Available online at www.sciencedirect.com
ScienceDirect
ELSEVIER
Data & Knowledge Engineering 64 (2008) 77–100
DATA & KNOWLEDGE ENGINEERING
www.elsevier.com/locate/datak

Continuous range monitoring of mobile objects in road networks

Dragan Stojanovic [a,*], Apostolos N. Papadopoulos [b], Bratislav Predic [a], Slobodanka Djordjevic-Kajan [a], Alexandros Nanopoulos [b]

[a] *Department of Computer Science, University of Nis AleksandraMedvedeva 14, 18000 Nis, Serbia*
[b] *Department of Informatics, Aristotle University 54124, Thessaloniki, Greece*

Available online 12 September 2007

Abstract

In contrast to regular queries that are evaluated only once, a continuous query remains active over a period of time and has to be continuously evaluated to provide up to date answers. We propose a method for continuous range query processing for different types of queries, characterized by mobility of objects and/or queries which all follow paths in an underlying spatial network. The method assumes an available 2D indexing scheme for indexing spatial network data. An appropriately extended R∗-tree, that primarily is used as an indexing scheme for network segments, provides matching of queries and objects according to their locations on the network or their network routes. The method introduces an additional pre-refinement step which generates main-memory data structures to support efficient, incremental reevaluation of continuous range queries in periodically performed refinement steps.
© 2007 Elsevier B.V. All rights reserved.

Keywords: Mobile objects; Location based services; Continuous range queries; Query processing

1. Introduction

Advances in wireless communication technologies, mobile positioning and Internet-enabled mobile devices have given rise to a new class of mobile applications and services. Location based services (LBS) deliver geo-information and geo-processing services to the users according to their current location, or locations of the objects of their interests. Such services, like automatic vehicle location, fleet management, tourist services, transport management, traffic control and digital battlefield are all based on mobile objects and the management of their continuously changing location.

In several applications, the object motion is constrained by an underlying spatial network, i.e., objects can not move freely in space, and their position must satisfy the network constraints. Network connectivity is

* Corresponding author. Tel.: +381 18 529 235; fax: +381 18 588 399.
E-mail addresses: dragans@elfak.ni.ac.yu (D. Stojanovic), apostol@delab.csd.auth.gr (A.N. Papadopoulos), bpredic@elfak.ni.ac.yu (B. Predic), sdjordjevic@elfak.ni.ac.yu (S. Djordjevic-Kajan), alex@delab.csd.auth.gr (A. Nanopoulos).

0169-023X/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.datak.2007.06.021

usually modeled by a graph representation, comprising a set of nodes (intersections) and a set of edges (segments). Depending on the application, the graph may be *weighted* (a cost is assigned to each edge) and *directed* (each edge has an orientation).

Several types of location-dependent queries are significant in LBS, such as range queries, k-nearest neighbor (*k*-NN) queries, reverse neighbor queries, distance joins, closest pair queries and skyline queries. In this paper, we address the problem of processing continuous range queries over mobile objects, whose motion is constrained by a spatial network. The fundamental type of query for the purpose of monitoring and tracking mobile objects is the range query. The query range may represent a user selected area, a map window, a polygonal feature, a part of the road segment or an area specified by the distance from a reference point. The map window of the LBS client represents the simplest continuous range query that must be supported in a monitoring and tracking LBS application. Using such a query, up-to-date information about moving/static objects in user's surrounding is continuously represented in the map window. Thus, in contrast to regular queries that are evaluated only once, a continuous query remains active over a period of time. A major challenge for this problem is how to provide efficient processing of continuous queries with respect to CPU time, I/O time, main-memory utilization and network bandwidth.

The rest of the article is organized as follows. The next section presents related work in the area and describes our contributions. Section 3 describes in detail the proposed framework, analyzing the methodology AND describes the data structures needed for the query processing algorithms. Section 4 presents algorithms for different steps in continuous query processing methodology. Section 5 presents the results of the experimental evaluation of query processing algorithms. Finally, Section 6 concludes the paper and shortly outlines directions for the future work.

2. Related work and contribution

One of the challenges in LBS development is how to handle different types of queries in a mobile environment, where both queries and objects are mobile. The MOST data model proposed in Sistla et al. [27] identified three categories of such queries:

- Instantaneous, for which the query is evaluated immediately after the query issuance and the answer is transmitted to the user;
- Continuous, which need to be evaluated at every time instant in order to ensure the correctness and validity of the query answer;
- Persistent, which need to be evaluated at every time instant as well, but the query evaluation must include all previous mobile object database states starting from the time of query issuance (database history).

According to the mobility of query (client) and the objects queried by the clients, such queries can be further classified into three categories:

- Mobile objects querying static objects (e.g. tourist services, m-commerce);
- Static clients querying mobile objects (e.g. fleet management, traffic control and management);
- Mobile clients querying mobile objects (e.g. tourist services, digital battlefield, mobile games).

Continuous spatiotemporal query processing in a location-aware environment is an active area of research, resulting in the proposal of many query processing methods, techniques and indexing schemes. In Prabhakar et al. [25], velocity constrained indexing and query indexing (Q-index) has been proposed for efficient evaluation of static continuous range queries. According to the proposed method in-memory data structures and algorithms are developed and presented in Kalashnikov et al. [16]. The Q-index method assumes static range queries over mobile objects. The queries are indexed by an R-tree and mobile objects examine the index structure to find the queries in whose answer they may participate. By indexing queries not mobile objects, Q-index method avoids frequent updates of the index structure and thus expensive maintenance of this structure. In addition, it utilizes the concept of safe regions which oblige the mobile object to issue an update only if such mobile object leaves the region and thus influence the answer of some other queries.

The MQM method [3,4] focuses on static continuous range queries. It is based on partitioning the query space into rectangular sub-domains, and assignment the resident domain to each mobile object in the system. A mobile object is aware only of the range queries intersecting its resident domain, and reports its current location to the server only if it crosses the boundary of any of these queries and participates in the query answer. Thus, the part of the continuous query processing is deployed on the mobile client. When an object exits its resident region, it request and obtain the new one from the server. To determine the new resident domain for the object, the server uses a binary partitioning tree which maintains for each sub-division of the space the queries that intersect it.

Gedik and Liu [10] propose a method and a system for distributed query processing, called *Mobieyes*. *Mobieyes* ships some part of the query processing to the mobile clients while the server mainly acts as a mediator between mobile objects. The method tries to reduce the load on the server and save communication costs between mobile objects and the server. The workspace is partitioned using grid and the monitoring regions of the queries are maintained at the server. The monitoring region of a query is defined as the union of the grid cells it can potentially intersect. When mobile object falls in the monitoring region of the query, it receives the information about the query position and velocity, and notifies the server only when it enters or leaves the predicted query region. The mobile object stores locally the information about queries in whose answer it can participate and monitors its relationships with such queries through time. The queries, actually mobile objects with specified range about them, issue updates to the server when they change velocity vector, or when they move out of their current cell. In the paper [11] the authors propose a scheme called motion adaptive indexing (MAI) which enables optimization of continuous query evaluation according to the dynamic motion behavior of the objects. They use the concept of motion sensitive bounding boxes (MSB) to model and index both moving objects and moving queries. This enables decreasing of the number of updates performed on the indexes, end such a scheme is independent of the underlying index structures, i.e. any spatial index method, such as an R-tree, can be used to index MSBs.

Wang et al. [29] propose a system leveraging the computing capacities of mobile devices for continuous range query processing. In their proposal, continuous range queries are mainly processed on the mobile device side, which is able to achieve real-time updates with minimum server load. They introduce a distributed server infrastructure to partition the entire service region into a set of service zones and cooperatively handle requests of continuous range queries. This feature improves the robustness and flexibility of the system by adapting to a time-varying set of servers. Second, they propose a novel query indexing structure, which records the difference of the query distribution on a grid model. This approach significantly reduces the size and complexity of the index so that in-memory indexing can be achieved on mobile objects with constrained memory size.

The concept of mobile continuous queries is considered in Lazaridis et al. [19]. They define a dynamic query as a temporally ordered set of snapshot queries. They specify the index structures for trajectories of moving objects and describe their efficient usage for evaluation of dynamic queries that represent predictable or non-predictable movement of an observer. Such problem definition and solution is in accordance with application scenario and aims to support rendering of objects in virtual tour-like applications (Table 1).

Mokbel et al. [21] present SINA, a server-side method based on shared execution and incremental evaluation of continuous queries. Shared execution is achieved by implementing query evaluation as a spatial join between the mobile objects and the queries. Incremental evaluation means that the query processing system produce only the positive or negative updates of the previously reported answer, not the complete answer for every evaluation of the query. Both the object and query indexes are implemented as disk-based regular grids. The method consists of three phases, namely hashing, invalidation and joining phase performed on in-memory and disk-based index structure. It does not take into account the current location of a mobile object according to its motion vector (predicted motion), i.e. the objects change their location in a stepwise manner.

Hu et al. [12] propose a generic framework for monitoring continuous spatial queries over moving objects, both range and k-NN queries. Besides reducing the evaluation cost of continuous query processing, they also address the location update mechanism which is aware of currently running queries. Each moving object is aware of its safe region, the rectangular area computed at the server in such a way that the answers of all queries remain valid as long as all objects reside in their safe regions. The moving object issues location updates to the server only when it moves out of the safe region. The database server receives this update, reevaluates all

Table 1
Notation for analyzing the cost of pre-refinement step in query processing methodology

Symbol	Meaning
MOi	Number of MO in *olist* of the SR*-tree leaf node entry in whose *qlist* is CQi
AQi	Number of MO in answer set of CQi after the pre-refinement step
FQi	Number of MO in final answer after the refinement setp
p	$0 < p \leqslant 1$, the selection characteristics of the pre-refinement step ($p = AQi/MOi$)
r	Probablity of MO to issue continuous query (to be the reference object of the query) ($0 \leqslant r \leqslant 1$)
Tp	Time needed for the pre-refinement step
Tu	Time needed for update of pre-refinement data structures upon receiving location update from MO
Te^1	Time needed for incremental evaluation based on pre-refinement data structures
Te^2	Time needed for incremental evaluation without pre-refinement data structures
Tr	Time needed for the refinement step when using the pre-refinement step
Tr'	Time needed for the refinement step without the pre-refinement step
Tinc	Time needed for finding the incremental answer
LoS	Time needed for calculation of the new MO location along the segment based on previous location and speed
Up	Time needed for update of answer period given by the expressions in the previous section
Inside1D	Time needed to determine whether the 1D point is within 1D interval
Inside2D	Time needed to determine whether the 2D point is within 2D range
FindIncAnswer	Time needed to search if MO belongs or not to the previous query answer in order to be included in the incremental answer (depends on the size of previous query answer)
Nu	Number of updates of all mobile objects that are part of the temporal query answer between two successive evaluations

affected queries and computes a new safe region for the object. The framework maintains two index structures at the server; the object index that stores the current safe regions of all the objects in the form of on-disk R-tree, and the query index that stores the quarantine area of the query in the form of in-memory grid index. The quarantine area specifies that the current answer is unchanged as long as all answer objects stay inside it and all non-answer objects stay outside it. Based on those structures, detailed algorithms for range/k-NN query evaluation/reevaluation and safe region computation are provided.

To allow for incremental query (re)evaluation, various kinds of virtual constructs (VCs) have been proposed for building query indexes, such as virtual construct rectangles (VCRs) [31]. The VCR-based query indexing was main-memory based and was shown to outperform other query indexing approaches, such as the cell-based [16], and the covering tile-based [32] approaches, in terms of total query (re)evaluation time. It uses a set of predefined VCRs to decompose query regions. Search operations are conducted indirectly via VCRs. However, many of the VCRs are redundant, unnecessarily slowing down the index search time and the query (re)evaluation time. Wu et al. [33] introduce a new query indexing method, called CES-based indexing, for incremental processing of continuous range queries over moving objects. The method is based on a set of predefined containment-encoded squares (CES), which represent virtual constructs used to decompose query regions and are used to store indirectly precomputed search answers. CES-based indexing aims to further minimize the total query (re)evaluation time so that the accuracy of the query answers can be additionally improved via more frequent (re)evaluations. The set of defined set in a query space is smaller than corresponding set of VCRs, which significantly shorten the search time. Containment relationships among the CESs and containment encodings make index search operations very efficient. These VC-based approaches neglect the dynamic nature of mobile objects which continuously move between successive updates and follow the paths in an underlying network.

Illari et al. [13] propose an interesting solution to distributed processing of continuous location-dependent queries in mobile environments using mobile agents. The agents are in charge of tracking the location of interesting mobile objects and refreshing the answer to a query efficiently. They present a scalable distributed architecture as a completely decentralized solution for processing of continuous mobile queries without engaging the mobile client devices and argue for usefulness and feasibility of their approach.

Papadias et al. [24] describe a framework to support query processing in spatial network. They take advantage of location and network connectivity to efficiently prune the search space. They show the application of the framework to the most popular spatial queries, namely range, nearest neighbors, closest pairs and e-distance joins, in the context of static queries and objects positioned on the spatial network.

Several reported work study processing of range queries over past trajectories of mobile objects moving in networks and are related to our work since they use the extension of the R-tree to represent network segments and the trajectories of mobile objects over those segments. Two index structures for indexing the past trajectories of mobile objects in networks have been proposed. The Fixed Network R-Tree (FNR-Tree) [9] consists of a top level 2D R-Tree, whose leaf nodes contains pointers to 1D R-Trees. The 2D R-Tree is used to index the edges of the network, and for every leaf node of the 2D R-Tree, there is a 1D R-Tree indexing the time-interval of each object's movement inside the line segments of the network. The main disadvantages of this approach is very high number of entries and lots of updates in the index structures, as well as the limitation that an object cannot end or change its movement within the network edge, but only at nodes. The other index structure is proposed by Almeida and Gueting in De Almeida and Guting [6] and is called Moving Objects in Networks Tree (MON-Tree). They describe two network models that can be indexed by the MON-Tree. The first model is edge oriented and represents the network by edges and nodes, and the second one is route oriented, represents the network by routes and junctions and is more suitable for road networks. The MON-Tree was experimentally evaluated against FNR-Tree and it showed better performance in answering range queries on complete moving object trajectories. In their tests, the MON-Tree indexing the route oriented network model showed the overall best results.

There is also a lot of work on continuous k-nearest neighbor (*k*-NN) queries. Some of the methods, such as SINA [21,12] provide extensions of range query processing to processing of nearest neighbor queries as well, and some new methodologies for *k*-NN query processing are developed independently [34,36]. Yoo and Shekhar [35] propose the methods and algorithms that based on given a static dataset and a query path, retrieve the object that causes the shortest detour if included in the path. Jensen et al. [14] propose the road network model suitable for processing of *k*-NN queries over the network constrained mobile objects and present a prototype system for such query processing. Recent work of Mouratidis et al. studies *k*-NN monitoring in road networks where the distance between a query and a data object is determined by the length of the shortest path connecting them [23]. Although their research is focused on continuous *k*-NN queries, they propose similar main-memory index and data structures to support query processing. They propose three main-memory index structures: the spatial index on the network edges (PMR quad tree), the edge table, which maintains network and data object information, and the query table, which maintains information about the queries. They propose two methods employing these data structures, the first one that maintains the query answers by processing only updates that may invalidate the current NN sets, while the second one follows the shared execution paradigm to reduce processing time. Their data structures for representing and indexing spatial network and object and queries that continuously move along the edges of the network looks very similar to ours. The network representation of the road network is a simple graph (nodes are connected by straight line segments and the degree of nodes can be 2 as well). For the second method, group monitoring algorithm (GMA) they introduce sequence (segments in our proposal) as polyline path between two nodes/intersections. But they also neglect the real dynamics of the environment where objects and queries change their location according to current motion parameters (speed, direction, route, etc.) without explicit updating of the mobile object database and employ of stepwise motion of the objects (up to 20% of them move change their location between two successive timestamps).

In this work, we propose a framework for continuous range query processing for objects moving on network paths. The framework introduces the methodology, the data structures and the query processing algorithms for processing continuous range queries over mobile objects, when queries may be both static and mobile. Similar to most continuous monitoring algorithms and methods reviewed in this section, we assume main-memory query evaluation. Our methodology is based on an extension of main-memory R∗-tree index that indexes network data, named SR∗-tree (Segment R∗-tree) and memory-resident data structures that support query processing and maintaining up-to-date query answers. The SR∗-tree is also used for map matching purpose. The methodology introduces an additional step in traditional spatiotemporal query processing strategy (filter refinement), namely, the pre-refinement step. The filter step selects the candidate objects according to fulfillment of the spatial query condition using the SR∗-tree index on the spatial network. The pre-refinement step is performed after the filter step, to further refine the mobile objects obtained by the filter step and to build the main-memory data structures to support periodical and incremental refinement steps. The pre-refinement step aims to surpass the shortcomings of the query processing methods and algorithms reviewed so far

that neglect the real nature of the movement that between two successive location updates objects move at certain speed. Those methods mostly assume the stepwise motion of the mobile objects [27]. Thus, at time instant t_c, $t_i < t_c < t_{i+1}$, where t_i and t_{i+1} are time instants of location updates, the location of an object is on $(t_c - t_i) \cdot v_c$ distance along the route in the network from the location determined at t_i.

The benefits of the pre-refinement step are in generating and maintaining temporal answers of continuous queries which are only updated upon receiving update of the motion parameters (speed, segment change) of the mobile objects. The answer generated by the pre-refinement step is represented by the set of pairs (*mo*, *tp*), indicating that the mobile object *mo* is the answer of the continuous query from time *tp.begin* until the time *tp.end*. The refinement step is performed periodically by processing in-memory data structures generated by the pre-refinement step and generates the incremental query answer.

3. Proposed methodology and data structures

The methodology for processing continuous range queries in a mobile environment is developed as a part of ARGONAUT, a service platform for mobile object data management [26]. We base our approach on the application scenario appropriate in LBS for monitoring and tracking mobile objects. In this scenario, users have wireless devices (e.g., mobile phones or PDAs) that are online via some form of wireless communication network. We assume that users can obtain their positions using global positioning system (GPS) technology. A setting is assumed in which a central database at the LBS server stores a representation of each mobile object's current position. Each mobile object stores locally its position assumed by the server. Then, an object updates the database whenever the deviation between its actual position (as obtained from a GPS device) and the local copy of the position that the server assumes exceeds the uncertainty threshold. The mobile objects move along the paths in an underlying spatial network. In order to perform tracking with as few updates as possible, the LBS server matches the position received from the mobile object to the network segment, by a map matching technique. Map matching is a technique that positions an object on a network segment, at some distance from the start of that segment, based on location information from a GPS device. Knowing the mobile object's speed and the time of the position update, the server determines the current position of the object till the next intersection (node) assuming that it moves at a constant speed. Reduction of updates reduces communication between clients and the server, as well as server-side update processing. When a mobile object reaches the end of his current segment (the intersection) and enters the new segment on its route the server must determine its new segment. The possible approach is that the server assumes that the predicted position remains at the end of the segment until the mobile object issues an update when reach the uncertainty threshold [5]. Another approach requires storage capacities on the mobile object to store road network data (as in current AVL systems) and obligation of mobile object to send an update also when it changes the network segment. The possible approach proposed in Ding and Guting [7] requires extension of the road infrastructure where in every intersection there are a group of installed sensors. Whenever a moving object transfers from one segment to another via intersection it gets notification from the sensors which will trigger the location update.

The ARGONAUT methodology employs an incremental continuous query evaluation paradigm [21]. Thus the server reports to the clients only the changes of the answer from the last evaluation time of their continuous queries. This significantly saves the network bandwidth by limiting the amount of transmitted data to the updates of the answer only rather than the whole query answer. Two types of updates are distinguished: positive updates and negative updates. The positive/negative update indicates that a certain object needs to be added/removed to/from the query answer.

The methodology employs and maintains two representations of the network data in two corresponding main-memory data structures. These structures support the filter step of the query processing algorithm and the map matching procedure. The first representation organizes network segments according to Euclidean distance in a main-memory R*-tree index structure [1], that stores MBR (Minimal Bounding Rectangle) representation of segments belonging to a spatial network. This structure is named SR*-tree (Segment R*-tree). The graph representation of a spatial network is maintained by the Network Connectivity Table (NCT), a data structure that stores information about the connectivity of network segments. Both network representations are interconnected, i.e. there is a reference from SR*-tree segment representation to the corresponding NCT segment representation and vise versa.

Since both objects and queries move on a spatial network, their spatial properties (location or partial route) are indexed within the same index structure. The leaf nodes of the SR∗-tree is appropriately modified to enable indexing of both mobile objects and queries. The leaf node entry of the general R∗ tree has the form *(o, mbr)*, where *o* is either a unique identifier of the spatial object or an address of the spatial object in the main-memory/secondary storage, and *mbr* is its minimal bounding rectangle. The leaf node entry of our extended SR∗ tree is represented as a quadruple (*segid, mbr, olist, qlist*), where *olist* and *qlist* is the list of identifiers of objects and queries, respectively moving along the segment identified by *segid*. The SR∗-tree structure enables efficient map matching in order to find the network segment and position on that segment from the start of the segment according to the location (*X*, *Y* coordinates) received from the mobile object.

The connectivity graph of the spatial network is maintained in main memory by the Network Connectivity Table (NCT) which stores information about the connectivity of network segments. A NCT entry is described as (*segid, rtEntry, segLength, startCon, endCon*), where *rtEntry* is the pointer to the SR∗-tree leaf node entry which represents that network segment (*segid*) whose length is given by *segLength*, and provide access to the lists of objects and queries that reside or move along the segment. The elements *startCon* and *endCon* are pointers to the lists of records described as (*segid, dir*), where *segid* represents the identifier of the segment connected to the start/end node and *dir* is the direction of that segment in the connection. The NCT is indexed on the *segid* attribute. Any main-memory index structure such as hash index or B-tree index can be used for the implementation of the NCT. The NCT is maintained to improve the updating of the index structure when mobile object/query issues a location update after changing its network segment. The NCT also improves map matching performed at the server in order to find the appropriate network segment according to the current location of a mobile object. Also, the connectivity structure maintained in NCT is more effective for LBS queries (range, *k*-NN, etc.) over objects on the spatial network, where the distance between two objects is not determined by the Euclidean metric, but by means of a network-based metric.

The insertion of the object into the index structure and the generation of the object list (*olist*) attached to the leaf node entries are performed according to the following rules:

- A static object is inserted in the *olist* attached to the leaf node entry of the SR∗-tree index that corresponds to the network segment at which such object resides.
- A mobile object is inserted in the *olist* attached to the leaf node entry of the SR∗-tree index that correspond to the network segment along which the object currently moves (the segment is the known route of the mobile object).

The insertion of the query in the same SR∗-tree index is performed according to following rules:

- A static range query is inserted in each *qlist* attached to the leaf node entry of the SR∗-tree index if its range overlaps the MBR of the corresponding network segment.
- A mobile range query is inserted in each *qlist* attached to the leaf node entry of the SR∗-tree index if its known route overlaps with the MBR of the network segment. The known route of the mobile query is defined by MBR of the network segment along which its reference object moves, extended by the query range.

The SR∗-tree index structure performs the matching between mobile objects and mobile/static queries according to their spatial relations and fulfillment of the spatial condition of a query. It enables the calculation of the initial answer set of the continuous query (filter set). Since, the index structure maintains only the spatial properties of objects and queries, it is not selective enough, and the initial answer set contains false positive answers. The methodology thus introduces an additional step in query processing scheme, the pre-refinement step, that creates additional main-memory data structures and support subsequent incremental refinement steps. The pre-refinement step refines the initial query answer set with regard to temporal information of objects' and queries' motion, as well as their exact geometries. The pre-refinement step creates the data structures in main memory to support incremental refinement steps.

In addition to the previous data structures two additional main-memory data structures are defined. The continuous query table (CQT) is used to store information about continuous range queries. A CQT entry

is of the form (*qid*, *oid*, *range*, *validPeriod*, *answerSet*) and stores information regarding continuous queries. The table is indexed on the *qid* attribute which is the unique query identifier. The *oid* is the identifier of the reference object of the query, static or mobile, and *range* defines the shape of the spatial query range around the reference query object, or the distance along the network that represents the network range. The *validPeriod* represents the period for which the query is valid. The *answerSet* is the initial query answer obtained by the filter step with additional, temporal information about satisfaction of a query condition. The initial answer is a list of elements of type *CQAnswer* defined as (*moid*, *aperiod*, *status*), where *moid* is mobile object identifier which belongs/will belong to the answer of the query during the period *aperiod*, while its current status in the query answer is described by the *status* attribute. The values of the status attribute are INIT_ ANSWER, NEW_ANSWER, OLD_ANSWER and NO_ANSWER. In the simplified case, when the period is a single time period, the mobile object, during its motion and/or query motion, change all status values sequentially. Thus, an object has INIT_ANSWER status when it will be the answer of the query in some period(s) in the future. The status NEW_ANSWER is associated to an object, when it becomes the new member of the query answer. The status OLD_ANSWER is associated to an object when it is the member of the current query answer, as well as was a member of the query answer in the previous evaluation of the query. An object has status NO_ANSWER when it is not a member of the current query answer, nor will be in the future, but was the member of previous query answers. Due to time progress, every mobile object candidate for the query answer changes its status in a strict sequence [INIT_ANSWER → NEW_ANSWER → OLD_ ANSWER]n→ NO_ANSWER, where [...]n indicates that the part of the status sequence can be repeated if the answering period of an object is a multi period (Fig. 1).

For each mobile object in the system, an in-memory Mobile Object Table (MOT) is created and maintained. The mobile object entry is described as (*moid, loc, time, speed, querySet*), where *moid* is the unique mobile object identifier, *loc* is the last received location, *time* is the time instant of the location update and *speed* is the last received speed, which has positive or negative value depending on the mobile object's direction along the segment (from the start to the end intersection or vice versa). The *querySet* attribute represents the list of queries in which such object participates, either in a query answer, or as a reference object of a query. Each query in this list is represented by an element of type *CQReference* which contains two attributes: *qid*, the query identifier and *ans*, the reference to the appropriate *CQAnswer* element of this query maintained for the mobile object. If a mobile object is the reference object of a query, this reference is set to NULL value.

To support fast and efficient updates of leaf node entries (object and query lists), which do not require changes in the SR∗-tree non-leaf node entries and the structure of the SR∗-tree, we extend two auxiliary main-memory index structures according to proposals of LUR-tree [17] and bottom-up update of R-tree [20]. For each mobile object in MOT there is a pointer to the corresponding entry in the leaf node of SR∗-tree (*mind*), which represent the current segment of the mobile object. This MOT extension enables fast access to the *olist(s)* of the corresponding leaf node entry(-ies). Analogously, the CQT is extended by a *qind* pointer to the list of SR∗-tree leaf node entries that store that query in corresponding *qlists*. Fig. 2 illustrates the structure of the SR∗-tree and the main-memory data structures mobile for objects and continuous queries.

Mobile objects in a real-world setting very often move following a particular route, which represents a path in road networks (Brinkhoff, 2003; [15]). Most humans do not move around aimlessly, but move towards a known destination choosing the best/fast/shortest paths depending on the cost criteria (time, distance, etc.). Therefore, LBS server can predict correctly the route on which a service user travels as an ordered set of

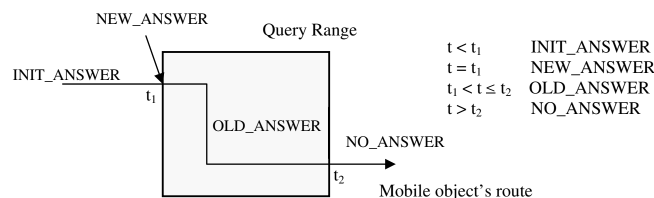

Fig. 1. Changing the status of a mobile object in a continuous query answer.

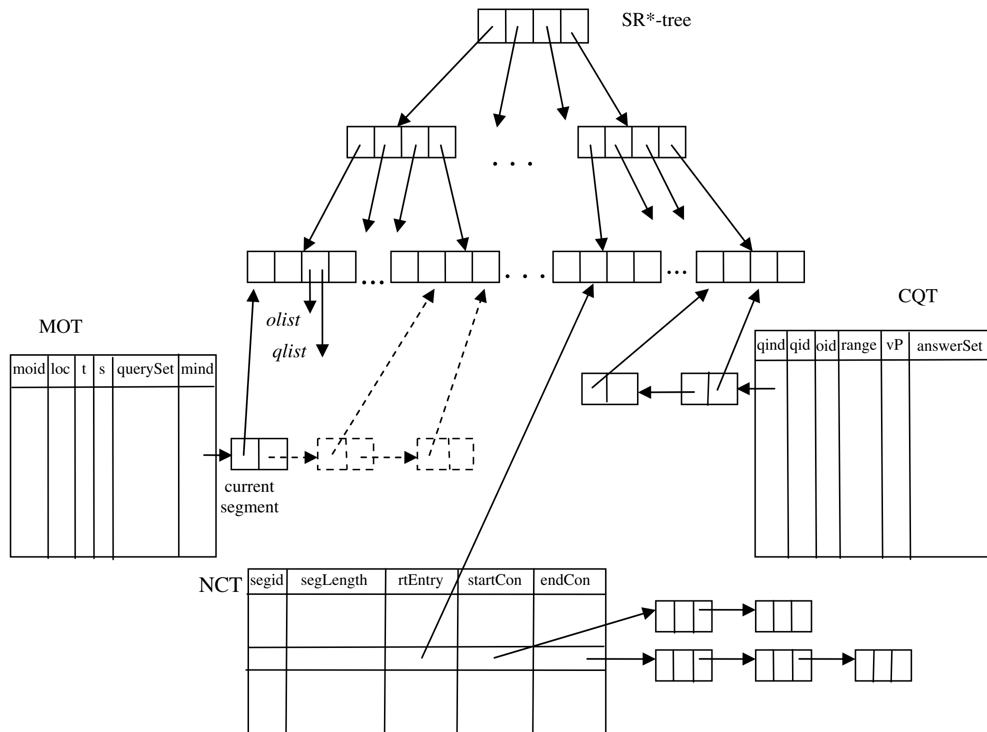

Fig. 2. Structure of SR∗-tree, as well as MOT and CQT auxiliary structures.

segments from the starting position to the destination. Using the correct route in place of a network segment reduces the number of updates needed to maintain a user's position within uncertain threshold as well as maintains the correct answer (valid period of answer) of continuous queries. Updates occur only because of speed changes. When a user changes its future route, because of sudden circumstances on that route (traffic accident, traffic jam, etc.) the server needs to calculate the new route for the user towards the same destination. If the partial or complete route of the mobile object is known in advance, the last element of the MOT entry is transformed in the head of the list of SR∗-tree leaf node entries that represent road segments as the known future route of the mobile object. In that case, the first element in the list contains the pointer to the leaf node entry of the current segment, and the other contains pointers to the leaf node entries which represent the known future route of the mobile object.

4. Continuous range monitoring algorithms

In this section we describe the algorithms for continuous range monitoring, which include the algorithms for the filter step, the pre-refinement step, and the refine step. The continuous range monitoring generates the incremental query answer. Also the algorithm for maintenance of temporal query answer generated by the pre-refinement step upon location/speed/segment updates of mobile objects/queries is given in Section 4.1. Section 4.2 gives numerical analysis of the cost of the pre-refinement step and estimates the threshold for which this step is justified. This section discusses the highly dynamic situation, where the size of incremental query answer can be at most two times larger than the full answer.

4.1. Algorithms

The insertion of the mobile object into the SR∗-tree index structure and the generation of the object list attached to the leaf node entries are performed according to the following algorithm (Fig. 3).

Algorithm Insertion of mobile object in SR*-tree
Input: *MO* - the new mobile object

1. Search SR*-tree on *MO.loc* and find SR*-tree leaf node entries
2. **for each** *rtEntry* **in** found SR*-tree leaf node entries
3. Apply ***Map matching*** to find *rtEntry.segid* for *MO.loc*
4. **end for**
5. *rtEntry.olist*.Add(*MO*)

Fig. 3. Insertion of mobile object in SR*-tree.

The insertion of the query in the same SR*-tree index is performed according to the algorithm in Fig. 4, where range is represented as rectangular area, and in Fig. 5 for range represented as distance along the network from reference object.

The algorithms for creating these tables are slightly different for the cases of static and mobile continuous queries. The input argument of the algorithms is the set of mobile objects that represent the potential query answers obtained by the filter step, by the examination of the SR*-tree index. The pre-refinement algorithm is performed on exact spatial and temporal geometries of mobile objects and mobile/static queries and creates necessary data structures. The algorithm for static queries over mobile objects is given in Fig. 6.

After updating the CQT and MOT tables (lines 2–4), the algorithm examines the spatial relation *intersects* on a mobile objects route (the geometry of the current network segment or the predicted route) and the static query range (line 8). This step enables removing false spatial positives obtained by the filter step. False positives are mobile objects whose MBR of the current segment overlaps with the query range, but their actual route does not intersect the query range. For those objects satisfying the spatial relation *intersects*, the time period (multi time period) in which the mobile object is (was, will be) *within* the query range is calculated, based on current motion parameters (speed, route) of the mobile object (line 9). The operators for spatiotemporal geometric calculations and topological relations in space+time (such as *Intersects, Within, When*, etc.) are developed and integrated in the ARGONAUT mobile object data management framework [28]. For objects whose answer period ends somewhere in the future, the new *CQAnswer* and *CQReference* elements

Algorithm Insertion of a query in SR*-tree
Input: *CQ (o - reference object, range)*

1. **if** *CQ.o* is a static object **then**
2. Search SR*-tree on *CQ.range* around *o.loc* and find SR*-tree leaf node entries
3. **for each** *rtEntry* **in** found SR*-tree leaf node entries
4. *rtEntry.qlist*.Add(*CQ*)
5. **end for**
6. **else**
7. Search SR*-tree on *o.loc* and find SR*-tree leaf node entries
8. **for each** *rtEntry* **in** found SR*-tree leaf node entries
9. Apply ***Map matching*** to find *rtEntry.segid* for *o.loc*
10. **end for**
11. Construct *extendedMBR* from *rtEntry.mbr* and *cq.range*
12. Search SR*-tree on *etxendedMBR* and find SR*-tree leaf node entries
13. **for each** *rtEntry* **in** found SR*-tree leaf node entries
14. *rtEntry.qlist*.Add(*CQ*)
15. **end for**
16. **end if**

Fig. 4. Insertion of static/mobile query in SR*-tree according to range in Euclidean space.

```
Algorithm Insertion of a query in SR*-tree
Input: CQ (o - reference object, distance)

1.  Search SR*-tree on CQ.o.loc and find SR*-tree leaf nodes
2.  for each rtEntry in SR*-tree leaf node
3.      Apply Map matching to find rtEntry.segid with CQ.o.loc
4.  end for
5.  Get NCT entry for rtEntry.segid
6.  if CQ.o is a static object then
7.      Using NCT entries, segLength, startCon and endCon obtain the list of segments (NCT entries) that are
8.      within network distance from CQ.o.loc
9.  else
10.     Using NCT entries, segLength, startCon and endCon obtain the list of segments (NCT entries) that are
11.     within network distance from startIntersection and endIntersection from rtEntry.segid
12. end if
13. for each ncte in selected NCT entries
14.     ncte.rtEntry.qlist.Add(CQ)
15. end for
```

Fig. 5. Insertion of static/mobile query in SR∗-tree according to range in a network.

```
Algorithm Pre-refinement step for static queries over mobile objects
Input: filterSet: a list of MO obtained by accessing the SR*-tree

1.  Add new entry SMquery in CQT
2.  for each MO in filterSet
3.    if not exist MO in MOT with such MO.moid then
4.        Add new MOT entry for MO
5.    else
6.        Find MOT entry with MO.moid
7.    end if
8.    if MO.route() Intersects SMquery .range then
9.        ap ∈TimePeriod and ap ← When (MO.loc Within SMquery.range)
10.       if ap.end > currentTime then
11            cqanswer ← new CQAnswer(MO.moid, ap, INIT_ANSWER);
12.       SMquery.answerSet.Add(cqanswer);
13.       cqref ← new CQReference(SMquery.qid, cqanswer);
14.       MO.querySet.Add(cqref);
15.       end if
16.   end if
17.end for
```

Fig. 6. Pre-refinement step for static query-mobile objects.

are created and added to the lists of corresponding CQT and MOT entries (lines 11–14). Such condition selects the mobile objects which currently are, or will be, the part of the query answer.

The pre-refinement step in processing of mobile query over mobile objects is slightly different, because the relationship between two moving objects is not linear in time. Thus it is impossible to exactly determine the answer period for each mobile object as potential answer of the mobile query. Therefore, for mobile queries, the spatial condition given in line 8 of the previous algorithm is changed. Instead of examining the query range in lines 8 and 9, the query route is examined, which is determined as a Minkowski sum of the query's network segment and defined range.

The periodic, incremental evaluation of continuous range queries is performed by scanning and examining the CQT by refinement step. The refinement step is performed periodically and evaluates the temporal query

condition. It determines the incremental answer in regard to the previous evaluation (refinement step), which is sent to the mobile client issuing a continuous query. The incremental answer represents the set of *IncAnswer* elements containing the *moid* of the object and the Boolean attribute indicating that the object becomes the part of the answer of the query (*true* value), or that it is not the answer any longer (*false* value). The incremental answer can be location- aware [13] if the user is interested not only in the set of objects that satisfy the constraints in the query, but also in their current geographic location (for instance, to locate such objects on a map). In that case the incremental answer contains location and other useful attributes of the mobile objects included in the answer. The refinement step is performed for all continuous queries active in the system, according to the algorithm shown in Fig. 7.

If the answering period contains current time, the mobile object is either the positive update of the query answer (lines 6–9) and thus the part of the incremental answer, or is already included in some previous query answer, and thus is not included in the new incremental answer (line 10). If a mobile object enters the query range several times during its motion (the answer period is a set of time periods), when one period from the set is expired, the objects status is INIT_ANSWER again, until the beginning of the next time period (lines 13 and 14). If the answer period of an object expires, the negative query answer update is generated and the corresponding element is removed from the query answer (lines 16–19). The refinement step for the case of mobile queries over mobile objects must include an additional test of the spatial condition on exact mobile object and query location, for those objects that already satisfy the temporal condition. Thus, the line 4 of the algorithm in Fig. 7 introduces the changed condition and becomes:

4. **if** *cqanswer.period* **contains** *currentTime* and *MO.currentLoc*() **within** *CQ.currentRange*() **then**

The functions *currentLoc*() and *currentRange*() calculate the location/range of the mobile object/query at the current time, given the last received location, time and speed of the object/query reference object, as well as its current route (network segment). As mentioned previously, a mobile object moves on its network segment with the last reported speed. When its predicted location (obtained by the *currentLoc*() function) differs

Algorithm Refinement step for static queries over mobile objects
Output: *incAnswerSet,* a set of *IncAnswer* objects

```
1. for each CQ in CQT
2.   if CQ.validPeriod Contains currentTime then
3.     for each cqanswer in CQ.answerSet
4.       if cqanswer.aperiod Contains currentTime then
5.         if cqanswer.status = INIT_ANSWER then
6.           cqanswer.status ← NEW_ANSWER;
7.           ia ← new IncAnswer (cqanswer.moid, true)
8.           incAnswerSet.Add(ia);
9.         else
10.          cqanswer.status ← OLD_ANSWER
11.        end if
12.      else if cqanswer.status = OLD_ANSWER then
13.        if cqanswer.aperiod is a TimePeriodSet and has_ periods_in_the_ future then
14.          cqanswer.status ← INIT_ANSWER
15.        else
16.          cqanswer.status ← NO_ANSWER
17.          ia ← new IncAnswer(cqanswer.moid, false)
18.          incANSWERSet.Add(ia)
19.          remove cqanswer from CQ.answerSet
20.        end if
21.      end if
22.    end for
23.  end if
24. end for
```

Fig. 7. Refinement step for the set of static continuous queries over mobile objects.

from its exact location by the specified threshold, the object must send location, time and speed updates. Upon receiving updates, the server must determine if the mobile object changes its network segment using map matching techniques. If the mobile client is augmented with road network data or network infrastructure is extended with specific sensors, the mobile object itself could determine the change of the network segment and report it to the server (as mentioned at the beginning of the section).

The location update for each mobile object requires scanning and updating both tables, MOT and CQT (Fig. 8). If the mobile object remains on its network segment, the update algorithm scans the pre-refinement

Algorithm Location/speed/network segment update of a mobile object
Input: *newloc, newspeed, newtime* of the mobile object MO

```
1.  Update MOT entry for MO.moid;
2.  MO.loc←newloc;
3.  MO.speed←newspeed;
4.  MO.time←newtime;
5.  Search SR*-tree on MO.loc and find SR*-tree leaf nodes
6.  for each rtEntry in SR*-tree leaf node
7.      Apply Map matching to find rtEntry.segid for MO.loc
8.  end for
9.  if MO not change network segment according to entry MO.moid in MOindex then
10.     for each qr in MO.querySet
11.         cq ←CQ with qr.qid
12.         if qr.ans != NULL then
13.             Update(cq.answerSet, qr.ans, MO)
14.         else
15.             for each res in cq.answerSet
16.                 Update(cq.answerSet, ALL, MO)
17.             end for
18.         end if
19.     end for
20. else  // MO changes the network segment
21.     Remove the first element in MOIndex entry list associated to MO.moid which points to the old_rtEntry
22.     old_rtEntry.olist.Remove(MO)
23.     if there is a predefined route then
24.         Check if MO follows its route
25.     else
26.         Add the pointer to the new_rtEntry of the new segment in MO.mind list associated to MO.moid
27.     end if
28.     new_rtEntry.olist.Add(MO)
29.     for each CQ in old_rtEntry.qlist \ new_rtEntry.qlist
30.         CQ.answerSet.Remove(MO)
31.     end for
32      for each CQ in new_rtEntry.qlist
33.         if CQ exists in MO.querySet then
34.             Update corresponding CQAnswer in CQ.answerSet
35.         else
36.             Add new CQAnswer to CQ.answerSet and new CQReference to MO.querySet
37.         end if
38.     end for
39.     for each CQ with MO as a reference object
40          for each rtEntry pointed by CQ.qind list elements for CQ.qid
41.             Remove CQ from the rtEntry.qlist
42          end for
43.         Insert CQ in SR*-tree (Algorithms on Figure 4 or 5)
44.     end for
45. end if
```

Fig. 8. Location/speed/network segment update of a mobile object.

data structures and updates the corresponding MOT entry, as well as all CQT entries of the affected queries and their answers using the simple expressions (lines 10–19). If the mobile object leaves its network segment and starts moving along a new one, the pre-refinement data structures (*querySets, answerSets*) related to the mobile object and the new set of queries obtained by the new SR∗-tree *qlist*, should be updated or new entries should be added according to the algorithms for the pre-refinement steps depicted in Fig. 6(lines 29–38). If the mobile object is also the reference object of the mobile query, the new SR∗-tree *olists* must be examined and the corresponding *answerSet* and *querySet* should be updated (lines 39–44).

The update of the answer period of an object (*aperiod*) issuing location and speed updates at a certain time instance is based on the threshold value *ut*, previous speed v_o, the new speed v_n, and the time t_n of the new update. The *update* function (lines 13 and 16 of Fig. 8) updates the answer period *aperiod* of a mobile object (or every time period in the set of time periods) according to the following expressions:

$$aperiod.start = \frac{v_o(aperiod.start - t_n) \pm ut}{v_n} + t_n \quad aperiod.end = \frac{v_o(aperiod.end - t_n) \pm ut}{v_n} + t_n$$

The uncertainty threshold value *ut* in this formula is added if the mobile object is advanced in regard to its predicted location, and it is subtracted if it is late in regard to its predicted location. Thus, the system provides an up to date and accurate answer for every continuous range query it maintains, according to location/speed updates of mobile objects.

4.2. Analysis of algorithms

We now analyze the cost of pre-refinement step in terms of CPU time needed to maintain continuous query answer up-to-date for static queries over mobile objects. The notation used in this section is summarized in Table 1. We compare the time needed to perform incremental query evaluation using the pre-refinement step with time for continuous range query processing without this step, using only SR∗-tree and NCT. The proposed methodology includes the times needed for the pre-refinement step, for the update of temporal query answer upon change of motion parameters of mobile objects within the same segment and for periodic incremental evaluation. When the methodology does not use the pre-refinement step, to perform periodic query evaluation we need the time for calculation of mobile object locations and query ranges at the time of evaluation, as well calculation of spatial predicate inside for point and rectangular geometric object. The analysis gives the number of updates per network segment that justifies the inclusion of pre-refinement step in the query processing methodology. We do not analyse the cost of changing network segment upon receiving mobile object's update, since both evaluation strategies perform the same actions in SR∗-tree when such change occurs.

The costs of the continuous query evaluation with the pre-refinement step and without the pre-refinement step are as follows:

$$Te^1 = Tp + Nu \cdot Tu + Tr \qquad Te^2 = Tr\prime + Tinc \tag{1}$$

We are interested in the value of Nu that satisfies $Te^1 < Te^2$. We simplify the expression for Te^1 and assume that all mobile objects that are the part of the CQ temporal answer set simultaneously update their location/speed. We take that the average probability that a mobile object issue continuous query is between 0 and 1. The cost of the pre-refinement step is mostly influenced by the determination whether the route of the mobile object intersects the query range and determination of the time stamps between which the mobile objects is/will be within the query range.

$$\begin{aligned} &\underbrace{f_1(MO_i, LoS, Inside2D)}_{Tp} + \underbrace{Nu \cdot f_2(MO_i, p, Up, r)}_{Tu} + f_3(M\underbrace{O_i, p, Inside1D)}_{Tr} \\ &\quad < \underbrace{g_1(MO_i, LoS, Inside2D)}_{Tr} + \underbrace{g_2(FQi, MOi)}_{Tinc} \end{aligned} \tag{2}$$

This analytical expression shows that for particular query and defined settings: the number of mobile objects obtained in the filter step, the number of mobile objects obtained in the pre-refinement step (the selectivity of pre-refinement), the characteristics of the underlying network (long polyline segments, short straight

segments, etc.), the query range, there is a threshold in *Nu* value which gives the benefits when applying the pre-refinement step. The number of updates between successive evaluations is directly proportional to the duration of evaluation period and agility of mobile objects/queries (the frequency of changing motion parameters between successive evaluations). We will experimentally determine the threshold for the period between successive evaluations for various parameter settings in Section 5.2.

For highly dynamic situation where especially both objects and queries are mobile, and the longer period between successive continuous query evaluations, the size of incremental query answer can be larger than the full answer. Let $Answer(t_i)$ be the answer of continuous query CQ in t_i, and $Answer(t_{i+1})$ be the answer of continuous query generated at t_{i+1}. Then, incremental query answer $IAnswer(t_{i+1})$ at t_{i+1} is represented as:

$$IAnswer(t_{i+1}) = (Answer(t_i) \setminus Answer(t_{i+1}), 0) \cup (Answer(t_{i+1}) \setminus Answer(t_i), 1), \tag{3}$$

where 0 stands for objects that those objects are removed from the query answer, and 1 defines objects that must be added to the query answer. Depending on speed profile of mobile objects and eventually queries and characteristics of the network for duration of time between successive evaluation $t = t_{i+1} - t_i$ the size of incremental answer is greater than the size of the full answer, $IAnswer(t_{i+1}) > Answer(t_{i+1})$ (we do not take into account the status bit of mobile objects in the incremental query answer). In highly dynamic environment and long periods between successive continuous query evaluation the answers of two successive evaluations can be completely different, i.e. $Answer(t_{i+1}) \cap Answer(t_i) = \varnothing$. In that case $IAnswer(t_{i+1}) = Answer(t_{i+1}) \cup Answer(t_i)$, For uniform distribution of objects and queries, $Size(Answer(t_{i+1})) \approx Size(Answer(t_i))$, and then $Size(IAnswer(t_{i+1})) \approx 2 \cdot Size(Answer(t_{i+1}))$. Thus, the size of the incremental answer at certain time stamp can be equal to the double size of the full answer. In the experimental section we perform experiments to determine the duration of period between successive evaluations for which incremental evaluation is not justified.

5. Experimental evaluation

In this section, we present the results of some experiments to analyze the performance of our continuous query processing methodology.

5.1. Experimental setup

We used the Network-based Generator of Moving Objects [2] to generate a set of 10,000 mobile objects and 1000 mobile queries. The input to the generator is the road map of Oldenburg (a city in Germany). The road network consists of 2873 intersections and 3803 segments. The output of the generator is a set of moving objects that move on the road network of the given city with probability of 5% to report its location update at each time stamp. Thus, every mobile object changes its motion parameters within 20 seconds period. We choose some objects randomly and consider them as reference objects and centers of rectangle range queries. All the experiments have been conducted on an Intel Pentium IV CPU 3.0 GHz with 512 MB RAM running Windows XP Professional. The page size is set to 4 KB. We have implemented the SR*-tree based on the original implementation of R*-tree [1]. ARGONAUT continuous query processing algorithms and main-memory data structures have been implemented using Microsoft Visual Studio C++ and the STL library. Our performance measures are: (i) the CPU time required for the pre-refinement step and the creation of in-memory data structures, (ii) the CPU time for the refinement step and the generation of incremental answers and (iii) the CPU time for updating the main-memory data structures upon receiving location/speed/network segment update of a mobile object. Since the performance for access and update of SR*-tree which index network segments highly depends on size and characteristics of the underlying spatial network, we do not present the results of those experiments.

We start our performance consideration using the set of objects produced by the filter step, which are the candidates for the final answer according to their spatial characteristics, i.e., location and range for the static queries and routes for mobile objects/queries. We perform the experiments for 10,000 mobile objects and 1000 mobile (static) queries and measure the average CPU time (in milliseconds) necessary for the pre-refinement step of a continuous query and the generation of the necessary data structures in main-memory for refinement steps which are performed periodically and produce incremental query answers. The average number of

mobile objects per query obtained in the filter step for different query ranges is given in Fig. 9. We vary the query range from 0.001 to 0.2 of each dimension of the data space.

5.2. Experimental results

We perform experiments to obtain the performance of the filter step in the query processing methodology. In our experimental setup, inserting 10,000 mobile objects in SR∗-tree requires 0.34 seconds. Insertion of the continuous queries in the SR∗-tree depends on query range and the query type, i.e. whether the query is static or mobile. The performance of the filter step for 1000 continuous queries, both static and mobile is presented in Fig. 10, when query range varies between 0.001 and 0.2 of the data space. The difference in performance of the filter step for the static and mobile queries is caused by the larger MBR of the mobile queries which is represented either as the extension of the MBR of the current segment by the actual query range, as described in Algorithm on Fig. 4, or by distance from the start and end intersections of the current segment, as described in Algorithm in Fig. 5.

The experiments on the performance of the pre-refinement step show that it requires a very small amount of CPU time (milliseconds) (Fig. 11). More specifically, for 10,000 mobile objects and 1000 continuous range queries, the pre-refinement step needs approximately 1.8 milliseconds for queries whose query range represents 0.1 of the data space.

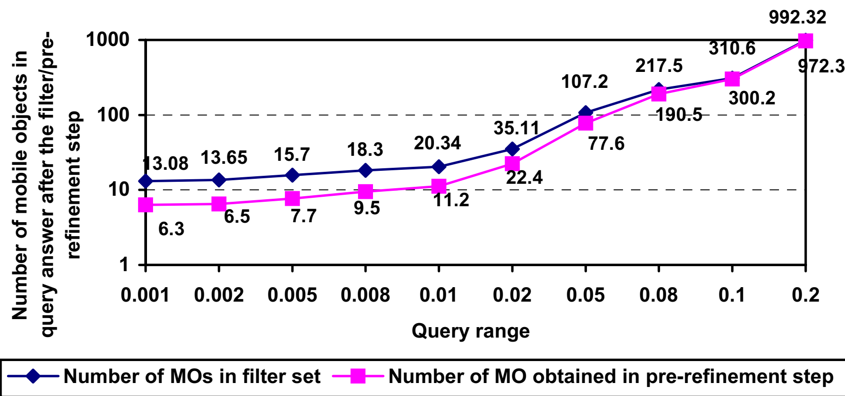

Fig. 9. The average number of mobile objects obtained in the filter step and pre-refinement step for 10,000 MO/1000 static continuous queries.

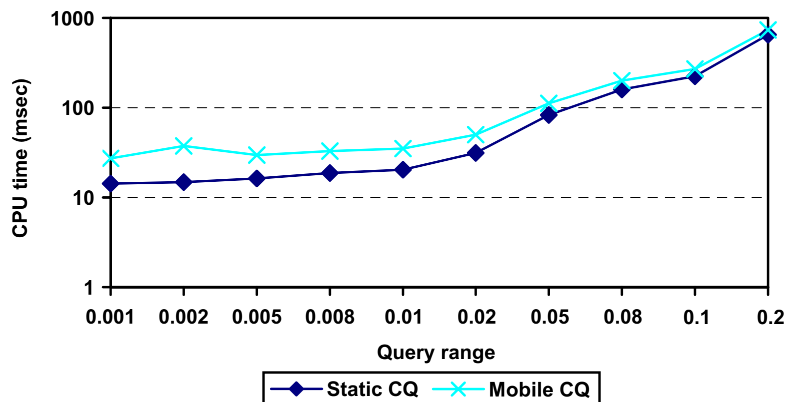

Fig. 10. The CPU time required for the filter step for 1000 static/mobile continuous queries.

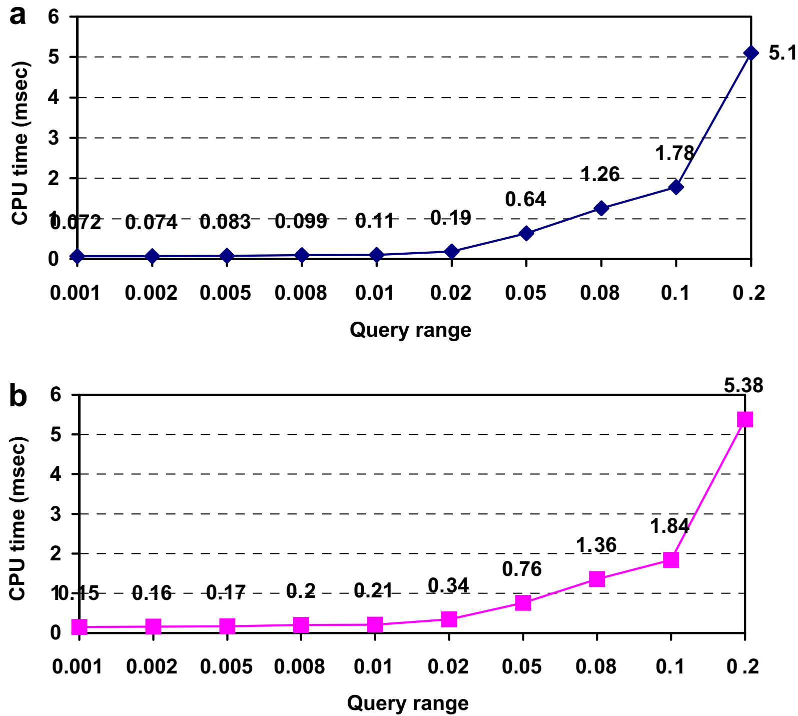

Fig. 11. The CPU time for the pre-refinement step per query for 10,000 MO/1000 (a) static CQ (b) mobile CQ.

During the refinement step an access to the main-memory data structures and the examination of the temporal part of the query condition are performed. The refinement step is performed periodically, i.e., once every T seconds. For each object in the answer of the continuous query, the refinement step should select those objects that constitute the current answer of the query as well as to generate the incremental answer in regard to the previous refinement step and evaluation of the query. We examine the duration of the refinement step for 1000 continuous queries over 10,000 objects, using different query ranges. Fig. 12 depicts the CPU time required for the incremental evaluation of 1000 static and mobile continuous queries over mobile objects depending on query range, according to the algorithm shown in Fig. 7 and its extension for mobile queries.

The average number of mobile objects in the query answer is given in Fig. 13 for different query ranges. The filter characteristics of SR∗-tree and pre-refinement data structures, can be determined as the size of the real answer divided by the size of the average filter answer (depicted in Fig. 9). For our experimental settings the filter characteristics of SR∗-tree take values from 0.12 for query range 0.001 of the data space, to 0.9 for the query range 0.2 of the data space.

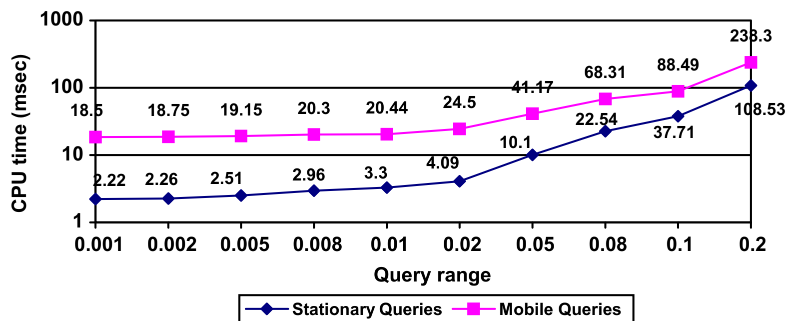

Fig. 12. The CPU time needed for refinement of 1000 mobile/static queries over 10,000 mobile objects.

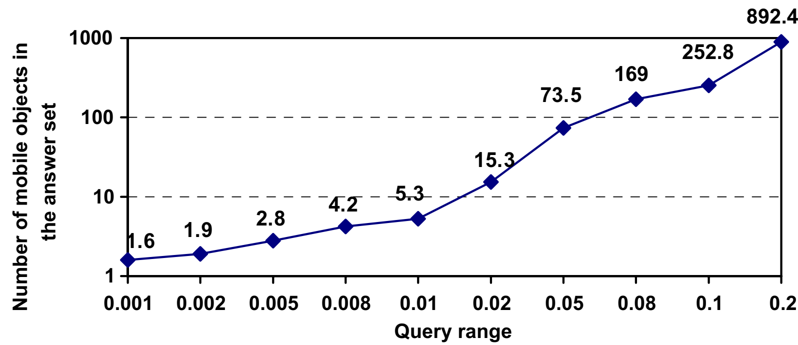

Fig. 13. The average number of mobile objects obtained in the refinement step for 10,000 MO/1000 static continuous queries.

When the mobile object (query) issues a location/speed/segment update, the main-memory data structures should be updated appropriately, to reflect the new location and the motion parameters of the mobile object/query, as well as its new route (network segment). According to the algorithm shown in Fig. 8, a mobile object should update the answering periods for all continuous queries in which it participates, according to its new location and new reported speed. When a mobile object is the reference object of a query (queries) the answering periods of all answers in the set should be updated accordingly. The CPU time per mobile object required for updating the main-memory data structures, under the load of 10,000 mobile objects and 1000 continuous queries is presented in Fig. 14. The CPU time shown represents the average time per object needed for the update of MOT and CQT data structures that preserves the consistency of the temporal query answer sets. If a mobile object changes its underlying network segment, it is necessary to update also the SR∗-tree, according to the filter algorithm. In that case it is necessary to perform the filter and the pre-refinement step for each such object and generate the new *querySets* and *answerSets* elements of the MOT and CQT entries, respectively. Thus, this experiment shows the cost of update of main-memory data structures SR∗-tree, MOT and CQT when the mobile object move to another network segment.

To prove the benefits of the pre-refinement step in the continuous query processing methodology the set of experiments were performed to prove the numerical analysis in Section 4.2. We compare the CPU time for incremental query evaluation without the pre-refinement step, and the CPU time for query evaluation which include the pre-refinement step, necessary updates of pre-refinement data structures upon location/speed update of mobile objects and the refinement step based on pre-refinement data structures. We do not take into account the CPU time needed for changing network segment upon receiving mobile object's update, since both evaluation strategies perform the same actions in SR∗-tree when such change occurs. We set the period

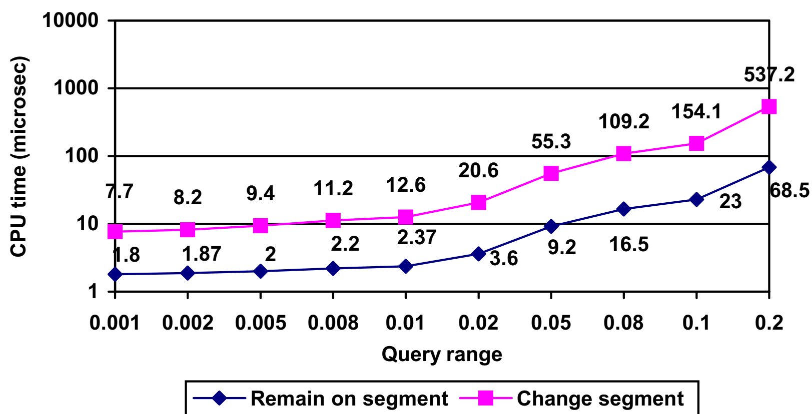

Fig. 14. The average CPU time per mobile object needed for data structures update upon location/speed/segment update.

between two successive evaluations at 10 time stamps (seconds), and vary the agility of mobile objects in range 5%, 10% and 20%. The agility of 5% mobile objects per time stamp means that within evaluation period 50% of mobile objects change their motion parameters (location/speed). Analogously, for 10% and 20% agilities, all mobile objects change the motion parameters once or twice, respectively between two successive evaluations. To compare the performance of our data structures SR∗-tree and NCT in continuous query evaluation, we implement the hash-based approach in the form of basic 40×40 main-memory grid according to Kwon et al. [18]. The performance results are given in Fig. 15.

The experiments show that our methodology and algorithms based on main-memory SR∗-tree and other data structures, both with and without the pre-refinement step, outperforms the simple hash-based approach using grid. The reason for this behavior is worse selectivity features of grid for objects in the road networks in regard to SR∗-tree, and thus is expected. The experiments also show that there is a threshold for the benefits of the pre-refinement step. For smaller query range the query evaluation with pre-refinement step needs less CPU time then the query evaluation without pre-refinement step for all agilities. In our experimental settings and according to Fig. 9, the filter characteristic of the pre-refinement step, the parameter p in Section 4.2, vary from 0.5 from query range 0.001 of data space to almost 1 for the maximum query range. Because we assume 10,000 mobile objects and 1000 continuous queries, the parameter r defined also in Section 4.2 is set to 0.1. For larger query ranges and greater mobile objects agilities the CPU cost of updating the large answer set generated by the pre-refinement step surpass the CPU costs of maintaining incremental query answer for query evaluation without pre-refinement and thus degrades the overall performance of query evaluation with

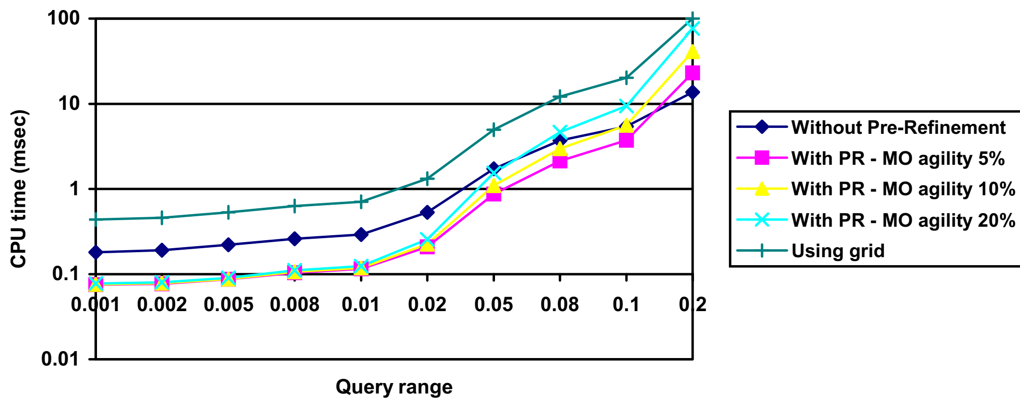

Fig. 15. The average CPU time for evaluation of continuous query with and without the pre-refinement step for different MO agilities and the evaluation period of 10 seconds.

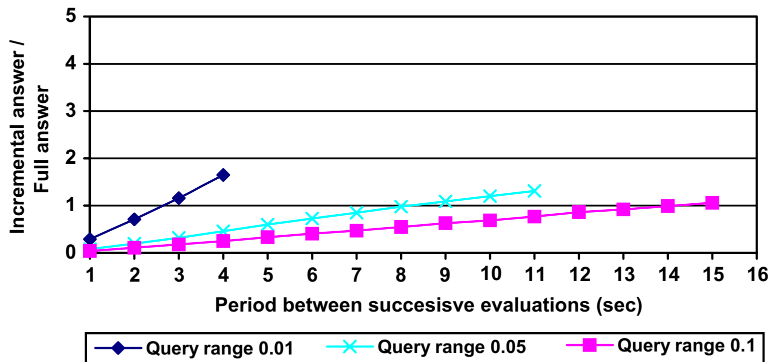

Fig. 16. The relation between incremental and full continuous query answer for different evaluation periods.

pre-refinement step. Therefore, for longer network segments, small query ranges and less frequent location/speed updates of mobile objects, the inclusion of the pre-refinement step is fully justified.

We perform the experiments to determine the duration of period between successive continuous query evaluations for which the size of incremental answer becomes greater than the size of the full answer (Fig. 16). We measure average sizes of the full answers and incremental answers of static continuous queries.

According to experimental settings and characteristics of simulated mobile objects, for evaluation period of 14 seconds for continuous queries with range of 0.1% of data space the incremental answer becomes greater than the full answer and thus continuous query processing methodology can switch to response with full answer rather than incremental answer. For queries with smaller range, i.e. less than 0.1 of the data space and especially for mobile queries which brings more dynamics in query answers, the threshold for which the incremental answer become greater then the full answer is even shorter. Most of the research work neglects this dynamics in query answer, which is also continuously changed with motion of objects and/or queries. For example, the SINA incremental evaluation experiments based also on data from Brinkhoff generator [2] adopt the query with side length 0.01, and reevaluation for every 10 seconds, while 10% of mobile objects explicitly change their locations within that period (stepwise motion). Kwon et al. in Ref. [18] use similar experimental setup, where 10% of objects report their locations every 3 min moving in a stepwise manner and the query window is set to 0.002 of the entire range.

5.3. Summary of the experiments

With respect to the above experimental results, the ARGONAUT methodology offers acceptable performance in updating the answering periods of the continuous query answers, when receiving new values for the location or speed of a mobile object. The optimal solution for the continuous query processing methodology is to employ the pre-refinement step for longer segments and less frequent location/speed updates between successive query evaluations, and bypass the pre-refinement with higher agility of mobile objects and longer evaluation periods. For longer periods between successive evaluations the methodology must switch to issuing the full answer rather than the incremental answer. Thus, the pre-refinement step offers better performance for evaluation of continuous queries over objects in highways and main motorways; in situations that are characterized by less frequent changes and updates of motion parameters (speed) and moderate frequency of evaluation. For continuous queries over objects in city streets, which frequently change their speed and acceleration, the pre-refinement step causes unnecessary overhead and must be avoided. In all cases the use of main-memory index and data structures, SR$*$-tree, NCT, MOT and CQT is fully justified. Therefore, the proposed methodology can be used to solve real-life problems and aid real-life applications which require the storage and manipulation of mobile objects moving in road networks.

6. Concluding remarks

This paper introduces the ARGONAUT framework and methodology for evaluating continuous range queries over mobile objects moving on a spatial network. The ARGONAUT methodology is based on both traditional indexing schemes (SR$*$-tree) implemented in main-memory and main-memory data structures. It employs an incremental evaluation paradigm to achieve scalability and efficiency in processing of continuous range queries. It introduces an additional step in traditional spatiotemporal query processing strategy (filter/refinement), the pre-refinement step. The filter step selects the candidate objects according to fulfillment of the spatial query condition and using appropriately extended SR$*$-tree index on spatial network. The pre-refinement step is performed after the filter step to further refine the mobile objects obtained by the filter step and builds the required main-memory data structures to support periodical and incremental refinement steps. The filter and pre-refinement steps are performed only once, unless the reference query object changes its underlying network segment. The refinement step is performed periodically by processing in-memory data structures generated by the pre-refinement step.

To the best of our knowledge this is the only reported work that studies continuous range queries over mobile objects whose motion is constrained by a spatial network. We perform comprehensive experiments measuring the required CPU times needed for the pre-refinement step, for the update of pre-refinement data

structures upon receiving location/speed update from a mobile object, and for the periodical refinement steps with variable query parameters. We also determine threshold at which it is justified to include the pre-refinement step in query processing strategy, and provide experiments to define the threshold for which the size of incremental query answer becomes greater than the full answer and thus utilize more network bandwidth for query answer response. The experimental results show that the performance of the ARGONAUT query processing methodology is satisfactory for real-world settings in LBS applications for monitoring and tracking mobile objects.

With this goal in mind and according to Jensen [15] we choose to implement our query processing methodology in wireless/wired Web prototype applications for monitoring and tracking mobile objects and fleet management. There are a only few reported implementations of continuous query processing systems, with regard to many proposed access methods and indexing structures dedicated to continuous query processing, reviewed in Section 2. Those are DOMINO (Databases fOr MovINg Objects) [30], PLACE (Pervasive Location-Aware Computing Environments) [22] and MODIS (Moving Object Database Interface System) [8]. We implement ARGONAUT Server as a mobile object server with mobile object data management and query processing functionality. The ARGONAUT Server provides Web service interface for ARGONAUT clients, which can be classical Web clients or mobile clients. The Web client for monitoring and tracking mobile objects is implemented using AJAX technology and SVG. Mobile client is implemented on J2ME technology. The continuous query processing methodology implemented within the ARGONAUT Server enables efficient processing of static and mobile queries with respect to the CPU time and main-memory utilization of the server, as well as network bandwidth in communications with mobile clients.

We plan to continue working on query processing in mobile environments, especially on the issues of distributed and mobile query processing techniques which ship some part of the query processing to the mobile objects which have the computational and storage capabilities to perform some part of the query processing algorithms. The promising direction will be to include some form of adaptation in our methodology, mentioned in Section 5.3 that will provide adaptive techniques to efficiently support the dynamic workload of objects and queries and current characteristics of their motion. We also plan to extend our query processing methodology to processing of different types of continuous queries, like k-NN queries, as well as to provide processing of persistent queries, which are continuous in nature and span over a time-interval and require reevaluation over the history of mobile objects' motion.

Acknowledgement

Research supported by the 2004–2006 Serbian–Greek joint research and technology program and by ARCHIMEDES project 2.2.14, "Management of Moving Objects and the WWW", of the Technological Educational Institute of Thessaloniki (EPEAEK II).

References

[1] N. Beckmann, H.-P. Kriegel, R. Schneider, B. Seeger, The R*-tree: an efficient and robust access method for points and rectangles, in: Proceedings ACM SIGMOD Conference, 1990, pp. 322–331.
[2] T. Brinkhoff, A framework for generating network-based moving objects, GeoInformatica 6 (2) (2002) 153–180.
[3] Y. Cai, K. Hua, G. Cao, Processing range-monitoring queries on heterogeneous mobile objects, in: Proceedings MDM Conference, 2004, pp. 27–38.
[4] Y. Cai, K.A. Hua, G. Cao, T. Xu, Real-time processing of range-monitoring queries in heterogeneous mobile databases, IEEE Trans. Mobile Comp. (2005) 931–942.
[5] A. Civilis, C. Jensen, S. Pakalnis, Techniques for efficient road-network-based tracking of moving objects, IEEE Trans. Knowl. Data Eng. 17 (5) (2005) 698–712.
[6] V. De Almeida, R. Guting, Indexing the trajectories of moving objects in networks, GeoInformatica 9 (1) (2005) 33–60.
[7] Z. Ding, R. Guting, Managing moving objects on dynamic transportation networks, in: Proceedings 16th SSDBM Conference, 2004, pp. 287–296.
[8] D.H. Francis, S. Madria, C. Sabharwal, MODIS: a moving object database interface system, Network Inform. Syst. J. 10 (5) (2005) 59–94.
[9] E. Frentzos, Indexing objects moving on fixed networks, in: Proceedings 8th SSTD Symposium, 2003, pp. 289–305.

[10] B. Gedik, L. Liu, Mobieyes: Distributed processing of continuously moving queries on moving objects in a mobile system, in: Proceedings EDBT Conference, 2004, pp. 67–87.
[11] B. Gedik, K. Wu, P. Yu, L. Liu, Motion adaptive indexing for moving continual queries over moving objects, in: Proceedings CIKM Conference, 2004, pp. 427–436.
[12] H. Hu, J. Xu, D. Lee, A generic framework for monitoring continuous spatial queries over moving objects, in: Proceedings ACM SIGMOD Conference, 2005, pp. 479–490.
[13] S. Illari, E. Mena, A. Illarramendi, Location-dependent queries in mobile contexts: distributed processing using mobile agents, IEEE Trans. Mobile Comp. 5 (8) (2006) 1029–1043.
[14] C. Jensen, J. Kolar, T.B. Pedersen, I. Timko, Nearest neighbor queries in road networks, ACM GIS (2003) 1–8.
[15] C. Jensen, Geo-Enabled, Mobile Services – a tale of routes, detours and dead ends, in: Proceedings DASFAA Conference, LNCS, vol. 3882, 2006, pp. 6–19.
[16] D. Kalashnikov, S. Prabhakar, S. Hambrusch, Main memory evaluation of monitoring queries over moving objects, Distrib. Parallel Dat. 15 (2) (2004) 117–135.
[17] D. Kwon, S. Lee, S. Lee, Indexing the current positions of moving objects using the lazy update R-tree, in: Proceedings 3rd MDM Conference, 2002, pp. 113–120.
[18] D. Kwon, S. Lee, W. Choi, S. Lee, An adaptive hashing technique for indexing moving objects, Data Know. Eng. 56 (3) (2006) 287–303.
[19] I. Lazaridis, K. Porkaew, S. Mehrotra, Dynamic queries over mobile objects, in: Proceedings 8th EDBT Conference, 2002, pp. 269–286.
[20] M.-L. Lee, W. Hsu, C.S. Jensen, B. Cui, K.L. Teo, Supporting frequent updates in R-trees: a bottom–up approach, in: Proceedings 29th VLDB Conference, 2003, pp. 608–619.
[21] M. Mokbel, X. Xiong, W. Aref, SINA: Scalable incremental processing of continuous queries in spatio-temporal databases, in: Proceedings ACM SIGMOD Conference, 2004, pp. 623–634.
[22] M. Mokbel, X. Xiong, M. Hammad, W. Aref, Continuous query processing of spatio-temporal data streams in place, Geoinformatica 9 (4) (2005) 343–365.
[23] K. Mouratidis, M.L. Yiu, D. Papadias, N. Mamoulis, Continuous nearest neighbor monitoring in road networks, Proceedings 32nd VLDB Conference, 2006, pp. 43–54.
[24] D. Papadias, J. Zhang, N. Mamoulis, Y. Tao, Query processing in spatial network databases, Proceedings 29th VLDB Conference, 2003, pp. 802–813.
[25] S. Prabhakar, Y. Xia, D. Kalashnikov, W. Aref, S. Hambrusch, Query indexing and velocity constrained indexing scalable techniques for continuous queries on moving objects, IEEE Trans. Comput. 51 (10) (2002) 1124–1140.
[26] B. Predic, D. Stojanovic, A framework for handling mobile objects in location based services, in: Proceedings AGILE Conference, 2005, pp. 419–427.
[27] P. Sistla, O. Wolfson, S. Chamberlain, S. Dao, Modeling and querying moving objects, in: Proceedings 13th ICDE Conference, 1997, pp. 422–432.
[28] D. Stojanovic, S. Djordjevic-Kajan, Modeling and querying mobile objects in location based services, Facta Universitatis: Series Mathematics and Informatics 18 (1) (2003) 59–80.
[29] H. Wang, R. Zimmermann, W.-S. Ku, Distributed continuous range processing on moving objects, in: Proceedings 17th DEXA Conference, 2006, pp. 655–665.
[30] O. Wolfson, H. Cao, H. Lin, G. Trajcevski, F. Zhang, N. Rishe, Management of dynamic location information in DOMINO, in: Proceedings EDBT Conference, 2002, pp. 769–771.
[31] K.-L. Wu, S.-K. Chen, P.S. Yu, Processing continual range queries over moving objects using VCR-based query indexes, in: Proceedings IEEE Conference on Mobile and Ubiquitous Systems: Networking and Services, 2004, pp. 226–235.
[32] K.-L. Wu, S.-K. Chen, P.S. Yu, Efficient processing of continual range queries for location-aware mobile services, Inform. Syst. Frontiers 7 (4-5) (2005) 435–448.
[33] K.-L. Wu, S.-K. Chen, P.S. Yu, Incremental processing of continual range queries over moving objects, IEEE Trans. Knowl. Data Eng. 8 (11) (2006) 1560–1575.
[34] X. Xiong, M. Mokbel, W. Aref, SEA-CNN: Scalable processing of continuous K-Nearest neighbor queries in spatio-temporal databases, in: Proceedings ICDE Conference, 2005, pp. 643–654.
[35] J.S. Yoo, S. Shekhar, In-route nearest neighbor queries, Geoinformatica 9 (2) (2005).
[36] X. Yu, K.Q. Pu, N. Koudas, Monitoring K-Nearest neighbor queries over moving objects, in: Proceedings ICDE Conference, 2005, pp. 631–642.

Dragan Stojanovic is currently an assistant professor at the Computer Science Department, Faculty of Electronic Engineering, University of Nis, Serbia. He received his Ph.D., M.S., and B.S. degrees in Computer Science from the Faculty of Electronic Engineering, University of Nis, Serbia, in 2004, 1998 and 1993, respectively. His current research and development interests include context-aware and location-based services and systems, mobile data management, spatio-temporal databases and geographic information systems.

Apostolos N. Papadopoulos was born in Eleftheroupolis, Greece in 1971. He received his 5-year Diploma degree in Computer Engineering and Informatics from the University of Patras and his Ph.D. degree from Aristotle University of Thessaloniki in 1994 and 2000 respectively. He has published several research papers in journals and proceedings of international conferences. From March 1998 to August 1998 he was a visitor researcher at INRIA Research Center in Rocquencourt, France, to perform research in spatial databases.

His research interests include spatial and spatiotemporal databases, data stream processing, data mining and information retrieval. His research work has over 270 citations in scientific journals and conference proceedings. He has served as a track co-chair of ACM SAC DTTA (Database Technologies Techniques and Applications) Track 2005, 2006, 2007 and 2008. He is a member of the Technical Chamber of Greece. Currently, he is a Lecturer in the Department of Informatics of Aristotle University of Thessaloniki.

Further information can be found at http://delab.csd.auth.gr/~apostol.

Bratislav Predic is currently research and teaching assistant and Ph.D. student at the Computer Science Department, Faculty of Electronic Engineering, University of Nis, Serbia. He received his B.Sc. (5-year Diploma) degree in computer science from the Faculty of Electronic Engineering, University of Nis, Serbia, in 2003. His research interests include mobile and ubiquitous computing, location-based services and locating systems, traffic management and geographic information systems.

Slobodanka Đorđević-Kajan is a full professor of computer science and the head of the CG&GIS Lab at the Computer Science Department, Faculty of Electronic Engineering, University of Nis, Serbia. She received her Ph.D., M.S., and B.S. degrees in Computer Science from the Faculty of Electronic Engineering, University of Nis, Serbia, in 1987, 1980 and 1968, respectively. Her current professional and scientific interests include context-aware and location-based services and systems, spatio-temporal and multimedia databases, and application of ontologies to geographic information systems and control and command systems.

Alexandros Nanopoulos was born in Craiova, Romania, in 1974. He graduated from the Department of Informatcis, Aristotle University of Thessaloniki, Greece, on November 1996, and obtained a PhD from the same institute, on February 2003. The subject of his dissertation was: "Techniques for Non Relational Data Mining". He is co-author of more than 30 articles in international journals and conferences. He has also co-author the monograph "Advanced Signature Techniques for Multimedia and Web Applications" and "R-trees: Theory and Applications". His research interests include data mining, web information retrieval, and spatial database indexing.