QR-tree: An efficient and scalable method for evaluation of continuous range queries

HaRim Jung, Yong Sung Kim, Yon Dohn Chung

Abstract

In this paper, we explore the problem of the scalable evaluation of continuous range queries (CRQs) over moving objects, each of which continually retrieves the moving objects that are currently located within a query region of interest. Most existing methods assume that moving objects continually communicate with the server to report their current locations and the server continuously updates the results of queries. However, such an assumption degrades the system performance, because the communication cost is huge and the server workload is increased when the number of moving objects and queries is enormous.

In this paper, we propose a novel query indexing structure, referred to as the Query Region tree (QR-tree), which allows the server to cooperate with moving objects efficiently by leveraging the available computational resources of the moving objects to improve the overall system performance. In addition, we present another version of the QR-tree, called the Bit-vector Query Region tree (BQR-tree), for the evaluation of CRQs that specify additional non-spatial selections. The BQR-tree stores a summary of the non-spatial information specified by CRQs in the form of bit-vectors. Through a series of comprehensive simulations, we verify the efficiency of the QR-tree and the BQR-tree in terms of the communication cost and server workload.

1. Introduction

With the technological advances in wireless networks and the development of navigation systems, such as the global positioning system (GPS), location-based services (LBSs) have attracted much attention as one of the most promising applications in mobile/ubiquitous computing environments [3–5,7,9–19,22,25,27,28]. Many useful LBSs, including mobile advertising and emergency monitoring, usually rely 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 a client specifies. Consider the scenario of a mobile advertising service as an example, where a restaurant (i.e., client) plans to send e-discounts to the nearby potential customers (i.e., moving objects) who have opted into the mobile advertising service. Then, the service provider (i.e., server) must be able to keep track of the locations of the customers and report their proximity to the restaurant, whenever needed. A commercial example of such a mobile advertising service is ShopAlerts supplied by the location-based service provider.
AT&T. ShopAlerts enables an advertiser (e.g., restaurant) to specify a region of interest. When any customer who has opted into the service enters this region, she will be alerted to any valuable information (e.g., e-discounts) offered by the advertiser.

In many real-life LBSs, however, clients of diverse interests often additionally constrain the target moving objects by specifying their non-spatial selection criteria. Continuing with the above example, suppose the restaurant focuses on attracting only a specific class of customers (e.g., 30 ≤ Age ≤ 40, $40,000 ≤ Annual income ≤ $80,000 and Dietary preference = Vegetarian). In this case, the service provider should report to the restaurant only the nearby customers whose profiles match the above criteria. In this paper, we investigate a set of methods for scalable evaluation of conventional CRQs, as well as CRQs with non-spatial selections.

There is a large body of work on CRQ evaluation over moving objects, which can be classified into two categories according to the mobility of query regions: one deals with stationary or quasi-stationary query regions [3,9,15,22,27,28], whereas the other deals with moving query regions [4,7,10,17–19]. Our study belongs to the former category. The majority of existing methods for CRQ evaluation assume that moving objects periodically send location updates to the server via wireless connections and the server keeps the results of the registered queries up to date [15,18,27,28]. However, when the server involves a large number of moving objects and queries, the overall system performance may deteriorate drastically due to a severe communication bottleneck and overwhelming server workload [6].

The safe region technique, which helps reduce the frequency at which the moving objects send location updates, was introduced in [9,22]. The safe region, assigned to each moving object o, is the area that (i) contains o and (ii) guarantees that the current results of all the queries will remain valid as long as o does not exit it. Therefore, 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 improves the overall system performance to a certain degree, because the size of a safe region assigned to each object o is typically small, o easily exits its current safe region and contacts the server in order to receive a new safe region. Thus, the server must frequently determine o’s safe regions, which requires intensive computation.

Monitoring Query Management (MQM), which aims to reduce the communication cost and server workload by leveraging the available (memory and computational) capabilities of moving objects, was introduced in [3]. MQM splits the rectangular workspace into a set of subdomains. In particular, if the number of query regions on the workspace exceeds a threshold τ, the workspace is split into two subdomains of equal size. When a query region q.R overlaps with a subdomain N, the common area between q.R and N (i.e., q.R ∩ N) is called the monitoring region. The split process recursively continues until every subdomain N has no more than τ monitoring regions that overlap with N. In MQM, the binary partitioning tree (BP-tree) and the additional data structure are used for indexing queries based on monitoring regions. Assuming τ = 1, Fig. 1 shows an example of the workspace split for query regions q₁, R ~ q₅, R.

MQM assigns each moving object o (i) a rectangular collection of subdomains, referred to as the resident domain that contains o and (ii) all the 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 is measured by the number of monitoring regions o can load and process at a time. For example, suppose the capability o.Cap of the moving object o in Fig. 1 is 3. Then, o is assigned (i) the rectangular collection of subdomains N₂₁₁, N₂₁₂, and N₂₁₂ as its resident domain and (ii) three monitoring regions R₄₁, R₄₂, and R₄₃, which overlap with o’s resident domain.

Only when each moving object o exits its resident domain or crosses any of its assigned monitoring regions, does it contact the server in order to receive a new resident domain (together with new monitoring regions) or let the server update the query results (if necessary). As such, in MQM, moving objects and the server share the process of CRQ evaluation, which

Fig. 1. Example of the workspace split in MQM.
lightens the server workload. Since the moving objects know exactly when they should contact the server, the communication cost can be also reduced. However, MQM has the following limitations:

- First, when a query region overlaps with many subdomains, it may produce a substantial number of monitoring regions. This leads MQM to assign a very small resident domain to each moving object \( o \) because its capability \( \text{Cap} \) is measured against a large number of monitoring regions instead of a much smaller number of original query regions. As a result, \( o \) may have to frequently contact the server to receive new resident domains. Assuming default parameter settings in our performance evaluation, Fig. 2 shows the number of monitoring regions produced according to the number of query regions (see Table 1 in Section 4 for details).

- Second, because each moving object \( o \) checks its movement against the monitoring regions without the knowledge of the original query regions, \( o \) may unnecessarily contact the server to update the query results. For example, when the moving object \( o \) in Fig. 1 moves from monitoring region \( R_{41} \) to monitoring region \( R_{42} \), it sends two notification messages to the server, one that it has exited \( R_{41} \) and the other that it has entered \( R_{42} \). However, \( o \) does not cross the original query region \( q_4 \); thus, the movement of \( o \) does not affect the result of query \( q_4 \).

In this paper, we introduce a novel index structure, called the Query Region tree (QR-tree), to overcome the above limitations of MQM. The QR-tree indexes queries based on the original query regions instead of the monitoring regions. Compared to the BP-tree used in MQM, the QR-tree allows the server to assign each moving object \( o \) a much larger resident domain together with the original query regions. Consequently, the frequency at which \( o \) contacts the server to receive new resident domains or update the query results is reduced. In addition, for CRQs that involve non-spatial selections, we present another version of the QR-tree, referred to as the Bit-vector Query Region tree (BQR-tree), which stores the additional bit-vector information required to describe the non-spatial information specified by the queries.

The remainder of this paper is organized as follows. In Section 2, the system overview is provided. In Section 3, we present the QR-tree and the BQR-tree, and explain how the server and the moving objects can complement each other to evaluate CRQs. In Section 4, we provide the performance evaluation and verify the efficiency of our proposed methods. In Section 5, some related work is reviewed. Finally, in Section 6, we present our conclusions.

![Fig. 2. Number of monitoring regions (\( t = 25 \)).](image)

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>0.5–5 km (2.5 km)</td>
</tr>
<tr>
<td>Number of moving objects</td>
<td>20,000–200,000 (100,000)</td>
</tr>
<tr>
<td>Max. speed of moving objects</td>
<td>10–100 km/h (50 km/h)</td>
</tr>
<tr>
<td>Update rates of queries</td>
<td>1–10% (0%)</td>
</tr>
</tbody>
</table>

Please cite this article in press as: H. Jung et al., QR-tree: An efficient and scalable method for evaluation of continuous range queries, Inform. Sci. (2014), http://dx.doi.org/10.1016/j.ins.2014.02.061
2. System overview

The goal of our study is to design a system in which the server and moving objects share the evaluation process of CRQs. To achieve this, we use the resident domain concept. Similarly to the models presented in most existing work [3, 9, 10, 22, 27, 28], the system model we consider consists of a set of moving objects, clients who issue queries, and the central server. Fig. 3 illustrates the system model.

The moving objects and the clients do not communicate directly, but use the server as an intermediary. Each moving object \( o \) is aware of its location (e.g., is equipped with a GPS receiver), and has some available (memory and computational) capability, \( o.\text{Cap} \). We assume that each moving object \( o \) has heterogeneous capability \( o.\text{Cap} \), which is measured by the number of query regions it can load and process at a time, and that \( o.\text{Cap} \geq \theta \), where \( \theta \) is a system parameter that indicates the minimum number of query regions each moving object should be able to process; thus, a moving object with more powerful capability is assigned a larger resident domain together with a greater number of original query regions. It should be noted that, because it is beyond the scope of this paper to explain how the capabilities of moving objects are measured, we assume that the system has benchmark-based descriptions of the capabilities of moving objects. 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 \( q.R \). The former is for the purpose of receiving a new resident domain, whereas the latter is to allow the server to update the result of the corresponding query \( q \). Each client is able to issue multiple queries to the server and receives the results of these queries from the server through wireless or high-speed wired 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. The following three main tasks are performed by the server. The details of the cooperation between the server and the moving objects to evaluate queries will be covered in the next section:

- **Query registration and de-registration:** When a new query \( q \) is issued (or \( q \) is terminated) by a client, the task of query registration (or de-registration) is performed, which consists 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:** The task of resident domain assignment is performed in response to the registration of a new moving object or the message sent by a moving object that exits its current resident domain. A new resident domain is searched by traversing the QR-tree. Then, it is broadcast with several nearby query regions and an object identifier \( oid \).
- **Query result update:** The task of query result update is performed mainly in response to the message sent by a moving object whose movement crosses one of its assigned query regions \( q.R \). Then, 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.

3. The proposed index structures

In this section, we first propose the QR-tree, which aims to assign (i) the largest possible resident domain and (ii) the original nearby query regions to each moving object \( o \). This helps reduce the frequency at which \( o \) contacts the server to receive new resident domains or update the query results. Consequently, the overall performance of the system for evaluating CRQs is greatly improved as compared with that of the safe region technique [9] and MQM [3] in terms of the communication cost and server workload. Then, we present another version of the QR-tree, namely, the BQR-tree, which is specialized for evaluating CRQs with non-spatial selections. The cooperation between the server and the moving objects for query evaluation is also described in this section.
3.1. The Query Region tree (QR-tree)

3.1.1. Description

Before we present the QR-tree, we classify the overlap relationship between a query region \( q.R \) and a (sub) domain \( N \) into four categories as shown in Fig. 4: covers (see Fig. 4a), is covered by (see Fig. 4b), partially intersects (see Fig. 4c), and equals (see Fig. 4d).

The QR-tree is a binary tree index of queries, which is built by splitting the entire workspace recursively. 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 continues 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. The threshold value \( t (\geq 0) \) is determined by the moving object with the minimum capability among all the moving objects registered at the server.

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 \( ptr \). Hereafter, without ambiguity, we use the symbol ‘\( N \)’ to denote both a tree node and its corresponding (sub) domain. The QR-tree satisfies the following properties:

1. A query identifier \( qid \) of a query \( q \) is stored in a leaf node \( N \) 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 \( (ptr; \bar{N}) \) stored in a non-leaf node \( \bar{N} \), \( \bar{N} \) represents one of the equal halves of \( \bar{N} \)’s domain.
4. Every (leaf and non-leaf) node additionally stores a variable \( Count \) and is associated with a special list, called the covering list (CL).

**Definition 1** (Count). Given a set of queries \( Q \) and a (leaf or non-leaf) node \( N \) of the QR-tree, the value of a count variable, \( N.Count \), stored in \( N \) is determined as

\[
N.Count = \sum_{q \in Q} \text{Check}_{bc}(q.R),
\]

where \( \text{Check}_{bc}(q.R) \) returns 1 iff \( q.R \) is covered by or partially intersects \( N \), otherwise, it returns 0.

**Definition 2** (Covering list (CL)). Given a set of queries \( Q \) and a (leaf or non-leaf) node \( N \) of the QR-tree, \( N \)’s associated covering list, \( N.CL \), is a list that stores the query identifier \( qid \) of every query \( q (\in Q) \) whose query region \( q.R \) covers or equals \( N \).

---

2 In this paper, a domain is assumed to be split along its longer dimension. When two dimensions of the domain are equal in length, it is assumed to be split horizontally.
The benefits of associating a covering list \( N \text{CL} \) with each QR-tree node \( N \) are twofold. First, the server can assign \( N \) to a qualifying moving object \( o \) as its resident domain, together with only the query regions that are covered by or partially intersect \( N \), because the query regions that cover or equal \( N \) need not be assigned to \( o \) based on the following observation:

**Observation 1.** Given a QR-tree node \( N \) and a query \( q \) whose identifier \( \text{qid} \) is stored in \( N \text{CL} \) (i.e., query \( q \) whose query region \( q.R \) covers or equals \( N \)), every moving object \( o \) that currently resides in \( N \) is guaranteed not to affect the result of \( q \) as long as it is moving within \( N \) (see Fig. 5b).

Therefore, in the QR-tree method, the server can assign a larger resident domain to each moving object \( o \) because its capability \( o.Cap \) can be measured against the number of \((\text{only})\) original query regions that are covered by or partially intersect each QR-tree node \( N \). As we will describe later, the server assigns \( N \) to \( o \) as its resident domain if \( N \) contains \( o \)'s current location and \( N.Count \leq o.Cap \). It is important to note that the result update of each query \( q \) whose identifier \( \text{qid} \) is stored in \( N \text{CL} \) is additionally handled at the server (if necessary) when \( o \) contacts the server to receive a new resident domain.

Second, the case where the common area among \( t+1 \) query regions overlaps with a leaf node \( N \) can be efficiently resolved. In such a case, the split process of \( N \) may never terminate, which is one of the main reasons why MQM constructs the BP-tree based on monitoring regions. Assuming \( t = 1 \), Fig. 5a shows an example where the common area between the query regions \( q_1, R \) and \( q_2, R \) (i.e., \( q_1.R \cap q_2.R \)) overlaps with the leaf node \( N \). In MQM, \( N \) is recursively split until \( q_1.R \cap q_2.R \) is partitioned into the distinct monitoring regions \( R_{13} \) and \( R_{14} \) (see Fig. 5b). On the other hand, in the QR-tree method, \( N \) is recursively split until one of \( t+1 \) query regions \( q.R \) covers or equals its descendent node \( N \), because the query identifier \( \text{qid} \) of the corresponding query \( q \) is not stored in \( N \) (by the Property 1). Then, \( \text{qid} \) is inserted into \( N \text{CL} \) (by Definition 2). As shown in Fig. 5c, \( N \)'s descendent nodes \( N_{112} \) and \( N_{12} \) are not further split although \( q_1.R \cap q_2.R \) overlaps with them, because \( q_1.R \) and \( q_2.R \) cover \( N_{112} \) and \( N_{12} \), respectively. Then, the query identifiers of the corresponding queries \( q_1 \) and \( q_2 \) are inserted into \( N_{112} \text{CL} \) and \( N_{12} \text{CL} \), respectively. Fig. 6 shows an example of the QR-tree for the query regions \( q_1.R \sim q_3.R \) shown in Fig. 1, assuming \( t = 1 \).

### 3.1.2. Cooperative evaluation of CRQs

When a new moving object is registered at the server, the search algorithm for its resident domain is invoked. **Algorithm 1** is the pseudocode of the search algorithm on the QR-tree. Given a moving object \( o \) with its current location and capability \( o.Cap \), the search algorithm starts from the root and recursively visits only the nodes that contain \( o \)'s current location until reaching the node \( N \) such that \( N.Count \leq o.Cap \). Now, \( N \) becomes \( o \)'s resident domain.

Then, the algorithm invokes **FINDQUERYREGIONS**, which is a simple depth-first search algorithm that takes \( N \) as an input and retrieves all the distinct query identifiers stored in each \( N \)'s descendent leaf node \( N \) and its associated covering list \( N \text{CL} \). It should be noted that the query identifier \( \text{qid} \) of every query \( q \) stored in \( N \text{CL} \) must be retrieved because, although the query region \( q.R \) of \( q \) covers or equals \( N \), it may be covered by or partially intersect \( N \) (by Definition 2); if \( \text{qid} \) is stored in both \( N \text{CL} \) and \( N \text{CL} \), **FINDQUERYREGIONS** filters it out because \( q.R \) covers or equals \( N \). In the worst case, \( N \) may be a leaf node. In this case, **FINDQUERYREGIONS** retrieves only the query identifiers stored in \( N \).

**Algorithm 1. Search**(\( N, o \))

```plaintext
Input: \( N \): A QR-tree node initially set to the root, \( o \): A moving object
Output: \( R \): \( o \)'s resident domain, \( Q \): A set of distinct query identifiers
1: Initialize an empty set \( Q \);
2: if \( N.Count \leq o.Cap \) then
3: \( R = N \);
4: \( Q = Q \cup \text{FINDQUERYREGIONS}(N) \);
5: Return \( R \) and \( Q \);
6: else // \( N.Count > o.Cap \)
7: Find the entry (\( \text{ptr}, \hat{N} \)) stored in \( N \) such that \( \hat{N} \) contains \( o \)'s current location;
8: Search(\( \hat{N}, o \)); // Recursion
9: end if
```

After **Algorithm 1** terminates, the server searches all the queries (in the query table) referred to by the retrieved query identifiers and assigns the moving object \( o \) its resident domain \( N \) together with query identifier and query region pairs. For example, the server assigns the moving object \( o \) with \( o.Cap = 3 \) shown in Fig. 6 node \( N_2 \), together with query identifier and query region pairs \((q_3, q_1.R), (q_4, q_4.R), \) and \((q_5, q_5.R)\).
Now, we describe how the moving object \textit{o} in Fig. 6 cooperates with the server to evaluate CRQs. Suppose that the movement of \textit{o} crosses \textit{q}_1.R, as shown in the figure. Then, \textit{o} sends an UpdateResult(\textit{oid}, \textit{oloc}, \textit{q}_5, \textit{q}_3, \textit{R}) message to the server, where \textit{oloc} is the current location of \textit{o}. In response, the server updates the result of \textit{q}_5 stored in the query table, i.e., inserts \textit{q}_5 into the result of \textit{q}_o. On the other hand, if \textit{o} exits its current resident domain \textit{N}_2, it sends a RequestDomain(\textit{oid}, \textit{oloc}, \textit{N}_2) message. Then, the server assigns a new resident domain together with new query identifier and query region pairs to \textit{o}. As mentioned before, when the server receives a RequestDomain(\textit{oid}, \textit{oloc}, \textit{N}) message from a moving object, it should additionally visit \textit{N}.\textit{CL} and look up the query table (using query identifiers stored in \textit{N}.\textit{CL}) to check whether the movement of the moving object crosses each query region \textit{q}.\textit{R} that covers or equals \textit{N}. If so, the server updates the result of the corresponding query \textit{q}. Since \textit{N}_2.\textit{CL} in Fig. 5 is empty, the server skips this additional process when it receives the RequestDomain(\textit{oid}, \textit{oloc}, \textit{N}_2) message from \textit{o}.

3.1.3. Building and maintaining the QR-Tree

When a new query \textit{q} is issued by a client and is inserted into the query table, the insert algorithm recursively follows the paths of the QR-tree, each of which consists of non-leaf and leaf nodes with which the query region \textit{q}.\textit{R} of \textit{q} overlaps. Algorithm 2 is the pseudocode of the insert algorithm.

**Algorithm 2. Insert(\textit{N}, \textit{q})**

<table>
<thead>
<tr>
<th>Input</th>
<th>\textit{N}: A QR-tree node initially set to the root, \textit{q}: A query</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>\textbf{if} \textit{N} is a non-leaf node \textbf{then}</td>
</tr>
<tr>
<td>2</td>
<td>\textbf{if} \textit{q}.\textit{R} covers or equals \textit{N} \textbf{then}</td>
</tr>
<tr>
<td>3</td>
<td>Insert \textit{qid} into \textit{N}.\textit{CL};</td>
</tr>
<tr>
<td>4</td>
<td>\textbf{else} // \textit{q}.\textit{R} is covered by or partially intersects \textit{N}</td>
</tr>
<tr>
<td>5</td>
<td>Increase \textit{N}.\textit{Count} by 1;</td>
</tr>
<tr>
<td>6</td>
<td>\textbf{end if}</td>
</tr>
<tr>
<td>7</td>
<td>\textbf{for} each entry (\textit{ptr}, \textit{\hat{N}}) stored in \textit{N} \textbf{do}</td>
</tr>
<tr>
<td>8</td>
<td>\textbf{if} \textit{q}.\textit{R} overlaps with \textit{\hat{N}} \textbf{then}</td>
</tr>
<tr>
<td>9</td>
<td>Insert(\textit{N}, \textit{q});</td>
</tr>
<tr>
<td>10</td>
<td>\textbf{end if}</td>
</tr>
<tr>
<td>11</td>
<td>\textbf{end for}</td>
</tr>
<tr>
<td>12</td>
<td>\textbf{else} // \textit{N} is a leaf node</td>
</tr>
<tr>
<td>13</td>
<td>\textbf{if} \textit{q}.\textit{R} covers or equals \textit{N} \textbf{then}</td>
</tr>
<tr>
<td>14</td>
<td>Insert \textit{qid} into \textit{N}.\textit{CL};</td>
</tr>
<tr>
<td>15</td>
<td>\textbf{else} // \textit{q}.\textit{R} is covered by or partially intersects \textit{N}</td>
</tr>
<tr>
<td>16</td>
<td>Insert \textit{qid} into \textit{N};</td>
</tr>
<tr>
<td>17</td>
<td>Increase \textit{N}.\textit{Count} by 1;</td>
</tr>
<tr>
<td>18</td>
<td>SplitNode(\textit{N}) if \textit{N}.\textit{Count} &gt; \textit{t};</td>
</tr>
<tr>
<td>19</td>
<td>\textbf{end if}</td>
</tr>
<tr>
<td>20</td>
<td>\textbf{end if}</td>
</tr>
</tbody>
</table>

At a non-leaf node \textit{N} in each path, the algorithm checks whether \textit{q}.\textit{R} covers or equals \textit{N}. If so, the query identifier \textit{qid} of \textit{q} is inserted into \textit{N}.\textit{CL} (Lines 2–3). Otherwise (i.e., \textit{q}.\textit{R} is covered by or partially intersects \textit{N}), the algorithm increases \textit{N}.\textit{Count} by 1 (Lines 4–5). When reaching a leaf node \textit{N} in the path, the algorithm checks whether \textit{q}.\textit{R} covers or equals \textit{N}. If this is the case, it inserts \textit{qid} into \textit{N}.\textit{CL} (Lines 13–14). Otherwise, the algorithm inserts \textit{qid} into \textit{N} and increases \textit{N}.\textit{Count} by 1 (Lines 15–17). When \textit{N} overflows (i.e., \textit{N}.\textit{Count} > \textit{t}), the split algorithm is invoked (Line 18). Algorithm 3 is the pseudocode of the split algorithm.
Algorithm 3. SplitNode(N)

**Input** N: An overflowed QR-tree leaf node

1. Create two new empty leaf nodes N_{left} and N_{right};
2. Create a new empty non-leaf node N_{new};
3. Insert entries (ptr, N_{left}) and (ptr, N_{right}) into N_{new};
4. Insert all the query identifiers stored in N.CL into N_{left}.CL, N_{right}.CL, and N_{new}.CL;
5. Set N_{new}.Count to N.Count;
6. Find the entry (ptr, N) stored in N’s parent and redirect ptr to point to N_{new};
7. **for** each entry (ptr, N_{new}) stored in N_{new} **do**
   // We use N_{new} to denote N_{left} or N_{right}
   8. **for** each q referred to by each qid stored in N **do**
   9. **if** q.R covers or equals N_{new} **then**
   10. Insert qid into N_{new}.CL
   11. **else if** q.R is covered by or partially intersects N_{new} **then**
   12. Insert qid into N_{new};
   13. Increase N_{new}.Count by 1;
   14. **end if**
   15. **end for**
16. **end for**
17. Discard N;
18. **for** each entry (ptr, N_{new}) stored in N_{new} **do**
19. SplitNode(N_{new}) if N_{new}.Count > t;
20. **end for**

Given an overflowed leaf node N, the split algorithm creates two new empty leaf nodes N_{left} and N_{right}, and a new non-leaf node N_{new} that stores entries (ptr, N_{left}) and (ptr, N_{right}), where N_{left} or N_{right} represents one of the equal halves of N (Lines 1–3). Now, N_{left} and N_{right} become N_{new}’s children. Next, the algorithm (i) inserts all the query identifiers stored in N.CL into N_{left}.CL, N_{right}.CL, and N_{new}.CL and (ii) sets N_{new}.Count to N.Count, after which it finds the entry (ptr, N) stored in N’s parent to redirect ptr to point to N_{new} (Lines 4–6). Now, N’s parent becomes N_{new}’s parent. Then, the algorithm checks for each query q referred to by each query identifier qid stored in N whether q.R covers or equals N_{left} (or N_{right}). If so, it inserts qid into N_{left}.CL (or N_{right}.CL) (Lines 9–10). On the other hand, if q.R is covered by or partially intersects N_{left} (or N_{right}), the algorithm inserts qid into N_{left} (or N_{right}) and increases its Count value by 1 (Lines 11–13). Finally, the algorithm discards N (Line 17). This split process propagates downward if necessary (Line 19).
The server broadcasts an $\text{InsertQuery}(\text{qid}, q, R)$ message to all moving objects after inserting $q$ into the QR-tree, in order to notify them of this fact. In response, each moving object $o$ sends an $\text{UpdateResult}(\text{oid}, \text{o.loc}, \text{qid}, R)$ message if it currently lies within $q, R$, after which $o$ checks whether $q, R$ is covered by or partially intersects its current resident domain. If so, $o$ adds $(\text{qid}, q, R)$ to its assigned query identifier and query region pairs if available; otherwise, it sends a $\text{RequestDomain}(\text{oid}, \text{o.loc}, N)$ message for a new resident domain.

When an existing query $q$ is terminated by a client and is deleted from the query table, the delete algorithm is invoked. Similarly to the insert algorithm, the delete algorithm recursively follows the paths of the QR-tree, each of which consists of the non-leaf and leaf nodes that overlap with the query region $q, R$ of $q$. Algorithm 4 is the pseudocode of the delete algorithm.

Algorithm 4. Delete($N, q$)

```
Input $N$: A QR-tree node initially set to the root, $q$: A query
1: if $N$ is a non-leaf node then
2: if $q, R$ covers or equals $N$ then
3: Delete $\text{qid}$ from $N.\text{CL}$;
4: else // $q, R$ is covered by or partially intersects $N$
5: Decrease $N.\text{Count}$ by 1;
6: end if
7: for each entry (ptr, $N$) stored in $N$ do
8: if $q, R$ overlaps with $N$ then
9: $\text{DELETE}(N, q)$;
10: end if
11: end for
12: else // $N$ is a leaf node
13: if $q, R$ covers or equals $N$ then
14: Delete $\text{qid}$ from $N.\text{CL}$;
15: else // $q, R$ is covered by or partially intersects $N$
16: Delete $\text{qid}$ from $N$;
17: Decrease $N.\text{Count}$ by 1;
18: $\text{MERGE}(N$'s parent);
19: end if
20: end if
```

At a non-leaf node $N$, the algorithm deletes the query identifier $\text{qid}$ of $q$ from $N.\text{CL}$ if $q, R$ covers or equals $N$ (Lines 2–3). Otherwise, it decreases $N.\text{Count}$ by 1 (Lines 4–5). On the other hand, at a leaf node $N$, the algorithm deletes $\text{qid}$ from $N.\text{CL}$ if $q, R$ covers or equals $N$ (Lines 13–14), otherwise, it deletes $\text{qid}$ from $N$ and decreases $N.\text{Count}$ by 1 (Lines 15–17). Then, the algorithm invokes the merge algorithm, which takes $N$'s parent as an input, to condense the tree if possible (Line 18). Algorithm 5 is the pseudocode of the merge algorithm.

Algorithm 5. MergeNode($N$)

```
Input $N$: A non-leaf node of the QR-tree
1: if $N.\text{count} \leq t$ then
2: Create a new leaf node $N_{\text{new}}$;
3: Insert all the query identifiers stored in $N.\text{CL}$ into $N_{\text{new}}.\text{CL}$;
4: Set $N_{\text{new}}.\text{Count}$ to $N.\text{Count}$;
5: Find the entry (ptr, $N$) stored in $N$'s parent and redirect ptr to point to $N_{\text{new}}$;
6: for each entry (ptr, $N$) stored in $N$ do
7: Insert all the distinct qids stored in $N$ into $N_{\text{new}}$;
8: for each $q$ referred to by each qid stored in $N.\text{CL}$ do
9: if $q, R$ is covered by or partially intersects $N_{\text{new}}$ then
10: Insert qid into $N_{\text{new}}$ if it is not stored in $N_{\text{new}}$;
11: end if
12: end for
13: end for
14: Discard $N$ and $N$'s children;
15: $\text{MERGE}(N_{\text{new}}$'s parent);
16: end if
```

Please cite this article in press as: H. Jung et al., QR-tree: An efficient and scalable method for evaluation of continuous range queries, Inform. Sci. (2014), http://dx.doi.org/10.1016/j.ins.2014.02.061
Given a non-leaf node \( N \), the algorithm first checks whether \( N \) count \( \leq t \). If so, the algorithm creates a new empty leaf node \( N_{\text{new}} \), inserts all the query identifiers stored in \( N \) into \( N_{\text{new}} \), and sets \( N_{\text{new}} \) count to \( N \) count (Lines 2–4). Next, the algorithm finds the entry \( \text{ptr}, N \) stored in \( N \)’s parent to redirect \( \text{ptr} \) to point to \( N_{\text{new}} \) (Line 5). Then, the algorithm (i) inserts all the distinct query identifiers stored in each \( N \)’s child \( N \) into \( N_{\text{new}} \) (Lines 6–7) and (ii) checks for each query \( q \) referred to by each query identifier \( qid \) stored in \( N \). If \( q \) \& \( R \) is covered by or partially intersects \( N_{\text{new}} \) (Lines 8–10). Finally, the algorithm discards \( N \) and its children (Line 14). This merge process propagates upward until the node that does not satisfy the merge condition is reached (Line 15).

After deleting \( q \) from the QR-tree, to notify them of such a change, the server broadcasts a DeleteQuery\( (qid, q, R) \) message to all the moving objects. In response, each moving object \( o \) deletes \( (qid, q, R) \) from its assigned query identifier and query region pairs if it has \( (qid, q, R) \).

3.2. The Bit-vector Query Region tree (BQR-tree)

3.2.1. Motivation

The BQR-tree is a variant of the QR-tree, which focuses on evaluating CQRs with non-spatial selections. We formalize the problem as follows.

**CRQs with non-spatial selections.** Suppose each moving object \( o \) is associated with a set of \( n \) non-spatial attributes \( A = \{ a_1, a_2, \ldots, a_n \} \). Each non-spatial attribute \( a_i \in A \) (\( 1 \leq i \leq n \)) is either numeric (e.g., Age and Annual income) or categorical (e.g., Dietary preference). We denote the non-spatial attribute values of \( o \) by \( o.A = \{ a_1, o.a_2, \ldots, o.a_n \} \). A query \( q \) is represented as \( (q, q, V) \), where \( q \) \& \( R \) denotes a query region and \( q.V = \{ q.v_1, q.v_2, \ldots, q.v_m \} \) denotes a set of non-spatial intervals or non-spatial values specified on a subset of non-spatial attributes \( A \) (\( \subseteq A \) = \{ \( a_1, a_2, \ldots, a_m \} \). We assume in this paper that \( q.v_i \) is an interval if \( a_i \) is a numerical attribute. For a query \( q = (q, q, V) \), the server should continuously retrieve all moving objects \( o \) that are currently located within \( q \) \& \( R \) and satisfy: \( \forall o, a_i \in o.A, o.a_i \) lies in \( q, v_i \) (or \( o.a_i = q.v_i \) if \( a_i \) is a categorical attribute).

For example, let us assume that the moving object \( o \) with \( o.\text{Cap} = 3 \) in Fig. 7 is associated with three non-spatial attributes \( A = \{ a_1: \text{Age}, a_2: \text{Annual income}, a_3: \text{Dietary preference} \} \) and \( o.A = \{ a_1 = 35, a_2 = \$93,000, a_3 = \text{Vegetarian} \} \). Suppose the queries \( q_1, q_2, q_3, q_4 \) involve non-spatial selections \( q_1, q_2, q_3, q_4 \) on a subset of \( A \), as shown in Fig. 7. In addition to the query regions \( q_1, q_2, q_4, q_5 \). Using the QR-tree (\( t = 1 \)), the server assigns an object the resident domain depicted in the figure together with the query regions \( q_1, q_2, q_3, q_4, q_5 \).

However, \( o \) can be assigned a much larger resident domain than its current resident domain (e.g., the whole workspace) because \( o \) does not satisfy the non-spatial selection criteria that the queries \( q_1, q_2, q_3, q_4 \), and \( q_5 \) specify. Therefore, \( o \) need not check its movement against the query regions \( q_1, q_2, q_4, q_5 \), and \( q_3, q_4 \), because its movement does not affect the results of \( q_1, q_2, q_3, q_4 \), and \( q_5 \). We say a query region \( q.R \) is non-spatially matched to a moving object \( o \) if \( o \) satisfies the non-spatial selection criteria that the corresponding query \( q \) specifies.

3.2.2. Description

In the BQR-tree, each (leaf or non-leaf) node additionally stores a summary of non-spatial intervals (or values) of queries in the form of a bit-vector. We represent the non-spatial attribute values of moving objects and non-spatial intervals (or values) of queries as object bit-vectors and query bit-vectors, respectively. Before we introduce the BQR-tree, we first define the object bit-vector and the query bit-vector. For a numerical attribute \( a_i \), its mapping function \( f_i \) divides its domain into \( |IV| \) disjoint intervals \( I_{v_1}, I_{v_2}, \ldots, I_{v_m} \) of equal length. Then, given a moving object \( o \) \& \( f \) maps \( o.a_i \) into a bit-string \( b_i b_1 b_2 \ldots b_{|IV|} \) such that \( b_i = 1 \) if \( o.a_i \) lies in \( I_{v_j} \), otherwise, \( b_i = 0 \). Similarly, it maps the interval \( q.v \) that a query \( q \) specifies on \( a \) into |IV| bit-string \( b_1 b_2 \ldots b_{|IV|} \) such that \( b_i = 1 \) if \( q.v \) overlaps with \( I_{v_j} \), otherwise, \( b_i = 0 \). On the other hand, for a categorical attribute \( a_i \) with \( |C| \) categories \( c_1, c_2, \ldots, c_{|C|} \), a mapping function \( f \) maps \( o.a_i \) (or the specified value \( q.v \) on \( a_i \)) into a bit-string \( b_1 b_2 \ldots b_{|C|} \) such that \( b_j = 1 \) if \( o.a_i = c_j \) (or \( q.v = c_j \)), otherwise, \( b_j = 0 \).

**Definition 3** (Object bit-vector). Suppose that there is a mapping function \( f_i \) (\( 1 \leq i \leq n \)) for each non-spatial attribute \( a_i \in A \) (\( 1 \leq i \leq n \)). Then, an object bit-vector generated for \( o.A \) is \( f_1(o.a_1) + f_2(o.a_2) + \ldots + f_n(o.a_n) \), where \( + \) denotes the bit-string concatenation operator.

**Definition 4** (Query bit-vector). A query bit-vector generated for \( q.V \) is \( f_1(q.v_1) + f_2(q.v_2) + \ldots + f_n(q.v_n) \). It should be noted that when \( q.V \) does not contain the specified interval (or value) \( q.v_i \) on \( a_i \) (\( 1 \leq i \leq n \)), the bit-string for \( f(q.v_i) \) becomes \( \ast \ast \ast \ast \ldots \ast \) with its length being equal to \( f_i(o.a_i) \), where the symbol \( \ast \) denotes a do not care condition.

In the following, we show an example of generating an object bit-vector and query bit-vectors using the non-spatial attribute values \( o.A \) of the moving object \( o \) in Fig. 7 and the sets of non-spatial selections \( q_4, V \) and \( q_5, V \) that the queries \( q_4, q_5 \) specify on a subset of non-spatial attributes \( A \) in Fig. 7. Suppose that there are three mapping functions:
respectively.

We omit the planar representation of the BQR-tree, because it is exactly same as that of the QR-tree shown in Fig. 6.

3.2.3. Cooperative evaluation of CRQs with non-spatial selections

Algorithm 6 is the pseudocode of the search algorithm on the BQR-tree for assigning each moving object its resident domain together with non-spatially matched query regions that are covered by or partially intersect N. Given a moving object o with its current location, available capability o.Cap, and set of non-spatial attribute values o.A, the search algorithm first

\[
\begin{align*}
f_1(x) &= \begin{cases}
1000 & \text{if } x \text{ lies in (or overlaps with) } [0, 20); \\
0100 & \text{if } x \text{ lies in } (20, 40); \\
0010 & \text{if } x \text{ lies in } [40, 60); \\
0001 & \text{otherwise.}
\end{cases} \\
f_2(x) &= \begin{cases}
1000 & \text{if } x \text{ lies in (or overlaps with) } [0, 40, 000); \\
0100 & \text{if } x \text{ lies in } (40, 000, 80, 000); \\
0010 & \text{if } x \text{ lies in } [80, 000, 120, 000); \\
0001 & \text{otherwise.}
\end{cases} \\
f_3(x) &= \begin{cases}
10 & \text{if } x \text{ is Meat eater;} \\
01 & \text{if } x \text{ is Vegetarian.}
\end{cases}
\end{align*}
\]

Then, the object bit-vector generated for o.A is 01000011001 (1000 + 0010 + 01). On the other hand, the query bit-vectors generated for o.V and q.V are \(* * * * 001001 (\text{=} * * * * + 0010 + 01)\) and 1000 \(* * * * * * * * * (\text{=} 1000 + * * * * * * * * * )\), respectively.

Now, we present the BQR-tree. A leaf node of the BQR-tree stores at most 3 entries of the form (qid, q.bv), where qid refers to a query q in the query table and q.bv is the query bit-vector of q (more specifically, the query bit-vector generated for q.V). On the other hand, a non-leaf node stores two entries of the form (ptr, N.N.bv), where ptr and N are defined as in the QR-tree, and N.bv is the node bit-vector of a node N. The BQR-tree satisfies the following properties:

1. An entry (qid, q.bv) for a query q is stored in a leaf node N only if q.R is covered by or partially intersects N.
2. An entry (qid, q.bv) for a query q can be redundantly stored in several leaf nodes if q.R partially intersects these leaf nodes.
3. For each entry (ptr, N, N.bv) stored in a non-leaf node N, N represents one of the equal halves of N’s domain, whereas N.bv is a node bit-vector of N, which is formed by bitwise OR-ing every query bit-vector q.bv of a query q whose query region q.R is covered by or partially intersects N.
4. Each (leaf or non-leaf) node of the BQR-tree stores a variable Count and is associated with a covering list CL.

It should be noted that, while the definition of a variable Count in the BQR-tree is exactly the same as Definition 1, the definition of a covering list CL in the BQR-tree given in Definition 5 is slightly different from Definition 2.

Definition 5 (Covering list (CL)). Given a set of queries Q and a (leaf or non-leaf) node N of the QR-tree, N’s associated covering list N.CL is a list, which stores the entry of the form (qid, q.bv) for every query q \((\in Q)\) whose query region q.R covers or equals N.

Assuming \(t = 1\), Fig. 8 shows the BQR-tree for the queries q_1 = (q_1, R, q_1, V), q_2 = (q_2, R, q_2, V), \ldots, q_5 = (q_5, R, q_5, V) in Fig. 6. We omit the planar representation of the BQR-tree, because it is exactly same as that of the QR-tree shown in Fig. 6.

Please cite this article in press as: H. Jung et al., QR-tree: An efficient and scalable method for evaluation of continuous range queries, Inform. Sci. (2014), http://dx.doi.org/10.1016/j.ins.2014.02.061
generates object bit-vector $o.bv$ and initializes queue $Queue$ (Lines 2–3). Then, starting from the root, the algorithm recursively descends the BQR-tree while checking whether each node $N$ contains $o$’s current location. If $N$ contains $o$’s current location, according to two cases, the algorithm proceeds as follows:

- **Case (1):** If $N.bv \land o.bv \neq o.bv$ (\(\land\) denotes bit-wise AND-ing), the algorithm sets $N$ as $o$’s resident domain. In this case, $N$ stores no overlapped query regions that are non-spatially matched to $o$, and thus, the algorithm does not invoke $\text{FINDQUERYREGIONS}$ (Lines 6–7).

- **Case (2):** If $N.bv \land o.bv = o.bv$, the algorithm further checks whether $N.\text{Count} \leq o.Cap$. If so, it sets $N$ as $o$’s resident domain and invokes $\text{FINDQUERYREGIONS}$ (Lines 8–11). $\text{FINDQUERYREGIONS}$, similarly to that in the QR-tree, retrieves every distinct entry $(qid, q.bv)$ stored in each $N$’s descendent leaf node $N$ and its associated covering list $N.CL$. If $(qid, q.bv)$ is stored in both $N.CL$ and $N.CL$, or $q.bv \land o.bv \neq o.bv$, $\text{FINDQUERYREGIONS}$ filters it out. In case $N$ is a leaf node, $\text{FINDQUERYREGIONS}$ retrieves each entry $(qid, q.bv)$ stored in $N$ if $q.bv \land o.bv \neq o.bv$.

On the other hand, when $N$ does not contain $o$’s current location, the algorithm inserts $N$ into $Queue$ if $(N.bv \land o.bv \neq o.bv$ (Lines 16–17). After searching $o$’s resident domain and retrieving a set of distinct qualifying entries, denoted by $QE$, the algorithm further checks whether $|QE| < o.Cap$. If so, it starts de-queueing elements iteratively until the de-queueing element $E$ touches $o$’s resident domain (or $Queue$ is empty) (Lines 21–23). Then, the algorithm sets $E$ as $o$’s additional resident domain and terminates (Line 24). It should be noted that $o$ can freely move within $E$ without contacting the server because all the query regions that are covered by or partially intersect $E$ are not non-spatially matched to $o$.

After Algorithm 6 terminates, the server searches all the queries (in the query table) referred to by the retrieved entries and assigns the moving object $o$ (i) its resident domain $N$, (ii) the pairs of non-spatially matched query region and the entry for the corresponding query, and (iii) the additional resident domain (if possible). For example, the moving object $o$ with $o.Cap = 3$ in Fig. 6 is assigned (i) node $N_2$ as its resident domain, (ii) the pairs of non-spatially matched query region and entry for the corresponding query ($\{(q_3, q_5, b.v), q_3, R\}$, $\{(q_4, q_6, b.v), q_4, R\}$), and (iii) the additional resident domain $N_1$ (because $N_1$ is the topmost element in $Queue$, which touches $N_2$).

**Algorithm 6. Search($N, o$)**

<table>
<thead>
<tr>
<th>Input $N$: A QR-tree node initially set to the root, $o$: A moving object</th>
</tr>
</thead>
<tbody>
<tr>
<td>Output $R$: $o$’s resident domain, $E$: $o$’s additional resident domain,</td>
</tr>
<tr>
<td>$QE$: A set of distinct qualifying entries</td>
</tr>
<tr>
<td>1: Initialize an empty set $QE$;</td>
</tr>
<tr>
<td>2: Map $o.V$ to $o.bv$;</td>
</tr>
<tr>
<td>3: Initialize an empty queue $Queue$;</td>
</tr>
<tr>
<td>4: for each entry (ptr.$N$) stored in $N$ do</td>
</tr>
<tr>
<td>5: if $N$ contains $o$’s current location then</td>
</tr>
<tr>
<td>6: if $N.bv \land o.bv \neq o.bv$ then</td>
</tr>
<tr>
<td>7: Set $R$ to $N$;</td>
</tr>
<tr>
<td>8: else if $N.bv \land o.bv = o.bv$ then</td>
</tr>
<tr>
<td>9: if $N.\text{Count} \leq o.\text{Cap}$ then</td>
</tr>
<tr>
<td>10: Set $R$ to $N$;</td>
</tr>
<tr>
<td>11: $QE \leftarrow QE \cup \text{FINDQUERYREGIONS}$;</td>
</tr>
<tr>
<td>12: else</td>
</tr>
<tr>
<td>13: Search($N, o$);</td>
</tr>
<tr>
<td>14: end if</td>
</tr>
<tr>
<td>15: end if</td>
</tr>
<tr>
<td>16: else (if $N$ does not contain $o$’s current location</td>
</tr>
<tr>
<td>17: Enqueue $N$ if $N.bv \land o.bv \neq o.bv$;</td>
</tr>
<tr>
<td>18: end if</td>
</tr>
<tr>
<td>19: end for</td>
</tr>
<tr>
<td>20: if $</td>
</tr>
<tr>
<td>21: repeat</td>
</tr>
<tr>
<td>22: Dequeue the next element $E$ from $Queue$;</td>
</tr>
<tr>
<td>23: until $E$ touches $R$ or $Queue$ is empty</td>
</tr>
<tr>
<td>24: Set $E$ (if it exists) as $N$’s additional resident domain;</td>
</tr>
<tr>
<td>25: end if</td>
</tr>
</tbody>
</table>

Please cite this article in press as: H. Jung et al., QR-tree: An efficient and scalable method for evaluation of continuous range queries, Inform. Sci. (2014), http://dx.doi.org/10.1016/j.ins.2014.02.061
4. Performance evaluation

In this section, we evaluate and compare the performance of the QR-tree and BQR-tree methods (denoted by QRT and BQRT, respectively) with that of the safe region technique [9] (denoted by SR) and MQM [3] for CRQ evaluation in terms of the server workload and communication cost. The server workload was measured in terms of the CPU-time that the server takes for CRQ evaluation. On the other hand, the communication cost was measured by the total amount of messages transmitted between the server and moving objects. The simulations were conducted on dual Intel Xeon x5860 6-core processors with 24 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 50 km \times 50 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 that we generated follow the random waypoint model [2], 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 25 and 100 query regions (or monitoring regions), and thus the threshold value $t$ of the QR-tree, BQR-tree, and BP-tree (used in MQM) was set to 25. For SR, we used the 64 grid indexes for indexing query regions and safe regions.

Each moving object is associated with five non-spatial attributes $a_1, a_2, \ldots, a_5$, where $a_1$, $a_2$, and $a_3$ are numerical, while $a_4$ and $a_5$ are categorical. The domain of $a_1$, $a_2$, and $a_3$ is $[0, 1023]$, and that of $a_4$ and $a_5$ is 32. The distribution of each non-spatial attribute value $a_1, a_2, \ldots, a_5$ of each moving object follows the Zipf distribution with skew coefficient $\alpha = 0.8$. Each non-spatial interval or value specified on a subset of $a_1, a_2, \ldots, a_5$ by each query $q$ in both Uniform and Skewed follows the same distribution. 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\textsuperscript{4} and measured the average of the CPU-time (in ms) and total amount of messages (in kb).

One important parameter that should be noted for BQRT is the size of a bit-string assigned for each non-spatial attribute. For categorical attributes \(a_4\) and \(a_5\), we set the size of the bit-string to 32 bits, because the domain size of \(a_4\) and \(a_5\) is 32. On the other hand, for numerical attributes \(a_1\), \(a_2\), and \(a_3\), we measured the size of the BQR-tree and compared it to that of the QR-tree and BP-tree by varying the size of the bit-string (i.e., the number of divided intervals of equal length) assigned to each of them. We set all the parameters in Table 1 to their default values. As shown in Fig. 9, the size of the QR-tree is much smaller than that of the BQR-tree and BP-tree. On the other hand, the size of the BQR-tree is smaller than that of the BP-tree, unless the size of each bit-string exceeds 1024 bits (see Fig. 9b). Therefore, in the remainder of our simulations, we set the size of the bit-string for each numerical attribute to 170 bits. Under these parameter settings, because the size of safe region and resident domain directly affects the performances of SR, MQM, QRT, and BQRT, we measured the average size of (i) safe regions assigned to moving objects in SR and (ii) resident domains assigned to the moving objects in MQM, QRT, and BQRT.

4.2. Simulation results

4.2.1. Effect of the number of query regions

In the first simulation, we varied the cardinalities of Uniform and Skewed from 1000 to 10,000 and 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 QRT and BQRT with regard to the number of queries. Fig. 10 shows the effect of the number of query regions on the CPU-time the server takes to perform CRQ evaluation. In MQM, QRT, and BQRT, the CPU-time performance is affected mainly by the search process for assigning resident domains to moving objects, whereas, in SR, the CPU-time performance is affected mainly by the safe region computation.

As shown in the figure, SR performs worst for Uniform and Skewed, because as the number of query regions becomes larger, the size of a safe region assigned to each moving object \(o\) becomes smaller. Therefore, \(o\) easily exits its current small safe region and contacts the server in order to receive a new safe region. This leads the server to determine \(o\)'s new safe region frequently, which requires intensive computation. It should be noted that \(o\)'s safe region is determined by the common area among all the query regions and the grid cell in which \(o\) resides. QRT performs much 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 drastically, 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. We recall that the BP-tree in MQM is built based on monitoring regions instead of the original query regions, and thus, the capabilities of moving objects are measured against the huge number of monitoring regions. On the other hand, the QR-tree

\textsuperscript{4} Note that the term 'time step' indicates an interval of CQR evaluation, which we set to 5 s.
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 performs worse than BQRT, because the BQR-tree stores bit-vector information to fully utilize the capabilities of moving objects by ignoring query regions that are not non-spatially matched to the moving objects. As a result, the server in BQRT can 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. BQRT takes 21.3%, 27.7%, and 80.7% of the server workload, as compared to SR, MQM, and QRT, respectively, for Uniform. Meanwhile, BQRT takes 31.6%, 39.4%, and 86.0% of the server workload, as compared to SR, MQM, and QRT, respectively, for Skewed.

Fig. 11 shows the effect of the number of query regions on the total amount of messages transmitted between the server and moving objects. As the number of query regions increases, the performances of all the methods degrade. However, SR and BQRT outperform MQM and QRT for Uniform and Skewed. This is because, in SR, although moving objects frequently move out of their small safe regions and contact the server, the size of the messages transmitted between the server and moving objects is much smaller than that in MQM, QRT, and BQRT (especially, the size of messages sent from moving objects to the server, each of which contains only the identifier and current location of a moving object). On the other hand, in BQRT, the server can assign moving objects large resident domains by ignoring query regions that are not non-spatially matched to the moving objects when measuring the capabilities of moving objects with the help of bit-vector information. 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. As expected, MQM performs worse than QRT because of the tremendous number of monitoring regions produced. It is also important to note that, because the QR-tree and the BQR-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 BQRT. As shown in Fig. 11, BQRT performs the best in all the cases. As compared to SR, MQM, and QRT, BQRT incurs 49.2%, 24.6%, and 38.4%, respectively, of the communication cost for Uniform. On the other hand, BQRT incurs 60.9%, 25.6%, and 42.3% of the communication cost as compared to SR, 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 0.5 km to 5 km to examine how the size of query regions affects the performances of SR, MQM, QRT, and BQRT. As shown in Fig. 12, QRT and BQRT perform much better and are less sensitive to this parameter than SR and MQM for Uniform and Skewed. 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 reduces the

<table>
<thead>
<tr>
<th>Method</th>
<th>Average size (Uniform) (km²)</th>
<th>Average size (Skewed) (km²)</th>
</tr>
</thead>
<tbody>
<tr>
<td>SR</td>
<td>0.19</td>
<td>0.08</td>
</tr>
<tr>
<td>MQM</td>
<td>9.89</td>
<td>6.74</td>
</tr>
<tr>
<td>QRT</td>
<td>21.87</td>
<td>17.44</td>
</tr>
<tr>
<td>BQRT</td>
<td>30.16</td>
<td>24.47</td>
</tr>
</tbody>
</table>

Fig. 10. CPU-time vs. number of query regions.

4.2.2. Effect of the size of query regions

In this simulation, we varied the side length of query regions from 0.5 km to 5 km to examine how the size of query regions affects the performances of SR, MQM, QRT, and BQRT. As shown in Fig. 12, QRT and BQRT perform much better and are less sensitive to this parameter than SR and MQM for Uniform and Skewed. 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 reduces the
size of the safe region assigned to each moving object \( o \), and thus, the server in SR should frequently determine \( o \)'s new safe region. The excessive overlap among query regions also increases the number of monitoring regions in MQM. This 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. On the other hand, QRT and BQRT are nearly not affected by the side length of query regions. This is because, although the excessive overlap among query regions causes the QR-tree and BQR-tree to be split, the capabilities of moving objects are still measured against the number of the original query regions that are covered by or partially intersect each QR-tree node or BQR-tree node. Among all the methods, SR performs worst, while BQRT performs best. As compared to SR, MQM, and QRT, BQRT incurs 58\%; 26\% , and 82\% of the server workload for Uniform. On the other hand, BQRT takes 32\%, 38\%, and 82\% of the server workload, as compared to SR, MQM, and QRT, respectively, for Skewed.

Fig. 13 shows the effect of the side length of each query region (i.e., the size of each query region) on the total amount of messages. As shown in the figure, SR and BQRT perform 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, BQRT achieves the best performance for Uniform and Skewed. As compared to SR, MQM, and QRT, BQRT incurs 58.7\%, 26.4\%,
and 39.9%, respectively, of the communication cost for Uniform. On the other hand, BQRT incurs 59.8%, 25.9%, and 40.8% of the communication cost as compared to SR, MQM, and QRT, respectively, for Skewed.

4.2.3. Effect of the number of moving objects

In this simulation, we increased the number of moving objects from 20,000 to 200,000 to study how the number of moving objects affects the performances of SR, MQM, QRT, and BQRT. As shown in Figs. 14 and 15, as 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. However, in all cases, BQRT outperforms SR, MQM, and QRT because only BQRT has the ability to fully utilize the capabilities of moving objects by ignoring the number of query regions that are not non-spatially matched to the moving objects. It can also be seen in the figures that SR performs worst in terms of the server workload, while MQM performs worst in terms of the communication cost.

4.2.4. Effect of the max speed of moving objects

In this simulation, we varied the maximum speed of moving objects from 10 km/h to 100 km/h to investigate how the speed of moving objects affects the performances of SR, MQM, QRT, and BQRT. As shown in Fig. 16, the performances of
all the methods in terms of the CPU-time deteriorate as the maximum speed of moving objects increases. This is obvious, because in SR, the CPU-time performance is affected mainly by the safe region computation, whereas in MQM, QRT, and BQRT, the CPU-time performance is affected mainly by the search process for assigning resident domains to moving objects. As the speed of moving objects increases, they may frequently exit their current safe regions or resident domains, and request new safe regions or resident domains. However, SR and MQM are more sensitive to this parameter than QRT and BQRT, confirming the superiority of QRT and BQRT in terms of the server workload. It can be seen in Fig. 16 that BQRT performs best in all cases. BQRT takes 22.2%, 22.6%, and 78.6% of the server workload, as compared to SR, MQM, and QRT, respectively, for Uniform. On the other hand, BQRT takes 28.8%, 34.9%, and 79.2% of the server workload, as compared to SR, MQM, and QRT, respectively, for Skewed.

Fig. 17 shows the effect of the maximum speed of moving objects on the total amount of messages transmitted. As expected, the performances of all the methods degrade as the maximum speed of moving objects increases. However, as shown in Fig. 17, BQRT performs best for Uniform and Skewed. As compared to SR, MQM, and QRT, BQRT incurs 17.5%, 24.8%, and 40.7%, respectively, of the communication cost for Uniform. On the other hand, BQRT incurs 23.6%, 18.5%, and 40.2% of the communication cost as compared to SR, MQM, and QRT, respectively, for Skewed. It is also observed from the figure that SR performs worst for Uniform, whereas MQM performs worst for Skewed.
4.2.5. Effect of the update rates of queries

Finally, we investigated how the updates (i.e., insertion and deletion) of queries affect the performance of SR, MQM, QRT, and BQRT by increasing update rates (from 1% to 10%) \(^5\) of queries in Uniform and Skewed. Fig. 18 shows the effect of the update rates of queries on the CPU-time. It can be seen in the figure that QRT and BQRT perform much better than SR and MQM for Uniform and Skewed. This is because, in SR, when a new query \(q\) is inserted, the server needs to re-compute safe regions for all the moving objects because the query region \(q_R\) of \(q\) could affect all their current safe regions [3]. In MQM, when \(q\) is inserted, \(q_R\) of \(q\) is partitioned into many monitoring regions. Then, the insertion operation of the BP-tree is performed for each of these monitoring regions. This increases the CPU-time drastically. On the other hand, in QRT (and BQRT), when a \(q\) is inserted (or deleted), the insertion (or deletion) operation of QR-tree (and BQR-tree) is performed only once for \(q_R\) of \(q\). It can also be seen in the figure that BQRT outperforms the other methods in all cases. BQRT takes 33.7%, 32.3%, and 90.9% of the server workload, as compared to SR, MQM, and QRT, respectively, for Uniform. On the other hand, BQRT takes 32.6%, 34.6%, and 91.3% of the server workload, as compared to SR, MQM, and QRT, respectively, for Skewed.

Fig. 19 shows the effect of the update rates of queries on the total amount of messages transmitted. As shown in the figure, MQM performs worst. This is because, in MQM, the server should communicate frequently with moving objects because

\(^5\) It should be noted that these update rates are sufficient for studying the performances of SR, MQM, QRT, and BQRT, because these methods focus on dealing with stationary or quasi-stationary queries.
the insertion (or deletion) of a query region produces (or eliminates) many monitoring regions. As expected, BQRT achieves the best performance in all cases for Uniform and Skewed. As compared to MQM, BQRT incurs only 21.9% and 23.9% of the communication cost for Uniform and Skewed, respectively.

5. Related work

Early studies assumed stationary objects and focused on developing efficient spatial access methods (e.g., the R-tree [8] and its variants [1,23]) and the evaluation of snapshot queries, which retrieves their results only once at a specific time point. Recently, motivated by LBSs, the focus has shifted toward continuous query evaluation over moving objects. Many studies have been performed on continuous range query evaluation, which can be classified into two categories depending on whether or not the queries also move. The first category focuses on stationary or quasi-stationary queries over moving objects [3,9,15,22,27,28], and the second category deals with moving queries over moving objects [4,7,10,17–19]. Since our work belongs to the first category, we elaborate on the review of the representative methods in the first category and briefly review the methods in the second category. Assuming that the trajectories of object movements are known a priori or predictable, Saltenis et al. [24] proposed the Time-Parameterized R-tree (TPR-tree) for indexing moving objects, where the location of each object is transformed into a linear function of time. Tao et al. [26] proposed an improved version of the TPR-tree, called the TPR*-tree, which uses the same data structure as the TPR-tree, but in which new insertion and deletion algorithms are applied. Later, some index structures were presented, such as the STRIPES [21] and the B*-tree [12], a variant of the B*-tree, to improve the performance of the TPR-tree family. However, the known-trajectory assumption 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), which leads these index structures to become too expensive to maintain.

Indexing queries, instead of indexing frequently moving objects with arbitrary velocities and directions, is considered to be an attractive strategy, which reduces the maintenance cost of an index structure, because queries remain active for a long period of time and are stationary. Prabhakar et al. [22] suggested using the R-tree to index queries, while Kalashnikov et al. [15] used the in-memory grid index. Wu et al. [28] proposed a new query indexing method, namely, Containment Encoded Square (CES) based indexing. In all these methods, it was assumed that objects proactively report their location updates to the server whenever they move. The server, meanwhile, continually (i) receives the location-update stream, (ii) determines the queries that are affected by the movements of the objects, and (iii) update their results. However, constant location updates generated by a large number of objects may incur a significant communication bottleneck and increase the workload of determining the affected queries and updating their results at the server. In addition, since the transmission of a location-update message over a wireless connection takes a substantial amount of energy, the handheld device carried by each object exhausts its battery life quickly. To help reduce the frequency at which each moving object reports its location update, the safe region technique was proposed in [9,22]. Cai et al. [3] proposed the MQM, which aims to reduce the communication cost and the server workload by leveraging heterogeneous computational capabilities of moving objects through the concept of resident domain. Recently, the safe region technique for moving circular range queries over stationary objects was also proposed in [4].

Focusing on the evaluation of continuous moving queries over moving objects, Mokbel et al. [18] proposed the Scalable Incremental hash based Algorithm (SINA) to achieve system scalability based on the notions of shared execution and incremental evaluation. Gedik et al. [7] presented MobiEyes, where moving objects play an active role in the query evaluation task as in MQM. In SINA, the objects blindly report their location updates periodically, while MobiEyes relies on location estimation to...
reduce the number of location updates of the objects, as well as moving query issuers (i.e., moving clients). Liu et al. [17] employed 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. Recently, assuming the objects periodically report their location-updates, Mouratidis and Bakiras [19] introduced the broadcast grid index (BGI), which employs periodic broadcasting for communications between the server and the query issuers to evaluate moving queries.

None of the methods reviewed above can adequately deal with continuous range queries with specifications for non-spatial attributes. Although some existing studies, such as [13,14,20], have addressed the spatial queries that involve non-spatial specifications, their methods are restricted to snapshot queries over stationary objects.

6. Conclusion

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 keep the results of CRQs up to date, while incurring the minimum communication cost and server workload by letting the moving objects evaluate several CRQs that are relevant to them. To achieve this, we used the resident domain concept and proposed a novel index structure, namely the Query Region tree (QR-tree). The QR-tree allows the server to cooperate efficiently with moving objects. In addition, we presented another version of the QR-tree, referred to as the Bit-vector Query Region tree (BQR-tree), for evaluating CRQs with non-spatial selections. This type of queries are common in many real-life location-based services. The BQR-tree, which stores the additional bit-vector information needed to describe the non-spatial information, greatly improves the overall system performance when queries involve non-spatial selections. We carried out a series of comprehensive simulations and demonstrated that the QR-tree and BQR-tree methods outperform the existing methods, namely, the safe region technique and Monitoring Query Management (MQM) that utilizes the Binary Partitioning tree (BP-tree), validating the effectiveness of the QR-tree and BQR-tree.

References


Please cite this article in press as: H. Jung et al., QR-tree: An efficient and scalable method for evaluation of continuous range queries, Inform. Sci. (2014), http://dx.doi.org/10.1016/j.ins.2014.02.061