

# Tracking Nearest Surrounders in Moving Object Environments

Ken C. K. Lee<sup>†</sup>   Josh Schiffman<sup>†</sup>   Baihua Zheng<sup>‡</sup>   Wang-Chien Lee<sup>†</sup>   Hong Va Leong<sup>§</sup>

<sup>†</sup>Department of Computer Science and Engineering, Pennsylvania State University,  
University Park, PA16802, USA. {cklee,jschiffm,wlee}@cse.psu.edu

<sup>‡</sup>School of Information Systems, Singapore Management University, Singapore. bhzheng@smu.edu.sg

<sup>§</sup>Department of Computing, The Hong Kong Polytechnic University,  
Hung Hom, Hong Kong. cshleong@comp.polyu.edu.hk

**Abstract**—This paper presents a system framework to support continuous *nearest surrounder* (NS) queries in moving object environments. NS query finds the nearest objects at individual distinct angles from a query point. This query distinguishes itself from other conventional spatial queries such as range queries and nearest neighbor queries by considering both distance and angular aspects of objects with respect to a query point. One of NS query applications is to monitor the nearest objects around an observation point. In our framework, a centralized server is dedicated to collect object location updates, to determine affected NS queries of each object location update, to compute the incremental result change of affected queries and to deliver result updates to corresponding interested users/applications that initiate the queries. In particular, we propose algorithms namely, *safe region formation* and *partial query evaluation*, that can significantly improve the system performance. Through simulations, we validate our proposed algorithms over a wide range of settings.

## I. INTRODUCTION

The convergence of wireless communication, positioning technology and portable computers, together with the advancement of data management technologies, has led to a wave of *location-related applications (LRAs)*. For many LRAs such as fleet management, cargo tracking, location-aware advertisement, and emergency services like E911 [6], tracking moving objects [13] is one of the essential functionalities. Considering a large population of moving objects and a high frequency of location update operations, much research effort has been put forth to optimize the cost of processing continuous queries, that involves evaluation of queries on moving objects over a period of time. Typically, evaluation of queries is initiated only when the location of a relevant object is changed [4], [16], [17], [20].

Many existing studies address continuous spatial queries based on the distance metric, either Euclidean distance or walking distance in road networks with respect to a given query point. However, in many practical situations, respective angles of objects to a query

point are also important. For instance, in robot football, robot  $A$ , a ball holder, attempts a ball pass to one of her teammates with minimum risk that the ball will be blocked or intercepted by any opponent. Obviously, conventional spatial queries like Nearest Neighbor (NN) or window query, which only considers the distance, cannot effectively provide  $A$  a good picture of all her surrounding opponents/teammates to determine whom her ball can be directly and safely passed to. Another example is soldiers fighting in battlefield. It is critical for a soldier to keep track of all his surrounding enemies in order to assure a clear firing path. Observing the importance of tracking nearest surrounding objects from these examples, we focus on the *nearest surrounder* (NS) query [9] for moving object environments in this paper.

In brief, the definition of NS query is described as follows. Given a set of objects  $O$  and a query point  $q$ , an NS query retrieves NNs at different angles from  $O$  with respect to  $q$ . The distance and angle used are referred to as Euclidean distance and polar angle, respectively. The result set, denoted by  $NS(q)$ , contains a set of tuples in form of  $\langle \text{object}:\text{angular range} \rangle$  that means the object (i.e. the object extent) is the nearest object to  $q$  within the associated angular range. For simplicity, we assume object extents are in *rectangular* shape. Let us consider a running example below.

*Example 1:* Given a query point  $q$  and a set of 10 objects,  $\{o_1, o_2, \dots, o_{10}\}$  as shown in Figure 1(a), the result set  $NS(q)$  is  $\{\langle o_1, [\alpha_g, \alpha_a] \rangle, \langle o_3, [\alpha_a, \alpha_b] \rangle, \langle o_6, [\alpha_b, \alpha_c] \rangle, \langle o_7, [\alpha_c, \alpha_d] \rangle, \langle o_8, [\alpha_d, \alpha_e] \rangle, \langle o_9, [\alpha_e, \alpha_f] \rangle, \langle \perp, [\alpha_f, \alpha_g] \rangle\}$  where  $\alpha_a$  through  $\alpha_g$  are distinct angles. The angular range  $[\alpha_f, \alpha_g]$  is associated with ‘ $\perp$ ’ that represents an empty sector within which no object is found. The angular ranges related to  $NS(q)$  are disjoint while their union equals  $[0, 2\pi)$ . In the example, all objects  $\in NS(q)$  are the nearest surrounders to  $q$  within corresponding angular ranges.  $\square$

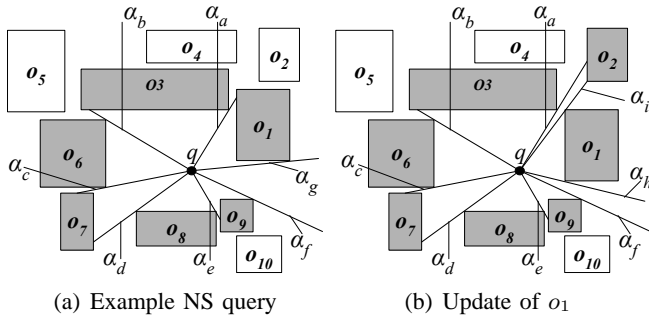


Fig. 1. Example NS query

In [9], we have proposed several search algorithms for NS query on static objects. When the locations of objects change, the answers to an NS query will become invalid. Let us look at another example in which the movement of an object invalidates an NS query result.

*Example 2:* This example continues Example 1. Here,  $o_1$  moves down while other objects remain stationary, as illustrated in Figure 1(b). Obviously,  $o_2$ , which was previously hidden by  $o_1$ , becomes a new member of  $NS(q)$  and the angular range  $[\alpha_f, \alpha_g]$ , which was previously an empty sector, is now partially occupied by  $o_1$ . The NS query result is thereafter changed to a new result,  $NS(q)'$ ,  $\{\langle o_1, [\alpha_h, \alpha_i] \rangle, \langle o_2, [\alpha_i, \alpha_a] \rangle, \langle o_3, [\alpha_a, \alpha_b] \rangle, \langle o_6, [\alpha_b, \alpha_c] \rangle, \langle o_7, [\alpha_c, \alpha_d] \rangle, \langle o_8, [\alpha_d, \alpha_e] \rangle, \langle o_9, [\alpha_e, \alpha_f] \rangle, \langle \perp, [\alpha_f, \alpha_h] \rangle\}$ .  $\square$

For each location update operation, how to efficiently locate all the invalidated NS queries and update corresponding results in a large scale system is crucial to the success of LRAs. In this paper, we address this update issue based on publish/subscribe model [3]. A straightforward solution is to scan all the NS queries to determine invalid ones (i.e., *query lookup*) and reevaluate those invalid queries (i.e., *query reevaluation*). In order to reduce the processing cost, we borrow the idea of *safe region* [14] to minimize the lookup overhead and *partial query reevaluation* to reevaluate only the invalid portions of query results. A safe region is a region bounding an NS query result. Given a safe region, an NS query  $Q$  needs to be re-evaluated only when 1)an object moves inside the safe region of  $Q$ ; 2)an object leaves the safe region of  $Q$ ; or 3)an object enters the safe region of  $Q$ . In other words, all the updates outside the safe region of  $Q$  can be safely ignored, since those objects are guaranteed to be excluded by the result set of  $Q$ . In this paper, we will explain how to form and represent safe regions for NS queries. On the other hand, partial query reevaluation determines only necessary result changes to existing query results rather than executing queries again from scratch. In brief, our contributions in this paper are four-fold:

- A system framework to manage object location updates, query reevaluation, and query subscription/withdrawal is presented to support continuous NS queries over moving objects.
- Formulation of safe regions for NS queries is devised to significantly improve the look up latency of invalidated NS queries.
- Partial query reevaluation and delivery of incremental result change are proposed to save both processing cost and network bandwidth.
- A comprehensive simulation is conducted to evaluate the proposal over a wide range of settings. The experimental results substantially prove the effectiveness of our proposal.

The rest of the paper is organized as follows. Section II reviews related work about NS query and continuous NN query. Section III details the architecture of the moving object monitoring system. Section IV and Section V describe the formulation of safe regions and the query reevaluation mechanism, respectively. Section VI discusses the simulation model and presents the experimental results. Section VII concludes this paper.

## II. RELATED WORK

NN search [15] has been well studied in both computational geometry and spatial database fields. However, as objects and queries are moving, extended NN queries have been defined in different semantics and various application situations. The work we present in this paper is to add a new dimension, i.e., the relative location of the object to the query point, to the NN search and focus on the update issue. In the following, we review the most relevant extended NN queries, Continuous NN (CNN) query and Nearest Surrounder (NS) queries. The former tackles the NN problem when the locations of a query point and objects keep changing, and the latter retrieves the nearest objects in a given angular space defined by the query.

### A. Nearest Surrounder Queries

The nearest surrounder search [9] evaluates an NS query based on both angle-based bounding properties and distance bounding properties of minimum bounding rectangle (MBR) in R-tree [1]. These properties dictate the traversal order of index nodes and provide pruning heuristics to adjust the searching space dynamically.

The search algorithms are based on the best-first approach that the index nodes and objects are queued in increasing order of object angles (from 0 to  $2\pi$ ) or object distance (from near to far) with respect to the query point. The query result is initialized with  $\langle \perp : [0, 2\pi) \rangle$  and it is refined when NS objects are found throughout the course of query execution. To save exhaustive result comparison and index navigation, once the MBR of an

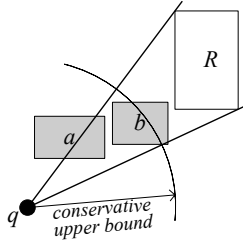


Fig. 2. NS query processing

index node or an object is examined not closer to a query point than existing NS objects within its angular range, the subtree referred by the index node or the object can be safely ignored from detailed inspection. This is illustrated in Figure 2. Suppose  $R$  (which is the MBR of an object or an index node extent) is being studied. Since existing NS objects,  $a$  and  $b$  cover the same angular range as  $R$  and provide a shorter distance than  $R$ , the corresponding index node or object is discarded. The maximum distance for  $a$  and  $b$  for the angular range is defined as “*conservative upper bound*”. The search algorithm terminates when the queue is empty.

### B. Continuous Nearest Neighbor Queries

The work on continuous NN queries can be roughly classified into two categories. The first category assumes the query and object trajectories/motions are known in advance, typically both query and objects are points and move in linear motion. With this assumption, elegant algorithms [2], [5], [10], [18] are devised. For static objects, a query trajectory, described as a line segment, can be divided into multiple shorter line segments [18] based on split points<sup>1</sup>. For linearly moving objects, time-varying distances between query and object trajectories are compared and the corresponding time intervals in which objects are nearer than others are determined [2], [10]. In addition, dealing with possible object motion changes, an even-queue based approach [5] schedules the NN result recomputation from time to time.

However, fixed object trajectory may not be realistic in many application scenarios; for instance, the object trajectories/motions are not always predictable as linear and constant [17]. Approaches in the second category make no assumption on object movement characteristics. Most of them [7], [12], [20] involve query reevaluation over updated object information. Similarly, without assuming any mobility pattern of objects, our work presented in the paper evaluates NS queries against updated object locations upon relevant change being detected.

## III. SYSTEM ARCHITECTURE

In this section, we start with the description of the moving object monitoring system, followed by the dis-

<sup>1</sup>A split point is equidistant to NN objects of two adjacent divided line segments.

ussion of the design of the database server, the core component of the system. The system performance is based on the *query result fresh latency*, which is the elapsed time from the moment that an object generates a location update to the moment that corresponding affected NS query results are updated. The proposed approaches to improve the system performance, namely *safe region formulation* and *partial query reevaluation*, will be discussed in Section IV and Section V respectively.

### A. System Environment

A typical moving object monitoring system consists of four parts: 1) a set of *target moving objects*; 2) multiple *base stations*; 3) a *database server*; and 4) *query clients* who issue NS queries. Figure 3 shows a high-level view of the system architecture. The geographical region covered by a base station is called *service area*. To simplify the discussion, the service area is assumed in *rectangular shape* and we only focus on the area covered by one base station. Every moving object equipped with a positioning device (e.g., GPS) keeps track of its current position. Periodically, it checks its location and initiates an location update report to a base station every time when the detected position is different from the previously reported one. The report includes object ID and its current location. The base station receives object location reports via the wireless channel and relays them to the database server via wired connection. The dissemination of location reports is in FCFS order.

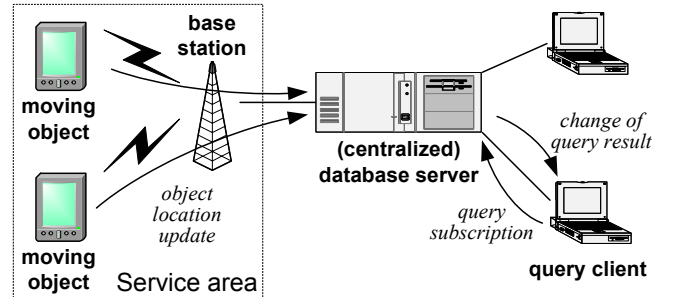


Fig. 3. System architecture

The central database server records the detailed locations of moving objects and is responsible for processing of NS queries. It also provides an interface through which remote query clients subscribe NS queries and/or cancel their query subscriptions. A detailed description of the database server is provided in the next subsection. We assume the clients maintain their NS query results and they are capable of incorporating the incremental result changes to original NS query results. To reduce processing cost and delivery cost, we adopt a partial query reevaluation approach and only deliver the difference between the current answer set and the previous

one. We also assume each issued NS query has a fixed query point and we fully focus this paper on the mechanism of efficiently updating query results according to the location changes of interested objects.

### B. Database Server

The structure of the database server, the main host to handle (object location) updates and query evaluation, is illustrated in Figure 4. It has five components, namely 1) *request queue*, 2) *object index*, 3) *query registry*, 4) *location updater* and 5) *query processor*. The database server manages one request at one time, either a location update report from an object via the base station or an NS query (issuing or canceling) from a remote client. Pending requests are placed into the request queue. When the server is idle, i.e., no other pending request is being processed by the server, the oldest pending request in the queue is dispatched and processed.

Assume that our system is stable and its throughput for processing requests is higher than the request incoming rate. Consequently, not too many requests would be queued for long and the request queue in a database server is resident in main memory. On the other hand, NS queries and location information of a large number of moving objects are stored on disk. Both indexing techniques and page caching are adopted to boost up the lookup performance of queries and objects and to reduce the I/O cost. R-tree, for its outstanding performance to support spatial queries and its MBRs' angle-based bounding properties in support of NS queries, is implemented to index the locations of all the moving objects. Besides, all objects are indexed by a hash table with their IDs as hash keys to support direct object lookup and to efficiently update the object index [11].

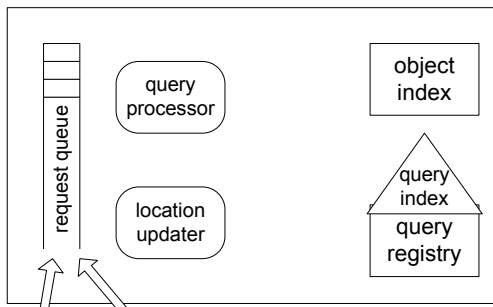


Fig. 4. The structure of the database server

The query registry maintains information of subscribed NS queries, namely, the query ID (assigned at the query subscription time), contact of the query client (e.g. IP address or name of a host issuing the query), the query point of the NS query, and the query result. As an object changes its location, results of some queries will become invalid. The mechanism *query invalidation* picks out all

the *invalidated queries*, i.e., those queries whose results become invalid. Recall an NS query result contains a set of  $\langle \text{object:angular range} \rangle$  tuples. Examining NS queries individually with exhaustive comparison between result tuples and the updated object location definitely incurs a high lookup overhead in the process of query invalidation. Alternatively, we adopt *safe regions*, approximated bounding regions outside which any object is guaranteed to be excluded by the answer sets of corresponding queries. These safe regions serve as filters to screen those invalidated NS queries and hence to leverage the high cost of exhaustive comparison. Further, these safe regions are maintained to construct a query index on the top of the query registry. In our work, the query index is implemented as an R-tree. Since both the shape and the size of safe regions affect the query lookup performance, we leave the discussion of safe region formulation in Section IV.

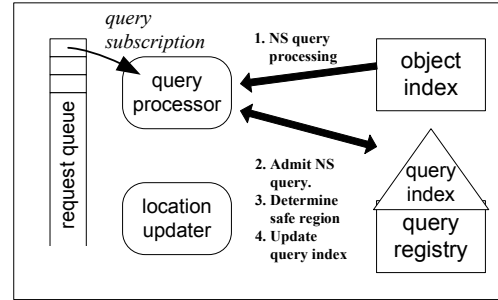


Fig. 5. The query processor's operation

The query processor is in charge of query subscription and query withdrawal. It is invoked whenever a new query is issued or an existing subscribed query is canceled. For a newly subscribed query  $Q$ , it assigns an ID, evaluates the query against the object index using one-time search algorithm [9], forwards the query information and query result to the query registry, and finally delivers the result to the query client as shown in Figure 5. The safe region corresponding to  $Q$  is then derived and indexed by the query index. On the other hand, handling query withdrawal is pretty straightforward. The query processor removes the corresponding safe regions and query information from the query index and the query registry, respectively.

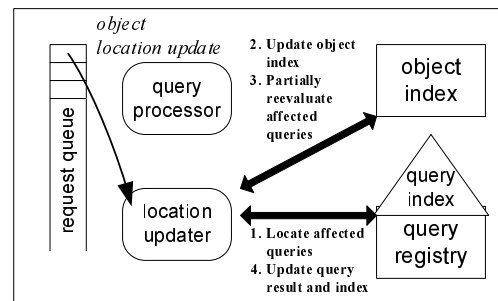


Fig. 6. The location updater's operation

The location updater is triggered by a location update

request launched from the request queue. It manipulates the object index and reevaluates NS queries invalidated by the update. The generalized mechanism is depicted in Figure 6. First of all, invalidated queries are identified by comparing the updated object against the safe regions of NS queries via the query index. Then, the object index is updated to reflect the new location of the object, and the invalidated queries are reevaluated. Finally, for each invalidated query  $Q'$  with updated query result, the location updater refreshes the query registry with the new query results and informs the query index of the new safe region of  $Q'$ . The detailed query reevaluation and determination of incremental result change will be detailed in Section V.

#### IV. SAFE REGION FORMULATION

In this section, we first introduce the detailed formulation of safe regions for *closed angular ranges* and *open angular ranges*, the only two types of angular ranges in the answer set, respectively. The former represents angular ranges in which the associated objects are identified, while the latter stands for those angular ranges without any identified objects. Next, we describe how to organize safe regions to speed up the query lookup process.

##### A. Safe Region for Closed Angular Range

For closed angular ranges of an NS query, we are concerned with whether the associated object (i.e., existing NS objects) is deleted or whether a new object is located closer to the query point than any existing NS objects. Given an NS query issued at a point  $q$ , a *hollow polygon*, which is a region formed by *facing edges* of all NS objects  $\in NS(q)$ , is compared with an updated object. Since the calculation of the overlap between an object extent, assumed in rectangular shape, and a polygon can be complicated, we use *Bounding Circle* as a safe region bounding the hollow polygon to save the checking cost. The formal formation of a bounding circle for the closed angular range is defined as follows.

**Bounding Circle.** The bounding circle of an NS query is centered at a query point  $q$  and has a radius equal to the conservative upper bound, that is the maximum of distance from  $q$  to all the NS objects, for all closed angular ranges.  $\square$

The bounding circle for the NS query discussed in Example 1 is shown in Figure 7(a). It centers at the query point and bounds the entire hollow polygon. Thereafter, the reevaluation of an NS query is initiated only when the location of deleted/inserted/updated object overlaps with the bounding circle, i.e., the minimal distance between the object extent and the query point does not exceed the

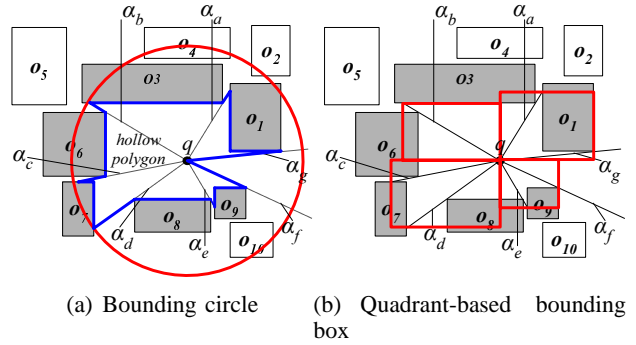


Fig. 7. Safe regions for closed angular ranges

radius of the corresponding bounding circle. To maintain bounding circles in R-tree, all circles are abstracted as MBR and then indexed.

Using a circle to approximate the hollow polygon improves the comparison cost, but it causes false hits because an object extent inside the circle is not necessarily located inside the hollow polygon. Therefore, we propose another type of safe regions, *quadrant-based bounding box*, as explained in the following.

**Quadrant-Based Bounding Box.** The space centered at a query point is split in four quadrants. For each quadrant, a bounding box is formulated to tightly cover the empty space for all closed angular ranges in the quadrant.  $\square$

Quadrant-based bounding box tries to minimize the difference between its covered area and that of the polygon. The four quadrant-based bounding boxes for the query discussed in Example 1 are depicted in Figure 7(b). It can be easily observed that quadrant-based bounding boxes bound the empty space more tightly than a bounding circle. The bounding boxes are directly indexed in the query index. The polygon of each NS query is represented by four bounding boxes, which occupy more memory space. However, it significantly reduces the number of unnecessary reevaluations due to the higher representation precision, which well justifies the approach.

##### B. Safe Region for Open Angular Range

For open angular ranges of an NS query, we are concerned with whether a new object enters the open angular range and becomes a new member to the NS query result. To speed up the query lookup performance, safe regions for open angular ranges are also defined. As shown in Figure 8(a), an open angular range covers an empty space. Based on the assumption that the service area is bounded, the empty space is a triangle (or multiple triangles if the open angular range touches more than one side of the service area) formed by two radial

lines of the open angular range and the boundary of the service area. Since no object falls inside the triangle, we name it *hollow triangle* in the following discussion. In this paper, we form the safe region as the bounding rectangle to approximate the hollow triangle, as depicted in Figure 8(a).

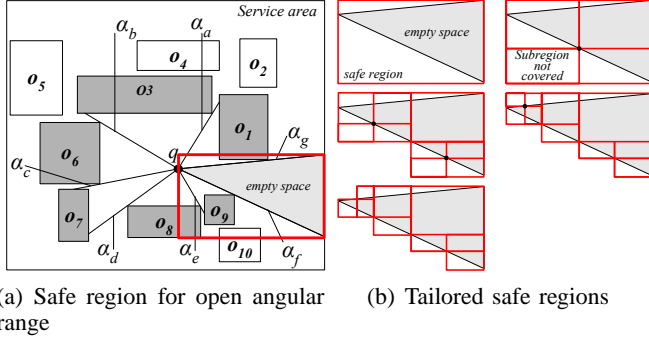


Fig. 8. Safe regions for open angular ranges

If we use a single rectangle to bound a triangle, the safe region at most doubles the space covered by the hollow triangle. As a result, objects whose extents overlap the safe region of a query may not fall inside the hollow triangle. In order to minimize the false hits, bounding box tailoring scheme, which breaks a large bounding box into smaller ones [8], is adopted. There are many different implementations of bounding box tailoring scheme. Because of the limited space, we present a simple approach here. The safe region is first partitioned into four smaller subregions. Thereafter, one of the subregion not covered by the triangle is removed. The remaining subregions are recursively tailored in the same fashion as illustrated in Figure 8(b).

Unconditional tailoring would run into a problem of generating an infinite number of fine safe regions. To control the quantity and quality (i.e. precision) of safe regions for open angular ranges, we define a metric called *coverage ratio* that is the percentage of the area covered by portions of hollow triangle to total area of the bounding regions. The partitioning terminates as the coverage ratio of a region is higher than a specified threshold (ranging from 0.0 to 1.0). When the threshold is set to 0.0, no tailoring is enabled. Meanwhile if it approaches to 1.0, infinite smaller bounding boxes will be generated.

### C. Organization of Safe Regions and Query Lookup

Since the natures of safe regions for open angular ranges and closed angular ranges are different, we build two query indexes, namely *Closed-QIndex* and *Open-QIndex*, in our implementation. All the safe regions for the closed angular ranges are stored in the *Closed-QIndex*, while the safe regions for the open angular ranges are stored in the *Open-QIndex*.

When the database server receives an update report from a moving object via the base station, it locates

all the invalidated queries with the query indexes. If the update is an insertion operation, a window search with object extent as window is conducted in both *Closed-QIndex* and *Open-QIndex*. Otherwise (when the update is a deletion operation), only *Closed-QIndex* is navigated. In both situations, the detected NS queries will be reevaluated, as to be discussed in the next section.

## V. NS QUERY REEVALUATION

When a moving object changes its location, invalidated NS queries have to be updated to reflect the new results. Assuming the capability of each query client to incorporate the incremental result change to the original answer set, we adopt the partial query reevaluation approach to evaluate only the invalidated portion of query results and deliver result changes for the invalidated part of the query result only. Consequently, the processing cost is reduced and bandwidth consumption of result delivery is improved, compared with answering each invalidated NS query from scratch. In the following, we present the detailed partial query reevaluation approach and explain result change delivery.

### A. Partial Query Reevaluation

In most cases if not all, update (either deletion of an object or insertion of a new object) invalidates only a portion of an NS query result. Therefore, reevaluation on the invalidated part of the result would be sufficient while the unaffected part can be retained.

Recall that an update happens when a new object enters the service area (insertion), or an existing object leaves the service area (deletion). Without loss of generality, we capture the movement of an object inside the service area as a deletion followed by a reinsertion of the object. Consequently, our following discussion considers only two basic operations: object insertion and object deletion.

When an object  $o$  is deleted, all NS queries that contain  $o$  in their answer sets are invalidated. Given an invalidated query  $Q$  issued at point  $q$  and assume  $\langle o : [\alpha, \beta] \rangle \in NS(q)$  (where  $[\alpha, \beta]$  is an angular range), substituting objects,  $o'$  in  $[\alpha, \beta]$ , need to be located by querying the object index using existing NS search algorithms. Thereafter, the old tuple  $\langle o : [\alpha, \beta] \rangle$  in the query result set will be replaced with the new tuples  $\langle o' : [\alpha, \beta] \rangle$ . In case that no object is found in the angular range  $[\alpha, \beta]$ , a tuple with  $\perp$  for the same angular range is created to substitute the old tuple in the result set. For a better illustration, we consider an example below.

*Example 3:* This example extends Example 1. Suppose  $o_1$  is deleted, and  $NS(q)$  is one of the invalidated answer sets as shown in Figure 9(a). Since  $o_1$  is the NS in  $[\alpha_g, \alpha_a]$  with respect to  $q$ , NS search algorithms are used to search other NS objects for the

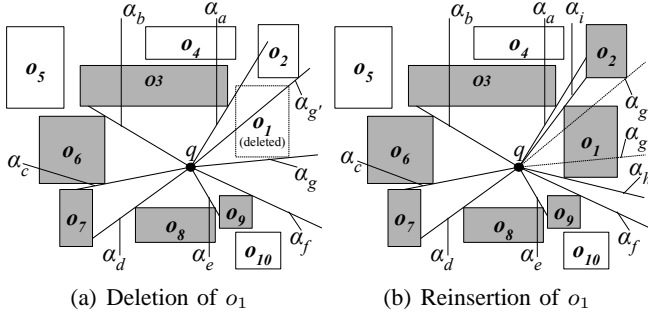


Fig. 9. Deletion and insertion

angular range  $[\alpha_g, \alpha_a]$ . Now,  $o_2$  is retrieved and it only covers a portion of the angular range, say  $[\alpha'_g, \alpha_a]$  as shown in Figure 9(b). The remaining part  $[\alpha_g, \alpha'_g]$  does not contain any object. After this partial query reevaluation, the NS query result maintained in the server becomes  $\langle \perp, [\alpha_g, \alpha'_g] \rangle$ ,  $\langle o_2, [\alpha'_g, \alpha_a] \rangle$ ,  $\langle o_3, [\alpha_a, \alpha_b] \rangle$ ,  $\langle o_6, [\alpha_b, \alpha_c] \rangle$ ,  $\langle o_7, [\alpha_c, \alpha_d] \rangle$ ,  $\langle o_8, [\alpha_d, \alpha_e] \rangle$ ,  $\langle o_9, [\alpha_e, \alpha_f] \rangle$ ,  $\langle \perp, [\alpha_f, \alpha_g] \rangle$ .<sup>2</sup> In other words, the result change set includes one tuple deleted (marked with superscript ‘-’) and two tuples inserted (with superscript ‘+’), i.e.,  $(\langle o_1 : [\alpha_g, \alpha_a] \rangle)^-$ ,  $(\langle \perp, [\alpha_g, \alpha'_g] \rangle)^+$ ,  $(\langle o_2, [\alpha'_g, \alpha_a] \rangle)^+$ .  $\square$

As shown in Example 3, it is important to notice that the union of angular ranges of all deleted tuples equals the angular ranges of all inserted tuples. When the client receives the result change, it removes tuples marked with ‘-’ from its result set and then reinserts the tuples marked with ‘+’.

Similar to object deletion, an insertion of a new object would invalidate portions of some NS query results as well. This time, it does not need to evaluate the query against the object index. The only necessary reevaluation is to check whether the inserted object is closer to the query point than any of the existing NS objects, as exemplified below.

*Example 4:* This example considers an insertion right after the deletion in Example 3. Now,  $o_1$  with a new location is reinserted and its new location around  $[\alpha_h, \alpha_i]$  from  $q$  invalidates  $NS(q)$  as shown in Figure 9(b). This time, the update is directly incorporated to the NS query result, i.e.,  $NS(q) = \{ \langle o_1, [\alpha_h, \alpha_i] \rangle, \langle o_2, [\alpha_i, \alpha_a] \rangle, \langle o_3, [\alpha_a, \alpha_b] \rangle, \langle o_6, [\alpha_b, \alpha_c] \rangle, \langle o_7, [\alpha_c, \alpha_d] \rangle, \langle o_8, [\alpha_d, \alpha_e] \rangle, \langle o_9, [\alpha_e, \alpha_f] \rangle, \langle \perp, [\alpha_f, \alpha_h] \rangle \}$ . The result change caused by this insertion involves three tuples deleted, plus three tuples added, that is,  $(\langle \perp : [\alpha_f, \alpha_g] \rangle)^-$ ,  $(\langle \perp, [\alpha_g, \alpha'_g] \rangle)^-$ ,  $(\langle o_2, [\alpha'_g, \alpha_a] \rangle)^-$ ,  $(\langle \perp : [\alpha_f, \alpha_h] \rangle)^+$ ,  $(\langle o_1 : [\alpha_h, \alpha_i] \rangle)^+$ ,  $(\langle o_2 : [\alpha_i, \alpha_a] \rangle)^+$ .  $\square$

<sup>2</sup>For illustration, we do not combine  $\langle \perp, [\alpha_g, \alpha'_g] \rangle$  and  $\langle \perp, [\alpha_h, \alpha_g] \rangle$ , both with  $\perp$  but our implementation does for space saving.

## B. Result Change Delivery

After result changes are determined by partial query reevaluation, the delivery of result changes back to query clients is straightforward. However, as an update is treated separately as a deletion-insertion pair, such simple result change passing would create a transient false image to query clients. For instance, if the query client incorporates the change produced in Example 3 before receiving the change in Example 4, the query client would see  $o_1$  disappeared; in fact it is moving to a new position.

To address this, result changes from composite deletion and insertion should be integrated. In place of individual result changes, accumulated result changes are used in our design. Due to the fact that any two tuples that have identical object and angular range but with different signs make no net effect to the final result, two effect-cancelling change tuples can be safely eliminated. Reconsider Example 3 and Example 4. It is not hard to observe that the result changes  $(\langle \perp, [\alpha_g, \alpha'_g] \rangle)^-$  and  $(\langle o_2, [\alpha'_g, \alpha_a] \rangle)^-$  obtained from Example 4 cancel the effect of  $(\langle \perp, [\alpha_g, \alpha'_g] \rangle)^+$  and  $(\langle o_2, [\alpha'_g, \alpha_a] \rangle)^+$  obtained from Example 3. As a consequence, the net result change caused by deletion and insertion of  $o_1$  is  $(\langle o_1 : [\alpha_g, \alpha_a] \rangle)^-$ ,  $(\langle \perp : [\alpha_f, \alpha_g] \rangle)^-$ ,  $(\langle \perp : [\alpha_f, \alpha_h] \rangle)^+$ ,  $(\langle o_1 : [\alpha_h, \alpha_i] \rangle)^+$ ,  $(\langle o_2 : [\alpha_i, \alpha_a] \rangle)^+$ . At last, a query client holding the old query result incorporates the change to its result by removing the deleted tuples marked ‘-’ and appending the added tuples marked ‘+’, accordingly. The updated result is already described in Example 2.

Furthermore, the accumulation of result change can be extended for the purpose of periodical result updates. If query clients prefer receiving the update at customized and regular time intervals, the result changes can be maintained, integrated and delivered at the specified time to the interested query clients.

## VI. PERFORMANCE EVALUATION

In this section, we run simulations to evaluate the proposed schemes, namely, *safe region formulation* and *partial query reevaluation*, against various environmental settings. In the following, we describe the simulation model and parameter settings followed by the detailed description of experiments and explanation of our findings. As to be presented later, the experimental results evident the efficiency (i.e., short response time and light IO overhead) of our proposed schemes in evaluating continuous NS queries.

### A. Simulation Model

The simulation is implemented with C++ and all experiments are conducted upon RedHat Linux Enterprise 3.0 on Intel® Xeon™ CPU 2.60GHz computers. A database server, with five components as described in

Section III-B, is implemented. In addition, we implement R\*-tree [1], a variant of R-tree, as underlying object index and query index structures with the node/page size fixed at 1KByte and the maximum node capacity is 50. The memory cache for each index is set to 25 pages for all experiments. As described in Section III, an efficient system should be able to provide a small query result refresh latency which is constituted by processing time and message transmission latency. Since the message transmission latency is not the focus of the evaluation, we measure the *processing time* for updates handled by the database server. The processing time is an elapsed time from the moment when a request starts to be processed at the server until the evaluation of all invalidated queries is completed. We also study *CPU time* and *IO cost* (e.g., the number of page accesses) as additional performance metrics. In the simulation, 1,000 object location updates are processed in each run and the reported performance metrics are averaged by 1,000 updates and presented in log scale.

The workload of the system is specified as follows. All datasets, including both synthetic and real datasets, are normalized in a fixed square service area of [1000, 1000]. Two real datasets from [19] representing geographical objects in Pennsylvania state and Rhode Island state (labelled as PA and RI respectively) of United States are evaluated. The number of objects in PA and RI (i.e., dataset size denoted by  $DS$ ) are 744k and 50k, respectively, and the average object sizes (denoted by  $OS$ ) in PA and RI are [0.33, 0.54] and [1.44, 1.06], respectively. Note that those object sizes are relative to the fixed service area<sup>3</sup>. In both datasets, landmark points from the same data source [19] serve as the NS query points. While object distribution in both PA and RI are uniform, synthetic datasets are generated as well to study the impact of  $DS$ ,  $OS$  and different object distribution. We vary the  $DS$  from 100k, to 500k and to 1000k and  $OS$  is set to [0.25, 0.25], or [0.5, 0.5], or [1.0, 1.0]. In addition to uniform object distribution, we have also investigated the skewed object distribution based on Gaussian distribution with mean set to [500, 500] and the variance set to [25, 25]. We further set the distribution of query points and object location updates to follow the object distribution. Table I summarizes the system parameters. The values in bold are default unless explicitly specified.

To justify our proposed approaches, i.e., **Safe Region** (SR) and **Partial Evaluation** (labeled as PE), we apply these techniques incrementally by adopting only SR, SR plus PE (labeled as SR+PE) in addition to a baseline approach, the **Brute Force** approach (labeled as BF) that

<sup>3</sup>The objects in RI are larger than that in PA on average because the  $DS$  of RI is smaller.

Parameters	Values
$DS$ (dataset size)	<b>100k</b> , 500k, 1000k (synthetic), 50k (RI) and 744k (PA)
$OS$ (object size)	[0.25, 0.25], [0.5, 0.5], [ <b>1</b> , <b>1</b> ] (synthetic), [1.44, 1.06](RI) and [0.33, 0.54](PA)
$NumQueries$	10, 100, <b>1000</b>

TABLE I  
EXPERIMENT PARAMETERS

does not have safe regions or query index, and evaluates all the invalidated queries from scratch. For SR and SR+PE, safe regions for closed angular ranges are based on quadrant-based bounding boxes and the coverage ratio for open angular ranges is 0, i.e., no tailoring is used. Because of space limitation, we report only the simulation results of four different sets of experiments. The first set of experiments evaluates the performance under real datasets; the second set studies the impact of dataset properties; the third set examines the impact of number of subscribed NS queries; the fourth set studies the impact of the forms of safe regions.

### B. Experiment #1. Real Dataset

Our first experiment examines the real practicality of the system by evaluating the system performance for the real datasets, i.e., PA and RI. The experiment results are shown in Figure 10. As observed, both SR and SR+PE perform pretty well as they both take less than 1 second to process an update, and significantly outperform BF for all measurements. Generally speaking, SR+PE consistently reduces at least two order of magnitudes the processing time (as shown in Figure 10(a)), CPU time (as bars in Figure 10(b)) and IO cost (as lines in Figure 10(b)) of BF for both datasets. This significant improvement shows the effectiveness of the proposed schemes.

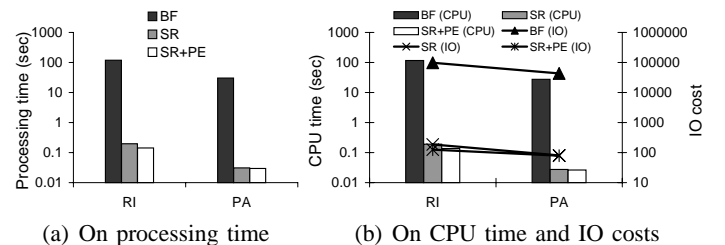


Fig. 10. Evaluation of real datasets

### C. Experiment #2. Properties of Dataset

The second experiment evaluates the impact of different factors, namely, dataset sizes ( $DS$ ), object sizes ( $OS$ ) and object distributions, on the processing time, CPU time and IO costs in order. The results are obtained based on synthetic datasets with various parameters. The results are shown in Figure 11 through Figure 13. Notice that both the dataset sizes and object sizes affect the object density. For higher object density, the result sets

of NS queries are expected to be smaller [9]. For all the results to be presented below, SR+PE is the best and SR consistently outperforms BF.

1) *Impact of Dataset Sizes*: Firstly, we fix the dataset distribution at uniform, fix  $OS$  at  $[1, 1]$ , and vary  $DS$  among 100k, 500k and 1000k to evaluate the impact of  $DS$ . As shown in Figure 11(a), the processing time of BF increases proportionally with respect to increased dataset sizes. This is caused by the heavy reevaluation overload, which is also reflected by IO costs in Figure 11(b). However, that of SR and SR+PE drops since a higher object density results in smaller NS result sets, hence fewer invalidated queries to be reevaluated. For the same reasons, we obtain similar observations in terms of CPU time and IO costs as shown in Figure 11(b).

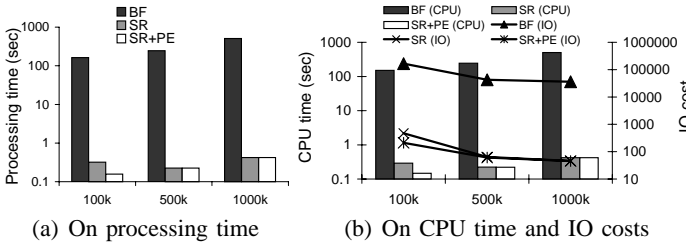


Fig. 11. The impact of dataset sizes ( $DS$ )

2) *Impact of Object Sizes*: Secondly, we study the impact of object sizes  $OS$  varied from  $[0.25, 0.25]$ , to  $[0.5, 0.5]$ , and to  $[1, 1]$  while  $DS$  and dataset distribution are fixed at 100k and uniform, respectively. The results are plotted in Figure 12. In general, the processing time of SR and SR+PE decreases as larger object sizes are experimented (see Figure 12(a)). This is because larger object sizes (i.e., higher object density) form smaller NS query result sets so that fewer queries become invalidated and reevaluated. On the other side, BF retains longer processing time for its high query reevaluation costs. In terms of CPU time and IO costs (shown in Figure 12(b)), similar performance results are obtained.

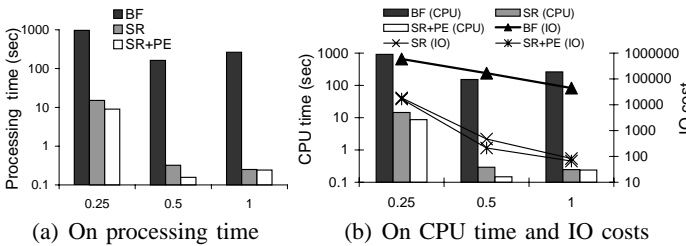


Fig. 12. The impact of object sizes ( $OS$ )

3) *Impact of Object Distribution*: Thirdly, we evaluate the performance under different object distributions, including both *uniform* and *skewed* distribution, and we fix  $OS$  at  $[1, 1]$  and  $DS$  at 100k, respectively. The results are plotted in Figure 13. Observed from the results, skewed object distribution slightly causes BF, SR and

SR+PE to take longer processing time. For all cases, SR+PE still outperforms SR and BF.

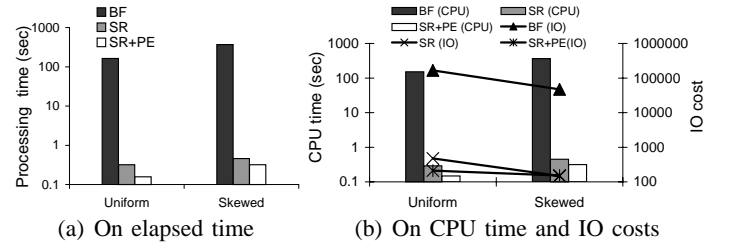


Fig. 13. The impact of object distribution

#### D. Experiment #3. Number of Subscribed Queries

This experiment evaluates the performance of the systems with different number of subscribed queries. Logically, the more the queries are subscribed, the larger the lookup overhead and the expected query evaluation cost. In this experiment, we change  $NumQueries$  from 10, to 100, to 1000 while other factors  $OS$ ,  $DS$ , and object distribution are fixed at  $[1, 1]$ , 100k, and uniform, respectively. The results are plotted in Figure 14. In general, the performance of all evaluated approaches increases linearly with the increased number of queries because the number of expected invalidated queries is increased proportionally, multiplying both the query lookup overhead and the reevaluation cost. Again, SR+PE performs the best.

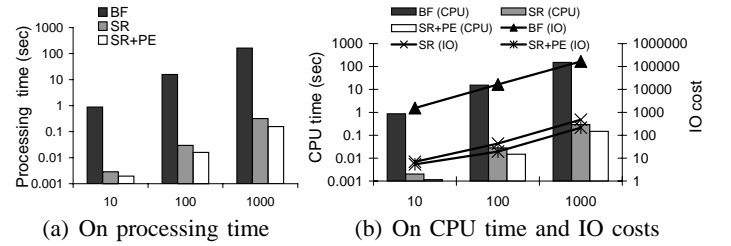


Fig. 14. The impact of the number of queries ( $NumQueries$ )

In addition, the average processing time to subscribe one NS query (that involves query result initialization and safe regions determination) is about  $0.1$  second.

#### E. Experiment #4. Forms of Safe Regions

The fourth experiment studies the impact of different forms of safe regions. Firstly, we vary the form of safe regions for closed angular ranges, namely bounding circle (labeled as BCircle) and quadrant-based bounding circle (labeled as QB BBox) described in Section IV-A, and we turn off the tailoring for safe regions for open angular ranges. We use both real and uniform synthetic datasets. The result is depicted in Figure 15. We can see QB BBox performs better than BCircle as it effectively reduces false hits.

Secondly, we evaluate the impact of threshold in tailoring safe regions for open angular range. In the

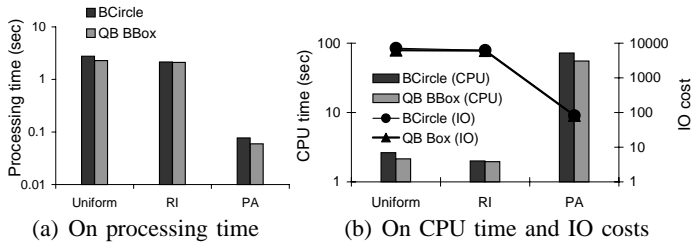


Fig. 15. The impact of forms of safe regions for closed angular ranges

experiment, the object density is high enough that only a few queries close to the service area boundary have open angular ranges and not too many updates invalidate those queries. As a result, the threshold setting has only minor impact to the overall performance. Due to limited space, the result is not presented.

## VII. CONCLUSION

This paper addresses the update issue of continuous NS queries in moving object environments. We discuss the system architecture and propose techniques such as safe region formulation and partial query reevaluation to support continuous NS queries. Through extensive performance evaluation on both real and synthetic datasets, the efficiency of our proposal is validated.

## ACKNOWLEDGEMENT

In this research, Wang-Chien Lee and Ken C. K. Lee were supported in part by US National Science Foundation grant IIS-0328881.

## REFERENCES

- [1] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R\*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data, Atlantic City, NJ, USA, May 23-25*, pages 322–331, 1990.
- [2] R. Benetis, C. S. Jensen, G. Karcauskas, and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *Proceedings of International Database Engineering and Applications Symposium, IDEAS'02, Edmonton, Canada, Jul 17-19*, pages 44–53, 2002.
- [3] F. Fabret, H.-A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha. Filtering Algorithms and Implementation for Very Fast Publish/Subscribe. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Santa Barbara, CA, USA, May 21-24*, 2001.
- [4] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *Proceedings of the 9th International Conference on Extending Database Technology, Heraklion, Crete, Greece, March 14-18*, pages 67–87, 2004.
- [5] G. S. Iwerks, H. Samet, and K. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany, Sep 9-12*, pages 512–523, 2003.
- [6] J. Reed, K. Krizman, B. Woerner, and T. Rappaport. An Overview of the Challenges and Progress in Meeting the E-911 Requirement for Location Service. *IEEE Personal Communications Magazine*, 3(5), Apr 1998.
- [7] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *Proceedings of the Thirtieth International Conference on Very Large Data Bases (VLDB), Toronto, Canada, Aug 31 - Sep 3*, pages 768–779, 2004.
- [8] G. Kollios, V. J. Tsotras, D. Gunopulos, A. Delis, and M. Hadjieleftheriou. Indexing Animated Objects Using Spatiotemporal Access Methods. *IEEE Transactions on Knowledge and Data Engineering*, 13(5):758–777, 2001.
- [9] K. C. K. Lee, W.-C. Lee, and H. V. Leong. Nearest Surrounding Queries. In *Proceedings of IEEE International Conference on Database Engineering, Atlanta, GA, USA, Apr 4-7*, page 85, 2006.
- [10] K. C. K. Lee, H. V. Leong, J. Zhou, and A. Si. An Efficient Algorithm for Predictive Continuous Nearest Neighbor Query Processing and Result Maintenance. In *Proceedings of the International Conference on Mobile Data Management (MDM), Ayia Napa, Cyprus, May 9-13*, pages 178–182, 2005.
- [11] 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 *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB), Berlin, Germany, Sep 9-12*, pages 608–619, 2003.
- [12] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, Jun 13-18*, pages 623–634, 2004.
- [13] E. Pitoura and G. Samaras. Locating Objects in Mobile Computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):571–592, 2001.
- [14] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.
- [15] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, CA, USA, May 22-25*, pages 71–79, 1995.
- [16] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, TX, USA, May 16-18*, pages 331–342, 2000.
- [17] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and Indexing of Moving Objects with Unknown Motion Patterns. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Paris, France, Jun 13-18*, pages 611–622, 2004.
- [18] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB), Hong Kong, China, Aug 20-23*, pages 287–298, 2002.
- [19] U.S. Census Bureau. 2002 TIGER/Line Files. [web] <http://www.census.gov/geo/www/tiger/tiger2002/tgr2002.html>.
- [20] X. Yu, K. Q. Pu, and N. Koudas. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *Proceedings of the 21st International Conference on Data Engineering (ICDE), Tokyo, Japan, Apr 5-8*, pages 631–642, 2005.