JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 33, 385-411 (2017)

DOI:10.1688/JISE.2017.33.2.7

QR*-Tree: An Adaptive Space-Partitioning Index for Monitoring Moving Objects

TIEN-KHOI PHAN, HARIM JUNG, HEE YONG YOUN AND UNG-MO KIM[†]
College of Information and Communication Engineering
Sungkyunkwan University
Jangan-gu, Suwon, 440-746 Korea
E-mail: {khoiphan; youn7147; ukim}@skku.edu; harim3826@gmail.com

A continuous range query over moving objects continually retrieves the moving objects that are currently within a given query region of interest. Most existing approaches assume that moving objects continually communicate with the server to report their current locations and the server updates the results of queries continuously. However, this assumption degrades the system performance because the communication cost and the server workload increase when the number of moving objects and queries becomes huge. The QR-tree is a query indexing structure, which helps the server cooperate with the moving objects efficiently by utilizing the available computational resources of the moving objects to improve the overall system performance. In this paper, we propose a variant of the QR-tree, namely, the QR*-tree, which helps reduce (i) the amount of location-update stream generated from moving object and (ii) the server work load for query evaluation. Through a series of comprehensive simulations, we verify the efficiency of the QR*-tree in terms of the wireless communication cost and the server workload.

Keywords: moving objects, location sensing, location-update stream, location-based services, range monitoring queries, query indexing, mobile/ubiquitous computing

1. INTRODUCTION

With the growing popularity of mobile devices equipped with location sensing technology and the advances in wireless networks and navigation systems, location-based services (LBSs) have been widely acknowledged to be the most promising applications in ubiquitous computing environments [1-18]. Many convenient LBSs are often based on the functionality of evaluating continuous range queries (CRQs), each of which continually retrieves the moving objects that are currently located within a spatial query region of interest. For instance, a transportation company wants to track vehicles' locations; a restaurant manager wants to send advertising messages to potential customers who are surrounding his restaurant; a tour guide needs to monitor some groups in different areas; a traffic management department wants to monitor the traffic conditions in some areas. In such applications, the query results should be updated when the moving objects exit or enter the regions of interest.

The studies on CRQ evaluation can be broadly categorized into two types which depend on the movement of the queries. In the first category, the studies focus on stationary or quasi-stationary queries over moving objects [1, 5, 10, 15, 17, 18], while the second category deals with moving queries over moving objects [2, 4, 6, 12-14]. Our work belongs to the former category. The majority of existing solutions for CRQ evalua-

Received January 8, 2016; revised August 17, 2016; accepted September 16, 2016.
Communicated by Wen-Chih Peng.
[†] Corresponding author.

tion assume that moving objects periodically send location-update to the server via wireless connections, and the server identifies the affected queries and updates their results if necessary [10, 13, 17, 18]. However, if the number of moving objects (and queries) registered at the server becomes huge, the overall system performance may deteriorate considerably because of the overwhelming server workload and the severe communication bottleneck [19].

To address the above problem, the *safe region* technique was proposed in [5, 15]. The safe region, given to each moving object o, is the area that contains the current location of o and does not overlap with any query boundary. As a result, o need not send its location update to the server as long as it does not exit its safe region. Although the safe region technique generally improves the system performance to a certain degree, due to the fact that the size of a safe region allocated to each object o is relatively small, and thus o easily exits its current safe region and contacts the server for receiving a new safe region. As a result, the server must frequently determine o's safe regions, leading to intensive computational overhead.

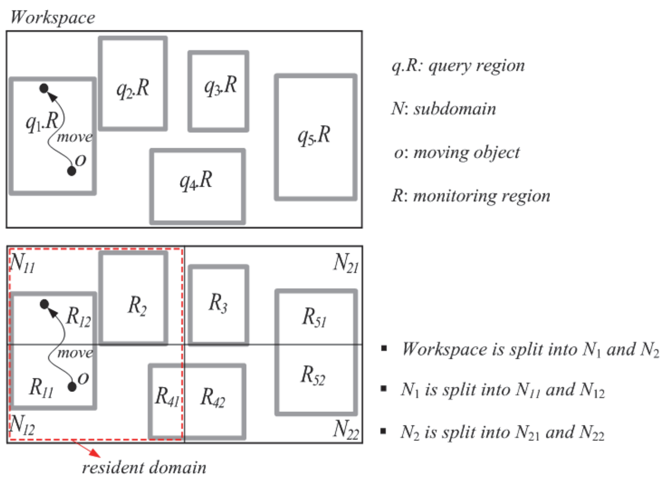
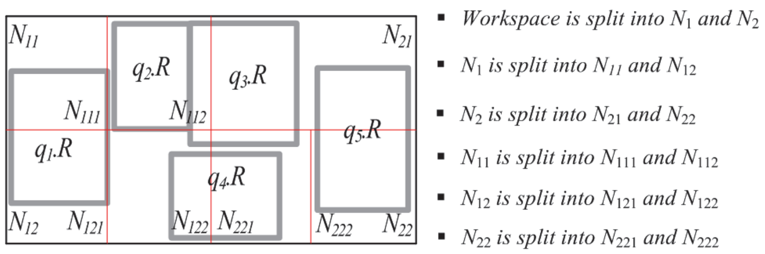


Fig. 1. Example of workspace split in MQM.

The *monitoring query management* (*MQM*), which aims to reduce the communication cost and the server workload by utilizing the available capabilities of moving objects, was introduced by Cai *et al.* in [1]. MQM partitions the rectangular workspace (or the database domain) into a set of disjoint subdomains. When a query region overlaps with a subdomain, the overlapping area is called a *monitoring region*. MQM uses the *binary partitioning tree* (*BP-tree*) and additional data structure for indexing queries based on monitoring regions. If the number of monitoring regions in a subdomain exceeds a predefined *split threshold t*, the subdomain is split into two smaller equal subdomains. This split process recursively continues until there is no subdomain that contains more than t monitoring regions. Fig. 1 shows an example of the workspace split for query regions $q_1.R \sim q_5.R$, assuming $t = 2$. In MQM, each moving object o is assigned (i) a rectangular subspace of the entire workspace, named the *resident domain*, which contains o; and (ii) all monitoring regions that overlap with *o*'s resident domain. The size of *o*'s resident domain is determined by *o*'s capability, *o.Cap*, which indicates the maximum number of (nearby) monitoring regions *o* can load and process at a time. Assuming the capability *o.Cap* of the moving object *o* in Fig. 1 is 4, for instance, *o* is assigned (i) the red-dotted rectangle, which includes two subdomains N_{11} and N_{12}, as its resident domain and (ii) four monitoring regions R_{11}, R_{12}, R_2, and R_{41}, which overlap with *o*'s resident domain. In MQM, the moving object *o* will contact the server to receive a new resident domain (together with new monitoring regions) or to let the server update the corresponding query result, respectively, only when *o* exits its resident domain or crosses any of the boundary of its assigned monitoring regions. As a result, the moving objects and the server share the CRQ evaluation, which helps the server workload degrade. The communication cost also decreases because the moving objects know exactly when they should send the messages to server. However, when a query region overlaps with many subdomains, the number of monitoring regions may increase rapidly. This leads MQM to assign a small resident domain to the moving object *o*; accordingly, *o* has to frequently contact to the server to receive a new resident domain. In addition, because *o* checks its movement against the monitoring regions instead of the original query regions, *o* may unnecessarily contact the server. As shown in Fig. 1, when the moving object *o* moves from monitoring region R_{11} to monitoring region R_{12}, it sends two messages to the server in order to notify that it exited R_{11} and entered R_{12}. Nevertheless, the query result of q_1 does not change because *o* is still in the query region $q_1.R$.

To overcome the limitations of MQM, Jung *et al*. proposed the *query region-tree method* (*QRT*) [20]. QRT uses the *Query Region tree* (*QR-tree*) for indexing queries based on the original query regions instead of the monitoring regions. This helps the QRT assign a much larger resident domain to the moving object *o*. Consequently, in QRT, the number of communications between the moving objects and the server is reduced. Similarly to MQM, QRT also partitions the workspace into a set of subdomains. Each leaf node in the QR-tree corresponds to a subdomain and is associated with a list of queries, which have the query regions overlap with the subdomain. When a leaf node is overflow (*i.e*., the number of queries overlapped with its corresponding subdomain exceeds the threshold *t*), the subdomain of this leaf node is split into two new equal-sized subdomains vertically or horizontally. This split method is called the *center split* method [1].



- *Workspace is split into* N_1 *and* N_2
- N_1 *is split into* N_{11} *and* N_{12}
- N_2 *is split into* N_{21} *and* N_{22}
- N_{11} *is split into* N_{111} *and* N_{112}
- N_{12} *is split into* N_{121} *and* N_{122}
- N_{22} *is split into* N_{221} *and* N_{222}

Fig. 2. Example of center split in QR-tree.

Fig. 2 shows an example of center split method in the QR-tree, assuming $t = 2$. The example is similar to the one in Fig. 1, except the size of query region $q_3.R$ is larger. This small difference obliges the QRT to divide the subdomains N_{11}, N_{12}, and N_{22} to {N_{111}, N_{112}}, {N_{121}, N_{122}} and {N_{221}, N_{222}}, respectively. It is easy to observe that the center split method incurs unnecessary splits. In particular, the QR-tree has the following drawbacks:

- First, the size of subdomains in the QR-tree is small. This leads the QRT to assign small resident domain to moving object *o*. As a consequence, *o* may have to frequently contact the server to receive a new resident domain.
- Second, when inserting a query, the recursive method *SplitNode* (See Algorithm 3 in [20]) can be invoked many times, this increases the server workload.
- Third, as the workspace is split into many subdomains, the size of QR-tree becomes large (*i.e.*, the number of nodes in QR-tree is much). Therefore, there will be more time consuming in order to search for a resident domain, or to insert/delete a query region in the QR-tree.

In this paper, we introduce a new variant of QR-tree, namely, the *QR^*-tree*, to overcome the above drawbacks of the QR-tree. The QR^*-tree uses a new splitting node method, called the *SmartSplit* method, which helps build a QR^*-tree with large subdomains, and the number of nodes in the QR^*-tree is less than that in the QR-tree. Because the insert/delete query operations of QR^*-tree is similar to that of QR-tree, in this work, we focus on describing the SmartSplit splitting node method.

The reminder of this paper is organized as follows. In Section 2, the QRT is summarized. In Section 3, the details of the SmartSplit method are described. In Section 4 the results of simulation experiments are presented. In Section 5, some related work is reviewed. Finally, Section 6 concludes the paper.

2. THE QUERY REGION TREE METHOD (QRT)

2.1 System Overview

In the QRT, the server and moving objects share the evaluation process of CRQs. In order to achieve this, QRT uses the resident domain concept. Similar to the models presented in most existing work [1, 5, 6, 15, 17, 18], the system model QRT considers includes a set of moving objects, the central server, and clients who issue queries (See Fig. 3).

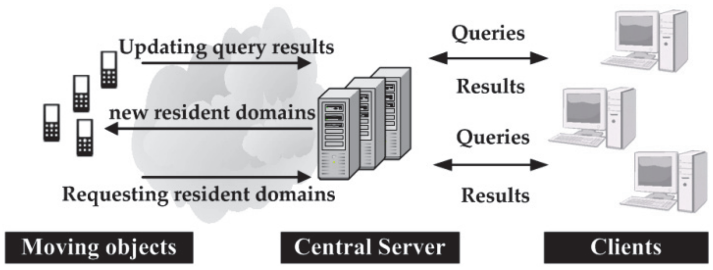


Fig. 3. System overview [20].

The moving objects and the clients communicate through the server. Each moving object *o* is conscious of its location (*e.g.*, is equipped with a GPS receiver), and has some available (memory and computational) capability, *o.Cap*. It is assumed that each moving object *o* has heterogeneous capability *o.Cap*, which is measured by the number of query regions it can load and process at a time, and that $o.Cap \geq \theta$, where θ is a system parameter that indicates the minimum number of query regions each moving object should be capable of processing; accordingly, a moving object with dominant capability is assigned a larger resident domain together with a bigger number of original query regions. Each moving object sends a message to the server via a wireless connection only when (i) it exits its current resident domain or (ii) it crosses any of its assigned query regions *q.R*. While the former is for receiving a new resident domain, the latter is to allow the server to update the result of the corresponding query *q*. Each client can issue multiple queries to the server and receives the results of these queries from the server via high-speed wired or wireless connections. The moving objects and the queries registered at the server are assumed to be identified by their unique identifiers.

The server maintains (i) a *query table*, hashed on query identifiers and (ii) the QR-tree. The query table stores, for each query *q*, an identifier *qid*, a query region *q.R*, and the result. Three main tasks are performed by the server as follows:

– **Query registration (or de-registration):** the task of query registration (or de-registration) is performed when a new query *q* is issued (or *q* is terminated) by a client, consisting of inserting *q* into (or deleting *q* from) the query table, updating the QR-tree, and broadcasting a message to all the moving objects in order to notify them of these changes.
– **Resident domain assignment:** when the registration of a new moving object or the message sent by a moving object that exits its current resident domain, the task of resident domain assignment is performed. A new resident domain is searched by traversing the QR-tree. Then, it is broadcast with a number of nearby query regions and an object identifier *oid*.
– **Query result update:** mainly in response to the message sent by a moving object whose movement crosses one of its assigned query regions *q.R*. The result of the corresponding query *q* is updated. This task may also be performed when *o* contacts the server to receive a new resident domain.

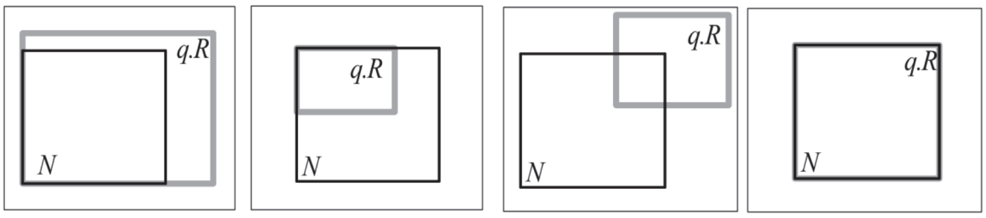
2.2 The Query Region Tree (QR-Tree)

In the QRT, there are four categories about the *overlap relationship* between a query region *q.R* and a (sub) domain *N* as shown in Fig. 4: *covers* (Fig. 4 (a)), *is covered by* (Fig. 4 (b)), *partially intersects* (Fig. 4 (c)), and *equals* (Fig. 4 (d)). The QR-tree built by splitting the entire workspace recursively is a binary tree index of queries. Given a set of query regions on the workspace that corresponds to the root, if the number of these query regions is greater than the split threshold *t*, it is split into two subdomains, each of which corresponds to a child node of the root. This process recursively lasts until every subdomain has no more than *t* query regions that are covered by or partially intersect the subdomain, and it corresponds to a leaf node. Note that the moving object with the minimum capability among all the moving objects registered at the server determines the threshold value t ($\geq \theta$).

A leaf node of the QR-tree stores at most t query identifiers, each of which refers to a query q in the query table. A non-leaf node stores two entries of the form (*ptr*, *N*), where *ptr* is a pointer to a child node (*i.e.*, leaf or non-leaf node) and *N* is a subdomain of the child node pointed to by *ptr*. From now, the symbol '*N*' denotes both a tree node and its corresponding (sub) domain. Each (leaf or non-leaf) node additionally stores a variable *Count* and is associated with a special list, called the *covering list* (*CL*). The QR-tree fulfills the following properties:

1. A leaf node *N* stores a query identifier *qid* of a query *q* only if $q.R$ is covered by or partially intersects *N*. It is important to note that although $q.R$ overlaps with *N*, *qid* is not stored in *N* if $q.R$ covers or equals *N*.
2. A query identifier *qid* of a query *q* can be redundantly stored in several leaf nodes if $q.R$ partially intersects these leaf nodes.
3. For each entry (*ptr*, $\acute{N}$) stored in a non-leaf node *N*; $\acute{N}$ represents one of the equal halves of *N*'s domain.
4. For each (leaf and non-leaf) node *N*, *N.Count* records the total number of query regions that are covered by or partially intersect *N*.
5. For each (leaf and non-leaf) node *N*, its associated covering list *N.CL* keeps every query identifier *qid* of a query *q* whose query region $q.R$ covers or equals to *N*.

Fig. 5 shows an example of the QR-tree for the query regions $q_1.R \sim q_5.R$ shown in Fig. 2.



(a) $q.R$ covers *N* (b) $q.R$ is covered *N* (c) $q.R$ partially intersects *N* (d) $q.R$ equals *N*

Fig. 4. Classification of the overlap relationship.

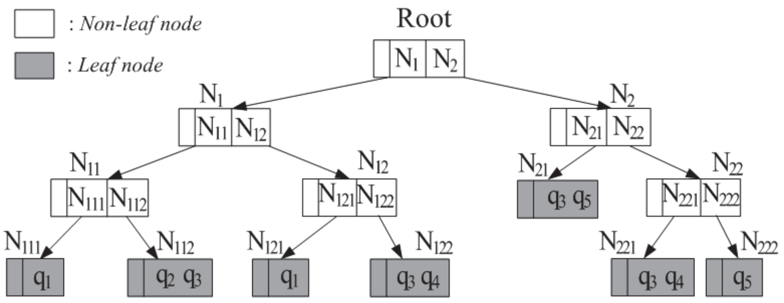


Fig. 5. Example of QR-tree with center split method.

3. THE SMARTSPLIT METHOD

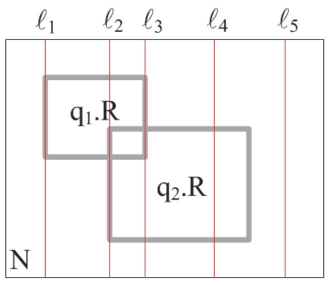
In this section, we present a new splitting node method, called *SmartSplit*, which helps build a QR*-tree with less number of nodes and larger subdomains than the QR-tree.

3.1 Split Strategy

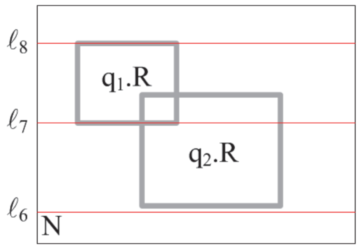
We assume that a (sub) domain N is split into two subdomains, N_1 and N_2, and we call the line ℓ that splits N the *splitting line* (*SL*). The SL ℓ can be horizontal ($\ell.y$ is the y-coordinate of ℓ) or vertical ($\ell.x$ is the x-coordinate of ℓ). Our split strategy is based on four rules:

First, the subdomains must have an overlap relationship with at least one query region, *i.e.*, $N_1.Count > 0$ and/or $N_1.CL \neq null$, and $N_2.Count > 0$ and/or $N_2.CL \neq null$. If N_1 does not overlap with any query region, the query regions in N_2 are the same with that in N and the split is therefore inefficacious. Figs. 6 (a) and (b) show examples of vertical SLs and horizontal SLs, respectively. In the figures, ℓ_1, ℓ_5, ℓ_6 and ℓ_8 are invalid SLs, while ℓ_2, ℓ_3, ℓ_4 and ℓ_{67} are valid SLs in accordance with the first rule.

Second, the number of SLs should be minimized. If there are more SLs, the smaller size of subdomains will be obtained. This rule helps maintain the size of subdomains as large as possible.



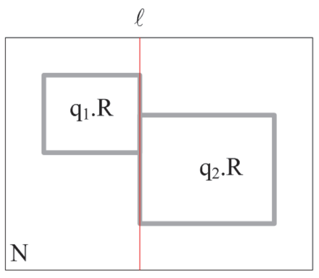
(a) Vertical SLs.



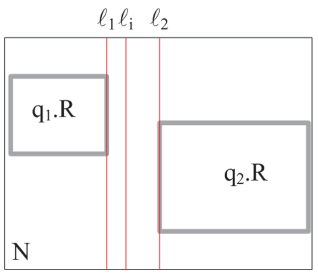
(b) Horizontal SLs.

Fig. 6. Examples of SLs.

Definition 1: (*Suboptimal SL*). A suboptimal SL is a SL, which makes $N_1.Count \leq t$ and $N_2.Count \leq t$.



(a) One suboptimal SL.



(b) Uncountable suboptimal SLs.

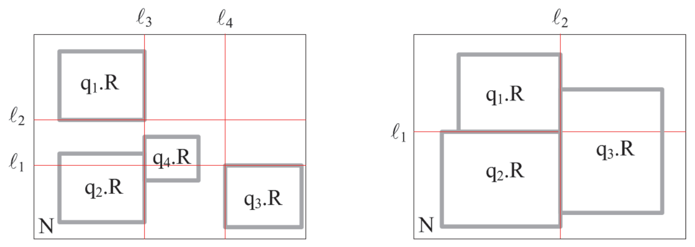
Fig. 7. Examples of suboptimal SLs.

It is clear that the suboptimal SL helps make only one split. Fig. 7 shows two examples of splitting (sub) domains with the threshold t being equal to 1. In Fig. 7 (a), there is one suboptimal SL ℓ, while in Fig. 7 (b), there are uncountable suboptimal SLs ℓ_i between $[\ell_1, \ell_2]$ (*i.e.*, $\ell_1.x \leq \ell_i.x \leq \ell_2.x$). If there is one suboptimal SL in the (sub) domain (*e.g.*, Fig. 7 (a)), it is chosen to split the (sub) domain. On the other hand, if there is no suboptimal SL (*e.g.*, Fig. 6), the third rule and the fourth rule will be evaluated. In case that there are more than one suboptimal SL (*e.g.*, Fig. 7 (b)), the third rule and the fourth rule will be considered among them.

Third, the number of queries in two subdomains N_1 and N_2 should be equal to the threshold value t or smaller than t. The best case is: $N_1.Count = N_2.Count = N.Count/2$. This rule helps the probability of splitting N_1 and/or N_2 into new subdomains when a new query is registered become low. To evaluate this rule, the variance of queries in N_1 and N_2 (denoted by *VarQ* of ℓ) is considered foremost; the smaller *VarQ*, the better. If there are many SLs with the same *VarQ*, the total number of queries in N_1 and N_2 (denoted by *SumQ* of ℓ) is considered subsequently; the smaller *SumQ*, the better. If *VarQs* of the SLs are equal, the fourth rule is evaluated. *VarQ* and *SumQ* of the SL ℓ are defined as follows:

$$VarQ(\ell) = (N_1.Count - N.Count/2)^2 + (N_2.Count - N.Count/2)^2, \tag{1}$$
$$SumQ(\ell) = N_1.Count + N_2.Count. \tag{2}$$

Fig. 8 (a) shows an example of splitting the domain N with the threshold t being equal to 3. In the example, there are many suboptimal SLs between $[\ell_1, \ell_2]$ and $[\ell_3, \ell_4]$. However, the SL ℓ_3 is chosen to split N because its *VarQ* is minimum (*i.e.*, 0). In Fig. 8 (b), assuming $t = 2$, there are two suboptimal SLs, ℓ_1 and ℓ_2. Because $VarQ(\ell_1) = VarQ(\ell_2) = 0.5$, *SumQ* is computed. The SL ℓ_2 is chosen to split N because $SumQ(\ell_1) = 4$ and $SumQ(\ell_2) = 3$.



(a) Choose SL with VarQ. (b) Choose SL with SumQ.
Fig. 8. Examples of choosing SLs based one VarQ and SumQ.

Fourth, the area of two subdomains N_1 and N_2 tends to be equal. The best case is: $N_1.Area = N_2.Area = N.Area/2$, where *N.Area* is the area of N. The fourth rule aims to create the subdomains with the relatively equal areas. This helps avoid the case of some too small subdomains and some too large subdomains. The variance of the areas of N_1 and N_2 (denoted by *VarA* of ℓ) is used to evaluate the fourth rule; the smaller *VarA*, the better. *VarA* of a SL ℓ is defined as:

$$VarA(\ell) = (N_1.Area - N.Area/2)^2 + (N_2.Area - N.Area/2)^2. \tag{3}$$

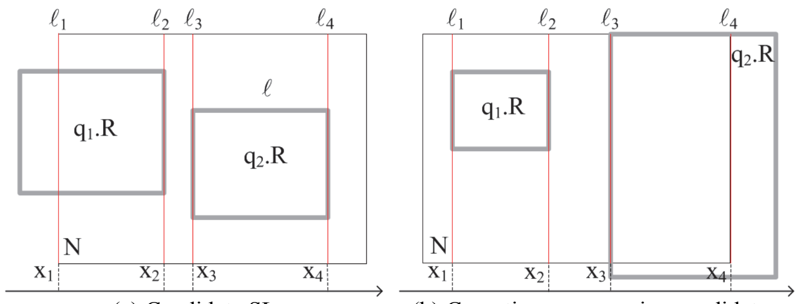
In the example in Fig. 7 (b), there are many suboptimal SLs between [ℓ_1, ℓ_2]. Checking the third rule, these suboptimal SLs have the same *SumQ* and *VarQ* (*i.e.*, 0 and 2, respectively). Then, the fourth rule is considered and the SL ℓ_2 is chosen, because its *VarA* is minimum.

3.2 The SmartSplit Algorithm

The SmartSplit algorithm has two phases, (i) sweeping the (sub) domain to find the horizontal and vertical SLs and (ii) evaluating the SLs to find the best SL based on the rules mentioned in Section 3.1.

3.2.1 The sweeping domain phase

In this section, we present the method to find the vertical SLs on a domain N (the horizontal SLs on N are found similarly). Let us assume that there is an imaginary vertical SL ℓ that sweeps from the left edge to the right edge of N. It is clear that when ℓ passes over the left edge or the right edge of a query region $q.R$, the number of queries on the subdomains created by ℓ may be changed. In particular, if ℓ passes over the left edge of $q.R$ (or ℓ enters $q.R$), the number of queries in the left subdomain (*i.e.*, N_1) created by ℓ is increased. If ℓ passes over the right edge of $q.R$ (or ℓ exits $q.R$), the number of queries in the right subdomain (*i.e.*, N_2) is decreased. We call the SLs at the left edge and the right edge of the query regions in N the *candidate SLs*, and the x-coordinates of the candidate SLs are called the *candidate positions*.



(a) Candidate SLs. (b) Coverting query region candidate.

Fig. 9. Examples of candidate SLs and covering query region.

In Fig. 9 (a), assuming $t = 1$, ℓ_1, ℓ_2, ℓ_3 and ℓ_4 are the candidate SLs, while x_1, x_2, x_3 and x_4 are the candidate positions that correspond to the candidate SLs. In the figure, because the left edge of $q_1.R$ is out of N, the SL ℓ_1 at the left edge of N becomes the candidate SL. It can be observed that every vertical SL ℓ_i between two continuous SLs ℓ_2 and ℓ_3 can split N into two subdomains N_1 and N_2, where $N_1.Count = 1$ and $N_2.Count = 1$. Fig. 9b shows an example of splitting the domain N with $t = 1$. For every SL ℓ_i between [ℓ_3, ℓ_4], the right subdomain N_2 is covered by $q_2.R$. We call $q_2.R$ the *covering query region*. It is important to note that if $q_2.R$ covers or equals to N_2, the query identifier of q_2 is added to the covering list of N_2.

Lemma 1: Given two continuous candidate SLs ℓ_i and ℓ_j, $VarQ(\ell_i) \leq VarQ(\ell_k)$ or $VarQ(\ell_j) \leq VarQ(\ell_k)$, where ℓ_k is a SL between (ℓ_i, ℓ_j) (*i.e.*, $\ell_i.x < \ell_k.x < \ell_j.x$).

***Proof*:** We prove this lemma by contradiction. Let us assume that $VarQ(\ell_k) - VarQ(\ell_i) < 0$ and $VarQ(\ell_k) - VarQ(\ell_j) < 0$.

The subdomains are created by ℓ_i, ℓ_j and ℓ_k are (N_{i1}, N_{i2}), (N_{j1}, N_{j2}) and (N_{k1}, N_{k2}), respectively. Without the loss of generality, we define:

$$N_{i1}.Count = x,\ N_{i2}.Count = y,$$

then, we have:

$$N_{k1}.Count = x + a,\ N_{k2}.Count = y$$
$$N_{j1}.Count = x + a,\ N_{j2}.Count = y - b,$$

where a is the number of query regions that have the left edges overlap with ℓ_i and b is the number of query regions that have the right edges overlap with ℓ_j. Because we do not consider the covering query regions in this lemma, the effect of the covering query regions will be verified later.

From Eq. (1) we have:

$$VarQ(\ell_k) - VarQ(\ell_i) = a(a + 2x - N)$$
$$VarQ(\ell_k) - VarQ(\ell_j) = b(-b + 2y - N).$$

By combining with our assumption, we obtain:

$$N > a + 2x \tag{4}$$
$$N > -b + 2y. \tag{5}$$

With the SLs ℓ_i and ℓ_j, we have:

$$x \geq N - y \tag{6}$$
$$x + a \geq N - y + b. \tag{7}$$

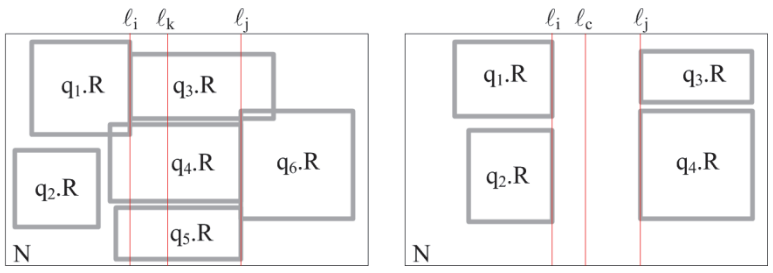
By combining Eqs. (4) and (5), we obtain:

$$2N > 2x + 2y + a - b. \tag{8}$$

By combining (6) and (7), we obtain:

$$2N \leq 2x + 2y + a - b. \tag{9}$$

Because of Eqs. (8) and (9), our assumption becomes wrong, and thus $VarQ(\ell_i) \leq VarQ(\ell_k)$ or $VarQ(\ell_j) \leq VarQ(\ell_k)$. ❑



(a) Continuous candidate SLs. (b) The central SL.

Fig. 10. Examples of two continuous SLs and central SL.

Fig. 10 (a) shows an example of two continuous SLs, ℓ_i and ℓ_j, assuming $t = 5$. In this example, $N = 6$, $x = 4$, $y = 4$, $a = 1$, $b = 2$, and $VarQ(\ell_i) = 2$, $VarQ(\ell_k) = 5$, $VarQ(\ell_j) = 5$.

Lemma 2: Given two continuous candidate SLs ℓ_i and ℓ_j, $SumQ(\ell_i) \leq SumQ(\ell_k)$ and $SumQ(\ell_j) \leq SumQ(\ell_k)$, where ℓ_k is a SL between (ℓ_i, ℓ_j).

***Proof*:** Similarly to the proof of Lemma 1, without the loss of generality, we define:

$N_{i1}.Count = x$, $N_{i2}.Count = y$.

We have:

$N_{k1}.Count = x + a,\ N_{k2}.Count = y$
$N_{j1}.Count = x + a,\ N_{j2}.Count = y - b.$

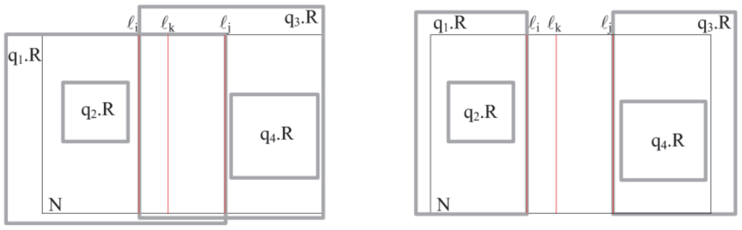
Then, $SumQ(\ell_i) = x + y$, $SumQ(\ell_k) = x + a + y$, $SumQ(\ell_j) = x + a + y - b$. It is easy to observe that $SumQ(\ell_i) \leq SumQ(\ell_k)$ and $SumQ(\ell_j) \leq SumQ(\ell_k)$. ❑

Lemma 3: Given two continuous candidate SLs ℓ_i and ℓ_j, the best SL between $[\ell_i, \ell_j]$ is ℓ_i or ℓ_j or ℓ_c, where ℓ_c is called *central SL* and $\ell_c.x = (N.Left + N.Right)/2$, $\ell_i.x < \ell_c.x < \ell_j.x$.[1]

***Proof*:** We only consider the scenario of $VarQ(\ell_i) = VarQ(\ell_k) = VarQ(\ell_j)$ and $SumQ(\ell_i) = SumQ(\ell_k) = SumQ(\ell_j)$, for other cases, from Lemma 1 and Lemma 2, ℓ_i and/or ℓ_j is the best SL based on the third rule. The fourth rule is considered with three situations. First, the central SL ℓ_c is between $[\ell_i, \ell_j]$, ℓ_c is the best SL because $VarA(\ell_c) = 0$. Second, ℓ_c is on the left of $[\ell_i, \ell_j]$, ℓ_i is the best SL because its *VarA* is minimum (ℓ_i is the nearest SL to ℓ_c). Third, ℓ_c is on the right of $[\ell_i, \ell_j]$, ℓ_j is the best SL. ❑

In Fig. 10 (b), assuming $t = 3$, ℓ_c is a central SL between two continuous candidate SLs, ℓ_i and ℓ_j.

[1] In this paper, we use *N.Left* and *N.Right*, the x-coordinates of the left edge and the right edge of *N*, respectively; *N.Bottom*, *N.Top* are the y-coordinates of the bottom edge and the top edge of *N*, respectively.



(a) Covering query regions cover [ℓ_i, ℓ_j]. (b) Covering query regions touch [ℓ_i, ℓ_j].
Fig. 11. Effect of covering query regions to SLs.

Fig. 11 (a) shows an example of covering query regions overlap two continuous candidate SLs, ℓ_i and ℓ_j ($t = 3$). In this case, $q_1.R$ and $q_3.R$ cover all SLs between [ℓ_i, ℓ_j]. We can see that $q_1.R$ and $q_3.R$ do not effect to the values *VarQ*, *SumQ* of the SLs between [ℓ_i, ℓ_j]. In Fig. 11 (b), assuming $t = 3$, $q_1.R$ and $q_3.R$ touch the candidates SLs, ℓ_i and ℓ_j, respectively. It is clear that the covering query regions help reduce the *Count* value of subdomains created by ℓ_i and ℓ_j (*i.e.*, N_{i1} and N_{j2}). The covering query regions have more positive influence on the candidate SLs than the SLs between candidate SLs.

As such, in the finding the SLs on the domain phase, the algorithm only needs to seek the candidate SLs and the central SL. The Algorithm 1, denoted by *SweepDomain*, describes the process of finding the SLs.

Algorithm 1: SweepDomain

```
Input N: a domain
Output SLlist: a list of splitting lines
1.  Initialize CQRlist, SLlist, binary search tree BST
2.  for each query region QR of each qid stored in N do
3.      if(QR.Left ≤ N.Left) then
4.          create new candidate position p = <N.Left, 1, 0>
5.          if(QR.Bottom ≤ N.Bottom and QR.Top ≥ N.Top)
6.              insert QR to CQRlist
7.      else
8.          create new candidate position p = <QR.Left, 1, 0>
9.      if(p is not in BST) then
10.         insert p into BST
11.     else
12.         increase LeftCount of p in BST by 1
13.     if(QR.Right ≥ N.Right) then
14.         create new candidate position ṕ = <N.Right, 0, 1>
15.         if(QR.Bottom ≤ N.Bottom and QR.Top ≥ N.Top)
16.             insert QR to CQRlist
17.     else
18.         create new candidate position ṕ = <QR.Right, 0, 1>
19.     if (ṕ is not in BST) then
20.         insert [illegible] into BST
21.     else
22.         increase RightCount of ṕ in BST by 1
23. for each candidate position cp in BST in in-order traversal do
24.     if(pre_cp is null) then
25.         num_left = cp.LeftCount
26.         num_right = cp.RightCount
27.     else
28.         if(pre_cp.Pos < (N.Left + N.Right)/2 < cp.Pos) then
29.             create a new SL $\ell_c$ = <v, (N.Left + N.Right)/2, num_left, N.Count − num_right>
```

```
30.             insert $\ell_c$ into SLlist
31.         num_right = num_right + cp.RightCount
32.         create a new SL $\ell$ = <v, cp.Pos, num_left, N.Count - num_right>
33.         insert $\ell$ into SLlist
34.         num_left = num_left + cp.LeftCount
35.     pre_cp = cp
36. remove the last SL in SLlist
37. for each SL $\ell$ in SLlist do
38.     for each query region QR in CQRlist do
39.         if(QR covers $\ell$ and $QR.Left \leq N.Left$) then
40.             decrease $\ell$.Count1 by 1
41.         else if(QR covers $\ell$ and $QR.Right \geq N.Light$) then
42.             decrease $\ell$.Count2 by 1
43. return SLlist
```

The SweepDomain has three steps as follows. First, SweepDomain collects information of candidate positions, from the query regions of queries stored in *N*. For each query region *QR*, SweepDomain creates the candidate position at the left edge first (lines 3-12). If the left edge of *QR* is out of *N*, SweepDomain creates a new candidate position at the left edge of *N* (line 4). The candidate position has the format of the form <*Pos, LeftCount*, *RightCount*>, where *Pos* is the x-coordinate of the candidate position, *LeftCount* is the number of query regions that have the x-coordinate of the left edge being equal to *Pos*, *RightCount* is the number of query regions that have the x-coordinate of the right edge being equal to *Pos*. SweepDomain also collects all covering query regions, and stores them in a list called *CQRlist* (line 6). SweepDomain uses a binary search tree (called *BST*) to store the candidate positions efficiently (lines 9-12). If there is an existing candidate position at the same position in *BST*, SweepDomain increases *LeftCount* of the existing candidate position in *BST* by 1 (line 12). SweepDomain does the similar way to the right edge of *QR* (lines 13-22).

Second, SweepDomain creates SLs at candidate positions by sweeping the domain *N* from the left to the right. When traversing *BST* in in-order traversal, the candidate positions are visited in the ascending order of their positions. At the first candidate position (the last candidate position), there is no query region overlap with the left subdomain N_1 (the right subdomain N_2) of the SL at this position, then SweepDomain does not create a SL at this position, following the first rule (lines 24 and 36). The SL has the format of the form <*Type, Pos, Count*1, *Count*2>, where *Type* states the SL horizontally or vertically (*h* indicates the horizontal SL, *v* indicates the vertical SL), *Pos* is the coordinate of SL, *Count*1 is the number of query regions that overlap N_1, *Count*2 is the number of query regions that overlap N_2. SweepDomain calculates *Count*1 and *Count*2 based on *LeftCount* value and *RightCount* value of the candidate positions that it meets when sweeping *N*. SweepDomain also creates a SL at the center of the domain (lines 28-30). All SLs are stored in a list called *SLlist* (lines 30 and 33).

Third, SweepDomain uses the covering query region list *CQRlist* in the first step to check the covering list of the subdomains of all SLs in *SLlist*. For each SL ℓ in *SLlist*, if there is a query region *QR* that covers or equalizes to the subdomain N_1, SweepDomain decreases ℓ.*Count*1 by 1 (lines 39-40). If *QR* covers or equalizes to the subdomain N_2, SweepDomain decreases ℓ.*Count*2 by 1 (lines 40-41). After the third step, the value ℓ.*Count*1 (or ℓ.*Count*2) is equal to the value N_1.*Count* (or N_2.*Count*); accordingly, the SL ℓ can be used in the evaluating SLs phase.

Lemma 4: SweepDomain runs in $O(t \log t + ct)$ time, where c is the number of covering query regions.

Proof: The number of query region is $t + 1$ and the number of candidate positions is at most $2t + 2$. Then, the time to finish the first step is $O(t \log t)$. The time to finish the second step and third step is $O(t)$ and $O(ct)$, respectively. Therefore, SweepDomain finishes in $O(t \log t + ct)$ time. ❑

3.2.2 The evaluating splitting lines phase

The process of evaluating the SLs is presented in the Algorithm 2, denoted by *EvaluateSL*. EvaluateSL is fairly simple and intuitive. For each SL ℓ, if there is no temporary best SL, ℓ becomes a temporary best SL (lines 2-3). Otherwise, EvaluateSL checks if ℓ is a suboptimal SL. First, if ℓ is a suboptimal SL and there is no suboptimal SL earlier, ℓ is chosen as the temporary best SL (lines 5-7). In case there is a suboptimal SL (temporary best SL) earlier, EvaluateSL compares two suboptimal SLs based on the rules in Section 3.1. The order of comparative values is *VarQ-SumQ-VarA*. If ℓ is better than the old one, ℓ replaces the old one to become the new temporary best SL (lines 9-16). Second, ℓ is not a suboptimal SL, EvaluateSL only evaluates ℓ if there is no suboptimal SL earlier because of the second rule. The evaluation is similar to the case of suboptimal SL with three comparative values, *VarQ-SumQ-VarA* (lines 18-26). EvaluateSL returns the best SL in accordance with the rules in Section 3.1.

Algorithm 2: EvaluateSL

```
Input N: a domain, SLlist: a SLs list
Output best_sl: the best splitting line
1.  for each SL $\ell$ in SLlist do
2.      if(best_sl is null) then
3.          best_sl = $\ell$
4.      else
5.          if($\ell$.Count1 $\leq t$ and $\ell$.Count2 $\leq t$) then
6.              if(best_sl.Count1 $> t$ or best_sl.Count2 $> t$) then
7.                  best_sl = $\ell$
8.              else
9.                  if(VarQ($\ell$) < VarQ(best_sl)) then
10.                     best_sl = $\ell$
11.                 else if (VarQ($\ell$) = VarQ(best_sl)) then
12.                     if(SumQ($\ell$) < SumQ(best_sl)) then
13.                         best_sl = $\ell$
14.                     else if (SumQ($\ell$) = SumQ(best_sl)) then
15.                         if(VarA($\ell$) < VarA(best_sl)) then
16.                             best_sl = $\ell$
17.         else
18.             if( (best_sl.Count1 $> t$ or best_sl.Count2 $> t$) then
19.                 if(VarQ($\ell$) < VarQ(best_sl)) then
20.                     best_sl = $\ell$
21.                 else if (VarQ($\ell$) = VarQ(best_sl)) then
22.                     if(SumQ($\ell$) < SumQ(best_sl)) then
23.                         best_sl = $\ell$
24.                     else if (SumQ($\ell$) = SumQ(best_sl)) then
25.                         if(VarA($\ell$) < VarA(best_sl)) then
26.                             best_sl = $\ell$
27. return best_sl
```

3.2.3 The overall algorithm

The overall algorithm *SmartSplit* is presented in Algorithm 3. SmartSplit sweeps the domain from left to right to get vertical SLs first, then sweeps the domain from bottom to top to get horizontal SLs (lines 1-2). After that, SmartSplit evaluates the SLs by invoking the function *EvaluateSL* in order to choose the best SL (lines 3-4). Depending on the type of this best SL, the suitable subdomains are created (lines 5-10). Fig. 12 shows the workspace split with SmartSplit splitting node method and the QR*-tree of the example in Fig. 2. The size of the QR*-tree is much smaller and the area of subdomains is larger than that of QR-tree in Fig. 5.

Algorithm 3: SmartSplit

```
Input N: a domain
Output <$N_1$, $N_2$>: two subdomains $N_1$ and $N_2$
1.  v_SLlist = invoke SweepDomain to sweep from left to right of N
2.  h_SLlist = invoke SweepDomain to sweep from bottom to top of N
3.  SLlist = h_SLlist merge with v_SLlist
4.  best_sl = EvaluateSL(SLlist)
5.  if(best_sl.Type = v) then
6.      create a subdomain $N_1$ = <N.Left, N.Bottom, best_sl.Pos, N.Top>
7.      create a subdomain $N_2$ = <best_sl.Pos, N.Bottom, N.Right, N.Top>
8.  else
9.      create a subdomain $N_1$ = <N.Left, N.Bottom, N.Right, best_sl.Pos>
10.     create a subdomain $N_2$ = <N.Left, best_sl.Pos, N.Right, N.Top>
11. return <$N_1$, $N_2$>
```

Because the operations of searching for resident domains, inserting/deleting queries and merging nodes of QR*-tree are similar to that of QR-tree, we omit the details of these operations.

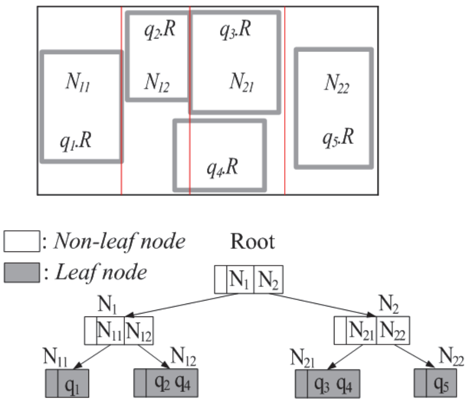


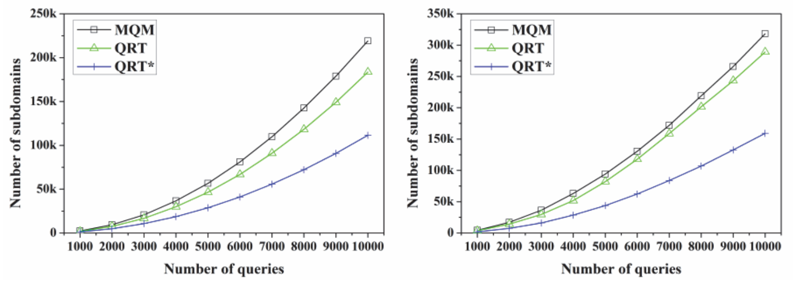
Fig. 12. QR*-tree with SmartSplit method.

4. PERFORMANCE EVALUATION

In this section, we evaluate and compare the performance of the QR*-tree (denoted by QRT[*]) with that of QR-tree [20] (denoted by QRT) and MQM [1] for CRQ evaluation in terms of the server workload and communication cost. The server workload was measured concerning the CPU-time that the server takes for CRQ evaluation. On the other hand, the communication cost was measured by the total number of messages transmitted between the server and moving objects. The simulations were conducted on dual Intel Xeon x5860 6-core processors with 8 GB RAM running on the Linux system.

4.1 Simulation Setup

Our simulations were based on two sets of queries, *Uniform* and *Skewed*, with the workspace fixed at 25km × 25km square. In *Uniform*, query regions are uniformly placed on the workspace. On the other hand, in *Skewed*, the distribution of query regions on the workspace follows the *Zipf* distribution with skew coefficient $\alpha = 0.8$. Each query region in both *Uniform* and *Skewed* is a square. The movements of the moving objects generated follow the random waypoint model (RWM) [21], which is one of the most widely used mobility models: each moving object chooses a random point of destination on the workspace and moves to the destination at a constant speed distributed uniformly from 0 to maximum speed. Upon reaching the destination, it remains stationary for a certain period of time. When this period expires, the moving object chooses a new destination and repeats the same process during the simulation time steps. The computational capability of each moving object was randomly selected from the range between 10 and 100 query regions (or monitoring regions), and thus the threshold value *t* of QR[*]-tree, QR-tree, and BP-tree (used in MQM) was set to 10. The total number of subdomains directly affects the performances of MQM, QRT, and QRT[*], because (i) if the number of subdomains is large, the size of subdomains is small, as a result, the size of resident domains assigned to moving objects is small; (ii) if the number of subdomains is large, the index tree (*i.e.*, BP-tree, QR-tree and QR[*]-tree) is large, and it takes more time to search resident domains or insert/delete queries. We measured the number of subdomains of MQM, QRT and QRT[*] for *Uniform* and *Skewed* according to the number of query regions (Fig. 13). As shown in the figure, the number of subdomains in QRT[*] is much smaller than that in MQM and QRT.



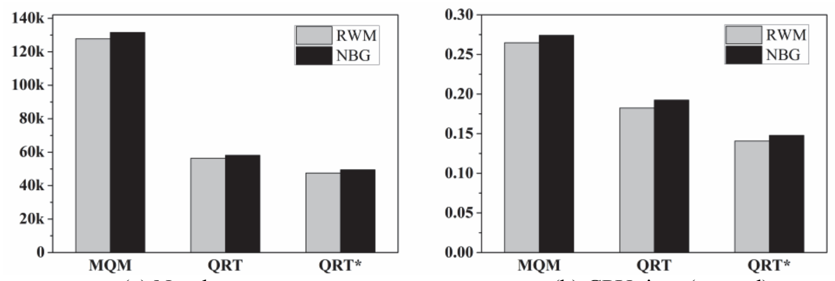
(a) Uniform. (b) Skew.
Fig. 13. Number of subdomains vs. number of queries.

We list the set of used parameters and their default values (stated in boldface) in the simulations in Table 1. In each simulation, we evaluated the effect of one parameter while the others were fixed at their default values. We ran each simulation for 1000 simulation time steps and measured the average of the CPU-time (in second) and total number of messages.

Besides RWM, we also used the network-based generator (NBG) [22] to generate moving objects. We generated the moving objects on the road network of San Joaquin County (18263 nodes, 23874 edges), scaled to fit the workspace 25km × 25km.

Table 1. Simulation parameters and their values.

Simulation parameter	Value used (Default)
Cardinality of *Uniform*/*Skewed*	1000 - 10,000 (5000)
Side length of query regions	0.4 - 4 km (2.0 km)
Number of moving objects	5,000 - 50,000 (25,000)
Update rates of queries	1 - 10% (0%)
Maximum speed of moving objects	10 - 100 km/h (50 km/h)



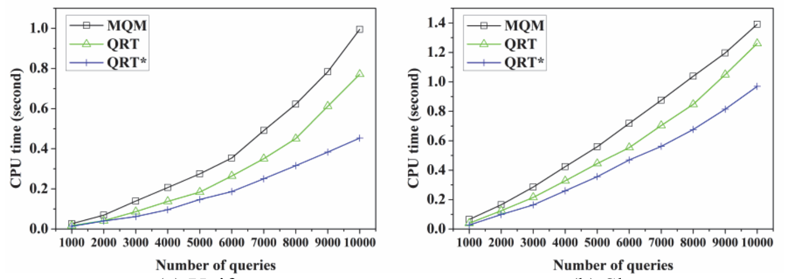
(a) Number messages. (b) CPU time (second).
Fig. 14. Performances on RWM and NBG.

Fig. 14 shows the performance comparison, among three methods MQM, QRT, and QRT*, for RWM and NBG in terms of (a) the number of messages and (b) the server work load (CPU time). The simulation run on *Uniform* with the default values of parameters (*i.e.*, Cardinality of *Uniform*: 5000, Side length of query regions: 2.0km, Number of moving objects: 25,000, Update rates of queries: 0%, Maximum speed of moving objects: 50 km/h). Because the performances on RWM and NBG are similar, we only present the simulation results using RWM.

4.2 Simulation Results

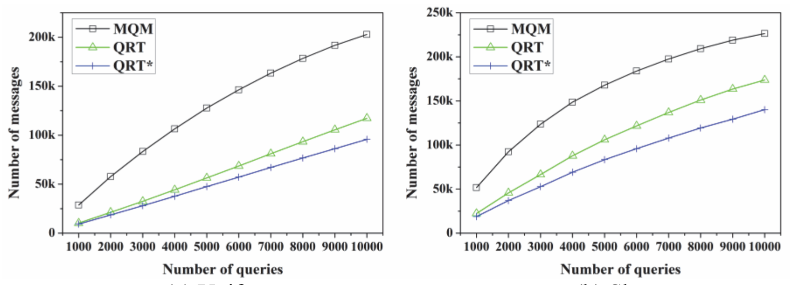
4.2.1 Effect of the number of query regions

In this simulation, we varied the cardinalities of *Uniform* and *Skewed* from 1000 to 10,000, then studied the effect of the number of query regions on the server workload and communication cost. The purpose of this simulation was to show the scalability of QR*-tree with regard to the number of queries. Fig. 15 shows the effect of the number of query regions on the CPU-time the server takes to perform CRQ evaluation. In MQM, QRT, and QRT*, the CPU-time performance is affected mainly by the search process for assigning resident domains to moving objects.



(a) Uniform. (b) Skew.
Fig. 15. CPU-time vs. number of queries.

As shown in the figure, QRT performs better than MQM for *Uniform* and *Skewed*. This is due to the fact that, as the number of query regions increases in MQM, the number of monitoring regions increases, which leads the server to assign small resident domains to moving objects. As a result, the server has to search a new resident domain frequently to assign each moving object that exits its current small resident domain. The BP-tree in MQM is built based on monitoring regions instead of the original query regions, subsequently the capabilities of moving objects are measured against the huge number of monitoring regions. On the other hand, the QR-tree is built based on the original query regions directly and the capabilities of moving objects are measured against only the number of original query regions that are covered by or partially intersect each QR-tree node. However, the QRT performance is not as effective as QRT*. In other words, with the SmartSplit splitting node method, the size of the subdomains and the number of nodes in QR*-tree are respectively larger and smaller than that of QR-tree. Accordingly, it is less time required for the server in QRT* to assign larger resident domains to the moving objects than that in QRT. This reduces the frequency at which a new resident domain is searched for each moving object. QRT* takes 50.3%, and 73.3% of the server workload, as compared to MQM and QRT, respectively, for *Uniform*. Meanwhile, QRT* takes 61.7% and 78.4% of the server workload, as compared to MQM and QRT, respectively, for *Skewed*.

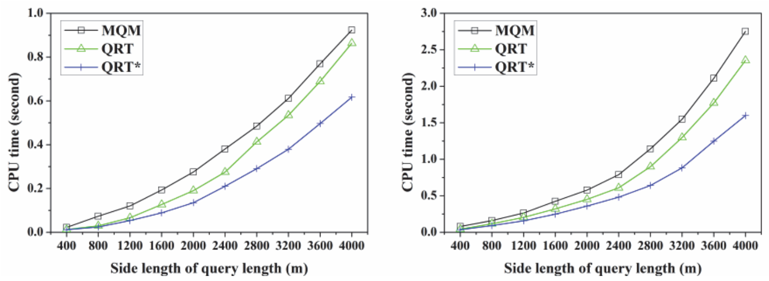


(a) Uniform. (b) Skew.
Fig. 16. Number of messages vs. number of queries.

Fig. 16 shows the effect of the number of query regions on the total number of messages transmitted between the server and moving objects. As the number of query regions increases, the performances of all the methods degrade. However, QRT and QRT* outperform MQM for *Uniform* and *Skewed*. This is because, in MQM, the server assigns small resident domains to moving objects because of the tremendous number of monitoring regions produced. On the other hand, in QRT and QRT*, the server can assign moving objects large resident domains. This leads to a reduction not only in the frequency at which the moving objects contact the server to receive new resident domains, but also in the number of messages the server sends to the moving objects to assign new resident domains. Notably, because the QR-tree and the QR*-tree index queries based on the original query regions instead of monitoring regions, a situation where moving objects send unnecessary messages to update the corresponding query results can be avoided in QRT and QRT*. The communication cost of QRT* is less than that of QRT because the size of subdomains in QRT* is larger, hence the resident domains assigned to moving object are larger than that in QRT, consequently reducing the number of messages to require a new resident domain in QRT*. As shown in Fig. 16, QRT* performs the best in all the cases. As compared to MQM and QRT, QRT* incurs 38.6% and 84.4%, respectively, of the communication cost for *Uniform*. On the other hand, QRT* incurs 49.9% and 79.8% of the communication cost as compared to MQM and QRT, respectively, for *Skewed*.

4.2.2 Effect of the size of query regions

In this simulation, we varied the side length of query regions from 0.4 km to 4 km to examine how the size of query regions affects the performances of MQM, QRT, and QRT*. QRT* performs better and are less sensitive to this parameter than MQM and QRT for *Uniform* and *Skewed* (Fig. 17). As the side length of each query region becomes longer (*i.e.*, the size of each query region becomes larger), an excessive overlap among query regions occurs. This increases the number of monitoring regions in MQM and causes the BP-tree to be split until all the common areas among the query regions are partitioned into a huge number of distinct monitoring regions. As a result, the server in MQM frequently searches a new resident domain for each moving object *o* that exits its small resident domain. The excessive overlap among query regions also increases number of subdomains in QRT, and thus, the server in QRT takes more time to search a resident domain for *o* and the size of this resident domain decreases consequently. On the other hand, the SmartSplit splitting node method helps QRT* keep the size of subdomains as large as possible; therefore, the size of QR*-tree is also kept as small as possible. Under the circumstances, QRT* takes less time to search a resident domain and assigns larger resident domain to *o* than MQM and QRT. As compared to MQM and QRT, QRT* takes 53.4% and

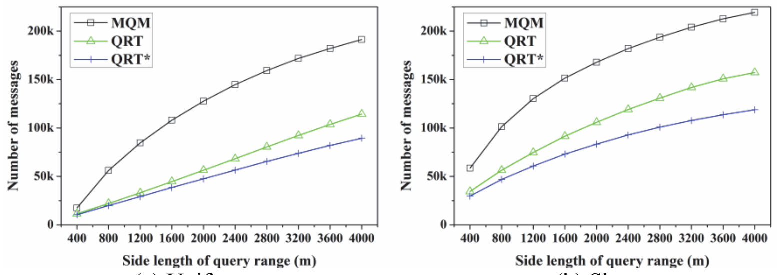


(a) Uniform. (b) Skew.

Fig. 17. CPU-time vs. size of query regions.

76.1%, respectively, of the server workload for *Uniform*. Besides, QRT[*] takes 57.3% and 75.3% of the server workload, as compared to MQM and QRT, respectively, for *Skewed*.

Fig. 18 shows the effect of the side length of each query region (*i.e.*, the size of each query region) on the total number of messages. QRT[*] performs better than MQM and QRT for *Uniform* and *Skewed* for the reason mentioned in the description of the first simulation. On the other hand, MQM performs worst because the longer side length of each query region (*i.e.*, larger size of each query region) negatively affects the performance of MQM. In all cases, QRT[*] achieves the best performance for *Uniform* and *Skewed*. In comparison with MQM and QRT, QRT[*] incurs 41.7% and 84.0%, respectively, of the communication cost for *Uniform*. On the other hand, QRT[*] incurs 50.5% and 79.1% of the communication cost as compared to MQM and QRT, respectively, for *Skewed*.

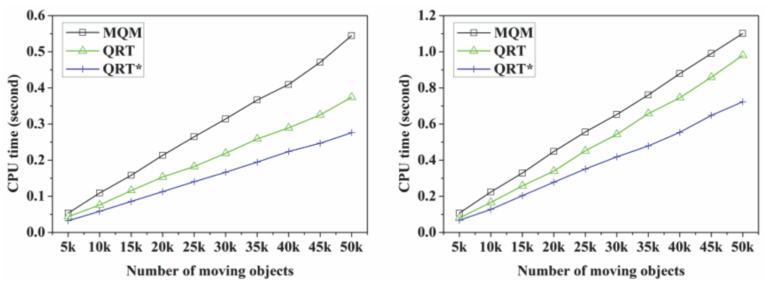


(a) Uniform. (b) Skew.

Fig. 18. Number of messages vs. size of query regions.

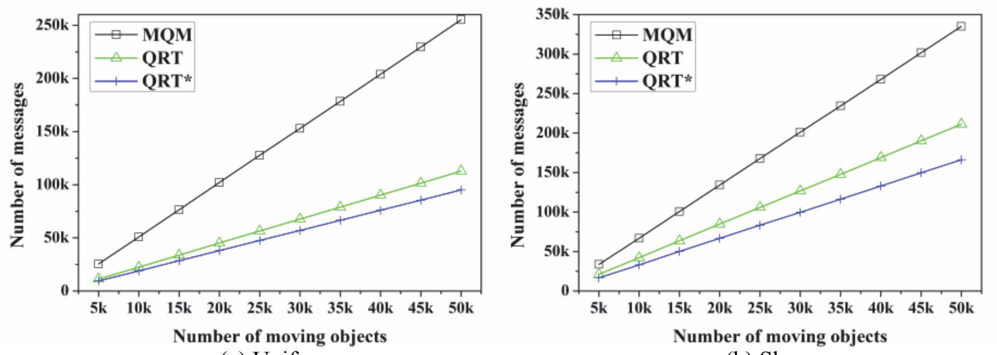
4.2.3 Effect of the number of moving objects

In this simulation, we increased the number of moving objects from 5,000 to 50,000 to study the way the number of moving objects affects the performances of MQM, QRT, and QRT[*]. As shown in Fig. 19 and Fig. 20, when the number of moving objects increases, the overhead of all the methods increases in terms of the CPU-time and the amount of messages transmitted between the server and moving objects. In all cases, QRT[*] outperforms MQM and QRT for the reason mentioned in the description of the first simulation. It is also clear that MQM performs worst regarding server load and communication cost.



(a) Uniform. (b) Skew.

Fig. 19. CPU-time vs. number of moving objects.

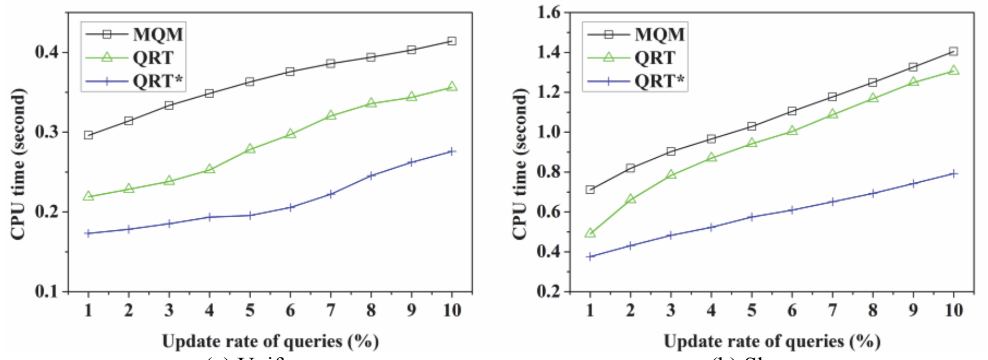


(a) Uniform. (b) Skew.

Fig. 20. Number of messages vs. number of moving objects.

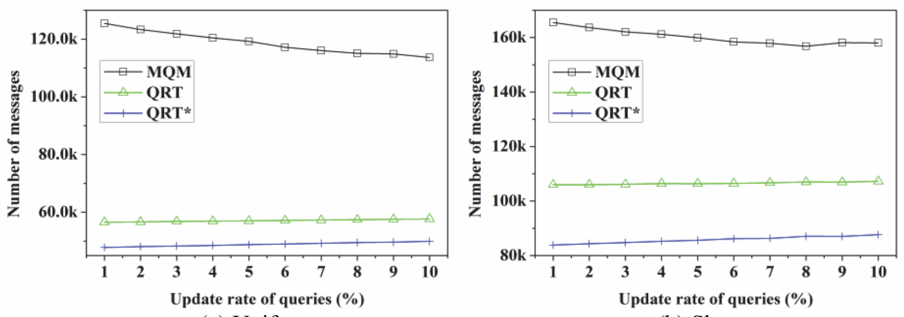
4.2.4 Effect of the update rates of queries

In this simulation, we investigated how the updates (*i.e.*, insertion and deletion) of queries affect the performance of MQM, QRT, and QRT* by increasing update rates (from 1% to 10%) of queries in *Uniform* and *Skewed*. Fig. 21 shows the effect of the update rates of queries on the CPU-time. As can be seen in the figure, QRT* performs much better than MQM and QRT for *Uniform* and *Skewed*. This is because, in MQM, when q is inserted, $q.R$ of q is partitioned into many monitoring regions. The insertion operation of the BP-tree is, therefore, performed for each of these monitoring regions. This increases the CPU-time drastically. The case where an existing query deleted is similar. On the other hand, in QRT (and QRT*), when a q is inserted (or deleted), the insertion (or deletion) operation of QR-tree (and QR*-tree) is performed only once for $q.R$ of q. However, the size of QR*-tree is much smaller than that of QR-tree, consequently, the cost of insertion (or deletion) operation in QR*-tree is less than that in QR-tree. In this figure, QRT* outperforms the other methods in all cases. QRT* takes 58.6% and 74.7% of the server workload, as compared to MQM and QRT, respectively, for *Uniform*. On the other hand, QRT* takes 54.7% and 62.4% of the server workload, as compared to MQM and QRT, respectively, for Skewed.



(a) Uniform. (b) Skew.

Fig. 21. CPU-time vs. update rates of queries.



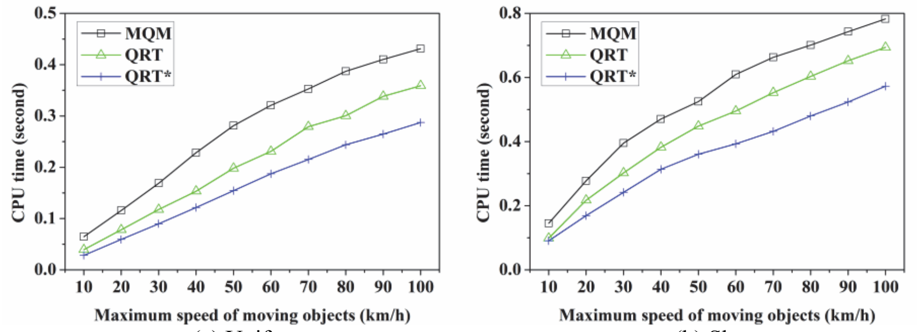
(a) Uniform. (b) Skew.

Fig. 22. Number of messages vs. update rates of queries.

Fig. 22 shows the effect of the update rates of queries on the total amount of messages transmitted. It is notable that MQM approach is worst, and as expected, QRT* achieves the best performance in all cases for *Uniform* and *Skewed*. From our experimental results, when the update rates of queries increase, the number of messages sent from server to moving objects increases in all methods. In QRT and QRT*, the number of messages from moving objects to server increases slightly, while in MQM considerably decreases (because of the effect of deleting queries), and in consequence, the total number of messages in MQM decreases. It can be seen that MQM is the most sensitive method with the update rates of queries.

4.2.5 Effect of the maximum speed of moving objects

Finally, we varied the maximum speed of moving objects from 10 km/h to 100 km/h to investigate the effect of the moving objects' speed on the performances of MQM, QRT, and QRT*.

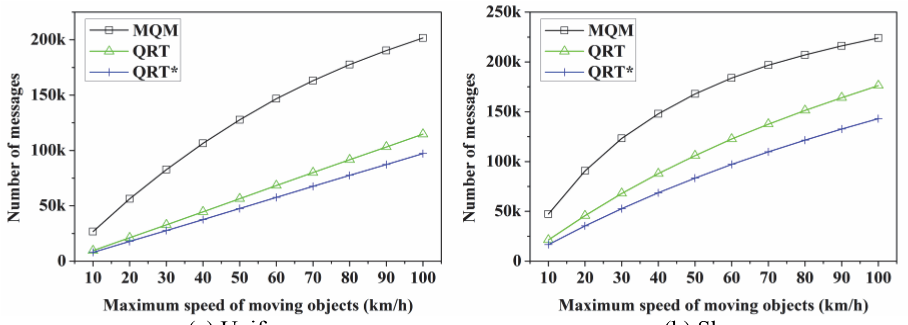


(a) Uniform. (b) Skew.

Fig. 23. CPU-time vs. maximum speed of moving objects.

As shown in Fig. 23, the performances of all the methods in terms of the CPU time degrade as the maximum speed of moving objects increases. The reason is that as the speed of moving objects increases, they may frequently exit their resident domains, and request new resident domains. It can be seen in Fig. 23 that QRT* performs best in all cases. In particular, QRT* takes 57.0% and 77.9% of the server workload, as compared to MQM and QRT, respectively, for *Uniform*. Meanwhile, QRT* takes 66.2% and 81.2% of the server workload, as compared to MQM and QRT, respectively, for *Skewed*.

Fig. 24 indicates the effect of the maximum speed of moving objects on the total amount of messages transmitted. As expected, the performances of all the methods decrease as the maximum speed of moving objects increases. It is also observed from the figure that MQM performs worst, whereas QRT* performs best for both *Uniform* and *Skewed*.



(a) Uniform (b) Skew

Fig. 24. Number of messages vs. maximum speed of moving objects.

5. RELATED WORK

In the past few decades, stationary objects were considered and efficient spatial access methods such as the R-tree [23] and its variants [24-26] were developed broadly. These approaches, however, have retrieve the results only once at a specific time point. In recent years, continuous query evaluation has been extensively attractive so much attention due to LBSs over moving objects. Continuous query evaluation can be categorized into two types which depend on the move of the queries. The first category deals with stationary or quasi-stationary queries over moving objects [1, 5, 10, 15, 17, 18], and the second one focuses on moving queries over moving objects [2, 4, 6, 12-14]. Because our work belongs to the first category, we elaborate on the review of the representative methods in the first category and evaluates the approaches in the latter category shortly. Considering the trajectories of object movements as a priori or predictable, Saltenis *et al.* suggested the Time-Parameterized R-tree (TPR-tree) for indexing moving object [27], where each object's location is transformed into a linear function of time. An upgraded version of the TPR-tree which uses the same data structure as the TPR-tree, called the TPR*-tree, was proposed by Tao *et al.* [28]. In this version, new insertion and deletion algorithm were applied. Further, some index structures such as the STRIPES [28] and the B^x-tree [30], a variant of the B^+-tree, were presented in order to enhance the performance of the TPR-tree family. The know-trajectory assumption, however, does not hold for many real-life LBS scenarios (*e.g.*, the velocity and direction of a typical customer on the road are frequently changed). This causes these index structures to be expensive to maintain.

Because queries remain active for a long period of time and are stationary, indexing queries seems to be a promising approach in comparison with indexing frequently moving objects. Kalashnkov *et al.* [10] proposed the in-memory grid index, while using the R-tree to index queries was suggested by Prabhakar *et al.* [15]. Wu *et al.* [18] used a new query indexing approach called Containment Encoded Square (CES) based indexing. To sum up, it was considered that objects proactively report their location updates to the server whenever they move. In the meantime, the server continually receives the location-update stream, regulates the queries affected by the movements of the objects, and updates their results. However, a remarkable communication bottleneck and the increasing of the workload of determining the affected queries may cause when constant location updates by a large number of objects. Besides, the handheld device carried by each object exhausts the battery life quickly due to the fact that the transmission of a location update message over a wireless connection takes a considerable amount of energy. In order to reduce the frequency of each moving object reporting its location update, the safe region technique was proposed [5, 15]. Cai *et al.* [1] and Jung *et al.* [20] proposed the monitoring query management method (MQM) and the QR-tree method (QRT), respectively, which aim to reduce the communication cost and the server workload by leveraging heterogeneous computational capabilities of moving objects through the concept of resident domain. In recent times, the safe region approach for moving circular range queries over stationary objects was suggested in [2]. Jung *et al.* [20, 31] proposed BQR-tree and GQR-tree, which deal with continuous range queries with specifications for non-spatial attributes.

The Scalable INcremental hash based Algorithm (SINA), which concentrates on the evaluation of continuous moving queries over moving objects, based on the notions of shared execution and incremental evaluation was suggested by Mokbel *et al.* [13]. Gedik *et al.* [4] proposed MobiEyes, where moving objects play an active role in the query evaluation task as in MQM. Liu *et al.* [12] presented two kinds of communication methods for moving query evaluation, on-demand access and periodic broadcasting, to reduce the communication costs and energy waste of handheld devices carried by the objects and the query issuers. In addition, the broadcast grid index (BGI) [14] was discussed by considering that the objects periodically report their location-updates. The method uses periodic broadcasting to communicate between the query issuers and the server in order to evaluate moving queries.

6. CONCLUSIONS

In this paper, we addressed the problem of the efficient and scalable evaluation of continuous range queries (CRQs). Given a set of geographically distributed moving objects, the primary goal of our study is to minimum communication cost and server workload by letting the moving objects evaluate several CRQs that are relevant to them. To achieve this, we proposed a variant of the Query Region tree (QR-tree), namely QR*-tree. We carried out a series of comprehensive simulations and demonstrated that the QR*-tree outperforms the existing methods, validating the effectiveness of the QR*-tree.

ACKNOWLEDGEMENTS

This research project was supported by Ministry of Culture, Sports and Tourism

(MCST) and from Korea Copyright Commission in 2016.

REFERENCES

1. Y. Cai, K.A. Hua, G. Cao, and T. Xu, "Real-time processing of range-monitoring queries in heterogeneous mobile databases," *IEEE Transactions on Mobile Computing*, Vol. 5, 2006, pp. 931-942.
2. M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang, "Continuous monitoring of distance-based range queries," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 23, 2011, pp. 1182-1199.
3. X. Chen, J. Pang, and R. Xue, "Constructing and comparing user mobility profiles for location-based services," in *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, 2013, pp. 261-266.
4. B. Gedik and L. Liu, "Mobieyes: a distributed location monitoring service using moving location queries," *IEEE Transactions on Mobile Computing*, Vol. 5, 2006, pp. 1384-1402.
5. H. Hu, J. Xu, and D. L. Lee, "A generic framework for monitoring continuous spatial queries over moving objects," in *Proceedings of ACM International Conference on the Management of Data*, 2005, pp. 479-490.
6. J. L. Huang and C.-C. Huang, "A proxy-based approach to continuous location-based spatial queries in mobile environments," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 25, 2013, pp. 260-273.
7. S. Ilarri, E. Mena, and A. Illarramendi, "Location-dependent query processing: where we are and where we are heading," *ACM Computing Surveys*, Vol. 42, 2010, pp. 1-73.
8. H. Jung, B. K. Cho, Y. D. Chung, and L. Liu, "On processing location based top-*k* queries in the wireless broadcasting system," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, 2010, pp. 585-591.
9. H. Jung, Y. D. Chung, and L. Liu, "Processing generalized *k*-nearest neighbor queries on a wireless broadcast stream," *Information Science*, Vol. 188, 2012, pp. 64-79.
10. D. V. Kalashnkov, S. Prabhakar, and S. E. Hambrusch, "Main memory evaluation of monitoring queries over moving objects," *Distributed and Parallel Databases*, Vol. 15, 2004, pp. 117-135.
11. K. C. K. Lee, B. Zheng, C. Chen, and C. Y. Chow, "Efficient index-based approaches for skyline queries in location-based applications," *IEEE Transactions on Knowledge and Data Engineering*, Vol. 25, 2013, pp. 2507-2520.
12. F. Liu, K. A. Hua, and F. Xie, "A hybrid communication solution to distributed moving query monitoring systems," *Electronic Commerce Research and Applications*, Vol. 10, 2011, pp. 214-228.
13. M. F. Mokbel, X. Xiong, and W. G. Aref, "SINA: scalable incremental processing of continuous queries in spatio-temporal databases," in *Proceedings of ACM SIGMOD*, 2004, pp. 623-634.
14. K. Mouratidis, S. Bakiras, and D. Papadias, "Continuous monitoring of spatial queries in wireless broadcast environments," *IEEE Transactions on Mobile Computing*, Vol. 8, 2009, pp. 1297-1311.

15. S. Prabhakar, Y. Xia, W. G. Aref, and S. Hambrusch, "Query indexing and velocity constrained indexing: scalable techniques for continuous queries on moving objects," *IEEE Transactions on Computers*, Vol. 51, 2002, pp. 1124-1140.
16. M. Song, H. Choo, and W. Kim, "Spatial indexing for massively update intensive applications," *Information Sciences*, Vol. 203, 2012, pp. 1-23.
17. K. L. Wu, S. K. Chen, and P. S. Yu, "Efficient processing of continual range queries for location-aware mobile services," *Information Systems Frontiers*, Vol. 7, 2005, pp. 435-448.
18. K. L. Wu, S. K. Chen, and P. S. Yu, "On incremental processing of continual range queries for location-aware services and applications," in *Proceedings of MobiQuitous*, 2005, pp. 261-269.
19. X. Ding, X. Lian, L. Chen, and H. Jin, "Continuous monitoring of skylines over uncertain data streams," *Information Sciences*, Vol. 184, 2012, pp. 196-214.
20. H. Jung, Y. S. Kim, and Y. D. Chung, "QR-tree: An efficient and scalable method for evaluation of continuous range queries," *Information Sciences*, Vol. 274, 2014, pp. 156-176.
21. J. Broch, D. A. Maltz, D. Johnson, Y.-C. Hu, and J. Jetcheva, "A performance comparison of multi-hop wireless ad hoc network routing protocols," in *Proceedings of ACM/IEEE MobiCom*, 1998, pp. 85-97.
22. T. Brinkhoff, "A framework for generating network-based moving objects," *GeoInformatica*, Vol. 6, 2002, pp. 153-180.
23. A. Guttman, "R-trees: a dynamic index structure for spatial searching," in *Proceedings of ACM SIGMOD*, 1984, pp. 47-57.
24. N. Roussopoulos and C. Faloutsos, "The R^+-tree: a dynamic index for multi-dimensional objects," in *Proceedings of International Conference on Very Large Database*, 1987, pp. 507-518.
25. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The R^*-tree: an efficient and robust access method for points and rectangles," in *Proceedings of ACM SIGMOD*, 1990, pp. 322-331.
26. I. Kamel and C. Faloutsos, "Hilbert R-tree: An improved R-tree using fractals," in *Proceedings of International Conference on Very Large Database*, 1994, pp. 500-509.
27. S. Saltenis, C. Jensen, S. Leutenegger, and M.A. Lopez, "Indexing the positions of continuously moving objects," in *Proceedings of ACM SIGMOD*, 2000, pp. 331-342.
28. Y. Tao, D. Papadias, and J. Sun, "The TPR^*-tree: an optimized spatio-temporal access method for predictive queries," in *Proceedings of International Conference on Very Large Database*, 2003, pp. 790-801.
29. J. M. Patel, Y. Chen, and V. P. Chakka, "STRIPES: an efficient index for predicted trajectories," in *Proceedings of ACM SIGMOD*, 2004, pp. 635-646.
30. C. S. Jensen, D. Lin, and B. C. Ooi, "Query and update efficient B^+-tree based indexing of moving objects," in *Proceedings of International Conference on Very Large Database*, 2004, pp. 768-779.
31. H. Jung, M. Song, H. Y. Youn, and U. M. Kim, "Evaluation of content-matched range monitoring queries over moving objects in mobile computing environments," *Sensors*, Vol. 15, 2015, pp. 24143-24177.

Tien-Khoi Phan received the B.S. and M.S. degrees in Computer Science and Engineering from Ho Chi Minh City University of Technology, Viet Nam in 2007 and 2011, respectively. He is a Ph.D. candidate in College of Information and Communication Engineering, Sungkyunkwan University, Korea. His research interests include database systems, spatial queries, GIS, and big data.

HaRim Jung received his B.S. degree in Computer Science from Kwangwoon University, Seoul, Korea, in 2004. He received his M.S. and Ph.D. degrees in Computer Science and Engineering from Korea University, Seoul, Korea, in 2007 and 2012, respectively. Currently, he is a research fellow at the School of Information and Communication Engineering, Sungkyunkwan University, Suwon, Korea. His research interests include location-based services and spatial data management in mobile/pervasive environments.

Hee Yong Youn received the B.S and M.S degree in electrical engineering from Seoul National University, Seoul, Korea, in 1977 and 1979, respectively, and the Ph.D. degree in computer engineering from the University of Massachusetts at Amherst, in 1988. Currently, he is Professor of College of Information and Communication Engineering and Director of Ubiquitous computing Technology Research Institute, Sungkyunkwan University, Suwon, Korea. His research interests include cloud and ubiquitous computing, system software and middleware, and RFID/USN.

Ung-Mo Kim received the B.E. degree in Mathematics from Sungkyunkwan University, Korea in 1981 and the M.S. degree in Computer Science from Old Dominion University, U.S.A. in 1986. He received Ph.D. degree in Computer Science from Northwestern University, U.S.A. in 1990. Currently, he is a Professor of College of Information and Communication Engineering, Sungkyunkwan University, Korea. His research interests include data mining, database security, data warehousing, GIS, and big data.