Distributed, Concurrent Range Monitoring of Spatial-Network Constrained Mobile Objects

Hua Lu[1,2], Zhiyong Huang[1,3], Christian S. Jensen[2], and Linhao Xu[1]

[1] School of Computing, National University of Singapore, Singapore
[2] Department of Computer Science, Aalborg University, Denmark
[3] Institute for Infocomm Research, Singapore

Abstract. The ability to continuously monitor the positions of mobile objects is important in many applications. While most past work has been set in Euclidean spaces, the mobile objects relevant in many applications are constrained to spatial networks. This paper addresses the problem of range monitoring of mobile objects in this setting, in which network distance is concerned. An architecture is proposed where the mobile clients and a central server share computation, the objective being to obtain scalability by utilizing the capabilities of the clients. The clients issue location reports to the server, which is in charge of data storing and query processing. The server associates each range monitoring query with the network-edge portions it covers. This enables incremental maintenance of each query, and it also enables shared maintenance of concurrent queries by identifying the overlaps among such queries. The mobile clients contribute to the query processing by encapsulating their host edge portion identifiers in their reports to the server. Extensive empirical studies indicate that the paper's proposal is efficient and scalable, in terms of both query load and moving-object load.

1 Introduction

In the management of moving objects, continuous monitoring queries have recently gained attention, as they provide the fundamental support to different mobile services, including services that monitor traffic at intersections, services that monitor fleets of vehicles such as buses or police cars, and a variety of services that monitor sensitive regions. Existing work on continuous monitoring queries has predominantly assumed that the moving objects are embedded into two-dimensional Euclidean space, and has relied on Euclidean distance as the relevant notion of distance. However, in many application scenarios, including the ones mentioned above, the moving objects are constrained to a spatial network, typically a road network. In this setting, Euclidean distance is not the relevant notion of distance—rather, network distance is.

In a spatial network setting, the ability to efficiently monitor the part of a network within a certain network distance of a query point for moving objects constitutes fundamental functionality for many mobile services. This paper proposes efficient techniques for such continuous range monitoring queries in spatial networks, which we term $CRMQ_N$ queries. Figure 1 exemplifies a $CRMQ_N$ query. In the partial road network displayed in the figure, a $CRMQ_N$ query q is issued with the position represented by a small square and with a 5 km distance of interest. The dashed circle identifies the range determined by the Euclidean distance, while the arrows with short bars identify the sub-network within network distance 5 km of the query point. Therefore, for the time point captured in the figure, moving objects A, B, C, and D belong to the result; object E does not, as the network path from query point q to E exceeds the query range. Note that a moving object's membership in the query result generally changes as time elapses due to the object's movements. The range monitoring query continuously maintains the correct query result as time elapses.

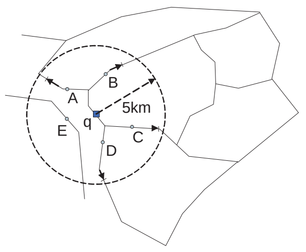


Fig. 1. $CRMQ_N$ query example

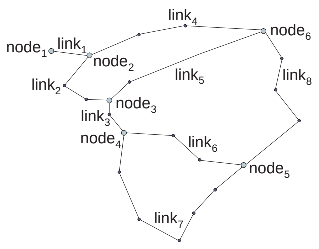


Fig. 2. Road network example

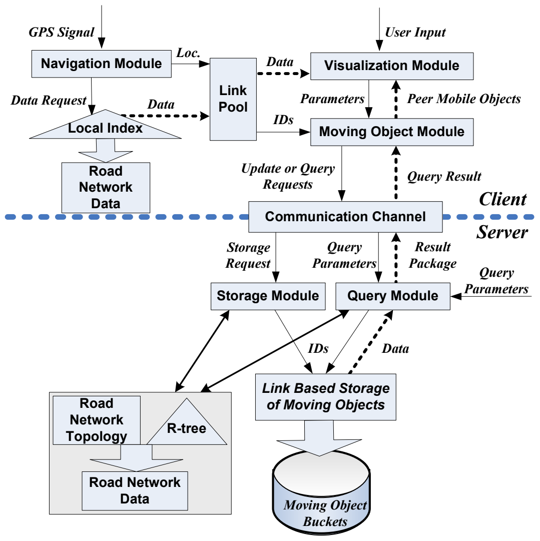


Fig. 3. System framework

We assume a client/server system architecture, in which the moving objects (clients) report their location information to the central server that is responsible for storing the locations of the moving objects and for processing the range monitoring queries. As the moving objects continually send their location reports to the server, the server must update the results of the active queries to maintain the query results. We also assume that query processing occurs in main memory, as do most online monitoring systems.

For each query, we initially use a network-expansion-based algorithm for identifying those network links that are either fully or partially covered by the query. Then in the subsequent query processing, all other ones are ignored as they have no impact on the query result. A *network link* (formally defined in Section 2.1) is a maximum part of a spatial network for which no exchange of traffic is possible .

For each link fully covered by the query, we report every object moving on it in the query result; for each link partially covered by the query, additional refinement based on the network distance is performed to discard non-qualifying objects. This identification of different link types enables us to reduce query processing cost by expending different amounts of effort on different types.

The identification of the links covered by each range monitoring query also discloses the overlaps among concurrent queries. In other words, different queries may cover the same link(s), although they have different query points and ranges. By exploiting such overlaps, we develop a shared maintenance mechanism for concurrent queries. This mechanism comprises particular data structures and relevant algorithms to efficiently update the results of concurrent queries.

In addition, we employ a novel client design that enables the mobile terminals to contribute their computing capabilities to the continuous query maintenance, thus reducing the server side computation. In particular, the host links of a moving object (the road link on which it is moving) is easily determined on client side, and its identifer is then encapsulated in the location report sent to the server, which uses this to facilitate the shared maintenance of concurrent queries.

The paper's key contributions are fourfold. First, we identify all links relevant to a range monitoring query, differentiating between those links that are fully covered versus partially covered by the query. This minimizes the subsequent query maintenance costs, and it also reveals the overlaps among concurrent queries, thus enabling a shared maintenance of these queries. Second, as realistic scenarios entail large numbers of concurrent queries, it is essential to be able to maintain multiple query results correctly and efficiently. We achieve this by means of shared maintenance based on the identification of relevant links. We propose hash-based structures with efficient processing algorithms for this purpose. Third, in link based query processing, the host links on which the mobile objects reside must be identified. To offload the server, we delegate the host link identification to the moving objects. This is enabled via a novel client design that adapts the terminal's navigation software module to determine the host link. Fourth, we report on extensive empirical studies that characterize the efficiency and scalability of our proposal.

In Section 2, we proceed to present some preliminaries. Section 3 describes the specific data management on the mobile terminals. Section 4 details the server-side design for concurrent continuous range monitoring queries of network constrained mobile objects. Section 5 experimentally evaluates the paper's proposals. Section 6 briefly reviews related work, and Section 7 concludes this paper.

2 Preliminaries

2.1 Data Model

Similarly to other proposals (e.g., [3]), we model a road network as a particular kind of spatial network that captures both the graph aspect of a road network and also captures the embedding of a road network into geographical space. Thus, a spatial network $\mathcal{SN} = (\mathcal{N}, \mathcal{L})$ consists of a set of nodes $\mathcal{N} = \{n_1, n_2, \ldots, n_N\}$ and a set of links $\mathcal{L} = \{l_1, l_2, \ldots, l_L\}$. A link consists of a pair of elements from $\mathcal{N}$, and a total function $weight : \mathcal{L} \rightarrow \mathbb{R}^+$ assigns a weight to each link.

The embedding into geographical space is accomplished with two total functions. First, $pos_{\mathcal{N}} : \mathcal{N} \rightarrow \mathbb{R}^2$ maps each node to a position in two-dimensional Euclidean space. Second, $pos_{\mathcal{L}} : \mathcal{L} \rightarrow \{(pos_n(n_s), p_1, ..., p_k, pos_n(n_e)) \mid n_s, n_e \in \mathcal{N} \wedge p_1, \ldots, p_k \in \mathbb{R}^2 \wedge k \geq 0\}$ maps each link to a polyline. A link $l = (n_s, n_e)$ is then mapped to a polyline with the positions of n_s and n_e as its delimiting positions. The k positions in-between these two are termed intermediate points. Consequently, a link is modeled by a polyline consisting of $k + 1$ line segments.

We will assume that the weight of a link is the length of the associated polyline. Next we assume a function d_N that takes any two positions on polylines of a spatial network as arguments and returns the (shortest) network distance between these. Such a function can be defined in straightforward fashion. We will also assume that links model bidirectional roads.

An example of road network is shown in Figure 2. Nodes are given by big dots and intermediate points by small dots. The road network encompasses 8 links, one of which does not contain any intermediate points (i.e., $k = 0$). Link 7 has the largest number of intermediate points ($k = 5$).

We assume a set $\mathcal{M}$ of moving objects that are constrained to the spatial network. Thus, a moving object at any point in time resides on a link in the network, and its position intersects with the polyline that represents the link. Function $pos_{\mathcal{M}}$ maps an object to its current position as known by the server.

2.2 Problem Statement

We assume a spatial network $\mathcal{SN}$, a set of moving objects $\mathcal{M}$ constrained to this network, and a network distance function d_N. Moving objects continually issue updates to the server, thus updating function $pos_{\mathcal{M}}$.

A *continuous network-distance-based range monitoring query* $\mathcal{R}$ takes $\mathcal{M}$ as argument and accepts two parameters: (p, r), where p is the (stationary) query point and r is the network query range. Such a query, termed $CRMQ_N$, is activated at some time t_s, the start time of the query, and it is terminated at some later time t_e, the end time of the query. The query result is maintained from time t_s to time t_e.

$$\forall t \in [t_s; t_e] \quad (o \in \mathcal{R}[p, r](\mathcal{M}) \Leftrightarrow o \in \mathcal{M} \wedge d_N(pos_{\mathcal{M}}(o), p) \leq r)$$

The problem addressed in this paper is that of providing a complete set of techniques that enable the correct and efficient maintenance of multiple such range monitoring queries.

2.3 Distributed System Architecture

Our proposal is based on a client/server architecture, in which the clients are moving objects equipped with mobile terminals and a central data server is in charge of the query processing. The clients communicate with the server via some form of wireless network, e.g., a 2.5 or 3G cellular network. The system framework is shown in Figure 3.

At the highest level of abstraction, a mobile terminal consists of three modules. A navigation module is responsible for retrieving spatial data that represents the object's current location; this is obtained from positioning hardware such as a GPS receiver. A visualization module is responsible for displaying the object's location and results of queries on a background map on the terminal's screen; and it is responsible for passing user input as query parameters to the moving object module. A moving object module issues update and query requests to the server, and passes query results back to the visualization module. The spatial network is organized in files and indexed by a composite structure that includes a compact R-tree and sequential indexes. The spatial data in use is maintained in a pool that is shared between the different modules. The data management on the clients is detailed in Section 3.

The important modules on the server are the storage and query modules. The former receives location reports from the moving objects and is responsible for storing this moving object information. The query module is responsible for processing queries issued by either the server itself or a moving object. Both modules access the moving objects via a network-link-based index, which also refers to road network data. The server-side data management is covered in Section 4.

3 Client-Side Data Management

Without loss of generality, we assume a client setting that resembles a GPS-enabled smartphone, such as a pocket PC. Such terminals are currently being sold with navigation software by, e.g., TomTom. (The HP iPAQ hw6915 is a concrete example.) Such terminals come with relatively limited flash storage and no disk. As the terminals may obtain power from the vehicle in which they are installed, we do not consider power issues.

This section first describes compact client-side data structures used for spatial networks, and then presents the client-side functionality that utilizes these structures.

3.1 Client-Side Index Structure

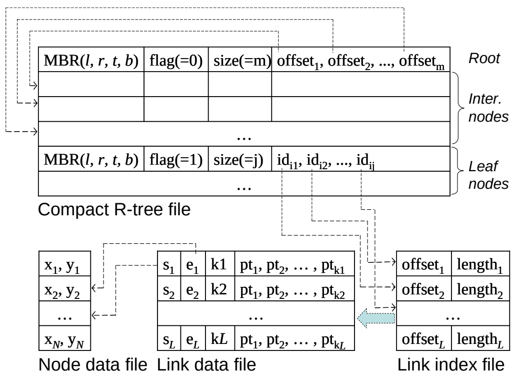


Fig. 4. Index structure on device

We store the nodes and links of a spatial network in two separate sequential data files. Due to the relatively limited storage available on a mobile terminal, it is not feasible to maintain all records from these files in main memory. For navigation purposes, we need to efficiently retrieve the data from the two files that relate to a region of interest. To enable this, we need a proper index structure. In particular, to support the range queries required for navigation and data display, we adapt the R-tree [10] into a compact structure suitable for a mobile environment. The client-side index structure, comprising the compact R-tree and the two sequential files, and an additional index file are illustrated in Figure 4 and explained next.

The sequential file of nodes stores the positions of the nodes, which are ordered according to their identifiers. Similarly, the sequential file of links stores for each link the two delimiting points of its polyline, as pointers to the node file; it stores the number of intermediate points; and it stores the coordinates of the intermediate points.

An R-tree is created on the set of polylines representing all the links in the spatial network. Each polyline is a unit to index. All non-leaf nodes are organized as in a standard R-tree. Each leaf node holds the identifiers of all the network links it covers, rather than pointers to the real link records in the link file. As link records are not of equal length, a dense sequential index file on the link file is used. To access the record of the link with identifier l_i, the ith record in the sequential index is retrieved. This record contains the offset and content length of the link record in the link file. By using a separate sequential index file, we retain the road link identifiers. This facilitates potential identifier-based random data access on the client side.

The R-tree is mapped to a sequential file in a specific way. All nodes are stored in a breadth-first order. Each non-leaf node is stored according to the format $\langle MBR, \mathit{flag}, \mathit{size}, \mathit{offsets} \rangle$, where MBR is the minimum bounding rectangle of a network link, flag indicates whether the node is a leaf node, size contains the number of child nodes, and $\mathit{offsets}$ are offsets into the file that point to the child nodes. The leaf-node format is similar, except that the field $\mathit{offsets}$ is replaced by a field ids that stores the integer identifiers of the network links covered by the leaf node; these identifiers correspond to row numbers in the sequential index on the link file. To conserve storage space, we allocate bits to the node fields in a compact way according to the ranges of parameters such as R-tree fanout, R-tree height, and the number of links in the road network.

3.2 Client-Side Navigation and Visualization

The navigation module receives so-called NMEA sentences with location data from the GPS receiver, retrieves pertinent link records via the data structures detailed in the previous section, and places the link records in a link pool that is used by the visualization module for display of part of the map containing the object's current location.

Initially, given the current location (x_c, y_c) of the object, the data within rectangle $(x_c - w, y_c + h, x_c + w, y_c - h)$ (left, top, right, bottom coordinates, as for bounding rectangles) is to be displayed on the terminal's screen, where w and h are configurable parameters. This rectangle is called the *active window*, and a window query against the data structure is used for retrieving the relevant data. Every link in the pool is of format $\langle id, MBR, \mathit{points} \rangle$, where id is the link's identifier, MBR is its minimum bounding rectangle, and points is the sequence of coordinates defining the link polyline.

When receiving a new position from the GPS receiver, the navigation module first checks whether the current location remains in the active window. If so, no new data retrieval is needed. Otherwise, a new active window win_{new} is computed, those currently pooled links that are outside win_{new} are discarded, and those links that intersect win_{new}, but are not in the pool, are retrieved via the data structure. This retrieval is achieved by a modified window query that uses two windows win_{new} and win_{old}. It returns the links that intersect with win_{new}, but do not intersect with win_{old}. Standard depth-first or best-first R-tree traversals can be adapted to process such modified window queries. Upon the retrieval, the active window is set to win_{new}. During the data retrieval, the network link and line segment on which the current location (x_c, y_c) resides are determined on the fly. We call this link (segment) the handset client's *host link* (*host segment*) with respect to its current location. If no new data is to be retrieved, the host link and segment are determined by the moving object module, covered next.

3.3 The Moving Object Module

The moving object module is responsible for ensuring that the server has up-to-date location information for the object. It continually receives location data from the navigation module, and maintains a record of the form $\langle loc_c, vel_c, t_c \rangle$ that captures the current location, velocity and the time of those for the object. It also maintains a record of the form $\langle lnk_h, seg_h \rangle$ that captures the host link and segment that correspond to the current location.

If the navigation module has not performed data retrieval for the most recent location, an update without a known host link and segment is sent to the moving object

```
Algorithm updateWithoutHosts($loc_n, vel_n, t_n$)
Input:  $loc_n$ is the new location
        $vel_n$ is the new velocity
        $t_n$ is the report time
    // Determine the host link and segment for $loc_n$
1.  if ($loc_n$ is still on $seg_h$)
2.      if ($|vel_n - vel_c| > \Delta v$)  // Marked velocity change
3.          Send UPT($oid, loc_n, lnk_h, t_n$) to the server;
4.      $loc_c = loc_n;\ vel_c = vel_n;\ t_c = t_n;$
5.  else
6.      for each segment $seg_i$ after/before $seg_h$ of link $lnk_h$
7.          if ($loc_n$ is on $seg_i$)
8.              Send UPT($oid, loc_n, lnk_h, t_n$) to the server;
9.              $loc_c = loc_n;\ vel_c = vel_n;\ seg_h = seg_i;\ t_c = t_n;$
10.             return;
11.     for each link $lnk_i \neq lnk_h$ in pool
12.         for each segment $seg_i$ of link $lnk_i$
13.             if ($loc_n$ is on $seg_i$)  // Moved to another link
14.                 Send UPT($oid, loc_n, lnk_h, lnk_i, t_n$) to the server;
15.                 $loc_c = loc_n;\ vel_c = vel_n;\ lnk_h = lnk_i;\ seg_h = seg_i;\ t_c = t_n;$
16.                 return;
```

Fig. 5. Update without known hosts

module. The relevant algorithm is shown in Figure 5. First, it is checked whether the new location is still on the currently recorded host segment. If so and if the velocity has changed markedly, an update message UPT is issued to the server and the records are updated (lines 2–4). A pre-specified threshold Δv of velocity change is used to judge whether an update to the server is necessary (line 2). For such an update involving the same host link, the UPT message includes the location, host link identifier and time. The server will use the host link identifier to efficiently locate the record corresponding to the moving object. If the object is not on the recorded host segment, the adjacent segments on the recorded host link are checked. The check is carried out sequentially in two directions one after the other: first from the segment after seg_h to the last one; then from the one before seg_h to the first one (line 6). When a new host segment is found for the location, an update message is sent to the server, the records are updated, and the algorithm terminates (lines 8–10). If the object is no longer on the recorded link, a new host link and segment are found by searching the link pool (lines 11–13). An update message is sent, the records are updated, and the algorithm stops (lines 14–16).

If the navigation module has just invoked a data retrieval for the new location, an update attached with the new host link and segment identifiers is sent to the moving object module. If the update is the first positioning report received, a registration message REG is sent to the server and the report is recorded. Otherwise, the process is similar to that of the previous algorithm, by distinguishing among three cases.

4 Server-Side Data Management

In this section, we first present how mobile objects are organized on the server side in accordance with the client design presented in Section 3. Then, we proceed to detail how concurrent continuous range monitoring queries are efficiently processed on the server side, which takes advantage of our specific system architecture.

4.1 Server-Side Mobile Object Management

Link-Based Moving Object Indexing On the server side, a hashing mechanism is used for indexing the moving objects according to their current network locations. The composite index structure is shown in Figure 6 and is explained next.

Recall that the L links in the spatial network are assigned integer identifiers from 1 to L. The top-level index is simply a sequential file with one entry for each link. With the same link index file as on the client side, a link record can be easily fetched given its identifier. Each link entry contains a pointer to a moving-object bucket, which keeps all objects currently moving on that link. The identifier of a link is also used

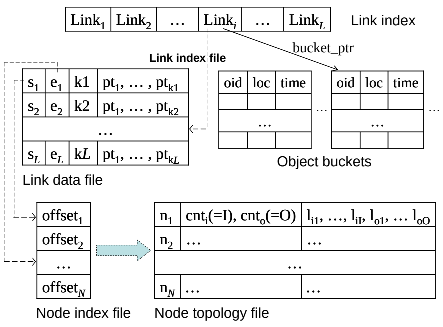


Fig. 6. Server-side index structure

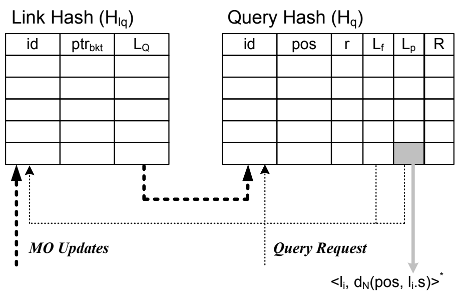


Fig. 7. Data structures and query execution

to locate the polyline associated with the link and the link's topology information. All links in polylines are organized in a sequential link file, as on the client side. Via a link index file, the link record is easy to fetch. Node topology information is organized in a separate file. Each node topology entry in this file contains the count of its incoming links, the count of its outgoing links, and the identifers of these two types of links sequentially. Link records and node topology information are associated via a node index file, which for each node stores its offset into the topology file. To speed up spatial operations needed in query processing, all network links are indexed by an R-tree, so do all network nodes. These R-trees are not illustrated in Figure 6. Different from previous indexes for disk-resident network constrained moving objects, ours does not employ hierachical R-trees [8, 16] or index trajectories of moving objects [2].

Each element of an object bucket has the format $\langle oid, loc, time \rangle$, where oid is an object identifier, loc is the most recently reported location, and $time$ is the time of the most recent report. To simplify the processing, we allocate oid's starting at 1. The hashing of a moving object works as follows. Given a moving object oid, its host link lnk is determined by its location loc. Identifier lnk is used to obtain the bucket pointer $bucket\_ptr$ in the corresponding link index entry. The object oid's record is kept in the object bucket pointed by $bucket\_ptr$. The hashing for an object is created or updated when the server receives an update report from that object, explained next.

Insertion An insertion occurs when a moving object issues a REG(oid, loc, lnk, $time$) message to the server. An insertion is quite straightforward. The field lnk is first used together with the index structure to locate the bucket into which the moving object should be inserted; then an object entry is created and inserted into that bucket.

Deletion An object may delete itself from the system. This can occur when a mobile terminal is switched off, e.g., the user's vehicle is parked. In this case, the terminal issues a delete message prior to powering down. A deletion is performed on the server when a DEL(oid, lnk) message is received from an object. Upon receiving this message, the server removes the relevant object from the database. Field lnk is used to locate the moving object bucket via the index structure, and then the bucket is scanned to remove the element in the bucket with identifier oid. As an option, by periodically checking object buckets we are able to identify objects that have been inactive for a long time, and delete them from the database.

Update Updates occur when the server received an UPT message from an object. As mentioned in Section 3.3, there are two types of UPT messages. For an UPT(oid, loc, lnk, $time$) message, which indicates that the object remains on the recorded host link, the object's entry is found, and the relevant record fields loc and $time$ are updated. In case of an UPT(oid, loc, lnk, lnk_n, $time$) message, which indicates that the object has moved out of the recorded host link, a deletion and an insertion are performed. Field lnk is used to locate the current bucket for the deletion, while lnk_n is used to find the new bucket for the insertion.

After each index operation is finished, the query maintenance function will be invoked to correctly adjust query results accordingly. This is discussed in Section 4.4.

Discussion The server-side index structure has several important properties. First, its overall performance is balanced. If a network link contains few moving objects, index maintenance is inexpensive. If the number of moving objects on a link is large, the query processing is less expensive. This balance between index maintenance and query processing costs is hard to achieve in moving object indexes [15]. Second, the index structure supports easy and high concurrency as it separates moving objects on different network links. Within each link, concurrency control on a hash table is much easier compared to that on any tree index. Third, the index structure efficiently supports queries based on network distance, which has not been addressed well in most previous work on the indexing of moving objects. We do not organize all moving objects in a global hash table, because a single large hash table is much more difficult to manage in terms of scalability and conflict rate, and it does not facilitate our query processing.

4.2 Solution Overview and Data Structures for Concurrent Queries

In a scenario with monitoring queries, multiple such queries will typically be active at a given point in time. We thus proceed to employ the *shared execution* [13, 18] philosophy to process concurrent queries. As the movements of all objects in $\mathcal{M}$ are constrained to the spatial network, only specific network links are relevant for each $CRMQ_N$ query. Further, the relevant links of different, concurrent $CRMQ_N$ queries may intersect. These observations provide the foundation for our shared query processing solution for $CRMQ_N$ queries.

Our solution identifies network links that are covered by multiple queries, organizes the relevant information together with the query results in two hash tables, and updates the hash table contents, including the query results, accordingly upon receiving moving object updates. The data structures and query execution are illustrated in Figure 7.

Given a query, those links that are relevant for the query, termed its *sensitive links*, are identified and classified into two categories: those completely within the query region, termed fully sensitive, and those that merely intersect with the query region, termed partially sensitive. The former links are kept in a list L_f, and the latter links are kept in a list in L_p. Then the sensitive links are searched for moving objects within the query region. For each fully sensitive link, the search simply retrieves all objects in its bucket. For each partially sensitive link, its moving-object bucket is searched to retrieve those objects whose network distance to the query point is within the query range.

We use a link-to-query hash table named H_{lq} that maps link identifiers to lists of queries for which the link is a sensitive link: for a link l_i the list L_{Qi} thus contains the queries for which l_i is sensitive. The query identifiers start from 1. For each query that has l_i as a fully sensitive link, its identifier is simply stored in list L_{Qi}. For each query that has l_i as a partially sensitive link, its identifier is stored in L_{Qi} with a negative sign. Additionally, a pointer to the bucket of all objects currently moving on l_i is stored in each element of H_{lq}.

All queries are stored in a hash table H_q that maps a query identifier q_i to an entry consisting of five relevant elements: the query point pos, the network query range r, the fully sensitive links L_f, the partially sensitive links L_p, and the current query result R. For each partially sensitive link l_i, L_p maintains a mapping of l_i to the network distance from the query point pos to the link's start or end node. If the link is partially covered by the query from its end node, the distance is attached with a negative sign. Otherwise the original distance value is used. This facilitates the network distance computation during query processing.

For continuous maintenance, the link identifiers enclosed in the update messages from the moving objects are used for accessing hash table H_{lq}. This way, the identifiers of all relevant queries, whose results may be affected by the update, are found. These identifiers are in turn used to explore hash table H_q, making it possible to update the results of multiple queries in a shared fashion. To uninstall a query q, the identifiers of its sensitive links, as stored in L_f and L_p in hash table H_q, are retrieved. These are then used for accessing hash table H_{lq} from where the query identifier stored in relevant link's query list L_Q is removed.

4.3 Query Initialization

The initialization of a single range monitoring query identifies all its sensitive links via incremental network expansion [17], and it searches these (mainly the partially sensitive ones) to determine the initial result. The algorithm, shown in Figure 8, does a network expansion from the query point pos and places all nodes within the network query range r in a priority queue Q. This Q gives priority to those nodes with shorter network distances to pos, and supports network expansion to identify all sensitive links. If the query point pos coincides with the position of a node, as determined by the function $isNode(pos)$ that searches the R-tree of all nodes, the node with a distance of 0 is pushed into queue Q (lines 2–3). If pos is not a node, its host link l_q is determined (line 5) by calling $find\_host(pos)$, which is implemented by searching the network link R-tree. Then l_q's two nodes are checked to determine whether they are within distance r of pos (lines 6–7). If both of them are within r, l_q is searched as a fully sensitive link and added to H_{lq} and $H_q(id).L_f$ (lines 8–9). Otherwise, l_q is searched as a partially sensitive link and is added to H_{lq} (line 11). In the link-to-query hash table H_{lq}, fully sensitive and partially sensitive links are distinguished by the sign attached to their query identifiers.

```
Algorithm monitorInit($pos, r, id$)
Input:  $pos$ is the query point
        $r$ is the network query range
        $id$ is the query identifier
Output: the initial result
1.  $R = \emptyset; Q = \emptyset;$
    // Determine starting links
2.  if $(isNode(pos))$
3.      $Q.enqueue(\langle node(pos), 0\rangle)$
4.  else
5.      $l_q = find\_host(pos);$
6.      if $(d_N(pos, l_q.s) < r)$  $Q.enqueue(\langle l_q.s, d_N(pos, l_q.s)\rangle);$
7.      if $(d_N(pos, l_q.e) < r)$  $Q.enqueue(\langle l_q.e, d_N(pos, l_q.e)\rangle);$
8.      if $((d_N(pos, l_q.s) \leq r)$ and $(d_N(pos, l_q.e) \leq r))$
9.          $R = R \cup l_q$'s bucket;  Add $id$ to $H_{lq}(l_q).L_Q;$  Add $l_q$ to $H_q(id).L_f;$
10.     else
11.         $R = R \cup search(l_q);$  Add $-id$ to $H_{lq}(l_q).L_Q;$  Add $\langle l_q, +/-d_N\rangle$ to $H_q(id).L_p;$
    // Expansion, search for sensitive links
12. while ($Q$ is not empty)
13.     $(n, d_N) = Q.pop();$
14.     foreach unvisited link $l_i$ connected to $n$
15.         if $(d_N + l_i.len > r)$  // A partially sensitive link
16.             $R = R \cup search(l_i);$  Add $-id$ to $H_{lq}(l_i).L_Q;$
                Add $\langle l_q, +/-(d_N + l_i.len)\rangle$ to $H_q(id).L_p;$
17.         else  // A fully sensitive link
18.             $R = R \cup l_i$'s bucket;  Add $id$ to $H_{lq}(l_i).L_Q;$  Add $l_q$ to $H_q(id).L_f;$
19.             if $(d_N + l_i.len < r)$  $Q.enqueue(\langle l_i.n \neq n, l_i.len\rangle);$
20. return $R$;
```

Fig. 8. Initialization of a $CRMQ_N$ query

A partially sensitive link l_q is also added to $H_q(id).L_p$ with a corresponding distance value (line 11). The distance value's sign is determined based on whether l_q's start node or end node is met.

Next, network expansion is repeated to identify all sensitive links until all unvisited nodes are too far away from pos (lines 12–19). For each fully sensitive link, all objects on it enter the initial result (line 18). Each partially sensitive link needs to be searched for the objects that are really covered by the query range (line 16). Similar to the case for l_q above (lines 8–11), relevant operations are carried out on hash tables H_{lq} and H_q.

4.4 Shared Concurrent-Query Maintenance

Shared maintenance of concurrent range monitoring queries occurs when the server receives a registration, deletion or update message from an object. The pseudo code for the shared maintenance mechanism is shown in Figure 9. The maintenance mechanism identifies all the relevant queries from H_{lq} by hashing given link identifiers. If the message from an object is a deletion (indicated by a negative oid in line 1), the object is removed from the results of all relevant queries by means of the link-to-query hash table (line 2). Upon receiving a first-time report, if link lnk is a fully sensitive link of some query ($i > 0$ in line 7) or the object is within the query range of pos on a partially sensitive link, the object is reported (line 8). For an update on the same link, any

```
Algorithm sharedMaintain($oid$, $loc$, $loc_n$, $lnk$, $lnk_n$)
Input:   $oid$ is the client object's identifier
         $loc$ is the client object's old position
         $loc_n$ is the client object's new position
         $lnk$ is the client object's old host link
         $lnk_n$ is the client object's new host link
1. if ($oid < 0$) // Object deletion
2.     foreach query identifier $i \in H_{lq}(lnk)$
3.         if (($i > 0$) or ($oid \in H_q(-i)$'s result)
4.             Remove $oid$ from $H_q(|i|)$'s result;
5. else if ($loc_n == null$) // First report
6.     foreach query identifier $i \in H_{lq}(lnk)$
7.         if (($i > 0$) or ($d_N(H_q(-i).pos, loc) \leq H_q(-i).r$))
8.             Add $oid$ to $H_q(|i|)$'s result;
9. else if ($lnk_n == null$) // Still on the old link
10.    foreach query identifier $i \in H_{lq}(lnk)$
11.        if ($i > 0$) continue;
12.        if ($oid \in H_q(-i)$'s result)
13.            if ($d_N(H_q(-i).pos, loc_n) > H_q(-i).r$)
14.                Remove $oid$ from $H_q(-i)$'s result;
15.        else if ($d_N(H_q(-i).pos, loc_n) \leq H_q(-i).r$)
16.            Add $oid$ to $H_q(-i)$'s result;
17. else // Link change
18.    foreach query identifier $i \in H_{lq}(lnk)$
19.        if (($i > 0$) or ($oid \in H_q(-i)$'s result)
20.            Remove $oid$ from $H_q(|i|)$'s result;
21.    foreach query identifier $i \in H_{lq}(lnk_n)$
22.        if ($i > 0$) // A fully sensitive link
23.            Add $oid$ to $H_q(i)$'s result;
24.        else if ($d_N(H_q(-i).pos, loc_n) \leq H_q(-i).r$)
25.            Add $oid$ to $H_q(-i)$'s result;
```

Fig. 9. Shared maintenance of concurrent queries

query having link lnk as a fully sensitive link is skipped (line 11) as no result change occurs, while any query having lnk as a partially sensitive link needs further checking (lines 12–16). For an update involving a link change, the object is removed from the results of current relevant queries if necessary (lines 18–20), and it is added to the results of new relevant queries if necessary (lines 21–25).

4.5 Support for Moving Queries

A monitoring query may change its position, e.g., because it is attached to a moving object. We term this a moving query. Naturally, such queries can be supported by terminating the old query and initiating the new query when such an update in the position of a query happens. This approach is already supported by the proposals presented thus far. However, we can do better. With our hash-based data structures available, we do not need to re-evaluate the new, or updated, query from scratch.

We apply a simplified variation of algorithm **monitorInit** (Figure 8). This algorithm uses network expansion to identify the updated fully (partially) sensitive link list L'_f (L'_p) with respect to the query's new query point without searching them. Then

```
**Algorithm queryUpdate**$(id, pos, L'_f, L'_p)$
**Input**:  $id$ is the query identifier
            $pos$ is the new location of query point
            $L'_f$ is the updated fully sensitive link list
            $L'_p$ is the updated partially sensitive link list
**Output**: the updated result
1. foreach link $l_i \in H_q(id).L_f$
2.     if $(l_i \notin L'_f)$
3.         Remove $l_i$ from $H_q(id).L_f$; Remove $l_i$'s objects from $H_q(id).R$;
4.     else Remove $l_i$ from $L'_f$;
5. foreach link $l_i \in L'_f$
6.     Add $l_i$ to $H_q(id).L_f$; Retrieve $l_i$'s objects to $H_q(id).R$;
7. foreach link $l_i \in H_q(id).L_p$
8.     if $(l_i \notin L'_p)$
9.         Remove $l_i$ from $H_q(id).L_p$; Remove $l_i$'s objects from $H_q(id).R$;
10. foreach link $l_i \in L'_p$
11.     Add $l_i$ to $H_q(id).L_p$; Search $l_i$ and add resultant objects to $H_q(id).R$;
12. return $R$;
```

Fig. 10. Support for moving queries

the update procedure shown in Figure 10 is invoked. Each expired fully sensitive link, together with those objects on it, is removed (lines 2–3); for any new fully sensitive link, objects on it are included in the result (lines 5–6). Each expired partially sensitive link, together with the objects on it, is removed (line 8–9). Each remaining/new partially sensitive link is searched, and the qualifying objects are included in the result (lines 10–11).

To support moving queries efficiently as detailed above, a minor change to the result (R) data structure in needed. It must consist of a hash table that maps link identifiers to the lists of objects on the links. This facilitates the necessary link-based objects removal in the algorithm above.

5 Empirical Performance Study

We first study the performance properties of the proposed server-side design and then consider the client side.

5.1 Server Side Experiments

Settings To obtain good server-side performance, we have proposed two strategies: that of shifting part of the server load to the clients by letting them report host links; and that of processing concurrent queries in a shared manner. These two strategies are orthogonal. Therefore, by varying each of them, we obtain four different system schemes, as listed in Table 1. In the horizontal dimension, clients report either their locations

	Client report	
Query execution	*Object Location*	*Host Link ID*
Isolated	IXY	IID
Shared	SXY	SID

Table 1. Different system schemes

Parameter	**Setting**
Total time stamps	1000
Moving object card.	1K, **2K**, ..., 10K
Max object vel.	10, 20, ..., **50**, ..., 100
Query sets card.	1K, **2K**, ..., 10K
Query range multiple	1, **2**, ..., 5

Table 2. Parameters used in experiments

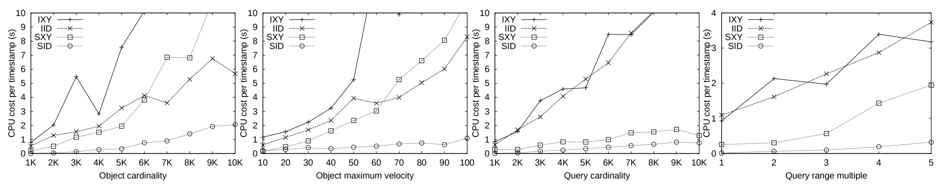
(XY) or their host links (ID). In the vertical dimension, the server processes concurrent queries either one by one in an isolated way (I) or in a shared manner (S). Consequently, IXY, IID, SXY, and SID represent four possible system schemes. In the *XY schemes, upon receiving a client report, the server invokes an additional procedure that identifies the host link via the link R-tree and link index before further processing. In the I* schemes, the link hashing table (H_{lq}) does not contain the list L_Q that contains the queries on a link. In the experiments, we compare these four system schemes to investigate the performance gains of the proposed strategies.

We used Brinkhoff's generator and the road network of Oldenburg [4], to generate network-based moving object workloads of 1K to 10K moving objects. Each workload is active for 1000 time stamps. At each time stamp, new objects come to be active, existing objects report to the server, and/or existing objects stop being active. The maximum object velocity is varied from 10 to 100 map units per time stamp. We generated static query sets of 1K to 10K queries. Each static query is produced as follows. First, a road network link is selected at random from all links. Then a position on that link is chosen at random as the query point (by choosing a line segment of that link at random and interpolating with a random ratio along that segment). Finally, the query range is decided as a multiple (randomly picked from 1, 2 to 5) of the length of the host link just selected. For the mobile query sets, we made the moving objects as query issuers, and determined the query ranges in the same way as for static queries. The parameters used in the experiments are listed in Table 2. The values in bold are the default settings used when their corresponding parameters are fixed. We implemented all system schemes in Java (JDK 1.4), and conducted all experiments on a Pentium IV desktop PC running MS Windows XP with a 3.00GHz CPU and 1GB RAM.

Experimental Results on Static Query Sets We consider two performance metrics: (1) the amortized CPU time spent on query processing per time stamp and (2) the average main memory consumption per time stamp. Each experiment ran for 1000 time stamps, with all queries active from start to finish.

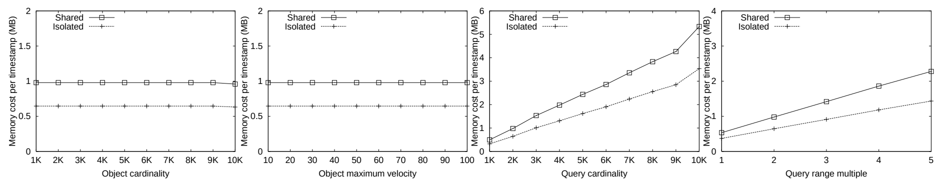
To understand the effect of object cardinality on CPU time, we varied the cardinality from 1K to 10K. The results are reported in Figure 11(a). SID always performs significantly better than other schemes, while IXY is always the worst. SID saves considerable CPU time in the processing of network distance based queries by not only executing relevant concurrent queries in a shared fashion, but also exploiting the clients' capabilities to determine host links. For object cardinalities up to 6K, SXY outperforms IID; the opposite is seen when the cardinality exceeds 6K. For small object sets, the shared execution of concurrent queries has the largest effect. However, for large object sets, the host link ID reporting strategy becomes crucial, because the server needs to process much more frequent update reports from the clients; thus, host link IDs in the reports save significant CPU time. As object cardinality increases the CPU time cost of SID grow slowly, which certainly demonstrates that it is scalable.

The results reported in Figure 11(b) show the effect on CPU time of the maximum object velocity, which we varied from 10 to 100 map units per time stamp. Not surprisingly, SID remains the best and most scalable scheme because it uses both efficient strategies. The relative performance for SXY and IID is alike to that for the previous experiment. As the objects move faster, their updates to the server become more frequent, which finally renders the host link ID reporting strategy crucial.



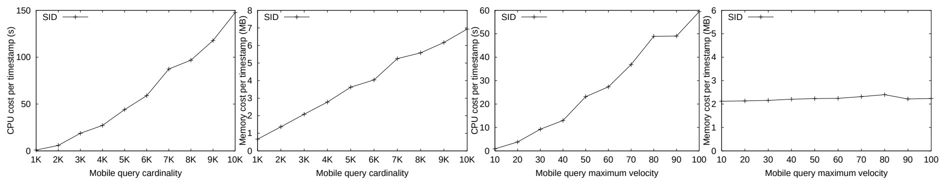
(a) Object cardinality (b) Object velocity (c) Query cardinality (d) Query range

Fig. 11. CPU times on static query sets



(a) Object cardinality (b) Object velocity (c) Query cardinality (d) Query range

Fig. 12. Memory costs on static query sets



(a) CPU vs cardinality (b) Mem. vs cardinality (c) CPU vs velocity (d) Mem. vs velocity

Fig. 13. Results on mobile query sets

To see the effect of query cardinality on CPU time, we varied it from 1K to 10K. The results are reported in Figure 11(c). Due to the lack of shared execution, IXY and IID degrade badly as more and more concurrent monitoring queries exist in the system. In contrast, SXY and SID both perform steadily because of the shared execution strategy. The slight improvements from 9K to 10K are due to more overlaps of sensitive links among different queries caused by the larger number of concurrent queries.

The results reported in Figure 11(d) show the effect on CPU time of the query range multiple, which we varied from 1 to 5. A larger query range means that more links are covered by queries. Shared execution helps alleviate the burden caused by additional sensitive links, as indicated by the better performance of SXY and SID. Nevertheless, SXY degrades much more than SID because the latter benefits substantially from host link ID reporting when many links are involved.

The experiments above also observed memory consumption, as shown in Figure 12. For memory cost, we only consider the data structures used to process the queries. As SXY (IXY) and SID (IID) use the same data structure on server side, we only compared the costs of the "Shared" and "Isolated" schemes. We see that the shared scheme consumes moderately more memory than the isolated scheme. This additional cost is due to the link-to-query mapping in H_{lq}. As the query cardinality increases, more entries are maintained in the query hash table (H_q). This leads to the memory cost increase in both schemes, as shown in Figure 12(c). As the query range increases, more links are maintained in the sensitive-link lists. This explains the observation in Figure 12(d) that the memory costs grow as query range increases. The (almost) linear growing patterns, together with the fact that neither object cardinality nor velocity impact the memory consumption, indicate that our techniques are scalable.

Experimental Results on Mobile Query Sets For this set of experiments, we let all moving objects send continuous range monitoring queries to the server with the query points being their current positions. Each moving object issues a continuous query when it sends a REG message to the server, and the query is updated when update reports are received by the server. When a DEL message is received, the object's mobile query is terminated. As SID is shown to be the most efficient scheme in Section 5.1, we only consider SID for the mobile query sets. Each experiment also ran for 1000 time stamps.

We first varied the mobile query cardinality, i.e., the number of moving objects, from 1K to 10K. The CPU costs are shown in Figure 13(a). As expected, the CPU cost is much higher compared to those obtained for static query sets—mobile queries invoke extra updates. Figure 13(b) shows that as the mobile query cardinality increases, the memory cost grows at a rate close to what is seen for static queries (Figure 12(c)). This is because mobile queries use the same query hash as do static queries, and almost need no extra memory space except the special mappings facilitating result removal.

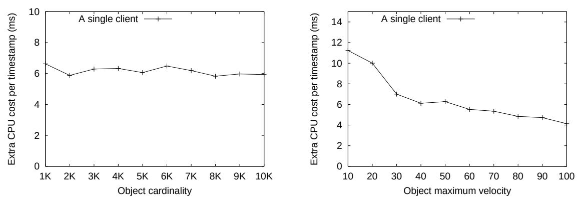
Next we fixed the mobile query cardinality at 2K and varied the query velocity from 10 to 100. Results are reported in Figure 13(c) and 13(d). Faster mobile queries lead to more frequent server-side computations, updating both the objects position information and the query results. This trend is consistent with the results shown in Figure 13(c). As seen in Figure 13(d), the memory cost does not change much as the query velocity changes; the memory cost is affected mainly by the query cardinality. We also varied the mobile query range multiple from 1 to 5 to observe its effect on the SID performance. Both CPU time and memory cost exhibit slight variation as the query range changes, and they are very close to their counterparts on the 2K mobile query set covered in Figure 13(a) and 13(b). Due to the space limitation we omit the graphs here.

5.2 Client Side Experiments

We implemented our client design using SuperWaba [1], a Java-based open-source platform for mobile terminal applications development. We conducted a set of experiments on an HP iPAQ h6365 pocket PC, running MS Windows Mobile 2003 with a 200MHz processor and 55MB user accessible SDRAM.

Our main concern is to determine how much extra CPU time is spent on our special client design that involves the reporting of host links, compared to a usual mobile handset client that only reports coordinate locations to the server. We used the moving-object workloads generated in Section 5.1. For each data set, our client program on the pocket PC processes all location updates for each network-constrained moving object. For each object, the extra CPU time is accumulated and finally amortized across all time stamps. We then report the average of all objects' amortized extra CPU time costs.

The results reported in Figure 14(a) are those gained for moving objects of 1K to 10K. It is seen that at each time stamp, the client's extra CPU time cost is close to 6 milliseconds, which is negligible compared to the gain we achieved by shifting the server load to the clients, according to the results reported in Figure 11(a). This demonstrates the benefit of our shifting strategy again. We also investigated the effect of the maximum object velocity on the client side CPU time. The results are reported in Figure 14(b). During a fixed active period, as an object moves faster, it produces more



(a) CPU vs object cardinality (b) CPU vs object velocity

Fig. 14. Additional CPU times on client side

location updates and tends to invoke more data retrieval because it moves out of the current display window. Consequently, the extra CPU time costs due to host link identification decreases as the navigation module in our design contributes more by triggering a simple update procedure with known hosts, described in Section 3.3. Therefore, the amortized cost over time stamps decreases in spite of more updates occurring. The observations provide evidence that our client design is effective in taking advantage of the navigation module, whose own costs increase as the object moves faster independently of whether our specific additional design is present or not.

6 Related Work

Recent years have seen some research work on spatial-network constrained moving objects. Vazirgiannis and Wolfson [20] proposed a variety of query types specific to the network-constrained moving objects, together with corresponding query processing algorithms. Shahabi et al. [19] employed an embedding technique to transform a road network to a higher dimensional space, where shortest paths between original network points are computed and used to help pruning in kNN search based on network distance. Jensen et al. [12] formalized the data model and problem definition for nearest neighbor queries in road networks, and adapted Dijkstra's algorithm [7] to compute the nearest neighbors for a mobile query point on the fly. Cho and Chung [6] considered continuous kNN queries aiming at static points of interest in a road network, whereas we query against mobile objects in this paper. Mouratidis et al. [14] addressed continuous kNN monitoring for spatial network constrained moving objects. The authors employed some specific tree structures to support the shared execution of multiple kNN monitoring queries, whereas we in this paper use simple yet efficient hashing tables for concurrent range monitoring queries. All these works [6, 12, 14, 19, 20] have assumed a central data server responsible for processing queries solely. Our work is significantly different in that we successfully exploit the computing capabilities of the mobile terminals to aid the server in the query processing.

The idea of shifting work from the server to the clients has been addressed in the context of monitoring free-moving objects. Gedik and Liu [9] proposed a distributed mobile system architecture, in which relevant monitoring query information is installed on mobile clients, enabling them to delay update reports to the server and process queries locally. Cai et al. [5] identified a resident domains for each mobile client, who in turn reports to the server only when it enters/leaves a resident domain so that queries can be updated correctly. Hu et al. [11] proposed to send mobile clients safe regions, within which continuous spatial monitoring results do not change. These techniques, however, are not applicable to our problem concerning network-constrained moving objects and network distance-based queries.

7 Conclusion

In this paper, we have proposed a full-fledged, novel solution to the continuous range monitoring of moving objects in a road network setting. The proposal takes advantage of the computing capabilities of mobile terminals to off-load computation from the server-side to the moving objects, and the server groups relevant concurrent queries together and then processes the queries in each group in a shared fashion. Extensive experiments with the prototype implementation capture the performance characteristics of the proposal, and suggest that the proposal is efficient and applicable in practice.

Acknowledgments

The work of Hua Lu, Zhiyong Huang and Linhao Xu was in part funded by A*STAR under grant no. 032 101 0026. Christian S. Jensen is also an adjunct professor at Agder University College, Norway.

References

1. Superwaba. http://www.superwaba.com, 2006.
2. V. T. de Almeida, R. H. Güting. Indexing the trajectories of moving objects in networks. *Geoinformatica*, 9(1): 33–60, 2005.
3. V. T. de Almeida, R. H. Güting. Using Dijkstra's algorithm to incrementally find the k-nearest neighbors in spatial network databases. In *Proc. ACM SAC*, pp. 58–62, 2006.
4. T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2): 153–180, 2002.
5. Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Proc. MDM*, pp. 27–38, 2004.
6. H.-J. Cho and C.-W. Chung. An efficient and scalable approach to CNN queries in a road network. In *Proc. VLDB*, pp. 865–876, 2005.
7. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1: 269–271, 1959.
8. E. Frentzos. Indexing objects moving on fixed networks. In *Proc. SSTD*, pp. 289–305, 2003.
9. B. Gedik and L. Liu. Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proc. EDBT*, pp. 67–87, 2004.
10. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*, pp. 47–57, 1984.
11. H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *Proc. ACM SIGMOD*, pp. 479–490, 2005.
12. C. S. Jensen, J. Kolář, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *Proc. ACM GIS*, pp. 1–8, 2003.
13. M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proc. ACM SIGMOD*, pp. 623–634, 2004.
14. K. Mouratidis, M. L. Yiu, D. Papadias, and N. Mamoulis. Continuous nearest neighbor monitoring in road networks. In *Proc. VLDB*, pp. 43–54, 2006.
15. B. C. Ooi, K.-L. Tan, and C. Yu. Frequent update and efficient retrieval: an oxymoron on moving object indexes? In *Proc. WISE Workshops*, pp. 3–12, 2002.
16. D. Pfoser, C. S. Jensen. Indexing of network constrained moving objects. In *Proc. ACM GIS*, pp. 25–32, 2003.
17. D. Papadias, J. Zhang, N. Mamoulis, and Y. Tao. Query processing in spatial network databases. In *Proc. VLDB*, pp. 802–813, 2003.
18. 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 Trans. Computers*, 51(10): 1124–1140, 2002.
19. C. Shahabi, M. R. Kolahdouzan, and M. Sharifzadeh. A road network embedding technique for k-nearest neighbor search in moving object databases. In *Proc. ACM GIS*, pp. 94–100, 2002.
20. M. Vazirgiannis and O. Wolfson. A spatiotemporal model and language for moving objects on road networks. In *Proc. SSTD*, pp. 20–35, 2001.