

# Real-Time Processing of Range-Monitoring Queries in Heterogeneous Mobile Databases

*Ying Cai* \*      *Kien A. Hua* †      *Guohong Cao* ‡      *Toby Xu* §

April 18, 2005

## Abstract

Unlike conventional range queries, a range-monitoring query is a continuous query. It requires retrieving mobile objects inside a user-defined region, and providing continuous update as the objects move into and out of the region. In this paper, we present an efficient technique for real-time processing of such queries. In our approach, each mobile object is associated with a resident domain and when an object moves, it monitors its spatial relationship with its resident domain and the monitoring areas inside it. An object reports its location to server when it crosses over some query boundary or moves out of its resident domain. In the first case, the server updates the affected query results accordingly while in the second case, the server determines a new resident domain for the object. This distributive approach achieves accurate and real-time monitoring effect with minimal mobile communication and server processing costs. Our approach also allows a mobile object to negotiate a resident domain based on its computing capability. By having a larger resident domain, a more capable object has less chance of moving out of it and having to request a new one. As a result, both communication and server processing costs are reduced. Our comprehensive performance study shows that the proposed technique can be highly scalable in supporting location-based services in a wireless environment that consists of a large number of mobile devices.

**KEYWORDS:** wireless communications, mobile database systems, range query, continuous query, location-based service.

---

\*Department of Computer Science, Iowa State University, Ames, IA 50011, E-mail: yingcai@cs.iastate.edu

†School of Computer Science, University of Central Florida, Orlando, FL 32816, E-mail: kienhua@cs.ucf.edu

‡Department of Computer Science and Engineering, Pennsylvania State University, University Park, PA 16802, E-mail: gcao@cse.psu.edu

§Department of Computer Science, Iowa State University, Ames, IA 50011, E-mail: tobyxu@cs.iastate.edu

# 1 Introduction

The advances in wireless technologies and positioning systems will enable billions of online mobile appliances that are location-aware in the coming years [1]. These battery-powered devices, which have vastly different CPU speed and memory capacity, will create an enormous and heterogeneous mobile computing environment. In light of this vision, we consider in this paper the challenges of processing *range-monitoring queries*: given a set of user-defined spatial regions, retrieve the mobile objects inside them and provide real-time update as the mobile objects move in and out of these regions. Efficient processing of range-monitoring queries could enable many useful applications. For instances, in a disaster such as 9-11, rescuers may mark the dangerous areas, which can change dynamically, and alert people who are within or approaching those regions [2]; a teacher, on a field trip, may need to monitor several groups of children in different areas; a restaurant might want to know about people in its vicinity during lunch hours in order to send advertising messages; similarly, we might want to track traffic condition in some area and dispatch more police there if the vehicle density exceeds a certain threshold. In such applications, it is highly desirable and sometime critical to provide accurate results and update them in real time whenever mobile objects enter or exit the regions of interest.

The range-monitoring queries defined above are different from the conventional *range queries* that retrieve objects inside a query window at some snap of time point. A range-monitoring query is a continuous query and stays active for a certain time period until it is terminated explicitly by the user. As objects continue to move, the query results keep changing and require continuous updates. To process range-monitoring queries, a simple strategy is to have each object report its position as it moves; and for each location update, the server identifies the affected queries and updates their results if necessary. Although this approach can provide real-time query results, it has two serious problems. First, the constant location updates from mobile objects can quickly exhaust their battery power, since sending a wireless message takes a substantial amount of energy [3, 4, 5], compared to other procedures such as arithmetic operations. Second, when the number of mobile devices is large, the excessive location updates

generated by this bruce-force approach will present the server not only a severe communication bottleneck, but also an overwhelming workload of determining the affected queries and updating their results.

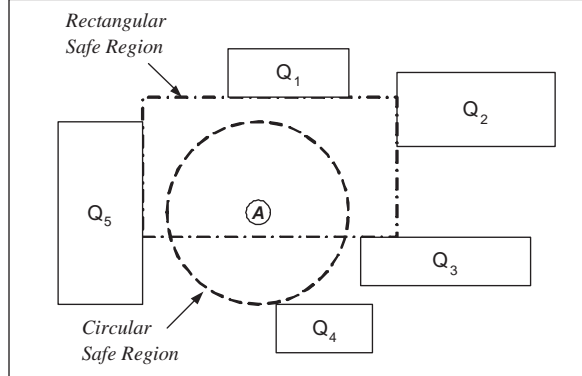


Figure 1: Examples of Safe Regions

To address the above problem, Prabhakar *et al* proposed a *Q-index* technique with a *safe region* concept [6, 7]. A safe region is defined to be either a circular or a rectangular region that contains the object’s current location and does not overlap with any query boundary. Figure 1 illustrates a mobile object *A* with its largest rectangular and circular safe regions. Since a safe region does not overlap with any query boundary, a mobile object does not need to report its location as long as its movement is limited within its safe region. The concept of safe region can dramatically reduce the number of location updates while providing real-time and accurate query results. Unfortunately, determining a safe region requires intensive computation. For example, computing a rectangular safe region takes from  $O(n)$  to  $O(n \log^3 n)$ , where  $n$  is the number of queries [6]. Since a safe region cannot overlap with any query boundary, it is typically very small in size. Thus, a mobile object can move out of its safe region quickly. Whenever this happens, the server needs to determine the object a new safe region and this may require to check the entire set of monitoring queries. The problem is even worse when adding a new query. In this case, the server may need to re-compute safe regions for all mobile objects because the new query rectangle could affect all existing safe regions. For instance, in Figure 1, the server must adjust *A*’s safe region if it overlaps with a new query. Because of these limitations, it is

unlikely that this approach can be used in a large scale real-time mobile system.

In this paper, we propose a scalable and adaptive technique, which we will refer to as *Monitoring Query Management* (MQM) [8, 9], for real-time processing of range-monitoring queries. Our goal is to support location-based services in a wireless environment consisting of a large number of mobile objects and monitoring queries. Similar to the safe region approach, MQM also leverages the computing capability of mobile devices to avoid constant location updates. However, our strategy is to allow mobile objects to monitor their movement directly against their nearby queries, instead of safe regions. Since an object can update the server whenever its movement affects any query result, real-time and accurate monitoring effects can be achieved. As we will show, making mobile objects aware of their nearby queries not only relieves the server from the overwhelming workload of continuous query evaluation, but also minimizes the expensive location updates. When an object moves, its nearby queries may keep changing and require continuous update. To address this problem, we propose a *resident domain* concept for mobile objects and develop a new spatial access method called *BP-tree* (Binary Partitioning tree) for efficient query management at the server side. We note that conventional database management systems are designed to manage data, not queries. Since range-monitoring queries are continuous queries, many can be active simultaneously. Existing database management systems need to be extended with real-time query management capability in order to support range-monitoring queries. The research reported in this paper could be viewed as a step toward enhancing databases with such functionality.

The remainder of this paper proceeds as follows. We present the design of our MQM technique in Section 2 and then introduce the BP-tree indexing technique in Section 3. In Section 4, we examine the performance results. We discuss other related work in Section 5. Finally, we give our concluding remarks in Section 6.

## 2 Monitoring Query Management

In this section, we first introduce the basic idea of the proposed technique and then present the server and mobile unit design in detail. In our discussion, we assume each range query is represented by a rectangular region and will refer to a range query as a region provided there is no risk of confusion. Without loss of generality, we assume that there is only one server. As in many mobile environments, we also assume that each mobile device has very limited computing resources in terms of CPU speed and memory capacity, and each is able to exchange information with a stationary server such as reporting its current position. The communications between server and mobile devices are through regular wireless broadcast. In practice, more efficient protocols such as *GeoCast* [10, 11] can be used for sending messages to mobile devices within a specific geographic region.

### 2.1 Basic Idea

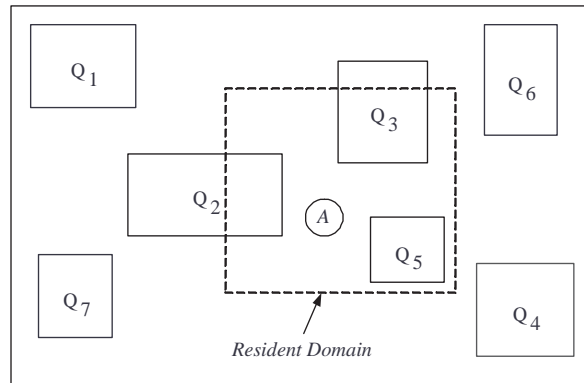


Figure 2: Example of Resident Domain

Our idea is to make each mobile object aware of its nearby queries directly, instead of a safe region. Specifically, we assign each object a *resident domain*, based on its current location, and notify it the queries that overlap with the domain. Figure 2 shows an object  $A$  with its resident domain and the overlapping queries  $Q_2$ ,  $Q_3$ , and  $Q_5$ . Since an object knows its resident domain and the queries inside it, the object can monitor its spatial relationship with them while it

moves. When it detects that it has crossed over some query boundary <sup>1</sup>, the object contacts the server to update the affected query results. Since an object knows exactly when such a report is needed, the mobile communication cost is minimized. We note that in safe region approach, an object knows only its safe region and needs to report to the server whenever it exits its safe region. However, moving out of a safe region in most cases does not affect any query result, because a safe region is just a small area that does not overlap with any query boundary. For example, in Figure 1, when object *A* exits its circular safe region, it needs to report the server, but no query result needs to be updated unless it moves into  $Q_4$ . In addition to minimizing the mobile communication cost, our approach also relieves the server from the overwhelming workload of query evaluation, because the queries are now actually evaluated distributively by their nearby mobile objects. Thus, the same server can be used to support many more mobile objects.

Compared to the safe region approach, our scheme requires more memory on mobile objects, because each object now needs to cache a number of query rectangles, instead of just a safe region. However, such overhead is minimal, considering that a query can be represented by 16 bytes and carrying 50 rectangles takes only 800 bytes. Our scheme also incurs more computation overhead on mobile objects. An object now needs to monitor its movement against a set of queries it carries, rather than a single safe region. It may first seem that such computation may result in more battery consumption. However, significant amount of energy is actually saved because of the substantial reduction on communication costs. In practice, power required by CPU is minimal compared to sending data over the wireless radio. For example, the energy cost of transmitting 1Kb over a distance of 100 meters is approximately 3 joules. By contrast, a general-purpose processor with 100 MIPS/W power could efficiently execute 3 million instructions for the same amount of energy [4]. In our case, it takes only 4 simple numerical comparisons to determine if a mobile object is inside a query rectangle and such calculation is needed only when an object moves.

---

<sup>1</sup>Given a rectangle  $R$  and an object's two consecutive sampling positions  $p_1$  and  $p_2$ , the object crosses one of  $R$ 's boundaries if  $R$  contains either  $p_1$  or  $p_2$ , but not both.

In addition to evaluating its nearby queries, an object also needs to monitor its movement against its resident domain, which actually can be treated as a query rectangle. When an object moves out of its resident domain, it needs to report to the server, which will then determine a new resident domain for the object. A technical problem here is how to determine a *suitable* resident domain. To minimize mobile communication and server processing costs, an object’s resident domain should be as large as possible. If it is too small, a moving object may have to frequently request for new resident domains – a problem similar in the safe region approach. However, if a resident domain is too large, it may contain too many queries and become unacceptable to a mobile object. In this paper, we measure the computing capability of a mobile object by the maximum number of queries it can load and process at a time. Thus, if an object’s processing capability is  $n$  queries, then its resident domain must contain its current location, should be as large as possible, but overlap no more than  $n$  monitoring rectangles. The problem of searching for such a resident domain may sound similar to that of searching for *K-Nearest Neighbors* (KNN) [12, 13] in the sense that we may try to find  $n$  query rectangles that are near an object’s current location. Existing KNN algorithms, however, are developed for point data, i.e., each neighbor is a point, and cannot be applied directly on spatial rectangular data.

In this paper, we address the problem of resident domain calculation by dynamically partitioning the database domain into many disjoint subdomains. Figure 3 shows an example of such partitioning. When a query overlaps with a subdomain, the overlapping area is called a *monitoring region* inside the subdomain and the query is a *relevant query* to the monitoring region. A query may create more than one monitoring region if it spans over more than one subdomain. For example,  $Q_1$  has only one monitoring region,  $R_1$ , while  $Q_2$  has two monitoring regions,  $R_{21}$  and  $R_{22}$ . On the other hand, a monitoring region can have multiple relevant queries if these queries overlap the same area in a subdomain. For example, both  $Q_3$  and  $Q_4$  are relevant to monitoring region  $R_{32}$ . With such domain partitioning and query decomposition, we can now use one or more subdomains as an object’s resident domain, as long as the number of monitoring regions inside it does not exceed the processing capability of the mobile object. This process is supported efficiently with a new spatial index structure called BP-tree (*Binary Partitioning*

tree), which we will discuss in detail in the next section.

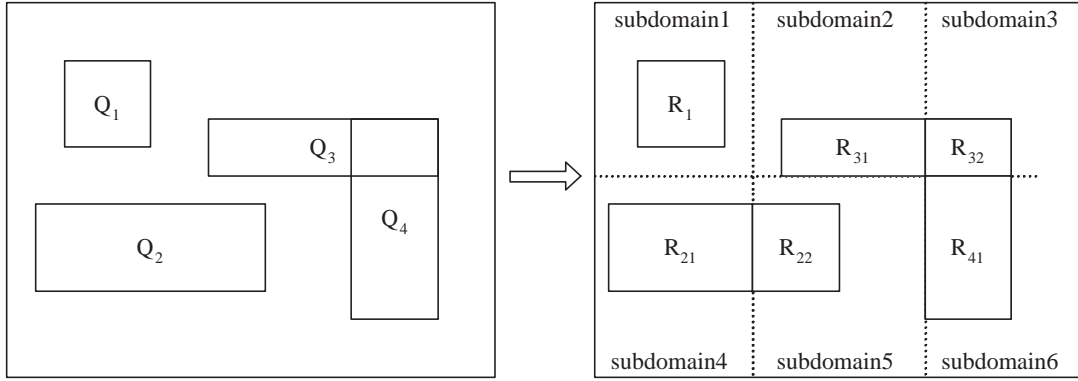


Figure 3: An Example of Domain and Query Decomposition

## 2.2 Server Design

At the server side, the subdomains and the monitoring regions are maintained using BP-tree. In addition, we use a binary relation, called *Relevance Table*, to track the queries and their monitoring regions. We recall that a query is considered *relevant* to a monitoring region if the query contains the monitoring region. Each row of the Relevance Table is a tuple of  $(r, q)$ , where  $r$  is a monitoring region and  $q$  is a query relevant to  $r$ . Many access structures can be used to retrieve the relevant queries efficiently given a monitoring region. For example, we can hash or build a B<sup>+</sup>-tree index on the monitoring-region field. Alternatively, we can also store the entire information in an adjacency matrix instead of a relational table. In the remainder of this paper, we will refer to this structure as a table and will not concern ourselves with its implementation details.

When a new range query  $q$  is submitted, the server searches the BP-tree for the subdomains it overlaps. For each of such subdomains, it determines the overlapping area, i.e., the monitoring region of this query inside this subdomain. The server then inserts a new tuple,  $(r, q)$ , to the relevance table, where  $r$  is the monitoring region. If the monitoring region does not already exist, it is also inserted to the BP-tree and the server broadcasts a message *AddMonitoringRegion*( $r$ ) to inform the mobile units that a new monitoring region is created. We will discuss how mobile

units respond to server messages shortly. We allow each subdomain to contain only a limited number of monitoring regions, determined by the minimum computing capability of mobile devices. When the number of monitoring regions in a subdomain exceeds a predetermined *split threshold*, the subdomain, say  $d$ , is further partitioned into two subdomains  $d_1$  and  $d_2$ . When this happens, the server broadcasts a *SplitDomain*( $d, d_1, d_2$ ) message to update the affected mobile objects.

When a query  $q$  is terminated, the server searches the relevance table and deletes all tuples containing  $q$  as the relevant query. If a tuple, say  $(r, q)$ , is deleted, and no other tuples in the table contain monitoring region  $r$ , then  $r$  is also deleted from the BP-tree. In this case, the server broadcasts a message *DeleteMonitoringRegion*( $r$ ). Deleting a monitoring region might cause a subdomain to underflow. To prevent sparse subdomains, we merge a subdomain with its split counterpart if the aggregate number of their monitoring regions drops below a predetermined *merge threshold*. In this case, the server broadcasts the message *MergeDomain*( $d_1, d_2, l$ ), where  $d_1$  and  $d_2$  are the two merging subdomains, and  $l$  is the combined list of monitoring regions inside  $d_1$  and  $d_2$ .

We assume that each mobile object is identified by a unique identifier. The server expects two types of messages from the mobile units, and processes them as follows:

- When an object  $oid$  enters or exits a monitoring region  $r$ , it sends an *UpdateQueryResult*( $r, oid, p$ ) message to the server, where  $p$  is the current position of the object. In response, the server searches the table for all queries that are relevant to this monitoring region. If a relevant query contains position  $p$ , then the object is inside the query region and  $oid$  should be in the query result. Otherwise, delete  $oid$  from the query result.
- When an object  $oid$  initializes itself or exits its current resident domain, it sends a message *RequestResidentDomain*( $oid, p, n$ ) to inquire its new resident domain, where  $p$  is the current position of the mobile object and  $n$  is the maximum number of monitoring regions it can accept. In response to this inquiry, the server searches the BP-tree to determine a resident domain for the mobile object. The server then broadcasts the message

*SetResidentDomain*(*oid*, *d*, *l*), where *d* and *l* denote the new resident domain of the object *oid* and the list of monitoring regions inside *d*, respectively.

## 2.3 Mobile Unit Design

The design of a mobile device consists of three main components: *Initialization*, *MessageListener*, and *RegionMonitor*. The following notations are used in the discussion of these components:

- *myID* : the unique identifier of the mobile unit;
- *myPos* : the current position of the mobile unit;
- *myDomain* : the current resident domain of the mobile unit;
- *myMRs* : the list of monitoring regions inside *myDomain*;
- *myCapacity* : the maximum number of monitoring regions acceptable to the mobile unit.

**Initialization**: This procedure is called when the mobile unit is powered on:

1. Set both *myDomain* and *myMRs* to *null*;
2. Spawn thread *MessageListener*;
3. Send message *RequestResidentDomain* (*myID*, *myPos*, *myCapacity*) to the server;
4. Spawn thread *RegionMonitor*.

**MessageListener**: The mobile unit listens to these messages and processes them as follows:

- *SetResidentDomain*(*oid*, *d*, *l*): If *oid* == *myID*, then do the following:
  - Set *OldDomain* = *myDomain*;
  - Set *myDomain* = *d*;
  - Set *myMRs* = *l*;

- If  $OldDomain == null$  (i.e., the object is in the initialization stage), then for each monitoring region  $r$  in  $myMRs$  that contains  $myPos$ , send an  $UpdateQueryResult(r, myID, myPos)$  message to the server.
- $AddMonitoringRegion(r)$ :
  - Add monitoring region  $r$  to  $myMRs$  if  $r$  is inside  $myDomain$ ;
  - If  $r$  contains  $myPos$ , send server a message  $UpdateQueryResult(r, myID, myPos)$ .
- $DeleteMonitoringRegion(r)$ : Delete monitoring region  $r$  from  $myMRs$  if  $r$  is inside  $myDomain$ .
- $SplitDomain(d, d_1, d_2)$ : If  $myDomain == d$ , then do the following:
  - If  $d_1$  contains  $myPos$ , set  $myDomain = d_1$ ; otherwise, set  $myDomain = d_2$ ;
  - For each monitoring region in  $myMRs$ , say  $r$ , do the following:
    - \* Delete  $r$  if it does not overlap with the new  $myDomain$ ;
    - \* Otherwise, replace  $r$  with the portion of the rectangle that is inside  $myDomain$ .
- $MergeDomain(d_1, d_2, l)$ : If  $myDomain$  overlaps with  $d_1$  or  $d_2$ , do the following steps:
  - Set  $myDomain$  to be the merged domain of  $d_1$  and  $d_2$ ;
  - Set  $myMRs = l$ .

**RegionMonitor**: When the mobile unit moves, it monitors its spatial relationships with its resident domain and the monitoring regions it knows as follows:

- If the object moves out of  $myDomain$ , then it requests for a new resident domain by sending the server a message  $RequestResidentDomain(myID, myPos, myCapacity)$ ;
- For each monitoring region  $r$  in  $myMRs$ , the object checks if it enters or exits  $r$  and when this happens, it sends a message  $UpdateQueryResult(r, myID, myPos)$  to update the server.

We note that *myCapacity* of a mobile device can be adjusted dynamically to reflect its processing capability at different times. When the device requires more CPU cycles and/or memory for other tasks with higher priorities, it can negotiate with the server for a smaller resident domain using a smaller *myCapacity*. Alternatively, we can consider allowing a mobile object to unilaterally reduce its resident domain to achieve the same effect. Although this option makes our technique even more flexible, we will not investigate it further in this paper. We also leave out the users of the location-based service. The users could connect to the server through conventional wired networks, or could be the mobile devices mentioned in the above discussion. For completeness, the server also needs to provide the interface for submitting queries and viewing query results. These issues are beyond the scope of this paper.

### 3 BP-Tree: Binary Partitioning Tree

A BP-tree consists of two types of nodes: *domain node* and *data node*. All internal nodes are domain nodes and all external nodes are data nodes. The main data structure for a domain node is two entries, each having the form  $(R, P)$ , where  $R$  holds the upper-left and lower-right coordinates of a rectangular subdomain, and  $P$  links another domain node or a data node. Each domain node represents a decomposition of a parent domain. As illustrated in Figure 4, the decomposition of subdomain  $d_2$  consists of two subdomains,  $d_{21}$  and  $d_{22}$ , each stored in one entry of the domain node representing  $d_2$ . In addition to the two entries, each domain node also uses a variable, *size*, to record the total number of monitoring regions indexed under this domain node. A data node stores the monitoring regions that are inside its parent subdomain. A data node contains an array of rectangles, each holding a monitoring region, and a variable *size*, recording the total number of monitoring regions. As an example, the data node linked by the domain entry of  $d_{11}$  in Figure 4 stores all monitoring regions inside  $d_{11}$ . We note that the size of data nodes is limited by the minimum processing capability assumed for mobile objects. This parameter is used to determine the split threshold for data nodes. Thus, a mobile object can load at least one data node.

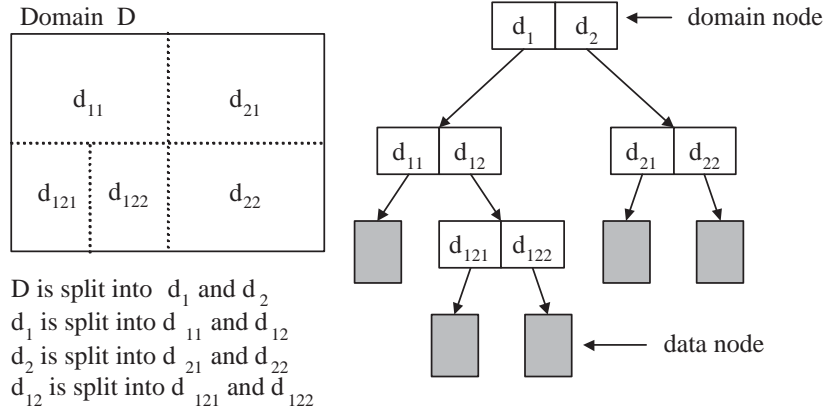


Figure 4: A BP-tree Example

The data structure of BP-tree efficiently supports the operation of resident domain search. Given a mobile object at position  $p$  with a processing capability of  $n$  queries, we determine its resident domain by searching from the root of BP-tree. If the number of monitoring regions inside the root domain is acceptable to the object (i.e., no larger than  $n$ ), the root domain becomes the object's resident domain. Otherwise, we descend the tree to check the subdomain that contains position  $p$ . The subdomain is the object's resident domain if the object can load all monitoring regions inside it; otherwise, we check the child domains of the subdomain and this process is done recursively until we find a subdomain in which the number of monitoring regions is acceptable to the object. We note that in the worst case, a mobile object takes a leaf subdomain as its resident domain. When a resident domain is determined, we then retrieve all monitoring regions inside it and send them to the requesting object.

With BP-tree, the monitoring regions are grouped according to their containing subdomains and each group is stored in one data node. Meanwhile, the domain decomposition hierarchy is captured by the organization of BP-tree domain nodes. Before we discuss the detailed operations of BP-tree, we define the following notations:

- Given an entry  $(R, P)$  in a domain node,  $R.P$  denotes the child node pointed at by  $P$ .
- Given a domain node  $D$ ,  $D.domain$  is the domain rectangle represented by this node,

$D.parent$  refers to the parent node who has an entry pointing to  $D$  ( $D.parent$  is *null* if  $D$  is the BP-tree root), and  $D.size$  is the total number of monitoring regions stored in the data nodes descending from  $D$ .

- Given two rectangles,  $R_1$  and  $R_2$ ,  $R_1 \cap R_2$  represents their overlapping area.

### 3.1 Search

When the server receives message  $RequestResidentDomain(oid, p, n)$ , it needs to determine a resident domain for the object  $oid$ , given its current position  $p$  and computing capability  $n$ . The resident domain should contain as many monitoring regions as possible, but no more than  $n$ . With BP-tree, this can be done efficiently by calling  $Search(root, p, n)$ , where  $root$  is the BP-tree root:

#### Search(D, p, n)

1. If  $D.size \leq n$ , then return  $D.domain$  and all monitoring regions indexed under  $D$ .  $D.domain$  is the new resident domain for the requesting mobile object;
2. Otherwise, search for the entry, say  $(R, P)$ , in  $D$  such that rectangle  $R$  contains position  $p$ ;
3. Recursively call  $Search(R.P, p, n)$ .

### 3.2 Insert

A BP-tree is initialized as a root domain node with one empty data node. That is, the first entry of the root is set to  $(R, P)$ , where  $R$  is the entire domain and  $P$  points at an empty data node. The variable  $size$  of the node is set to 0. When a new query arrives, the server descends the BP-tree to look for the data nodes whose domains overlap with the query rectangle. For each overlapping area, i.e., a monitoring region, say  $r$ , a new tuple  $(r, q)$  is added to the relevance

table. The monitoring region is also inserted into the BP-tree if it does not already exist in BP-tree. The fact that only distinct monitoring regions are stored in the BP-tree allows our technique to deal with overlapping queries. When a new monitoring region is inserted to a data node, the variable *size* of each domain node in the searching path, from the root to the data node, is increased by 1. Thus, given a domain node, we can know easily the total number of monitoring regions it contains.

An insert might cause a data node to overflow. When this happens, its domain is split. A number of decomposition schemes can be used to split a domain. A simple approach is *center split*, i.e., split the domain vertically or horizontally into two equal-sized subdomains. The direction of the split can be determined by comparing the dimensions of the domain. For example, we can split on the longer dimension to avoid having long and narrow subdomains. The monitoring regions spanning over the split line are also split and the relevance table is updated accordingly. Each time a data node is split, the server broadcasts a message  $SplitDomain(d, d_1, d_2)$  to notify mobile units that domain  $d$  has been decomposed into  $d_1$  and  $d_2$ . We have discussed how a mobile unit reacts to such a message.

When a new query  $q$  arrives, we call the following  $Insert(D, q)$  procedure, where  $D$  is set to be the BP-tree root:

**Insert(D, q)**

1. If  $D$  is a domain node, then for each entry  $(R, P)$  in  $D$ , call  $Insert(R, P, q)$  if  $R$  overlaps with  $q$ ;
2. If  $D$  is a data node, then do the following:
  - Set  $r$  to be equal to  $q \cap D.domain$ ;
  - Insert a new tuple,  $(r, q)$ , to the relevance table;
  - If no monitoring region in  $D$  is equal to  $r$ , then do the following:
    - Add monitoring region  $r$  to  $D$ ;

- Broadcast *AddMonitoringRegion(r)* message;
- Set  $D'$  equal to  $D.parent$  and repeat the following until  $D'$  is *null*:
  - \* Increase  $D'.size$  by 1;
  - \* Set  $D'$  equal to  $D'.parent$ .
- If  $D$  is full, call *SplitDataNode(D)*.

**SplitDataNode(D)**: Split BP-tree data node  $D$

1. Look for the entry, say  $(R, P)$ , in  $D.parent$  such that  $P$  points at  $D$ ;
2. Split domain  $R$  into two subdomains,  $R_l$  and  $R_r$ ;
3. Broadcast message *SplitDomain(R, R<sub>l</sub>, R<sub>r</sub>)*;
4. Create two new data nodes, *left* and *right*;
5. Create a new domain node,  $D'$ , and set its two entries to be  $(R_l, P_l)$  and  $(R_r, P_r)$ , where  $P_l$  and  $P_r$  point to data nodes *left* and *right*, respectively;
6. Redirect  $P$  in the entry  $(R, P)$  of  $D.parent$  to point at  $D'$ ;
7. For each monitoring region  $R_i$  stored in  $D$ , do the following:
  - If  $R_i$  overlaps with  $R_l$  and monitoring region  $R_i \cap R_l$  does not exist in *left*, then do the following:
    - Insert monitoring region  $R_i \cap R_l$  into *left*;
    - Increase *left.size* by 1.
  - If  $R_i$  overlaps with  $R_r$  and monitoring region  $R_i \cap R_r$  does not exist in *right*, then do the following:
    - Insert monitoring region  $R_i \cap R_r$  into *right*;
    - Increase *right.size* by 1.

- If  $R_i$  overlaps with both  $R_l$  and  $R_r$ , then for each tuple, say  $(r, q)$ , in the relevance table, do the following if  $r$  is equal to  $R$ :
    - Replace tuple  $(r, q)$  with  $(R_i \cap R_l, q)$ ;
    - Add new tuple  $(R_i \cap R_r, q)$  to the table.
8. Set  $D'.size$  equal to  $D.size$  and repeat the following until  $D'$  is *null*:
    - Increase  $D'.size$  by  $left.size + right.size - D.size$ ;
    - Set  $D$  equal to  $D'.parent$ .
  9. Discard  $D$ ;
  10. Call  $SplitDataNode(left)$  if  $left$  is full;
  11. Call  $SplitDataNode(right)$  if  $right$  is full.

### 3.3 Delete

*Delete* operation is used when a query, say  $q$ , needs to be terminated. The server first checks the relevance table and deletes all tuples containing  $q$  as the relevant query. If a tuple, say  $(r, q)$ , is deleted and there are no more tuples in the table containing monitoring region  $r$ , then  $r$  is also deleted from the BP-tree by calling  $Delete(D, r)$ , where  $D$  is the BP-tree root. A message  $DeleteMonitoringRegion(r)$  is then broadcast. Deleting a monitoring region may cause a subdomain to underflow, in which case the subdomain and its split counterpart are merged. When two subdomains, say  $d_1$  and  $d_2$ , are merged, the server broadcasts a message  $MergeDomain(d_1, d_2, l)$ , where  $l$  is the combined list of monitoring regions inside  $d_1$  and  $d_2$ . A more formal description of *Delete* operation is given below.

**Delete(D, r)**: Delete monitoring region  $r$  from the BP-tree rooted at node  $D$

1. Decrease  $D.size$  by 1;

2. If  $D$  is a domain node, then search for the entry, say  $(R, P)$ , in  $D$  such that  $R$  contains  $r$ , and call  $Delete(R.P, r)$ ;
3. Otherwise,  $D$  is a data node and do the following:
  - Remove monitoring region  $r$  from  $D$ ;
  - Call  $MergeUnderflows(D.parent)$ .

**MergeUnderflows(D)**: Merge the children nodes of  $D$  when necessary

1. If both children of  $D$  are data nodes and they can be merged into one, then do the following:
  - Create a new data node,  $D'$ ;
  - Move all monitoring regions stored in both children nodes of  $D$  into  $D'$ ;
  - Search for the entry, say  $(R, P)$ , in  $D.parent$  such that  $P$  points at  $D$ , and redirect  $P$  to point at  $D'$ ;
  - Discard  $D$  and its children nodes;
  - Call  $MergeUnderflows(D'.parent)$ .

## 4 Performance Study

For the purpose of performance comparison, we have implemented a detailed simulator for both Q-index [6, 7] and MQM techniques. The visualization of a simulation run can be found at [14], which clearly shows the difference of their performance. Since we are also interested in how the overall system performance is improved by taking advantage of the heterogeneous mobile computing capability, we implemented two versions of MQM:

- *Plain MQM* : This version allows each mobile object to cache and process only the monitoring regions stored in one data node. That is, it exploits only the minimum mobile computing capability, which is dictated by the least capable mobile device.

- *Adaptive MQM* : In this implementation, each mobile object is allowed to load monitoring queries as many as possible, according to their true computing capability. This is the proposed scheme in this paper.

## 4.1 Performance Metrics

The performance metrics selected for this study are as follows:

- *Server Processing Cost* : This cost is measured as the total number of index-tree nodes accessed in order to process requests from the mobile objects. This is a good measure of the server processing cost because server operations involve mainly navigating the BP-tree and processing the data stored in the leave nodes. The cost of searching the Relevance Table is ignored because it can be implemented as a hash file, and takes only  $O(1)$  to retrieve the relevant queries for a given monitoring region.
- *Server Communication Cost* : This cost is measured as the total number of messages transmitted from the server (to the mobile units). This is a good measure of the relative server communication cost because our messages are very short. For example, if we use 16 bytes to identify a monitoring rectangle, then it takes only 800 bytes to represent a full data node with 50 entries.
- *Mobile Communication Cost* : This cost is measured as the total number of messages sent by the mobile objects to the server. Again, because the messages are very short, this measure can reflect rather accurately the relative mobile communication costs under MQM and Q-index approaches.

We differentiate the two types of communication costs because sending a message requires a mobile device substantially more power than listening and receiving a message. The rationale for these three performance metrics is as follows:

- *Scalability Measure*: The server processing cost and server communication cost are good indicators of whether the server can become a bottleneck. They are good measures of system scalability.
- *Power Conservation Measure*: The mobile communication cost alone is a good measure of power conservation. We do not take into account the mobile computation cost because it is negligible compared to the power required for transmitting wireless data at a relatively high rate [4, 5].

We note that although the above model cannot predict the exact costs of these techniques, the intention of this performance study is to predict their relative performance under varying circumstances. Thus, the better technique, as measured by the above metrics, is able to scale up better to support a larger mobile community, and requires less battery power from the mobile devices.

## 4.2 Simulation Model

We implemented the BP-tree using a center-split strategy as discussed before - splitting at the middle of the longer dimension. The same BP-tree is used to index the monitoring regions for both MQM techniques and the safe region approach. This allows us to compare the server computation costs of these techniques fairly. The concept of *safe region* is implemented as follows. Given a mobile object, we first search the BP-tree for the subdomain that contains its current location. We then compute the largest circular region based on the monitoring regions within the subdomain as the safe region for this object, such that the safe region does not overlap with the boundaries of any query. We choose not to use rectangular safe region because its implementation is much more complicated while the achieved performance is quite similar to that of using circular safe region, as indicated in [6]. We note that we actually compare our technique with an improved safe region approach in this study. Our BP-tree approach avoids the excessive workload of computing a safe region by limiting the consideration to only the monitoring regions stored in one data node. It, however, can achieve a near-optimal safe region.

This is due to the fact that the size of a safe region is mainly determined by the query regions surrounding its host object. With the original Q-index approach, it would have to examine all the queries for the safe region of each mobile object. Obviously, this is not feasible for a real-time system. In fact, the technique discussed in [6] determined the safe region only once at system startup due to the high cost. The algorithm has a complexity of  $O(n^2)$ , where  $n$  is the total number of queries. It is not clear how they handle the situation when an object exits its current safe region. With the new strategy, we can compute a new safe region easily.

For each simulation run, we generated a number of square range-monitoring queries with random sizes ranging from  $10 \times 10$  to  $100 \times 100$  and placed them over a rectangular database domain of  $[0 \dots 100K, 0 \dots 100K]$  following a uniform distribution. The performance data under other distributions such as Gaussian and Poisson were also collected, but omitted here as they are quite similar. We then generated a number of mobile objects and placed them randomly in the database domain. The computing capability of these objects ranges from 50 to 500 monitoring regions, following a zipf distribution. Thus, the size of BP-tree data node is set to be 50 rectangles. We assume each object has constant processing capability throughout the simulations. Similar to [6], the velocities of these mobile objects follow a zipf distribution with a deviation of 0.7, and fall in between 0 and 20 per time unit. The velocity of each object is also constant throughout each simulation run. Their initial moving directions are set randomly. Each object moves linearly until it reaches any one of the four boundaries of the database domain, in which case it reflects its direction and continues to move at the same speed. This process is repeated and ended at 10,000 simulated time units. Given a same set of monitoring queries, which are uniformly sized and distributed, the number of query boundaries crossed by each individual object is likely to increase linearly as the simulation time period increases. Thus, the overall communication and processing costs under both MQM and Q-index also increase. We choose to report the performance results collected at 10,000 time units because we observed that after this time point, the factor of the performance gap between MQM and Q-index became rather stabilized. Table 1 summarizes the workload and system parameters used in the simulation.

Table 1: Parameters used for the simulation studies.

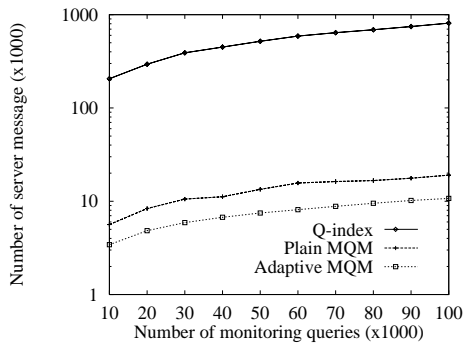
PARAMETER	DEFAULT	RANGE
Domain space	[0...100K, 0...100K]	N/A
Number of mobile objects	500	100 - 1,000
Number of monitoring queries	50,000	10,000 - 100,000
Skew of mobile processing capability	0.5	0.1 - 1.0
Mobile processing capability (queries)	N/A	50 - 500
Sizes of monitoring queries	N/A	10x10 - 100x100
Velocities of mobile objects	N/A	0-20
Velocity skew factor	0.7	N/A
Simulation time	10,000	N/A

### 4.3 Simulation Results

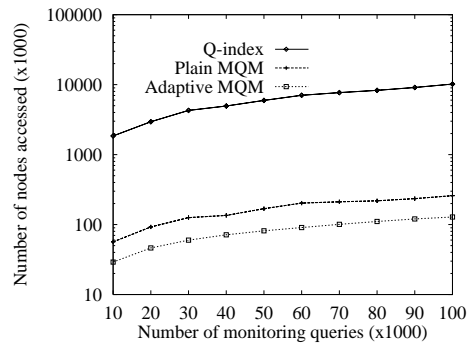
We are mainly interested in the scalability and robustness of the proposed technique. Therefore, we study how the performance metrics are affected by the number of monitoring queries, the number of mobile objects, and the skew factor of mobile computing capability. We report and explain the performance results as follows.

#### 4.3.1 Scalability w/r to number of monitoring queries

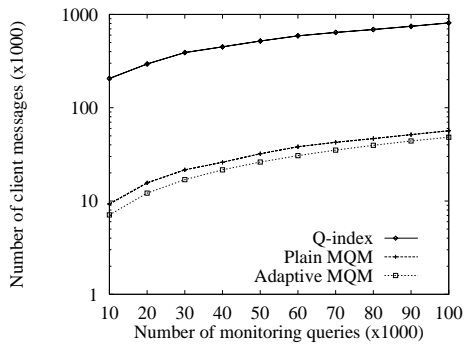
In this study, we generated 500 mobile objects and set the skew factor of mobile computing capability to be 0.5. We increased the number of monitoring queries from 10,000 to 100,000. The performance results are plotted in Figure 5. They show that Q-index performs much worse than the two MQM approaches. Under Q-index, each object is associated with a safe region. Each time an object moves out of its safe region, it needs to make a location update. Since a safe region cannot overlap with any query boundary, an object's safe region is normally many times smaller than its resident domain, which consists of at least one subdomain. As a result, the object may need to contact the server very frequently for new safe regions, generating high



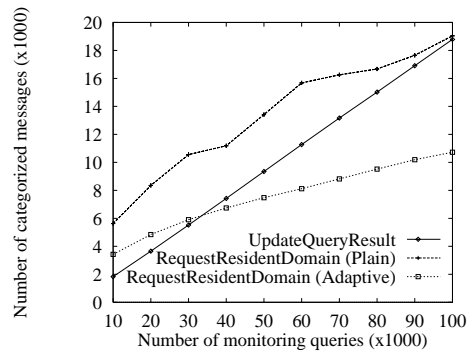
(a) Server communication cost



(b) Server processing cost



(c) Mobile communication cost



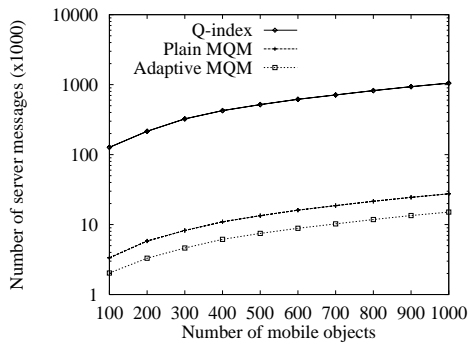
(d) Categorized mobile communication costs

Figure 5: Scalability w/r to number of monitoring queries

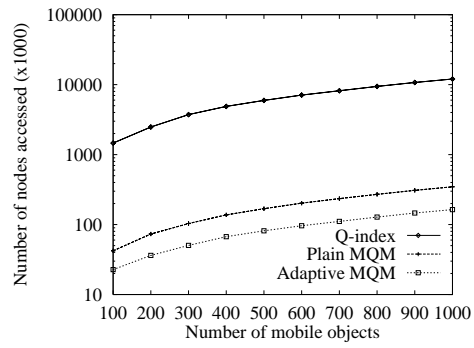
communication costs. When an object moves out of its safe region, the server needs to compute a new safe region and search the BP-tree to determine if any query results need to be updated. Such computation is often wasted since exiting a safe region does not imply that the object has crossed over some query boundary. In contrast, an object in MQM is aware of its nearby monitoring regions and can report the boundary-crossing events directly. As a result, MQM incurs much less server processing cost than Q-index, as showed in Figure 5(b). This study also shows that Adaptive MQM outperforms its Plain counterpart noticeably. Under MQM, there are two types of messages sent by mobile objects: *UpdateQueryResult* and *RequestResidentDomain*. The former message is sent when an object crosses a query boundary while the later one is sent when an object moves out of its current resident domain. Since the number of boundary-crossing events is fixed, both MQM techniques generate the same number of *UpdateQueryResult* messages. Thus, their performance difference is determined by the number of *RequestResidentDomain* messages. Figure 5(d) shows that Adaptive MQM incurs about 50% less *RequestResidentDomain* messages than Plain MQM. By loading monitoring queries as many as possible, an object can have a maximum size of resident domain. Thus, the object has less chance of moving out of its resident domain and has to request a new one, minimizing both communication overheads and server processing cost.

### 4.3.2 Scalability w/r to number of mobile objects

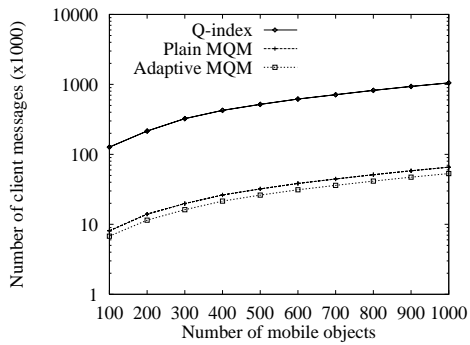
In this study, we generated 50,000 monitoring queries and set the skew factor of mobile processing capability to be 0.5. The number of mobile objects is varied from 100 to 1000. The simulation results are plotted in Figure 7. As the number of mobile objects increases, all three techniques incur higher communication and server processing costs. However, the costs under Q-index are many times higher than those under MQM approaches. Again, this is due to the fact that an object in MQM is aware of its nearby monitoring regions and can detect when its movement affects any query results, and in particular, an object’s resident domain is usually much larger than its safe region. This performance study again shows that leveraging the heterogeneous mobile computing capability can significantly reduce the communication and server processing



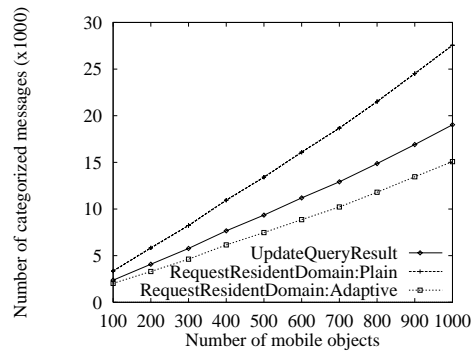
(a) Server communication cost



(b) Server processing cost



(c) Mobile communication cost



(d) Categorized mobile communication costs

Figure 6: Scalability w/r number of mobile objects

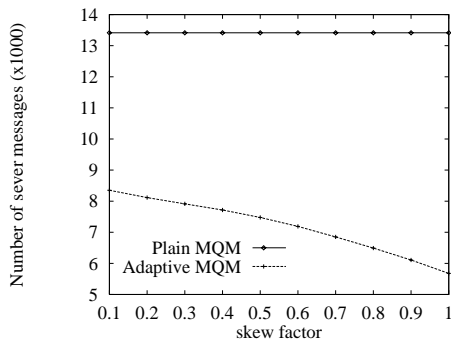
costs. Figure 7(d) shows that the number of *RequestResidentDomain* messages generated by Plain MQM is about one time more than that by Adaptive MQM. Accordingly, Adaptive MQM incurs much less server processing cost, as showed in Figure 7(b). Thus, this scheme is highly adaptive and more scalable in terms of supporting more mobile objects.

### 4.3.3 Effect of the skew of mobile computing capability

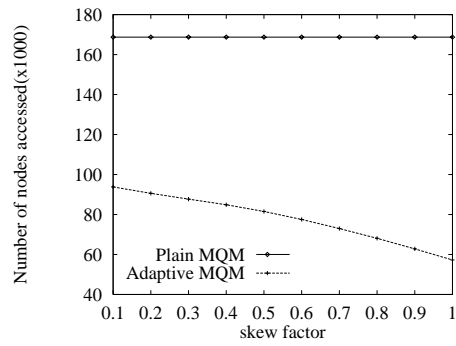
This study evaluated how the performances of two MQM techniques are affected by the skew of mobile computing capability. We generated 500 mobile objects and 50,000 monitoring queries. The skew of mobile computing capability is varied from 0.1 to 1.0, where a higher skew means a higher average of mobile processing capability. The simulation results are plotted in Figure 7. Since Plain MQM allows each mobile object to use only the minimally assumed computing capability, i.e., caching at most 50 queries in this simulation at one time, this approach cannot perform better even when mobile processing capability improves. Thus, its performance curves are flat. By contrast, both communication and server processing cost under Adaptive MQM drop as the skew factor increases. As showed in Figure 7(d), the number of *RequestResidentDomain* messages generated by Adaptive MQM decreases significantly when the objects become more and more capable in average in loading monitoring queries. This study confirms again that exploring heterogeneous mobile computing capability can effectively reduce both communication and server processing costs.

## 5 Related Works

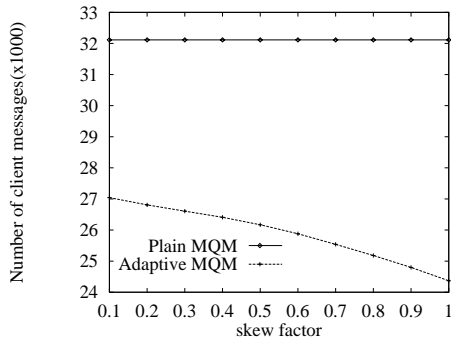
Extensive study has been done on range queries over a set of mobile objects. To avoid continuous location update, an object can report the server its initial position and velocity information, by which the server can estimate the object's future location given a time point. As a result, the object does not need to report its position until the deviation between its actual location and the *computed* location exceeds some threshold [15, 16, 17]. At the server side, the continuous movements of mobile objects can be approximated as many linear segments, which can then be



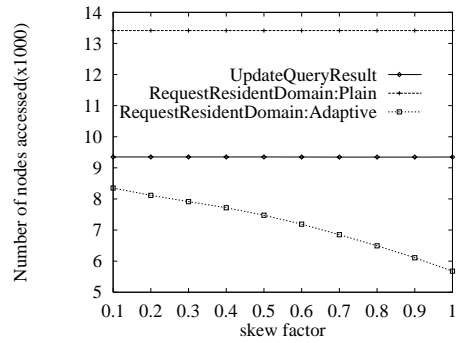
(a) Server communication cost



(b) Server processing cost



(c) Mobile communication cost



(d) Categorized mobile communication costs

Figure 7: Effect of the skew of mobile computing capability

indexed to support efficient range queries. Many techniques have been proposed for this purpose. For examples, [18] proposes to transform line trajectories into points in a higher-dimensional space, and then index these points using regular spatial indices. In [19, 20], the trajectories are indexed through their bounding rectangles that are time-parameterized. Another indexing scheme using external range trees [21] was presented in [22]. Performing range queries over historical data was discussed in [23, 24]. Other recent works on supporting spatial-temporal data can be found in [25, 26, 27, 28]. Although these techniques can efficiently approximate mobile objects inside a query window at some time point or within a certain time interval, they are server-centric and rely on location estimation, assuming that the behavior of mobile objects does not change frequently. In contrast, our MQM is fully distributed and can provide real-time and accurate<sup>2</sup> monitoring results. Since it does not use any location estimation, it is more robust in the sense that and can be used for all location management applications, especially those where the mobility pattern is unpredictable.

The problem of moving range queries over static objects was investigated in [29, 30]. A moving query is typically associated with a mobile object and the query window changes as the object moves. For example, a car driver may issue a moving query such as “retrieving the gas stations within 5 miles as I drive in the next 30 minutes”. Similar to our range-monitoring queries, a moving query is a continuous query and its query results may keep changing. However, a range-monitoring query does not change its query window and it returns mobile objects. In contrast, a moving range query changes its query region and in this case, it returns static objects. Continuous moving queries over a set of mobile objects was recently studied in [31, 32, 33]. The techniques proposed in [32, 33] are server-centric approaches and do not assume mobile computing capability. They rely on the location estimation and trajectory indexing to minimize mobile communication and server processing costs. *MobiEyes* proposed in [31] shares some similarity with our MQM in that both leverage the computing capability of mobile objects for distributed query evaluation. *MobiEyes*, however, relies on location estimation in order to

---

<sup>2</sup>In this paper, we ignore the small inaccuracy caused by positioning systems. Thus, each object knows its *true* position.

minimize the location updates of the mobile objects that issue moving queries and the mobile objects that are near these query regions.

Another related work is the location management and paging services in cellular communication systems [34, 35, 36, 37, 38]. A cellular network divides the whole service area into a collection of *cells*. Each cell has a *base station* which communicates with the mobile objects in the cell. These base stations are interconnected with high-speed backbone networks that serve as the bridges for the mobile users to communicate with each other. As a mobile device moves from one cell to another, its point of attachment to the fixed network changes. Therefore, a central issue of the cellular networks is how to store, query, and update the information about the cells in which mobile objects reside. At one extreme, the location information of all users is replicated and maintained at all network sites. In this case, any one of these databases can be queried to locate a user. However, each time an object moves into a different cell, the location databases at all network sites must be updated. At the other extreme, location information is not maintained, and therefore no update cost. Locating a mobile user, however, requires paging the entire service area. To balance the lookup and update costs, much effort has been done on various aspects of such location databases, such as architecture, placement and optimization, cache and replication, and so on. A comprehensive survey on this topic can be found in [39]. Although monitoring cellular-phone users within a fixed cell can be viewed as a *static range-monitoring query*, the above techniques cannot handle the ad hoc range-monitoring queries discussed in this paper.

## 6 Concluding Remarks

With the falling price of wireless connection and miniaturized electronic components, we are experiencing a dramatic increase in the number of online wireless appliances, such as mobile terminals, PDAs, wrist watches, cars, and so on. Take the cellular phone case as an example. In 1994, 16 million Americans subscribed to cellular phone services. As of April of 2004, this number has increased to more than 162 millions [40]. Some experts predict that worldwide

subscribership will reach 1.2 billion people by 2005 [41]. Together with Global Positioning Systems (GPS), a substantial portion of these mobile devices will be location-aware. In fact, under the E-911 Phase II mandate from the U.S. Federal Communications Commission (FCC), cellular phone companies must soon provide the means to track a caller's location on 911 calls made from their phones [42].

The perspective of a pervasive computing society has spurred a great research interest in database systems for location-based services. In this paper, we addressed the challenge of providing region monitoring services that require continuous and real-time update of query results. The proposed technique, *Monitoring Query Management*, will enhance traditional database management systems with the capability for real-time query management. The advantages of the proposed technique are as follows:

- *Low Communication Cost* : A mobile object does not need to report to the server unless the object enters or exits a monitoring region, or moves out of its current resident domain. This strategy significantly reduces its power consumption for wireless communication.
- *Scalability* : The range queries are evaluated distributively relieving the server from becoming a bottleneck. This feature makes MQM highly scalable allowing it to support a very large number of mobile objects and range-monitoring queries.
- *Reliability* : MQM is more reliable than techniques based on location estimation. These techniques typically require server to estimate the object locations according to some velocity model in order to save the update communication costs. For techniques on balancing the update cost and imprecision, interested readers are referred to [43], [15], [16], and [44]. The query results based on the estimated locations are approximated and could be inaccurate. As a contrast, MQM provides accurate and real-time query results.
- *Simplicity* : Another advantage of MQM is its simplicity. It does not use any velocity models or dead-reckoning algorithms [44].

To assess the performance of MQM, we implemented a detailed simulator to compare it with an

improved version of Q-index in terms of server computation cost, server communication cost, and mobile communication cost. The first two metrics are good indicators of the scalability of the proposed techniques, while the third metric is a good measure of power conservation. Our simulation results, under various workloads, indicate that MQM outperforms Q-index significantly in all three performance metrics.

## 7 Acknowledgements

The authors thank the associate editor and anonymous reviewers for their comments and valuable suggestions. We also gratefully acknowledge that this work was partially supported by funding provided through the Technology Commercialization Acceleration Program (Contract No. 400-65-87) at the Iowa State University.

## References

- [1] C. S. Jensen, A. Friis-Christensen, T. B. Pedersen, D. Pfoser, S. Saltenis, and N. Tryfona. Location-Based Services – A Database Perspective. In *Proc. of the Eighth Scandinavian Research Conference on Geographical Information Science*, pages 59–68, Norway, June 2001.
- [2] Students Create Global Positioning System Text Messages. Web page at [http://www.usatoday.com/tech/news/2004-06-28-search-and-rescue\\_x.htm](http://www.usatoday.com/tech/news/2004-06-28-search-and-rescue_x.htm).
- [3] M. Stemm and R. H. Katz. Measuring and Reducing Energy Consumption of Network Interfaces in Hand-held Devices. *IEICE Trans. on Communications*, vol.E80-B, no.8, p. 1125-31, E80-B(8):1125–31, 1997.
- [4] G.Pottie and W. Kaiser. Wireless Sensor Networks. *Communications of the ACM*, 43(5):51–58, May 2000.
- [5] S. Madden and M. Franklin. Fjording the Stream: An Architecture for Queries Over Streaming Sensor Data. In *Proc. of Intl. Conf. Data Engineering*, 2002.
- [6] S. Prabhakar, Y. Xia, D. Kalashnikov, W. G. Aref, and S. hambrusch. Queries as Data and Expanding Indexes: Techniques for Continuous Queries on Moving Objects. In *TR., Dept. of Computer Science, Purdue University*, 2000.
- [7] S. Prabhakar, Y. Xia, D. Kalashnikov, W. G. Aref, and S. hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 15(10):1124–1140, October 2002.

- [8] Ying Cai and Kien A. Hua. An Adaptive Query Management Technique for Efficient Real-time Monitoring of Spatial Regions in Mobile Environments. In *Proc. of the 21st IEEE Int'l Performance, Computing, and Communication Conference (IPCCC)*, pages 259–266, April 2002.
- [9] Ying Cai, Kien A. Hua, and Guohong Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *IEEE Int'l Conference on Mobile Data Management (MDM'04)*, pages 27–38, January 2004.
- [10] J. C. Navas and T. Imielinski. Geographic Addressing and Routing. In *Proc. of ACM/IEEE MobiCom'97*, Budapest, Hungary, September 1997.
- [11] T. Imielinski and J. C. Navas. GPS-Based Geographic Addressing, Routing, and Resource Discovery. *Communications of the ACM*, pages 86–92, April 1999.
- [12] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD*, 1995.
- [13] T. Seidl and H. P. Kriegel. Optimal Multi-Step k-Nearest Neighbor Search. In *SIGMOD*, 1998.
- [14] Monitoring Query Managment Project. Web page at [http://www.cs.iastate.edu/~yingcai/project/demo\\_mqm.html](http://www.cs.iastate.edu/~yingcai/project/demo_mqm.html).
- [15] O. Wolfson, S. Chamberlain, L. Jiang, and G. Mendez. Cost and Imprecision in Modeling the Position of Moving Objects. In *Proc. of Intl. Conf. Data Engineering*, pages 588–596, 1998.
- [16] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving Objects Databases: Issues and Solutions. In *Proc. of the 10th Int'l Conference on Scientific and Statistical Database Management*, pages 111–122, July 1998.
- [17] K. Lam, O. Ulusoy, T. S. H. Lee, E. Chan, and G. Li. An Efficient Method for Generating Location Updates for Processing of Location-Dependent Continuous Queries. In *DASFAA'01*, pages 218–225, Hong Kong, China, April 2001.
- [18] G. Kollios, D. Gunopulos, and V. J. Tsotras. On Indexing Mobile Objects. In *Proc. of ACM PODS'99*, pages 261–272, 1999.
- [19] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *ACM Proc. of SIGMOD'00*, pages 331–342, 2000.
- [20] Y. Tao, D. Papadias, and J. Sun. The TPR\*-Tree: An Optimized Spatio-temporal Access Method for Predictive Queries. In *VLDB*, 2003.
- [21] L. Arge, V. Samoladas, and J.S. Vitter. On Two-dimensional Indexability and Optimal Range Searching Indexing. In *Proc. of ACM PODS'99*, pages 346–357, 1999.
- [22] P. K. Argarwal, L. Arge, and J. Erickson. Indexing Moving Points. In *Proc. of ACM PODS'00*, pages 175–186, 2000.

- [23] D. Pfoser, Y. Theodoidis, and C. S. Jensen. Indexing Trajectories of Moving Point Objects. In *ChoroChronos Technical Report, CH-99-3*, 1999.
- [24] D. Pfoser, C. S. Jensen, and Y. Theodoidis. Novel Approaches in Query Processing for Moving Objects. In *Proc. of VLDB'00*, 2000.
- [25] L. Forlizzi, R. H. Guting, E. Nardelli, and M. Scheider. A Data Model and Data Structures for Moving Objects Databases. In *Proc. of ACM SIGMOD'00*, pages 319–330, 2000.
- [26] J. Tayeb, O. Ulusoy, and O. Wolfson. A Quadtree-based Dynamic Attribute Indexing Method. *The Computer Journal*, 41(3):185–200, 1998.
- [27] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing Access Methods for Bitemporal Databases. *IEEE TKDE*, 10(1):1–20, 1998.
- [28] B. Salzberg and V. J. Tsotras. A Comparison of Access Methods for Temporal Data. In *TimeCenter TR-13*, 1997.
- [29] Z. Song and N. Roussopoulos. K-nearest Neighbor Search for Moving Query Point. In *SSTD*, 2001.
- [30] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *VLDB*, 2002.
- [31] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT*, 2004.
- [32] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incrementable Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, 2004.
- [33] B. Gedik, K-L. Wu, P. Yu, and L. Liu. Motion Adaptive Indexing for Moving Continual Queries over Moving Objects. In *CIKM*, 2004.
- [34] M. Sidi and I. Cidon. A Multi-Station Packet-Radio Network. *Performance Evaluation*, 35(2):65–72, February 1988.
- [35] R. Steele. The Cellular Environment of Lightweight Hand Held Portables. *IEEE Communications Magazine*, pages 20–29, July 1988.
- [36] R. Steele. Deploying personal communication networks. *IEEE Communications Magazine*, pages 12–15, September 1990.
- [37] D. J. Goodman. Cellular Packet Communications. *IEEE Transaction on Communications*, 38:1272–1280, August 1990.
- [38] R. Ramjee, L. Li, T. LaPorta, and S. Kasera. IP Paging Service for Mobile Hosts. *Wireless Networks*, 8:427–441, 2002.
- [39] E. Pitoura and G. Samaras. Locating Objects in Mobile Computing. *IEEE Transactions on Knowledge and Data Engineering*, 13(4):571–592, July/August 2001.

- [40] Crunching Cell Phone Numbers.  
Web page at <http://www.electronichouse.com/default.asp?nodeid=2000>.
- [41] Cell Phone Facts and Statistics.  
Web page at <http://www.nwfusion.com/research/2001/0702featside.html>.
- [42] FCC sets tech standards for cellular 911 calls.  
Web page at <http://www.cnn.com/TECH/computing/9909/20/fcc.911.idg/>.
- [43] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and Querying Moving Objects. In *Proc. of Intl. Conf. Data Engineering*, pages 422–432, 1997.
- [44] O. Wolfson, A. P. Sistla, S. Chamberlain, and Y. Yesha. Updating and Querying Databases that Track Mobile Units. *Distributed and Parallel Databases*, 7(3):257–387, 1999.

## Biography of Authors

**Ying Cai** received his Ph.D. in computer science from the University of Central Florida in 2002. While studying at this university, Dr. Cai was the chief architect at nStor/StorLogic leading the effort to develop network storage technology. He developed the first remote and centralized RAID management system back in 1996 and the product was licensed by several major companies including SGI, Adaptec, and NEC. Currently, Dr. Cai is an assistant professor in the Department of Computer Science at the Iowa State University. His research interests include wireless networks, mobile computing, and multimedia systems.

**Kien A. Hua** received the B.S. degree in Computer Science, M.S. and Ph.D. degrees in Electrical Engineering, all from the University of Illinois at Urbana-Champaign, in 1982, 1984, and 1987, respectively. Form 1987 to 1990 he was with IBM Corporation. He joined the University of Central Florida in 1990, and is currently a professor in the School of Computer Science and the Interim Associate Dean for Research of the College of Engineering and Computer Science. Dr. Hua has published widely including three articles recognized as best papers and another three as top papers at various international conferences. He has served as General Chair, Vice-Chair, Associate Chair, Demo Chair, and Program Committee Member for numerous ACM and IEEE conferences. Currently, he is on the editorial boards of the IEEE Transactions on Knowledge and Data Engineering and Journal of Multimedia Tools and Applications.

**Guohong Cao** received his BS degree from Xian Jiaotong University, Xian, China. He received the MS degree and PhD degree in computer science from the Ohio State University in 1997 and 1999 respectively. Since then, he has been with the Department of Computer Science and Engineering at the Pennsylvania State University, where he is currently an Associate Professor. His research interests are mobile computing, wireless networks, and distributed fault-tolerant computing. His recent work has focused on data dissemination, cache management, network security and resource management in wireless networks. He is an editor of the IEEE Transactions on Mobile Computing and IEEE Transactions on Wireless Communications, has served as a co-chair of the workshop on Mobile Distributed Systems, and has served on the program committee of numerous conferences. He was a recipient of the Presidential Fellowship at the Ohio State University in 1999, and a recipient of the NSF CAREER award in 2001.

**Toby Xu** received his B.E. degree from the University of Science and Technology of China in 2003. He is currently a Ph.D. student in the Department of Computer Science at the Iowa State University. His research interests include wireless ad hoc networks and mobile computing.