A Threshold-Based Algorithm for Continuous Monitoring of k Nearest Neighbors

Kyriakos Mouratidis, Dimitris Papadias, Spiridon Bakiras, *Member*, *IEEE*, and Yufei Tao

Abstract—Assume a set of moving objects and a central server that monitors their positions over time, while processing continuous nearest neighbor queries from geographically distributed clients. In order to always report up-to-date results, the server could constantly obtain the most recent position of all objects. However, this naïve solution requires the transmission of a large number of rapid data streams corresponding to location updates. Intuitively, current information is necessary only for objects that may influence some query result (i.e., they may be included in the nearest neighbor set of some client). Motivated by this observation, we present a threshold-based algorithm for the continuous monitoring of nearest neighbors that minimizes the communication overhead between the server and the data objects. The proposed method can be used with multiple, static, or moving queries, for any distance definition, and does not require additional knowledge (e.g., velocity vectors) besides object locations.

Index Terms—Spatial databases, location-dependent and sensitive, query processing.

1 Introduction

The problem of monitoring changes in the results of continuous queries has been extensively studied in the context of computer network [6] and data stream [13] applications. Babcock and Olston [4] deal with an interesting case, where a central server continuously reports the largest k values obtained from distributed data streams. The main idea behind their methodology is to maintain arithmetic constraints at the stream sources to ensure that the most recently reported answers remain valid. Up-to-date information is obtained only when some constraint is violated, thus reducing the communication overhead. In this paper, we study an alternative to this problem suitable for spatio-temporal and mobile-computing applications. Assume a set of clients (e.g., hotels, taxi piazzas), each requiring continuous monitoring of its k nearest neighbors (NNs) among a large set of moving objects (e.g., taxis). Clients and objects may appear or disappear (e.g., a new taxi enters service and reports its position, another one goes off-duty). The clients and the objects do not communicate directly, but through a central server that collects the queries and the object locations. The server can broadcast messages to all objects, which can respond individually (e.g., similar to a taxi call-service). The server must be able to report at any time the k closest objects to every client (the value of k may be different for each client). Fig. 1 illustrates the general architecture of the system.

Each object is aware of its position (through a GPS device) and has some limited memory and processing capabilities so that it can store the current queries (broadcast by the server) and compute its distance from every query. For simplicity, we use examples in the Euclidean space, but the proposed solutions are independent of the distance definition (i.e., the network distance can be applied if the object can compute it based on a shortest path algorithm and a stored map of the area [16]). Although movement is continuous, we assume that updates are discrete; for example, there is a minimum time between two subsequent message transmissions from the same object. Concerning the server, it can either broadcast messages to all objects, or send unicast messages to objects individually. As in most real-life applications, we consider that the cost of broadcasting is much lower than that of sending a large number of individual messages.

Our goal is to minimize the communication cost, i.e., the number of messages exchanged between the server and the moving objects. Intuitively, although in practical applications there exist numerous objects that move with arbitrary velocities toward arbitrary directions, we only care about the ones that may influence some query (i.e., they may be included in the nearest neighbor set of some client). For the rest of the objects, we do not need up-to-date information; in accordance with the methodology of [4], we can avoid the continual transmission of a large number of rapid data streams corresponding to location updates.

Starting with a single static query q in Fig. 2, assume that we want to continuously monitor the 3-NNs (the initial NNs are p_1, p_2, p_3). We call the objects that belong to the result *inner*, as opposed to *outer*, for objects (p_4, p_5, p_6) that do not qualify the query at the current time. In addition, Fig. 2 contains three thresholds t_1, t_2, t_3 which define a range for each object, such that if its distance from q lies within the range, the result of the query is guaranteed to remain unchanged. Each threshold is set in the middle of the distances of two consecutive points from the query (the computation of the thresholds will be discussed in Section 3). The distance range for p_1 is $[0, t_1)$, for p_2 is $[t_1, t_2)$, for p_3 is $[t_2, t_3)$, and for p_4, p_5, p_6 is $[t_3, \infty)$. Every object is aware of its distance range, and when there is a boundary violation, it informs the server about the event.

- *K. Mouratidis, D. Papadias, and S. Bakiras are with the Department of Computer Science, Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.*
  E-mail: {kyriakos, dimitris, sbakiras}@cs.ust.hk.
- *Y. Tao is with the Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong. E-mail: taoyf@cs.cityu.edu.hk.*

Manuscript received 16 Sept. 2004; revised 1 Apr. 2005; accepted 28 Apr. 2005; published online 19 Sept. 2005.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-0326-0904.

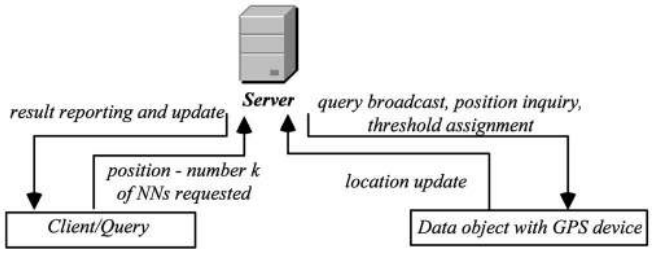

Fig. 1. Architecture of the system.

For instance, if the first threshold violation occurs when the distance $dist(p_1, q)$ becomes larger than t_1 (p_1 moves to a new position p'_1), then the order between the first two NNs may change (i.e., p_2 may become the first NN). In order to verify this, the server must acquire the current location of p_2. If p_2 has moved toward the query, it may become the first NN; otherwise (e.g., p_2 is static or has moved away from the query), the result remains the same. In any case, the value of t_1 is updated to reflect the new locations of p_1 and p_2. The positions of the other objects are not relevant, as they cannot influence the result before they trigger a threshold violation. Continuing the example, consider that the second event is triggered when p_3 moves further away from q than t_3, in which case any of the outer objects may become the third NN (e.g., since the last update, object p_6 has moved closer to the query). In order to restrict the set of potential candidates (that must send their new locations), the server broadcasts a request asking for the positions of all outer objects within distance $dist(p'_3, q)$ from the query point. A new value of t_3 is calculated based on the position of the new third (p_6) and fourth (p_3) NN and broadcast to all objects.

We propose a threshold-based algorithm for minimizing the network overhead, which can be employed with any distance definition, multiple (possibly moving) queries, and takes into account several real-life constraints (inaccuracy due to discrete updates, concurrent violations, appearance/disappearance of objects and queries). The rest of the paper is organized as follows: Section 2 surveys related work on snapshot and continuous NN queries. Section 3 presents our solutions for the single, static query case, while Section 4 extends our methodology to multiple moving queries. Section 5 experimentally evaluates the proposed techniques in terms of the communication cost and Section 6 concludes with directions for future work.

2 Related Work

Section 2.1 gives an overview of snapshot NN queries in client-server architectures and Section 2.2 presents methods for continuous monitoring of nearest neighbors.

2.1 Snapshot NN Queries

The first query processing techniques (e.g., [17], [8]) for NN retrieval considered static queries and data. Later, the focus shifted toward moving NN queries and/or objects in client-server architectures, where the goal is to provide, in addition to the current results, information about their validity. Zheng and Lee [22] propose a technique that

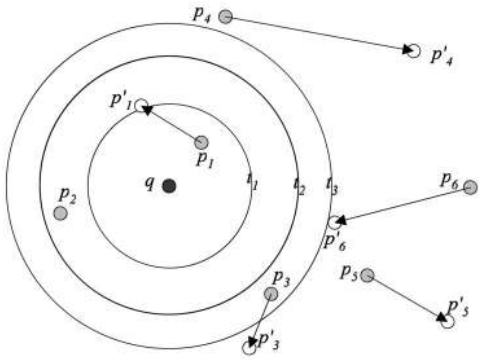

Fig. 2. Example of 3-NN monitoring.

reduces the network overhead for moving queries on static objects. When the server receives a query, it returns to the client the NN and a *validity period* T such that the NN is guaranteed to remain the same (even if the client moves before T expires, it does not need to issue another query in order to obtain up-to-date results). Zhang et al. [24] propose the concept of *location-based* queries that return the *validity region* around the query point where the result remains the same. The validity region is the Voronoi cell[1] of the NN, which is computed on-the-fly using an R-tree on the data points. As long as the client remains in the Voronoi cell, it can reuse the result without having to issue another query. The technique can also be applied for k NNs by computing order-k Voronoi cells.

For the same settings (moving query—static data objects), Song and Roussopoulos [18] attempt to reduce the number of future queries by introducing some redundancy in the results of current ones. In particular, when a k-NN query arrives, the server computes and returns to the client a number $m > k$ of neighbors. Let $dist(k)$ and $dist(m)$ be the distances of the kth and mth nearest neighbor from the query point q. If the client reissues the query at a new location q', it can be easily proven that the new k nearest neighbors will be among the m objects of the first query, provided that $2 \cdot dist(q', q) \leq dist(m) - dist(k)$. Tao and Papadias [19] study *time-parameterized* (TP) queries, assuming that the clients and the data objects move with linear and known velocities. In addition to the current result R, the output of a TP query contains its *validity period* T and the *next change* C of the result (that will occur at the end of the validity period). Given the additional information (C and T), the client only needs to issue another TP query after the expiry of the current result. Assuming a linearly moving client, a *linear* query [3], [19] returns all query results up to a future timestamp. In particular, the output is a set of tuples $< R_i, T_i >$, where R_i is the k NN set during interval T_i.

All the above techniques take as input a single query, and report its nearest neighbor set at the current time, possibly with some validity information (e.g., expiration time, Voronoi cell), or generate future results based on predictive features (e.g., velocity vectors of queries or data objects). On the other hand, continuous monitoring:

1. A data point is the NN for every (query) point that falls in its Voronoi cell.

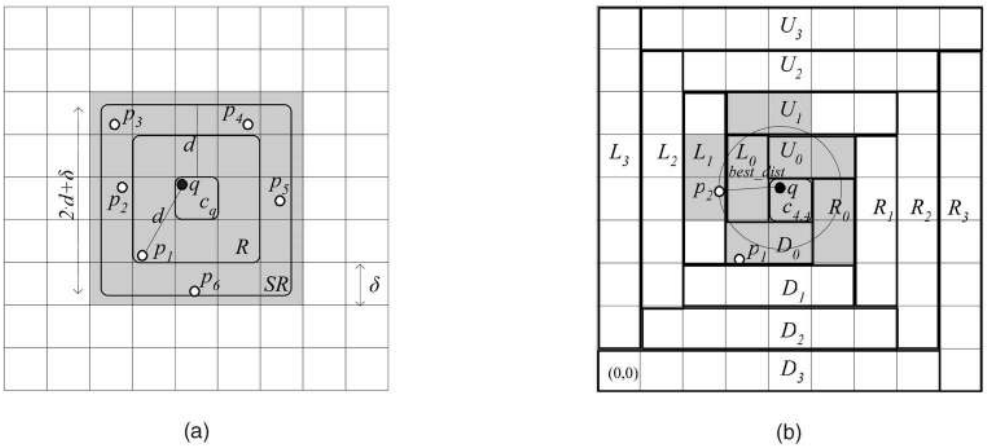

Fig. 3. YPK-CNN and CPM example. (a) NN search in YPK-CNN and (b) NN search in CPM.

1. involves multiple long-running queries (from geographically distributed clients),
2. is concerned with both computing *and* keeping the results up to date,
3. usually assumes main-memory processing to provide fast answers in an online fashion, and
4. attempts to minimize factors such as the CPU or communication cost (as opposed to I/O overhead).

2.2 Continuous NN Queries

Continuous monitoring of spatial queries is becoming increasingly important due to the wide availability of inexpensive and compact positioning devices, the evolution of mobile communications, and the need for improved location-based services. Consequently, several techniques have been developed during the last few years. Assuming static range queries, *Q-index* [15] indexes them using an R-tree. The moving objects probe the index to find the queries that they influence. *Q-index* utilizes the concept of *safe regions* to reduce the number of updates. In particular, each object p is assigned a circular or rectangular region, such that p needs to issue an update only if it exits this area. Kalashnikov et al. [11] show that a grid implementation of *Q-index* is more efficient (than R-trees) for main memory evaluation. MQM [5] and *Mobieyes* [7] exploit the object computational capabilities in order to reduce the processing load of the central server. SINA [14] performs a spatial join in three steps between moving objects and queries to continuously update the current results.

The aforementioned methods focus on range query monitoring, and their extension to NN queries is either impossible or nontrivial. Koudas et al. [10] describe a technique for approximate *k*-NN retrieval over streams of multidimensional points. Yu et al. [21] propose a method, hereafter referred to as YPK-CNN, for the continuous monitoring of exact *k*-NN queries. Objects are assumed to fit in main memory and are indexed with a regular grid of cells with size $\delta \times \delta$. YPK-CNN does not process updates as they arrive, but directly applies the changes to the grid. Each NN query installed in the system is re-evaluated every T time units. When a query q is evaluated for the first time, a two-step NN search technique retrieves its result. The initial step visits the cells inside an iteratively enlarged square R around the cell c_q covering q until k objects are found. Fig. 3a shows an example of a single NN query where the first candidate NN is p_1 with distance d from q; p_1 is not necessarily the actual NN since there may be objects (e.g., p_2) in cells outside R with distance smaller than d. To retrieve such objects, the second step searches in the cells intersecting the square SR centered at c_q with side length $2 \cdot d + \delta$, and determines the actual k NN set of q therein. In Fig. 3a, YPK-CNN processes p_1 up to p_6 and returns p_2 as the actual NN. The accessed cells appear shaded.

SEA-CNN [20] focuses exclusively on monitoring the NN changes, without including a module for the first-time evaluation of an arriving query q (i.e., it assumes that the initial result is available). Objects are stored in secondary memory, indexed with a regular grid. The *answer region* of a query q is defined as the circle with center q and radius $best\_dist$, where $best\_dist$ is the distance of the current kth NN. Book-keeping information is stored in the cells that intersect the answer region of q to indicate this fact. When updates arrive at the system, depending on which cells they affect and whether these cells intersect the answer region of the query, SEA-CNN determines a circular search region SR around q, and computes the new k NN set of q therein. To determine the radius r of SR, the algorithm distinguishes the following cases: 1) If some of the current NNs move within the answer region or some outer objects enter the answer region, SEA-CNN sets $r = best\_dist$ and processes all objects falling in the answer region in order to retrieve the new NN set. 2) If any of the current NNs moves out of the answer region, $r = d_{max}$ (where d_{max} is the distance of the previous NN that moved furthest from q), and the NN set is computed among the objects lying in SR.

CPM [12] assumes the same system architecture and indexing structures as YPK-CNN and SEA-CNN. When a query q arrives at the system, CPM computes its initial result by organizing the cells into conceptual (hyper) rectangles based on their proximity to q. Each rectangle *rect* is defined by a *direction* and a *level number*. The direction is U, D, L, or R (for up, down, left, and right), and the level number indicates how many rectangles are between *rect* and q. Fig. 3b illustrates the conceptual partitioning of the

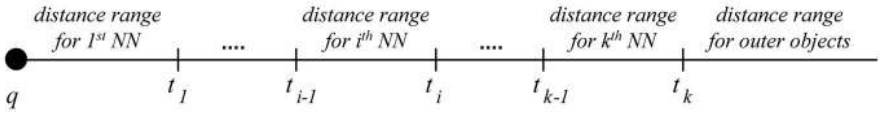

Fig. 4. Thresholds for inner and outer objects.

space around the cell $c_{4,4}$ of q in our running example. If $mindist(c,q)$ is the minimum possible distance between any point in cell c and q, the NN search considers the cells in ascending $mindist(c,q)$ order. In particular, CPM initializes an empty heap H and inserts 1) the cell of q with key equal to 0, and 2) the level zero rectangles for each direction DIR, with key $mindist(DIR_0,q)$. Then, it starts deheaping entries iteratively. If the deheaped entry is a cell, it examines the objects inside and updates accordingly the list $best\_NN$ of the closest NNs found so far. If the deheaped entry is a rectangle DIR_{lvl}, it inserts into H 1) each cell $c \in DIR_{lvl}$ with key $mindist(c,q)$ and 2) the next level rectangle DIR_{lvl+1} with key $mindist(DIR_{lvl+1},q)$. The algorithm terminates when the next entry in H (corresponding either to a cell or a rectangle) has key greater than the distance $best\_dist$ of the kth NN found. It can be easily seen that the NN search processes only the cells that intersect the circle with center at q and radius equal to $best\_dist$. This is the minimal set of cells to visit in order to guarantee correctness. In Fig. 3b, the search processes the shaded cells, and returns p_2 as the result.

Similar to YPK-CNN, SEA-CNN, and CPM, we target exact processing of continuous NN queries. However, the above three methods 1) assume that each object issues an update whenever it moves (independently of whether it influences a query or not), and 2) focus exclusively on minimizing the CPU cost at the server side. On the other hand, we 1) consider that the objects have some computational capabilities so that they can determine if they need to issue an update, and 2) aim at minimizing the network cost. In this sense, the proposed methodology is more related to previous techniques for the monitoring of range queries (e.g., [15], [5], [7]) that utilize the concept of safe regions to reduce the number of messages. However, NN queries are inherently more complex than ranges; in fact, as discussed in Section 3.3, the proposed methodology can capture range (and other) queries with straightforward adaptations. Furthermore, the threshold-based algorithm could be integrated with YPK-CNN, SEA-CNN, and CPM in one system that minimizes both the network overhead and the processing cost at the server.

3 Single Static Query

We assume a time-slotted system, where each object notifies (if necessary) the server about its new location at discrete timestamps, i.e., there is a minimum interval dt between two consecutive updates of the same object such that the round-trip time of a message between the server and any object is negligible compared to dt. Discrete updates are necessary for any realistic system as moving objects cannot report positions continuously. For simplicity, we consider that the updates are synchronous, although this is not a prerequisite for the proposed methods (the value of dt may be different for each object), which can also capture the fact that messages are sometimes lost due to the error-prone nature of the wireless medium. The discrete updates and potential message losses introduce inaccuracy because the reported result may become invalid at the interval between two subsequent updates. Here, we define correctness based on the most recent information received at the server side. For ease of presentation, we first describe our methodology for the continuous monitoring of a single static query before covering multiple moving queries. Section 3.1 describes the threshold-based algorithm for maintaining the k nearest neighbors, Section 3.2 presents cost models for measuring the communication overhead, and Section 3.3 discusses applications to other query types.

3.1 Threshold-Based Algorithm

At any time, the server maintains the ids of the k last reported NNs and the k thresholds that define the distance ranges where objects can move without causing a result change. Let p_i $(i = 1,\ldots,k)$ be the ith NN and d_i be its distance from the query point q. The permissible range $[t_{i-1}, t_i)$ of p_i is defined by thresholds t_{i-1} and t_i (see Fig. 4). Ideally, the threshold t_i between p_i and p_{i+1} should be set in a way that defers as much as possible a potential violation, e.g., if we expect both objects to move away from the query, a good value for t_i would be close to d_{i+1}. Since we do not assume any knowledge regarding the motion patterns, we initially set $t_i = (d_i + d_{i+1})/2$, i.e., in the middle of the distances of the two relevant NNs (for the first NN $t_0 = 0$). Each object locally keeps the position of the query and the relevant thresholds: Outer objects store t_k, while the ith NN stores t_{i-1} and t_i. When an object violates its threshold(s), it sends an uplink message to the server with its new location.

Once a query q arrives, the server first needs to retrieve the initial set of k-NNs. In order to achieve this, it broadcasts a message asking for the current locations of all objects within a distance r_1 from q. The value of r_1 can be tuned based on the trade-off between the cost of broadcasting and uplink messages. For instance, a large range is likely to contain more than necessary objects, but it will cause numerous uplink messages, several of which are redundant. On the other hand, a small value may return an insufficient number of objects so that the process has to be repeated incrementally around the query point. Assuming that the server only knows the cardinality $|N|$ of the object data set, but not its distribution (which, anyway, changes with time), we set:

$$r_1 = \sqrt{k/\pi \cdot |N|}.$$

The rationale is that for uniform data distribution in unit workspace, the circle (q, r_1) centered at q with radius r_1 is expected to contain k objects [1]. If the number k_1 of objects that respond is equal to or larger than k, the

server has enough information to compute the k-NNs and the thresholds.[2] Otherwise, it must broadcast a second query asking for the locations of objects in the circle (q, r_2), where r_2 is such that, based on the density information of the first query, the number of objects within distance r_2 from q is expected to be k. The set of responses to the server includes 1) objects with distance in the range $(r_1, r_2]$ that do not qualify the first query and 2) objects whose location has changed since their last transmission. The process is repeated until the server collects responses of at least k objects.[3] The radius r_i of the ith range query is computed using the value of r_{i-1} and the number k_{i-1} of objects that have responded up to the $i-1$th query:

$$r_i = r_{i-1} \cdot \sqrt{k/k_{i-1}}.$$

After the initial computation, the server has to continuously monitor the result and report changes. Let I and O be the sets of *incoming* (e.g., outer objects that come within distance t_k from q) and *outgoing* objects (e.g., inner objects that move out of t_k), respectively. Depending on the cardinalities of I and O at a particular timestamp, we distinguish two cases. The first one refers to the situation where $|I| \geq |O|$, i.e., the new value t'_k of the outer threshold is equal to or smaller than the previous one t_k (since more objects approach the query point). In this case, the new thresholds are only transmitted to the affected objects with unicast messages and broadcasting is avoided. On the other hand, if $|I| < |O|$ (the second case), then $t'_k > t_k$, implying that 1) some outer objects must send their updated positions since they may now belong to the query result and 2) the new value of t'_k must be broadcast to all outer objects.

Before illustrating these two cases in detail, we present in Fig. 5 the basic functionality of the system. Let T be the set of all objects that incur violations at some timestamp, i.e., $T = I \cup O \cup$ {inner objects that violate their assigned thresholds but not t_k}. If $T \neq \emptyset$, the interval $[t_{i-1}, t_i)$ of each inner object p_i $(\in T - I)$ that violates its threshold (i.e., moved to another interval or out of t_k) is marked as *orphan*. In the first step, the algorithm generates a new threshold for each nonorphan interval, assigning the initial values $t'_{i'}$ (i' is an index over the new threshold values). Then, the objects of $T - O$ (the violators whose current location is within t_k) are sorted according to their distance from q in list L_T and processed one-by-one (while-loop in Fig. 5). If an object falls in an orphan interval $[t_{i-1}, t_i)$, it produces a new threshold $t'_{i'} = t_i$. Otherwise, the updated location of the "owner" object is obtained, and $t'_{i'}$ is set in the middle of the two object distances. The process stops when the new result contains k objects or L_T becomes empty. If $|I| \geq |O|$, computing the new NN set and thresholds is completed at this point.

As an example of this case, consider the 6-NN query of Fig. 6a where p_2, p_4, and p_8 issue violations at the same timestamp, i.e., $T = \{p_2, p_4, p_8\}$, $I = \{p_8\}$, and $O = \emptyset$. Some objects (e.g., p_8) may violate multiple thresholds, if their ranges are small and they move fast (with respect to dt), or

2. If $k_i > k$, the outer threshold t_k is defined by the distances of the kth and $k+1$th NN. Otherwise, (i.e., if $k_i = k$), t_k is set to r_i.

3. Assuming that the message round-trip time is negligible, only objects with a distance in the range $(r_{i-1}, r_i]$ respond to the ith query, i.e., the positions of all objects in $(0, r_{i-1}]$, obtained by previous, are still valid.

```
Monitoring Algorithm
Initialization: Compute the initial k NNs and assign thresholds
// T: Set of all objects that incur violations
// p'i', for i' = 0, 1,..., k, are the new NNs
For each timestamp such that T≠∅ //there is at least one violation
 For each inner object pi ∈ T-I, mark interval [ti-1 , ti) as orphan
 i' = 1; t'0 = 0; // i' is an index over the new thresholds t'i'
 For i = 1 to k // i is an index over the previous thresholds ti
  If interval [ti-1, ti) is non orphan //1st step
    t'i' = ti; p'i' = pi;  i' = i' + 1; // assignment of initial NNs and t'i'
 Sort objects in T-O in ascending distance from q in sorted list LT
 While set of current NNs contains ≤ k objects and LT not empty
   Remove the next object p from LT
   Find interval [t'i'-1, t'i') that contains dp
   Find interval [ti-1, ti) that contains dp
   If [ti-1, ti) is orphan
     t'i' = ti; Mark [ti-1, ti) as non-orphan
   Else // not orphan - resolve with p'i' in interval [t'i'-1, t'i')
     If the current distance d'i' of p'i' is not known
       Obtain location of p'i'
       Insert p'i' into U // set of inner objects involved in conflicts
     t'i' = (d'i' + dp)/2;
   Insert p in the result
 End while
 If set of current NNs contains k objects
   Send unicast messages with new thresholds to objects in T∪U
 Else // Case 2, set of current NNs contains k' < k objects (|O| > |I|)
   Repeat
     Determine radius r around q that is expected to contain k objects
     Broadcast query asking for set UI of outer objects in circle (q,r)
     Insert to UI outgoing objects in [tk,r]
   Until |UI|+ k' ≥k
   Sort UI on distance
   Insert the first k-k' objects of UI in the result and in set U2
   Set the last k-k' thresholds accordingly
   Send unicast messages with thresholds to objects in (T-O)∪U∪U2
   Broadcast a message with threshold t'k
 If there are changes in the k-NN set, inform the client
```

Fig. 5. Threshold-based algorithm for single query.

previous update messages were lost. The algorithm first scans the original intervals (i.e., before the violations) and for each nonorphan one 1) it assigns a new threshold and 2) inserts the corresponding object in the current NN list. In Fig. 6b, there are only four NNs (p_1, p_3, p_5, p_6) and new thresholds $t'_1, \ldots, t'_4$ because $[t_1, t_2)$ and $[t_3, t_4)$ are orphan (they belong to objects p_2 and p_4 that issue violations). The first object (in $T - O$) processed is p_2 since it is the closest to q; p_2 falls in the nonorphan interval $[t_2, t_3)$ (according to the old thresholds), and in $[t'_1, t'_2)$ (according to the new ones). The current position of the object (p_3) assigned to $[t'_1, t'_2)$ is not known. Thus, the server has to obtain it in order to determine the order of p_2 and p_3 in the NN result. Then, p_2 is inserted in the appropriate order and a new threshold t'_2 is set between p_2 and p_3 (Fig. 6c). The processing of p_4 (falls in $[t'_2, t'_3)$) is similar (Fig. 6d), the only difference being that the current location of p_3 does not have to be obtained again. Object p_8 falls in orphan interval $[t_3, t_4)$, generating a new threshold $t'_5 = t_4$ (Fig. 6e) and the set of NNs is completed. The new outer threshold t'_6 equals t_5.

If the set of NNs (or their order) changes, the new result is transmitted to the client. In addition, the new thresholds are sent (by individual unicast messages) to 1) objects that triggered a violation ($T = \{p_2, p_4, p_8\}$) and 2) objects that updated their location (although they did not incur violations) because they were involved in NN conflicts ($U = \{p_3\}$). On the other hand, p_1 and p_5 maintain their old intervals ($[t'_0, t'_1) = [t_0, t_1)$ and $[t'_5, t'_6) = [t_4, t_5)$, respectively) since they have not issued a violation and are not involved in conflicts. The same is true for p_6 (previous sixth NN),

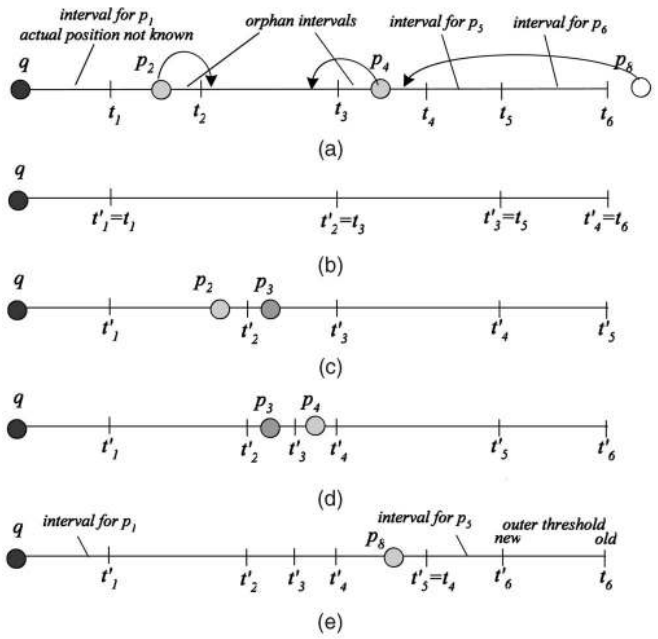

Fig. 6. Incoming objects are more than outgoing. (a) Before processing violations: $NN < p_1, p_2, p_3, p_4, p_5, p_6 >$. (b) After the first step: $NN < p_1, p_3, p_5, p_6 >$. (c) After inserting p_2: $NN < p_1, p_2, p_3, p_5, p_6 >$. (d) After inserting p_4: $NN < p_1, p_2, p_3, p_4, p_5, p_6 >$. (e) After inserting p_8: $NN < p_1, p_2, p_3, p_4, p_8, p_5 >$.

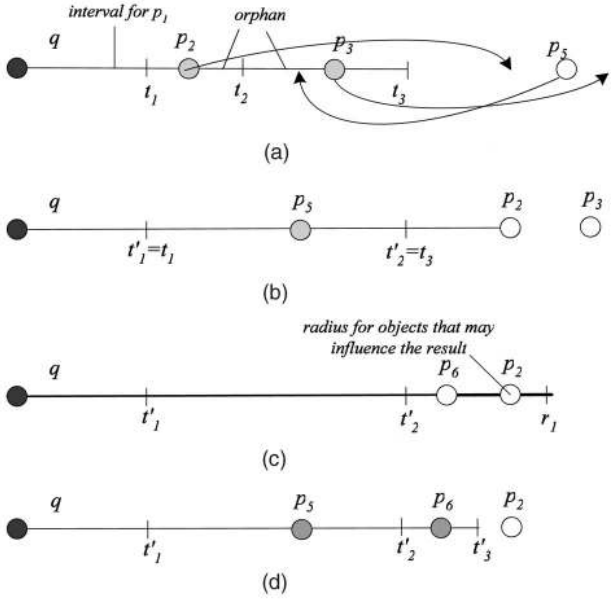

Fig. 7. Outgoing objects are more than incoming. (a) Before processing violations: $NN < p_1, p_2, p_3, >$. (b) After processing p_5: $NN < p_1, p_5, >$. (c) Range query for influencing objects. (d) After processing p_6: $NN < p_1, p_5, p_6 >$.

which keeps t_5 $(= t'_6)$ and t_6. If in the future p_6 crosses t_5, it has to be processed as an incoming object anyway; if it crosses t_6, it will issue a (*dummy*) violation, and the server will just send t'_6 to it. A similar *lazy* approach is followed for the other outer objects, which receive the new outer threshold t'_6 in future timestamps, only if they issue a violation of the outdated one t_6 or there is a broadcast due to subsequent violations. In Section 4, we explore an alternative for multiple queries, that piggy-backs threshold updates to broadcast messages issued due to violations incurred in other queries.

If $|O| > |I|$ (second case), the above process will not produce a sufficient number k of neighbors. Consider, for instance, the 3-NN query of Fig. 7a, where p_2 and p_3 move out of t_3, and p_5 moves in, i.e., $T = \{p_2, p_3, p_5\}$, $I = \{p_5\}$, and $O = \{p_2, p_3\}$. The while-loop of Fig. 5 will discover p_1 and p_5 as the two first NNs (Fig. 7b); the third NN can be any object p with $dist(p, q)$ between t_3 and d'_2, where d'_2 is the updated distance of the closest outgoing object p_2. In order to find such objects, the server could broadcast a query asking for all outer objects in the circle (q, d'_2). Since d'_2 might be large (i.e., p_2 has moved relatively far away from q), the qualifying objects may be more than enough, incurring unnecessary uplink cost. To avoid this, the server determines a radius r_1 that is expected to contain sufficient NNs, assuming uniform object distribution around q; r_1 is decided in the way discussed in the beginning of the section for acquiring the initial query result. For our example,

$$r_1 = t_3 \cdot \sqrt{3/2}.$$

Then, r_1 is compared with d'_2, and the smaller one (in this example, d'_2) defines the range of the query to broadcast.

Let U_1 be the set of outer objects in (q, d'_2), which also contains the outgoing objects falling in the circle (although they do not respond since they have issued a violation at the same timestamp). Assume that p_6 responds to the range query and $U_1 = \{p_6, p_2\}$. The objects of U_1 are sorted on their distance from the query point and the first one (in the general case, the first $|O| - |I|$) will complete the NN set (Fig. 7c). $U_2 (= \{p_6\})$ denotes the subset of U_1 that contains the new NNs. The new outer threshold[4] t'_3 is set to $(d'_6 + d'_2)/2$ and the objects of $(T - O) \cup U \cup U_2$ are informed about their new thresholds through unicast messages. In Fig. 7d, p_5 and p_6 receive (t'_1, t'_2) and (t'_2, t'_3), respectively. Object p_1 keeps its old thresholds since it did not incur a violation and was not involved in conflicts. The remaining (outer) objects use t'_3, which is transmitted through broadcasting.

Objects that appear/disappear (e.g., taxis that start/cease service) can be handled as incoming/outgoing objects, respectively. In particular, initially, a new object p reports its position to the server. If its distance from q is equal to or larger than t_k, p receives the threshold t_k and does not participate in the NN selection; otherwise, p is processed as an incoming object. Similarly, the disappearance of outer objects does not affect query processing. Disappearing inner objects are considered as outgoing ones that move infinitely far away from q and, thus, handled trivially by the monitoring algorithm of Fig. 5.

3.2 Cost Models

Independently of the monitoring method, the server has to receive a query request, compute the initial result, and report it back to the client together with subsequent updates. Thus, the difference among individual methods (and the focus of the subsequent models) is due to the overhead of messages between the server and the data

4. If no new objects are found in the range (i.e., $U_1 = \{p_2\}, U_2 = \{p_2\}$), the new outer threshold becomes $t'_3 = d'_2$.

objects for tracking result changes. Let C_u be the cost of an uplink message from an object to the server, C_d the cost of a downlink message (i.e., from the server to an object), and C_b the cost of broadcasting (from the server to all objects). We identify three different location update strategies.

3.2.1 Upper Bound

The upper bound corresponds to the naïve policy, where each object issues an update at every timestamp. Assuming that the cardinality $|N|$ of the data objects is constant, the cost for each timestamp is:

$$Cost_{UB} = |N| \cdot C_u.$$

3.2.2 Lower Bound

The lower bound for the communication cost may be computed by only considering the number of NN changes per timestamp. For instance, in Fig. 6 the (formerly outer) object p_8 overpasses p_5 and p_6 to become the new fifth NN. Thus, the positions of p_8, p_5, and p_6 are essential for determining the change. On the other hand, the updates of p_2, p_3, and p_4 are not required because their relative order in the result remains the same despite the violations. Similarly, in Fig. 7, we only need the current locations of p_2, p_3, p_5, and p_6 to determine the new result. In general, the set S of objects that need to issue updates contains:

1. inner objects that are involved in a NN order change (swap),
2. formerly outer objects that become part of the result, and
3. formerly inner objects that no longer belong to the result.

Formally,

$$Cost_{LB} = |S| \cdot C_u,$$

where $|S|$ is the cardinality of S. The maximum value of $Cost_{LB}$ is $2 \cdot k \cdot C_u$, and occurs when all k NNs are replaced by outer objects (in which case, both the old and the new k-NNs must issue location updates). Clearly, this is a hypothetical method, not useful in practice, since it requires prior knowledge of all results.

3.2.3 Threshold-Based

This policy corresponds to our approach and aims at minimizing the message overhead, without any prior knowledge of the results. In order to obtain its cost, we distinguish the two cases of Section 3.1. The first one refers to the situation where $|I| \geq |O|$, and incurs cost:

$$Cost_{TB1} = (|T| + |U|) \cdot C_u + (|T| + 2|U|) \cdot C_d,$$

where T is the set of objects that violate their assigned thresholds and U is the set of objects that must report their location (although they have not incurred violations). The rationale of the formula is that the objects in T need to inform the server about range violations (cost $|T| \cdot C_u$). Consequently, the server has to ask and obtain the locations of the objects in U (cost $|U| \cdot (C_u + C_d)$) in order to resolve the conflicts. Finally, it needs to update the threshold values for the objects of T and U (cost $(|T| + |U|) \cdot C_d$). In Fig. 6, $T = \{p_2, p_4, p_8\}$, $U = \{p_3\}$ (because p_2, p_4 fall in the range of p_3), and $Cost_{TB1} = 4 \cdot C_u + 5 \cdot C_d$.

In the second case ($|O| > |I|$), range queries have to be performed in order to find the outer influencing objects. For generality, assume that there also exist object disappearances and that the total number of range queries is n. The cost of the threshold-based policy now becomes:

$$Cost_{TB2} = (Cost_{TB1} - |O| \cdot C_d) + |U_1 - O| \cdot C_u + |U_2| \cdot C_d + (n+1) \cdot C_b,$$

where U_1 is the set of objects that satisfy some range query and $U_2 \subseteq U_1$ is the subset including objects in the new NN set. The first term $(Cost_{TB1} - |O| \cdot C_d)$ is because the second case also involves the computations of the first one, but we do not need to send individual messages to outgoing objects (due to subsequent broadcasting). The cost of the n broadcast queries is $n \cdot C_b$. Every object in U_1 that satisfies some range query, except for the ones in O (that have incurred violations and informed the server about their current positions), must send an uplink message (cost $|U_1 - O| \cdot C_u$). After the NN set is completed, the server sends the new thresholds to the inner objects with unicast messages ($|U_2| \cdot C_d$ since the cost of messages to other objects is included in $Cost_{TB1}$) and broadcasts the new outer threshold (cost C_b). In the example of Fig. 7, $n = 1$, $T = \{p_2, p_3, p_5\}$, $U = \emptyset$, $O = \{p_2, p_3\}$, $U_1 = \{p_2, p_6\}$, $U_2 = \{p_6\}$, and $Cost_{TB2} = 4 \cdot C_u + 2 \cdot C_d + 2 \cdot C_b$.

We expect that for most practical scenarios

$$Cost_{LB} < Cost_{TB} \ll Cost_{UB},$$

but the exact values depend on the specific problem parameters (e.g., moving patterns, object speed) and the underlying network (relative cost of downlink and uplink messages, cost of broadcasting). For instance, if the moving objects are power-limited, the value of C_u should be much larger than C_d since for mobile devices, more power is consumed in the "transmitting" than in the "receiving" mode [9]. On the other hand, an application implemented on top of a GPRS-like service should set these values according to the expected length (in bytes) of each message since the pricing policy of GPRS services is based on the amount of transferred bytes.

3.3 Application to Other Queries

The proposed techniques are not restricted to nearest neighbor search, but can be easily applied to other query types with practical relevance, such as range queries. In this case, the client specifies its location q and a range d, and asks for all objects within distance d from q. Continuous monitoring of range queries can be captured by a simplified version of our algorithm that works as follows:

1. Initially, the server broadcasts (q, d), acquires all objects in the range, and reports their ids to the client.
2. Each object p uses d as a single threshold and when it violates it (i.e., p moves in or out of the range), informs the server accordingly.
3. At each timestamp, the server collects the id of every object p that issues a violation and transmits it to the client.

If the client has already p (in its result list), p is deleted (it has moved out of the query); otherwise, it is inserted to the result.

In addition, we can solve other, nonconventional, variations of continuous monitoring that involve spatial conditions. Consider, for instance, an *inverse range* query, which specifies a point q, an integer k, and asks for the minimum range d_k around q that contains k objects (but not the object ids). The monitoring of such a query requires a combination of the techniques for range and k-NN search. In particular, the server must first acquire the k-NN set (in order to compute the initial value of d_k), but afterwards it only processes violations of the outer threshold since the order of the NNs is not important. Therefore, similar to range queries, each object keeps a single threshold t_k, which, however, changes with time (unlike range queries where it is set to a fixed value d).

Finally, it is worth pointing out that the proposed methods are independent of the distance definition since we do not make any assumptions that are particular to Euclidean space. For instance, in the taxi call-service scenario, the clients are more likely to request the NNs in terms of the network (as opposed to Euclidean) distance, or in terms of the expected travel time to the query point. Our techniques can be used directly, provided that each object can compute the measure of interest. The thresholds again define the range where each object can lie without influencing the result according to the new measure. For instance, if the network distances (travel time) of the first two NNs is 1 and 3km (minutes), respectively, the threshold between them could be set to 2km (minutes). In fact, as we show in the experimental evaluation, the behavior of the proposed algorithms is similar to the Euclidean case, independently of the application domain. Furthermore, the same concepts apply to spaces of higher dimensionality.

4 Multiple Moving Queries

Having solved the fundamental static-query case, we extend the threshold-based algorithm to multiple, possibly moving, queries. Let Q be the set of continuous queries that are simultaneously monitored by the system. For each query, the server stores its location, its last reported result, and the set of thresholds. Similarly, a data object p keeps the query locations and its threshold (pair of thresholds, if p is an inner object) with respect to every query. At each timestamp, the server collects the violations and processes all queries in parallel, trying to minimize the overhead by maximizing *information sharing*. In particular, if the location of an object is obtained for a query, this information is also shared by other ones that may use the object's position. In order to further reduce the network cost, range queries (needed to obtain the locations of outer objects that may influence some result) are broadcast together (i.e., a single broadcast asks for objects in the vicinity of multiple queries). Also, if an object satisfies multiple ranges, it reports its location only once. Transmission of thresholds occurs after all queries have been processed so that if an object has to receive an update for the needs of a query, it also receives the updated thresholds with respect to all other queries (in a single unicast message).

Furthermore, every time a message is broadcast (because the outer threshold of a query increases), it is appended with the outer thresholds of all other queries whose new thresholds have decreased since the last broadcast. Consider again the example of Fig. 6 where the outer threshold t'_6 has decreased, but the outer objects (including former NN p_6) have not been informed. If the server needs to broadcast the outer threshold of another query, the new value (t'_6) and the corresponding query id is also included in the message. Each object compares t'_6 with its previous threshold t_6 (for this query) and if $t'_6 < t_6$, it is kept as the most updated one. If an object has two thresholds (e.g., the object was or still is part of the result), t'_6 is compared with the smallest one t'. If $t'_6 \leq t'$, the object (e.g., p_6) has ceased to be part of the NN set and stores t'_6 as the new outer threshold. Otherwise, if $t'_6 > t'$. the object (e.g., p_1, p_2, p_3, p_4, p_8, and p_5 in Fig. 6) is still one of the k NNs and ignores t'_6.

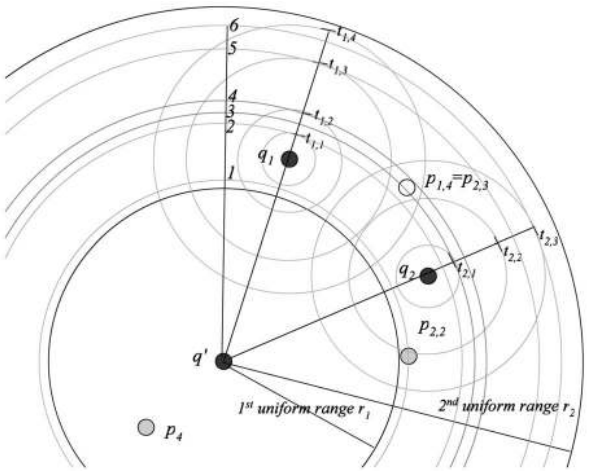

Fig. 8. Example of query addition.

Similar to the data objects, new queries may appear in the system, while existing ones may be terminated. Once a client requests the termination of a query, the server broadcasts a message to all objects, which simply remove the query location and the associated thresholds from their local memory. The handling of appearing (new) queries is more complex. Consider Fig. 8 that contains two running queries q_1, q_2 (requesting four and three NNs, respectively) and let p_{ij} be the jth neighbor of q_i. The fourth NN of q_1 is also the third NN of q_2, i.e., $p_{1,4} \equiv p_{2,3}$. The server knows the exact location of $p_{2,2}$ (e.g., it issued a violation), while for the remaining five objects, it only has the corresponding query thresholds. Assume that at this timestamp, a new 4-NN query q' arrives. According to the methodology of Section 3.1, a range query should be broadcast asking for all objects within the circle (q', r_1), where $r_1 = \sqrt{4/\pi \cdot |N|}$. If only one object (p_4) replies, a second range query should be performed, where $r_2 = 2 \cdot r_1$. Note that, because the area around q' is sparse, r_2 is larger than necessary since the circle already covers the six NNs of q_1 and q_2 (whereas we need only three more objects). In order to reduce the extend of these ranges (and, thus, the number of messages from qualifying objects), the server performs the following tasks. For each NN p_{ij} of some query, it computes a value d_{ij} which equals 1) the actual distance $dist(p_{ij}, q')$, if the current

location of p_{ij} is known (e.g., $p_{2,2}$), or 2) an upper bound $dist(q_i, q') + t_{ij}$, where t_{ij} is the right (i.e., largest) threshold of p_{ij} in q_i, if the current location of p_{ij} is not known. If p_{ij} is a NN of multiple queries, the minimum of the bounds is chosen, e.g., in Fig. 8

$$d_{1,4} = d_{2,3} = min(dist(q_1, q') + t_{1,4}, dist(q_2, q') + t_{2,3}).$$

Then, the objects are sorted according to their d_{ij}.

Intuitively, the first d_{ij} (in this case, $d_{2,2}$) defines a circle $(q', d_{2,2})$ that contains at least one object ($p_{2,2}$); the second one ($d_{1,1}$) defines a circle $(q', d_{1,1})$ that contains at least two objects ($p_{2,2}$ and $p_{1,1}$), etc. The number of objects in each circle is shown on the vertical line of Fig. 8. These numbers are lower bounds because 1) they only take into account objects that belong to the results of running queries and 2) if the actual position of an object is not known, it is assumed to have the farthest permissible distance from q'. The server utilizes the additional information by first deciding the appropriate circle that is guaranteed to contain the number (4) of objects requested by q', which in Fig. 8 corresponds to $(q', d_{1,2})$. The value of $d_{1,2}$ is compared with the range of the uniform estimation, and the smallest one (in this case, r_1) is used for the first broadcast query. Since the query returns a single object (p_4), the server chooses $d_{2,1}$ for the second range $(q', d_{2,1})$ because it is smaller than r_2 and is guaranteed to retrieve three more objects ($p_{2,2}$, $p_{1,1}$, and $p_{2,1}$).

Now, we remove the static query assumption and consider that, as in the case of objects, queries may issue location updates at discrete timestamps. The difference is that, for queries, we cannot have a threshold mechanism to minimize the amount of violation handling; even a small query movement may cause significant changes in the result and should be processed accordingly. Fig. 9 shows an example where at some timestamp that does not involve any other violations, a 4-NN query moves to a new position q', whose distance from the original location q_1 is d_q. If the objects are close to (but have not violated) their thresholds as shown in Fig. 9, the order of $p_{1,1}$, $p_{1,2}$, and $p_{1,3}, p_{1,4}$ changes, while some outer object may replace $p_{1,3}$ in the result. Since the actual positions of the previous NNs are not known, the server should broadcast a query asking for all objects in the circle $(q', t_{1,4} + d_q)$, which is guaranteed to retrieve at least $p_{1,1}$, $p_{1,2}$, $p_{1,3}$, and $p_{1,4}$.

In the above example, we intentionally assumed that d_q is small so that the new query location q' is close to the previous one q_1. In the general case, d_q may be arbitrarily large, so that 1) none of the NNs of q_1 belongs necessarily to the result of q', and 2) some NNs of other running queries may become NNs of q'. Such a situation is derived from Fig. 8, considering that q' corresponds to a location update of q_1 (instead of a new query). In this case, broadcasting a range query $(q', t_{1,4} + d_q)$ incurs unnecessary overhead since the server can utilize the available information about q_1 and q_2 to restrict the radius. In particular, query movement is handled as a deletion (of q_1) and insertion (of q') following the technique discussed in the context of Fig. 8, i.e., the server first broadcasts a "uniform" range (q', r_1) (retrieving p_4) and then a second one $(q', d_{2,1})$ (which contains at least $p_{2,2}$, $p_{1,1}$, and $p_{2,1}$). This method trivially captures the example of Fig. 9 and, in general, any query movement.

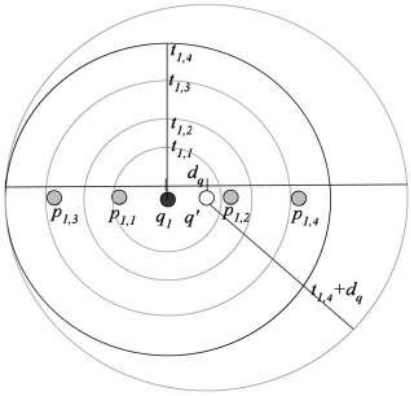

Fig. 9. Query movement without object violations.

5 Experimental Evaluation

In this section, we evaluate the performance of the proposed framework and compare it with the lower and upper bounds as discussed in Section 3.2. We use two data sets: In the first one, denoted as *spatial*, we randomly select the initial position and the destination of each object from a real dataset (Los Angeles, available at www.rtreeportal.org). The object follows a linear trajectory (with constant velocity) between the two points. Upon reaching the endpoint, a new random destination is selected and the same process is repeated. Distance is defined according to the Euclidean metric. The second data set, denoted as *road*, is created with the spatio-temporal generator of [2]. The input of the generator is the road map of Oldenburg (a city in Germany), consisting of 6,105 nodes and 7,035 edges. The output is a set of objects (e.g., cars, pedestrians) moving on this network, where each object is represented by its location at successive timestamps. An object appears on a network node, completes a shortest path to a random destination and then disappears. At each timestamp, we control the number of appearing objects so that the total number $|N|$ (a parameter) remains constant. The measure of interest is the *network distance*, which is defined as the length of the shortest path that connects the object to the query.

Table 1 summarizes the parameters under investigation, along with their ranges. Their default (median) values are typeset in boldface. In each experiment, we vary a single parameter, while setting the remaining ones to their default values. The reported results represent the average value over 20 simulations. For each simulation, the query point is one of the data points in the Los Angeles data set (for *spatial*) or a node in the network (for *road*), and the NNs are monitored for 1,000 timestamps.[5] We assume that the costs of uplink and downlink messages are equal (i.e., $C_d = C_u = 1$), whereas the cost of broadcasting C_b is a function of C_d (with default value $8 \cdot C_d$). In accordance with the presentation of the proposed algorithms, Section 5.1 evaluates the single static query case, while Section 5.2 deals with multiple, possibly moving queries.

5. Due to the huge space and time requirements of simulations in large network topologies, we use smaller values for $|Q|$ and $|N|$ in the *road* experiments.

TABLE 1
System Parameters (Ranges and Default Values)

Parameter	*Spatial*	*Road*
Object cardinality ($\|N\|$)	1, 4, 16, **64**, 256, 1024 (K)	1, 2, 4, **8**, 16, 32, 64 (K)
Number of NNs (k)	1, 2, 4, **8**, 16, 32, 64	1, 2, 4, **8**, 16, 32, 64
Object speed	slow, **medium**, fast	slow, **medium**, fast
Cost of broadcasting (C_b/C_d)	1, 2, 4, **8**, 16, 32, 64	1, 2, 4, **8**, 16, 32, 64
Query cardinality ($\|Q\|$)	1, 4, 16, **64**, 256, 1024	1, 2, 4, **8**, 16, 32, 64
Query moving period	1, 2, 4, **8**, 16, 32, 64	1, 2, 4, **8**, 16, 32, 64
Query speed	slow, **medium**, fast	slow, **medium**, fast

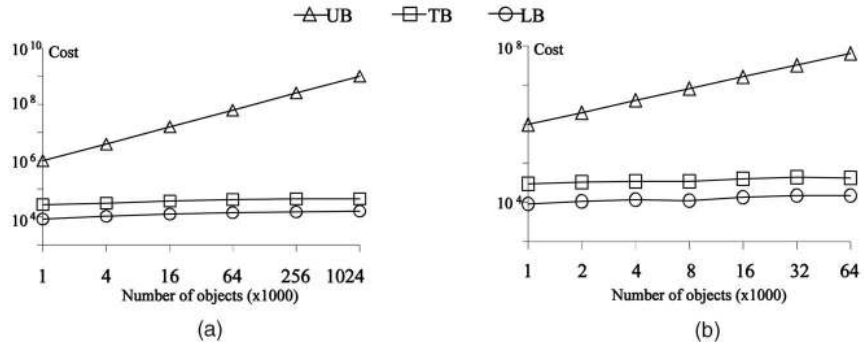

Fig. 10. Message cost versus object cardinality $|N|$: (a) *Spatial*. (b) *Road*.

5.1 Single Static Query

The first experiment evaluates the effect of the object cardinality $|N|$. Figs. 10a and 10b show the communication cost (over all timestamps), as a function of $|N|$, for the *spatial* and *road* data sets, respectively. Our threshold-based (TB) policy scales well with the population size since $|N|$ affects the cost only indirectly. In particular, the cost is determined entirely by the objects in the vicinity of the query point (as the distant ones cannot influence the result), whose number increases with $|N|$ due to the higher density. The results are similar for both data sets, indicating the generality of our techniques. The cost of the upper bound (UB) approach is $1,000 \cdot |N|$ since each object issues one update per timestamp. The lower bound (LB) strategy incurs between 9K and 15.5K messages in all cases. Recall that its cost (determined by the number of NN changes) lies in the range $[0, 2 \cdot k]$ per timestamp. Under these settings, the maximum number of messages over all timestamps can be 16K (if all nearest neighbors change at every timestamp), independently of $|N|$. As expected, the performance of TB is close to that of LB (around three times more expensive) and two to four orders of magnitude better than UB.

In order to provide some intuition about the network cost breakdown, Table 2 presents the number of uplink, downlink, and broadcast messages, for monitoring 8-NNs over a set of $|N|$ objects that move with medium speed during a period of 1K timestamps. In addition, the table contains (in parentheses), the percentage of the total cost incurred by each message category assuming that $C_u = C_d$ and $C_b = 8 \cdot C_d$. For both *spatial* and *road* data sets, uplink transmissions constitute the dominant factor in the total number of messages and, subsequently, in the overall cost (over 40 percent). This is because uplink messages are issued by objects that incur violations, and more importantly by objects that respond to broadcast (range) queries even though they do not necessarily influence the result. On the other hand, the number of broadcasts is small (only 3 percent to 4 percent of the total messages) and their contribution to the cost lies between 23 percent to 25 percent. The slightly higher relative cost of broadcasting for *road* is mainly due to the fact that it includes object deletions.

Nevertheless, despite their very different characteristics (cardinality, data distribution, type of movement, distance metric), the two data sets incur analogous overhead and number of messages per category. This trend is apparent in all subsequent experiments, where the performance is practically identical for both data sets, confirming the robustness of our methods independently of the application domain.

Figs. 11a and 11b illustrate the effect of k (number of NNs required by the query). In both cases, the cost of LB increases almost linearly with k, due to the reasons explained in the context of Fig. 10. TB behaves in a similar way because the number of thresholds (and, therefore, their possible violations) also increases linearly with k. On the other hand, the cost of UB is constant because it only depends on the object cardinality, and not on the number of NNs.

Fig. 12 shows the cost as a function of the object speed. Objects with *slow* speed cover a distance that equals 1/250 of the axis length per timestamp. *Medium* and *fast* speeds correspond to distances that are 5 and 25 times larger, respectively. The same concept is applied on both the *spatial* and *road* data sets, the only difference being that in the second case, objects are restricted to move on the network edges. The network cost is again constant for UB, and increases with the speed for TB and LB, because higher object agility causes a larger number of nearest neighbor changes.

Finally, in order to evaluate the applicability of our techniques to different network infrastructures, we vary the relative cost of C_b with respect to C_d. Fig. 13 measures the overall cost as a function of C_b in the range $[C_d, 64 \cdot C_d]$. UB

TABLE 2
Number of Messages per Data Set

Message Type	*Spatial* ($\|N\|$=64K)	*Road* ($\|N\|$=8K)
Uplink	15066 (42.9%)	14574 (41.7%)
Downlink	11912 (33.9%)	11571 (33.1%)
Broadcast	1021 (23.2%)	1104 (25.2%)
Total	35146	27249

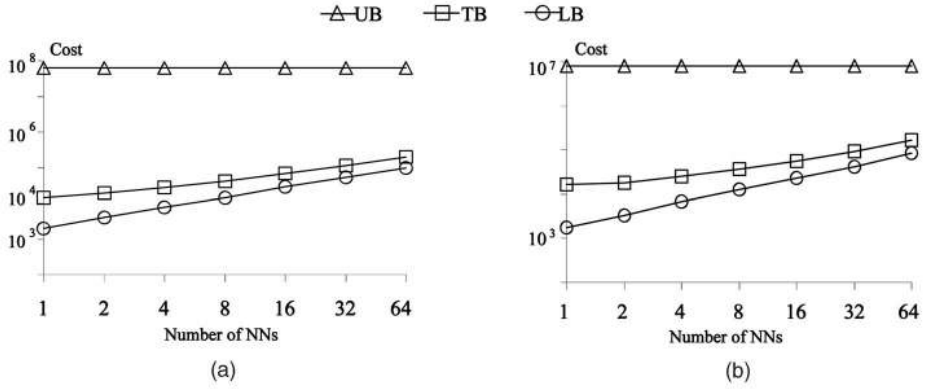

Fig. 11. Message cost versus k. (a) *Spatial* ($|N| = 64$K). (b) *Road* ($|N| = 8$K).

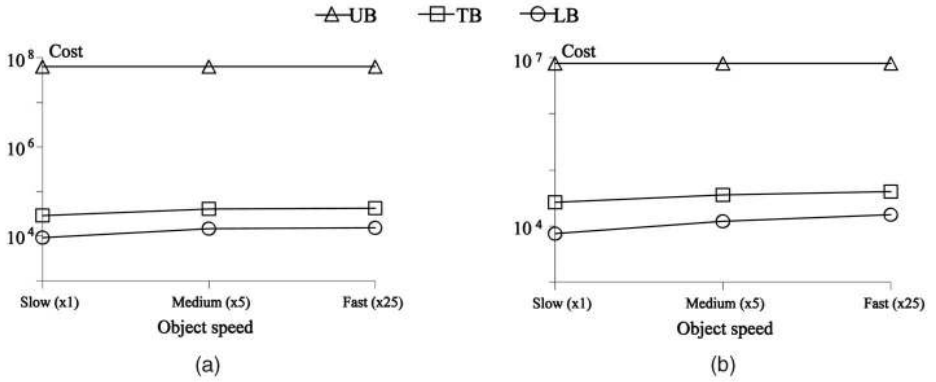

Fig. 12. Message cost versus object speed. (a) *Spatial* ($|N| = 64$K). (b) *Road* ($|N| = 8$K).

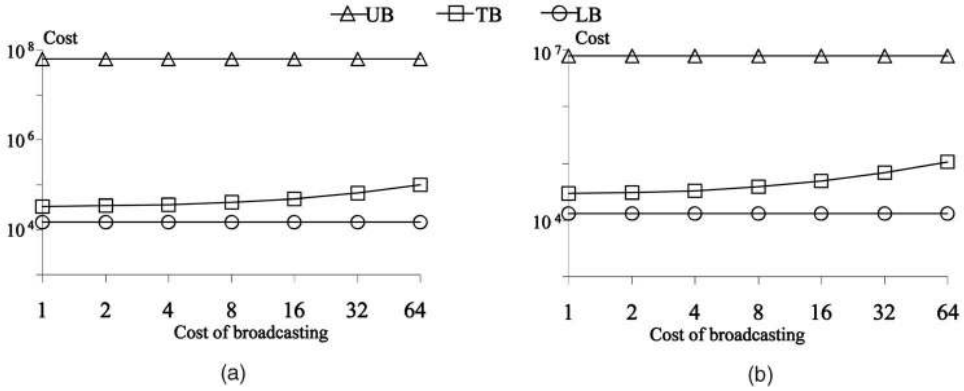

Fig. 13. Message cost versus C_b/C_d. (a) *Spatial* ($|N| = 64$K). (b) *Road* ($|N| = 8$K).

and LB do not involve any broadcasting and, therefore, they are not affected by C_b. On the other hand, the cost of TB does not grow much with C_b because, as shown in Table 2, broadcasting contributes a minor percentage in the total number of messages. Recall that due to the lazy update policy, broadcast is only required when the value of the outer threshold increases.

5.2 Multiple Moving Queries

The experiments for multiple moving queries use similar settings to the previous ones, but involve some additional parameters, namely, the cardinality $|Q|$ of the query set, the query speed, and the period of movement. First, assuming static queries, Fig. 14 explores the effect of $|Q|$ on the performance, where the remaining parameters are set to

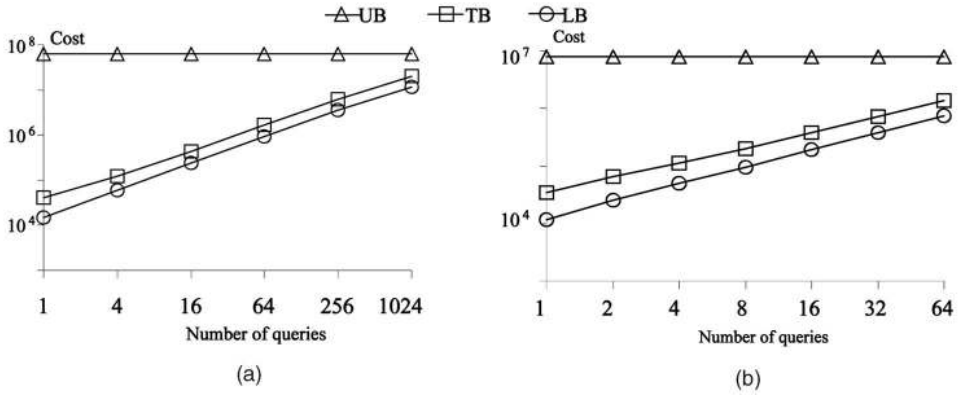

Fig. 14. Message cost versus query cardinality $|Q|$. (a) *Spatial* ($|N| = 64\text{K}$). (b) *Road* ($|N| = 8\text{K}$).

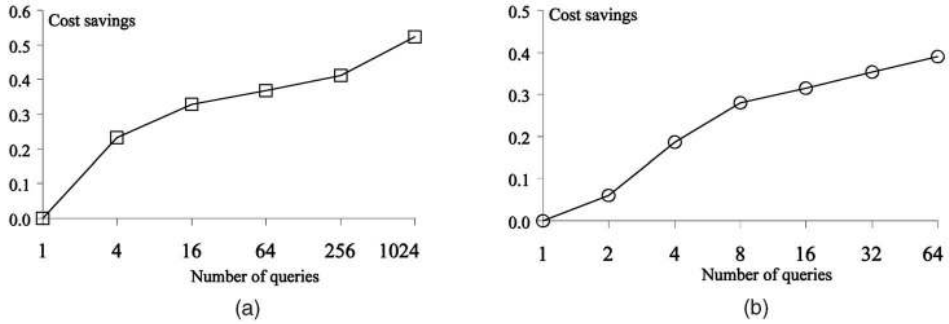

Fig. 15. Savings from information sharing. (a) *Spatial* ($|N| = 64\text{K}$). (b) *Road* ($|N| = 8\text{K}$).

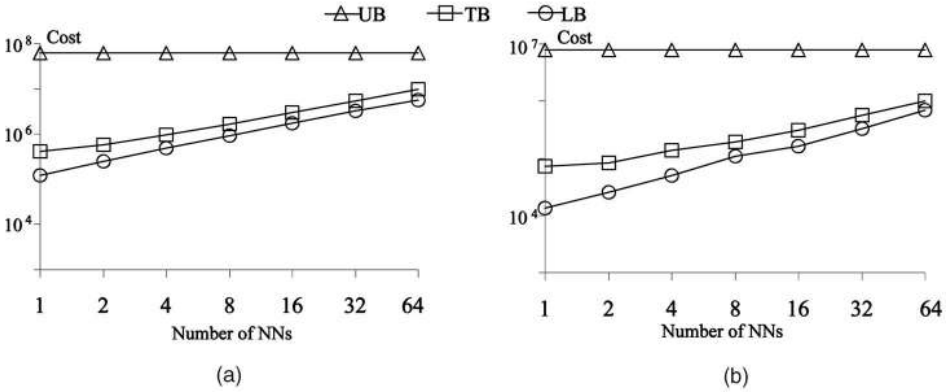

Fig. 16. Message cost versus k. (a) *Spatial* ($|Q| = 64$). (b) *Road* ($|Q| = 8$).

their default values, as shown in Table 1. Obviously, UB exhibits constant behavior independent of $|Q|$. The cost of LB lies in the range $[0, 2 \cdot |Q| \cdot k]$ per timestamp (i.e., in the worst case, all NNs of all queries may change at each timestamp). TB performs very well, and approaches the lower bound because the information at the server and the communication overhead is shared among the queries.

The concept of information sharing is further explored in Fig. 15, which shows the cost savings as a function of $|Q|$. In particular, the cost saving for $|Q|$ queries is defined as: $(|Q| \cdot Cost_1 - Cost_{|Q|})/|Q| \cdot Cost_1$, where $Cost_1$ is the overhead of processing a single query, and $Cost_{|Q|}$ the overhead of processing $|Q|$ queries simultaneously. As discussed in Section 4, the savings of simultaneous processing are due to the sharing of:

1. broadcast ranges,
2. uplink messages from objects that qualify multiple ranges at some timestamp, and
3. downlink messages to objects that influence various queries.

As $|Q|$ increases, so does the amount of sharing, and when, for instance, $|Q|$ reaches 64, $Cost_{|64|}$ is only 39 times higher than $Cost_1$ (instead of $64 \cdot Cost_1$) for the *road* dataset. Fig. 16 shows the cost as a function of k. The diagrams and their explanation are similar to the single query case (Fig. 11).

As opposed to the previous experiments that assume static queries, the following ones consider query movement. We first examine the effect of the query speed which, similar to the object speed, is classified as *slow*, *medium*, and *fast*, depending on the distance that the query moves from its previous location. We assume $|Q|$ concurrent queries,

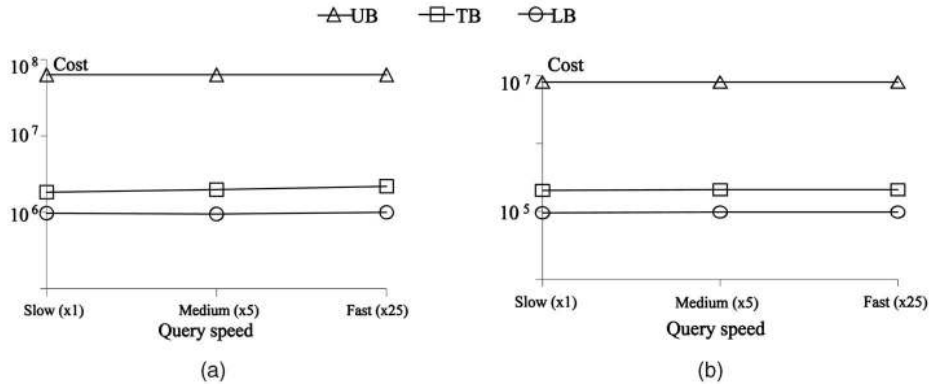

Fig. 17. Message cost versus query speed. (a) *Spatial* ($|Q| = 64$). (b) *Road* ($|Q| = 8$).

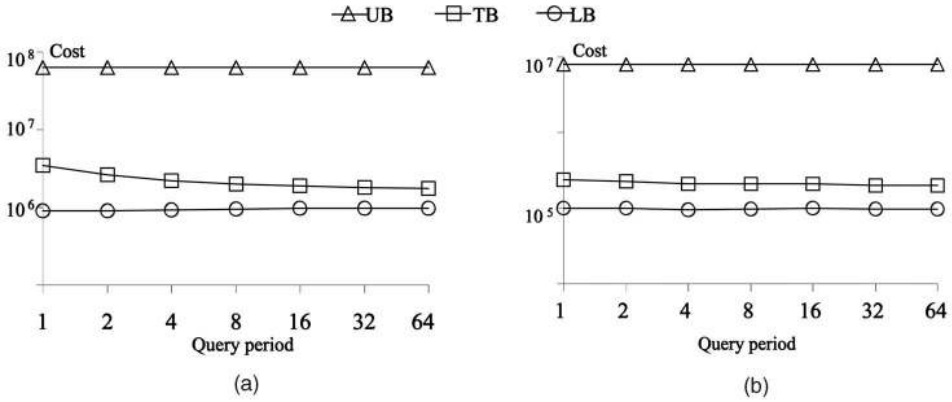

Fig. 18. Message cost versus query period. (a) *Spatial* ($|Q| = 64$). (b) *Road* ($|Q| = 8$).

each asking for $k = 8$ NNs and moving every eight timestamps. As shown in Fig. 17, the cost of LB is almost constant and close to its maximum value $2 \cdot |Q| \cdot k$ because the query movements are likely to invalidate all NNs (in addition to the invalidations due to object movements). The performance of TB is again closer to LB than to UB, and it is not very sensitive to the query speed.

Finally, Fig. 18 shows the communication cost as a function of the query period, i.e., the number of timestamps between two subsequent query movements. For instance, if the period is 1, all queries move at every timestamp, whereas if the period is 64, each query issues a location update every 64 timestamps. As expected, the cost of TB is higher for frequent movements because each movement requires some broadcast range queries and subsequent uplink messages to obtain the results. Nevertheless, its performance is robust (and much better than the naïve policy) even if all queries issue updates continuously.

6 Conclusions

This paper introduces and solves the problem of continuous k-NN monitoring over moving objects. Given a set of geographically distributed clients, the goal of the server is to constantly report the NN set of each client with the minimum communication overhead. Our contribution is a set of novel techniques, which are based on the intuition that up-to-date information is necessary only for objects that may influence some query result. The proposed methods can deal with multiple, static, or moving queries independently of the underlying distance definitions. Extensive experimental evaluation, using Euclidean and road network settings, confirms that they are robust and efficient, even when compared with a, practically infeasible, optimal method.

We believe that the continuous monitoring of spatial queries will play a central role in numerous applications related to mobile computing and spatio-temporal databases. At the same time, it poses several interesting research issues. A direction for further investigation refers to the extension of the proposed methodology to other query types. As briefly discussed in Section 3.3, variations of range search can be processed by a straightforward adaptation of our techniques. However, other forms of continuous monitoring are not trivial, as, for instance, spatial aggregate processing [23]. In the "monitoring" version of this problem, the client could require continuous reports about the value of some functions (e.g., number of cars, average traffic, maximum speed) in its vicinity. Another promising direction refers to approximate k-NN monitoring for situations where the exact results are not necessary and the communication resources are limited. A solution to this problem may use threshold-based techniques that adapt based on the trade-off between accuracy and network overhead. Furthermore, in this case, it would be interesting to devise probabilistic methods for providing guaranteed error bounds.

Acknowledgments

This work was supported by grants HKUST 6184/05E and CityU 1163/04E from Hong Kong RGC.

REFERENCES

[1] C. Bohm, "A Cost Model for Query Processing in High Dimensional Data Spaces," *ACM Trans. Database Systems,* vol. 25, no. 2, pp. 129-178, 2000.

[2] T. Brinkhoff, "A Framework for Generating Network-Based Moving Objects," *GeoInformatica,* vol. 6, no. 2, pp. 153-180, 2002.

[3] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis, "Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects," *Proc. 2002 Int'l Symp. Database Eng. & Applications (IDEAS),* 2002.

[4] B. Babcock and C. Olston, "Distributed Top-k Monitoring," *Proc. ACM SIGMOD Conf.,* 2003.

[5] Y. Cai, K. Hua, and G. Cao, "Processing Range-Monitoring Queries on Heterogeneous Mobile Objects," *Proc. 2004 IEEE Int'l Conf. Mobile Data Management,* 2004.

[6] M. Dilman and D. Raz, "Efficient Reactive Monitoring," *Proc. INFOCOM Conf.,* 2001.

[7] B. Gedik and L. Liu, "MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System," *Proc. Int'l Conf. Extending Database Technology,* 2004.

[8] G. Hjaltason and H. Samet, "Distance Browsing in Spatial Databases," *ACM Trans. Database Systems,* vol. 24, no. 2, pp. 265-318, 1999.

[9] C. Jones, K. Sivalingam, P. Agrawal, and J. Chen, "A Survey of Energy Efficient Network Protocols for Wireless Networks," *Wireless Networks,* vol. 7, no. 4, pp. 343-358, 2001.

[10] N. Koudas, B. Ooi, K. Tan, and R. Zhang, "Approximate NN Queries on Streams with Guaranteed Error/Performance Bounds," *Proc. Very Large Data Bases Conf.,* 2004.

[11] D. Kalashnikov, S. Prabhakar, and S. Hambrusch, "Main Memory Evaluation of Monitoring Queries Over Moving Objects," *Distributed and Parallel Databases,* vol. 15, no. 2, pp. 117-135, 2004.

[12] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias, "Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring," *Proc. ACM SIGMOD Conf.,* 2005.

[13] S. Madden, H. Shah, J. Hellerstein, and V. Raman, "Continuously Adaptive Continuous Queries over Streams," *Proc. ACM SIGMOD Conf.,* 2002.

[14] M. Mokbel, X. Xiong, and W. Aref, "SINA: Scalable Incremental Processing of Continuous Queries in Spatio-Temporal Databases," *Proc. ACM SIGMOD Conf.,* 2004.

[15] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, and S. Hambrusch, "Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects," *IEEE Trans. Computers,* vol. 51, no. 10, pp. 1124-1140, 2002.

[16] D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao, "Query Processing in Spatial Network Databases," *Proc. Very Large Data Bases Conf.,* 2003.

[17] N. Roussopoulos, S. Kelly, and F. Vincent, "Nearest Neighbor Queries," *Proc. ACM SIGMOD Conf.,* 1995.

[18] Z. Song and N. Roussopoulos, "K-Nearest Neighbor Search for Moving Query Point," *Proc. Int'l Symp. Spatial and Temporal Databases,* 2001.

[19] Y. Tao and D. Papadias, "Spatial Queries in Dynamic Environments," *ACM Trans. Databsase Systems,* vol. 28, no. 2, pp. 101-139, 2003.

[20] X. Xiong, M. Mokbel, and W. Aref, "SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-Temporal Databases," *Proc. Int'l Conf. Data Eng.,* 2005.

[21] X. Yu, K. Pu, and N. Koudas, "Monitoring K-Nearest Neighbor Queries over Moving Objects," *Proc. Int'l Conf. Data Eng.,* 2005.

[22] B. Zheng and D. Lee, "Semantic Caching in Location-Dependent Query Processing," *Proc. Int'l Symp. Spatial and Temporal Databases,* 2001.

[23] D. Zhang, V. Tsotras, and D. Gunopulos, "Efficient Aggregation over Objects with Extent," *Proc. ACM Symp. Principles of Database Systems,* 2002.

[24] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. Lee, "Location-Based Spatial Queries," *Proc. SIGMOD Conf.,* 2003.

Kyriakos Mouratidis received the BSc degree in computer science in 2002 from the Aristotle University of Thessaloniki, Greece. Currently, he is a PhD candidate in the Computer Science Department at the Hong Kong University of Science and Technology. His research interests include spatio-temporal databases, data stream processing, and mobile computing.

Dimitris Papadias is an associate professor at the Computer Science Department, Hong Kong University of Science and Technology (HKUST). Before joining HKUST in 1997, he worked and studied at the German National Research Center for Information Technology (GMD), the National Center for Geographic Information and Analysis (NCGIA, Maine), the University of California at San Diego, the Technical University of Vienna, the National Technical University of Athens, Queen's University (Canada), and University of Patras (Greece). He has published extensively and been involved in the program committees of all major database conferences, including SIGMOD, VLDB, and ICDE.

Spiridon Bakiras received the BSc degree in electrical and computer engineering from the National Technical University of Athens, the MSc degree (1994) in telematics from the University of Surrey, and the PhD degree (2000) in electrical engineering from the University of Southern California. Currently, he is a postdoctoral fellow in the Department of Computer Science at the Hong Kong University of Science and Technology. His research interests include high-speed networks, peer-to-peer systems, mobile computing, and spatial databases. He is a member of the ACM and the IEEE.

Yufei Tao received the diploma from the South China University of Technology in August 1999 and the PhD degree from the Hong Kong University of Science and Technology in July 2002, both in computer science. After that, he spent a year as a visiting scientist at the Carnegie Mellon University. Currently, he is an assistant professor in the Department of Computer Science, City University of Hong Kong. Dr. Tao is the winner of the Hong Kong Young Scientist Award 2002 from the Hong Kong Institution of Science. His research includes query algorithms and optimization in temporal, spatial, and spatio-temporal databases.