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
2066 Seobu-ro, Jangan-gu, Suwon 440-746, Korea
E-mail: khoiphan@skku.edu, harim3826@gmail.com, youn7147@skku.edu, ukim@skku.edu

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 Query Region tree (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, 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 object, location sensing, location-update stream, location-based service, range monitoring query, 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-
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.

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, \( o.\text{Cap} \), which indicates the maximum number of (nearby) monitoring regions \( o \) can load and process at a time. Assuming the capability \( o.\text{Cap} \) 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].

![Fig. 2. Example of center split in QR-tree](image)

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 $\text{SplitNode}$ (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, $QR^*$-tree, to overcome the above drawbacks of QR-tree. $QR^*$-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 $QR^*$-tree is less than that in 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 (Fig. 3).

![Fig. 3. System Overview [20]](image)

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 \(\theta\) 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

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. 4a), is covered by (Fig. 4b), partially intersects (Fig. 4c), and equals (Fig. 4d). 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 \((ptr, \tilde{N})\) stored in a non-leaf node \( N \), \( \tilde{N} \) 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.
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 \( \ell \) that splits \( N \) the splitting line (SL). The SL \( \ell \) can be horizontal \((\ell_y \text{ is the } y\text{-coordinate of } \ell)\) or vertical \((\ell_x \text{ is the } x\text{-coordinate of } \ell)\). Our split strategy is based on four rules:

First, the subdomains must have an overlap relationship with at least one query region, i.e., \( N_1.\text{Count} > 0 \) and/or \( N_1.\text{CL} \neq \text{null} \), and \( N_2.\text{Count} > 0 \) and/or \( N_2.\text{CL} \neq \text{null} \). 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 inefficient. Fig. 6a and Fig. 6b show examples of vertical SLs and horizontal SLs, respectively. In the figures, \( \ell_1, \ell_5, \ell_6 \) and \( \ell_8 \) are invalid SLs, while \( \ell_2, \ell_3, \ell_4 \) and \( \ell_7 \) 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.
Definition 1: (Suboptimal SL). A suboptimal SL is a SL, which makes $N_1.\text{Count} \leq t$ and $N_2.\text{Count} \leq t$.

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. 7a, there is one suboptimal SL $\ell$, while in Fig. 7b, there are uncountable suboptimal SLs $\ell_i$ between $[\ell_1, \ell_2]$ (i.e., $\ell_{i,x} \leq \ell_{r,x} \leq \ell_{2,x}$). If there is one suboptimal SL in the (sub) domain (e.g., Fig. 7a), 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. 7b), the third rule and the fourth rule will be considered among them.

![Fig. 7. Examples of suboptimal SLs](image)

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: $N_1.\text{Count} = N_2.\text{Count} = N.\text{Count}/2$. 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 $\text{VarQ of } \ell$) is considered foremost; the smaller $\text{VarQ}$, the better. If there are many SLs with the same $\text{VarQ}$, the total number of queries in $N_1$ and $N_2$ (denoted by $\text{SumQ of } \ell$) is considered subsequently; the smaller $\text{SumQ}$, the better. If $\text{VarQs}$ of the SLs are equal, the fourth rule is evaluated. $\text{VarQ}$ and $\text{SumQ}$ of the SL $\ell$ are defined as follows:

$$\text{VarQ}(\ell) = (N_1.\text{Count} - N.\text{Count}/2)^2 + (N_2.\text{Count} - N.\text{Count}/2)^2$$

$$\text{SumQ}(\ell) = N_1.\text{Count} + N_2.\text{Count}$$

Fig. 8a 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 $\ell_3$ is chosen to split $N$ because its $\text{VarQ}$ is minimum (i.e., 0). In Fig. 8b, assuming $t = 2$, there are two suboptimal SLs, $\ell_1$ and $\ell_2$. Because $\text{VarQ}(\ell_1) = \text{VarQ}(\ell_2) = 0.5$, $\text{SumQ}$ is computed. The SL $\ell_2$ is chosen to split $N$ because $\text{SumQ}(\ell_1) = 4$ and $\text{SumQ}(\ell_2) = 3$.

Fourth, the area of two subdomains $N_1$ and $N_2$ tends to be equal. The best case is: $N_1.\text{Area} = N_2.\text{Area} = N.\text{Area}/2$, where $N.\text{Area}$ 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 $\ell$) is used to evaluate the fourth rule; the smaller $VarA$, the better. $VarA$ of a SL $\ell$ is defined as:

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

In the example in Fig. 7b, there are many suboptimal SLs between $[\ell_1, \ell_2]$. 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 $\ell_2$ is chosen, because its $VarA$ is minimum.

Fig. 8. Examples of choosing SLs based on VarQ and SumQ

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 $\ell$ that sweeps from the left edge to the right edge of $N$. It is clear that when $\ell$ passes over the left edge or the right edge of a query region $q.R$, the number of queries on the subdomains created by $\ell$ may be changed. In particular, if $\ell$ passes over the left edge of $q.R$ (or $\ell$ enters $q.R$), the number of queries in the left subdomain (i.e., $N_1$) created by $\ell$ is increased. If $\ell$ passes over the right edge of $q.R$ (or $\ell$ 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. In Fig. 9a, assuming $t = 1, \ell_1, \ell_2, \ell_3$ and $\ell_4$ 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 $\ell_1$ at the left edge of $N$ becomes the candidate SL. It can be observed that every vertical SL $\ell_i$ between two continuous SLs $\ell_2$ and $\ell_3$ 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 $\ell_i$ between $[\ell_3, \ell_2]$, 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 identifi-
er of $q_2$ is added to the covering list of $N_2$.

![Diagram of Candidate SLs and Covering Query Region](image)

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

**Proof**. We prove this lemma by contradiction. Let us assume that $\text{VarQ}(\ell_j) - \text{VarQ}(\ell_i) < 0$ and $\text{VarQ}(\ell_j) - \text{VarQ}(\ell_i) < 0$. The subdomains are created by $\ell_i$, $\ell_j$ and $\ell_k$ are $(N_1^i, N_2^i)$, $(N_1^j, N_2^j)$ and $(N_1^k, N_2^k)$, respectively. Without the loss of generality, we define:

- $N_1^i$.Count = $x$
- $N_2^i$.Count = $y$
- $N_1^j$.Count = $x + a$
- $N_2^j$.Count = $y - b$

where $a$ is the number of query regions that have the left edges overlap with $\ell_i$ and $b$ is the number of query regions that have the right edges overlap with $\ell_j$. 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:

- $\text{VarQ}(\ell_j) - \text{VarQ}(\ell_i) = a(a + 2x - N)$
- $\text{VarQ}(\ell_j) - \text{VarQ}(\ell_i) = b(-b + 2y - N)$

By combining with our assumption, we obtain:

- $N > a + 2x$  \hspace{1cm} (4)
- $N > -b + 2y$  \hspace{1cm} (5)

With the SLs $\ell_i$ and $\ell_j$, we have:

- $x \geq N - y$  \hspace{1cm} (6)
- $x + a \geq N - y + b$  \hspace{1cm} (7)

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

- $2N > 2x + 2y + a - b$  \hspace{1cm} (8)

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

- $2N \leq 2x + 2y + a - b$  \hspace{1cm} (9)

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

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

**Lemma 2**: Given two continuous candidate SLs $\ell_i$ and $\ell_j$, $\text{SumQ}(\ell_i) \leq \text{SumQ}(\ell_k)$ and $\text{SumQ}(\ell_j) \leq \text{SumQ}(\ell_k)$, where $\ell_k$ is a SL between $(\ell_i, \ell_j)$.

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

$N_1^k.\text{Count} = x$, $N_2^k.\text{Count} = y$.

We have:

$N_1^i.\text{Count} = x + a$, $N_2^i.\text{Count} = y$.

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

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

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

In Fig. 10b, assuming $t = 3$, $\ell_c$ is a central SL between two continuous candidate SLs, $\ell_i$ and $\ell_j$.

Fig. 11a shows an example of covering query regions overlap two continuous candidate SLs, $\ell_i$ and $\ell_j$ ($t = 3$). In this case, $q_1.R$ and $q_3.R$ cover all SLs between $[\ell_i, \ell_j]$. We can see that $q_1.R$ and $q_3.R$ do not effect to the values $\text{VarQ}$, $\text{SumQ}$ of the SLs between $[\ell_i, \ell_j]$. In Fig. 11b, assuming $t = 3$, $q_1.R$ and $q_3.R$ touch the candidates SLs, $\ell_i$ and $\ell_j$, respectively. It is clear that the covering query regions help reduce the Count value of

---

1 In this paper, we use $N.\text{Left}$ and $N.\text{Right}$, the x-coordinates of the left edge and the right edge of $N$, respectively; $N.\text{Bottom}, N.\text{Top}$ are the y-coordinates of the bottom edge and the top edge of $N$, respectively.
subdomains created by \( \ell_i \) and \( \ell_j \) (i.e., \( N_1^j \) and \( N_2^j \)). The covering query regions have more positive influence on the candidate SLs than the SLs between candidate SLs.

![Covering query regions](image)

Fig. 11. Effect of covering query regions to 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 \text{SweepDomain}, describes the process of finding the SLs. The \text{SweepDomain} has three steps as follows.

### Algorithm 1 \text{SweepDomain}

**Input** \( N \): a domain

**Output** \text{SLList}: a list of splitting lines

1. Initialize \text{CQRlist}, \text{SLlist}, binary search tree \text{BST}
2. for each query region \( QR \) of each \( qid \) stored in \( N \) do
3.  if \((QR\_Left \leq N\_Left)\) then
4.      create new candidate position \( p = \langle N\_Left, 1, 0 \rangle \)
5.      if \((QR\_Bottom \leq N\_Bottom \text{ and } QR\_Top \geq N\_Top)\) then
6.          insert \( QR \) to \text{CQRlist}
7.  else
8.      create new candidate position \( p = \langle QR\_Left, 1, 0 \rangle \)
9.      if \( p \) is not in \text{BST} then
10.     insert \( p \) into \text{BST}
11.  else
12.     increase \text{LeftCount} of \( p \) in \text{BST} by 1
13.  if \((QR\_Right \geq N\_Right)\) then
14.     create new candidate position \( \hat{p} = \langle N\_Right, 0, 1 \rangle \)
15.     if \((QR\_Bottom \leq N\_Bottom \text{ and } QR\_Top \geq N\_Top)\) then
16.         insert \( QR \) to \text{CQRlist}
17.  else
18.     create new candidate position \( \hat{p} = \langle QR\_Right, 0, 1 \rangle \)
19.     if \( \hat{p} \) is not in \text{BST} then
20.         insert \( \hat{p} \) into \text{BST}
21.  else
22.     increase \text{RightCount} of \( \hat{p} \) in \text{BST} by 1
23. for each candidate position \( cp \) in \text{BST} in in-order traversal do
24.  if \((\text{pre}_\_cp \text{ is null})\) then
25.     \text{num\_left} = \text{cp\_LeftCount}
26.     \text{num\_right} = \text{cp\_RightCount}
27.  else
28.     if \((\text{pre}_\_cp\_Pos < (N\_Left + N\_Right)/2 < \text{cp\_Pos})\) then
29.         create a new SL \( \ell_c = \langle v, (N\_Left + N\_Right)/2, \text{num\_left}, N\_Count - \text{num\_right} \rangle \)
30.         insert \( \ell_c \) into \text{SLList}
31.     \text{num\_right} = \text{num\_right} + \text{cp\_RightCount}
32.     create a new SL \( \ell = \langle v, \text{cp\_Pos}, \text{num\_left}, N\_Count - \text{num\_right} \rangle \)
33.     insert \( \ell \) into \text{SLList}
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 $<Type, Pos, Count1, Count2>$, where $Type$ states the SL horizontally or vertically ($h$ indicates the horizontal SL, $v$ indicates the vertical SL), $Pos$ is the coordinate of SL, $Count1$ is the number of query regions that overlap $N_1$, $Count2$ is the number of query regions that overlap $N_2$. SweepDomain calculates $Count1$ and $Count2$ 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 $\ell$ in $SLlist$, if there is a query region $QR$ that covers or equalizes to the subdomain $N_1$, SweepDomain decreases $\ell.Count1$ by 1 (lines 39-40). If $QR$ covers or equalizes to the subdomain $N_2$, SweepDomain decreases $\ell.Count2$ by 1 (lines 40-41). After the third step, the value $\ell.Count1$ (or $\ell.Count2$) is equal to the value $N_1.Count$ (or $N_2.Count$); accordingly, the SL $\ell$ 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 \( \ell \), if there is no temporary best SL, \( \ell \) becomes a temporary best SL (lines 2-3). Otherwise, EvaluateSL checks if \( \ell \) is a suboptimal SL. First, if \( \ell \) is a suboptimal SL and there is no suboptimal SL earlier, \( \ell \) 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 \( \text{VarQ} - \text{SumQ} - \text{VarA} \). If \( \ell \) is better than the old one, \( \ell \) replaces the old one to become the new temporary best SL (lines 9-16). Second, \( \ell \) is not a suboptimal SL, EvaluateSL only evaluates \( \ell \) 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, \( \text{VarQ} - \text{SumQ} - \text{VarA} \) (lines 18-26). EvaluateSL returns the best SL in accordance with the rules in Section 3.1.

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

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.

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.

**Algorithm 3 SmartSplit**

```
Input N: a domain
Output <N1, N2>: two subdomains N1 and N2
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 N1 = <N.Left, N.Bottom, best_sl.Pos, N.Top>
7. create a subdomain N2 = <best_sl.Pos, N.Bottom, N.Right, N.Top>
8. else
9. create a subdomain N1 = <N.Left, N.Bottom, N.Right, best_sl.Pos>
10. create a subdomain N2 = <N.Left, best_sl.Pos, N.Right, N.Top>
11. return <N1, N2>
```

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 25 km x 25 km 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.

![Fig. 13. Number of subdomains vs. number of queries](image)

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 sim-
ulation time steps and measured the average of the CPU-time (in second) and total number of messages.

Table 1 Simulation parameters and their values

<table>
<thead>
<tr>
<th>Simulation parameter</th>
<th>Value used (Default)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cardinality of Uniform/Skewed</td>
<td>1000 – 10,000 (5000)</td>
</tr>
<tr>
<td>Side length of query regions</td>
<td>0.4 – 4 km (2.0 km)</td>
</tr>
<tr>
<td>Number of moving objects</td>
<td>5,000 – 50,000 (25,000)</td>
</tr>
<tr>
<td>Update rates of queries</td>
<td>1 – 10% (0%)</td>
</tr>
<tr>
<td>Maximum speed of moving objects</td>
<td>10 – 100 km/h (50 km/h)</td>
</tr>
</tbody>
</table>

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 25 km x 25 km. 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.

Fig. 14. Performances on RWM and NBG

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.
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.

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

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.

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.
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.

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.

![Fig. 21. CPU-time vs. update rates of queries](image1)

![Fig. 22. Number of messages vs. update rates of queries](image2)

### 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*. 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*. 
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*-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
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