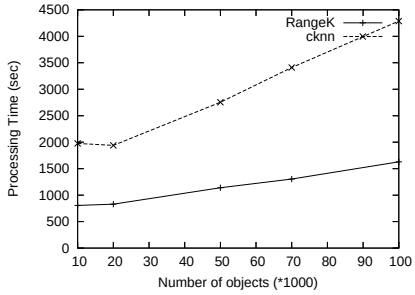


Figure 10. Effect of no

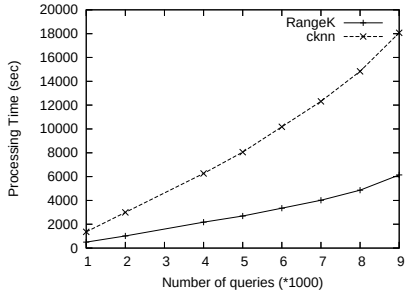


Figure 11. Effect of nq

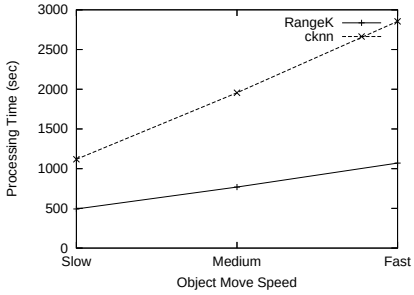


Figure 12. Effect of v

ure 12. RK outperforms CK in all speed settings. When the objects move faster, object movement is more likely to trigger continuous queries maintenance, thus the processing time of both RK and CK increases with objects' speed.

6. Conclusion

In this paper we investigated the problem of Continuous Reverse k-Nearest-Neighbor monitoring on moving objects. We show that a CRkNN query's processing can be clearly divided into continuous filter and continuous refinement, and continuous refinement will dominate the CRkNN processing time when $k > 1$. We propose an efficient continuous refinement method called CRange-k. In our method, the continuous refinement is formalized as continuous Range-k queries, which then are processed efficiently. Applying our *CRange-k* refinement method in CRkNN monitoring can effectively improve the efficiency of CRkNN monitoring.

References

[1] E. Achtert, C. Bohm, P. Kroger, P. Kunath, A. Pryakhin, and M. Renz. Efficient reverse k-nearest neighbor search in arbitrary metric spaces. In *SIGMOD*, 2006.

[2] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest and reverse nearest neighbor queries for moving objects. *The VLDB Journal*, 15(3):229–249, 2006.

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

[4] H. Ferhatosmanoglu, I. Stanoi, D. Agrawal, and A. E. Abbadi. Constrained nearest neighbor queries. In *SSTD*. 2001.

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

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

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

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

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

[10] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*. 2001.

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

[12] Y. Tao, M. L. Yiu, and N. Mamoulis. Reverse nearest neighbor search in metric spaces. *IEEE Transactions on Knowledge and Data Engineering*, 18(9):1239–1252, 2006.

[13] T. Xia and D. Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE*. 2006.

[14] M. L. Yiu, D. Papadias, N. Mamoulis, and Y. Tao. Reverse nearest neighbors in large graphs. *IEEE Transactions on Knowledge and Data Engineering*, 18(4):540–553, 2006.