An Adaptive Query Management Technique for Real-Time Monitoring of Spatial Regions in Mobile Database Systems*

Ying Cai Kien A. Hua

School of Electrical Engineering and Computer Science
University of Central Florida
Orlando, FL 32816-2362, U.S.A.
E-mail: {cai, kienhua}@cs.ucf.edu

Abstract

This paper presents a technique for real-time monitoring of mobile objects in user-defined spatial regions. We refer to such regions as range-monitoring queries. *Unlike conventional queries, which are based on an instant of the database at some moment in time, range-monitoring queries are continuous for monitoring purposes. Such queries can last for an extended period of time; and many can be active simultaneously. We present an efficient technique for managing such continuous queries. This capability is not available in conventional database management systems which are designed to manage data, not queries. In our environment, each mobile object is associated with a resident domain and is made aware of the monitoring areas inside it. When an object moves, it monitors its spatial relationship with its known monitoring areas. If it crosses any quary boundaries, it reports server and the server updates the affected query results accordingly. When an objet moves out of its resident domain, the server will determine a new one for the object. This process is supported efficiently with our new index structure called Domain tree. Our simulation results indicate that our technique is many times better than a recent method in terms of communication cost and server processing cost.*

1 Introduction

Many believe that the wireless revolution will bring even a more significant impact on society than today's commercial Internet. This emergent technology and the rapid advances in positioning systems have spurred a great research interest on database systems for location-based services. A major challenge of designing such a database system is to support range queries over the continuously moving objects. That is, given a rectangular area as a query, we want to retrieve objects that are currently inside the query window. A straightforward solution is to index the locations of mobile objects using some spatial access method [1], and update the index tree each time a mobile object moves. This simple strategy, however, requires excessive location updates and can quickly exhaust the battery power of the mobile units. In addition, when the number of mobile objects is large, the server also likely becomes a bottleneck since it has to process every move of the mobile objects. We note that each update consists of two expenses - the communication cost associated with reporting the new location to the server, and the cost of updating the index at the server side to reflect the new location.

To reduce the update costs, we can model the velocity of the object as a linear function of time $f(t)$, and use it to estimate the position of the object at different times. This scheme avoids excessive location updates because no explicit update is required unless the parameters of $f(t)$ change [2]. To support range queries efficiently, many techniques can be used to index the trajectory lines of each object. For examples, we can transform line trajectories into points in a higher-dimensional space, and then index these points using regular spatial indices [3]. The trajectories can also be indexed through their bounding rectangles that are time-parameterized. This indexing method was called TPR-tree in [4]. Another indexing scheme using external range trees [5] was presented in [6]. The techniques in [7] and [8] allow range queries performed over historical data. Other recent works on supporting spatio-temporal data can be found in [9, 10, 11].

It seems that with all these techniques, the prob-

*This research is partially supported by the National Science Foundation grant ANI-0088026.

lem of processing range queries over moving objects has been solved. However, in addition to retrieving objects currently inside a query window, many applications also need to monitor the window population over a time period. For instances, we might want to notify tourists of the updated information about their nearby hotels and restaurants as they traverse different regions. It is also desirable if we can provide an alert in case anyone enters into some dangerous zone. Similarly, we might want to continuously monitor the traffic condition and provide alert in case the number of vehicles within some area exceeds a certain threshold. In these applications, the system must be able to provide the accurate query results and update them in real time whenever some mobile object enters or exits the region defined by the query. We refer to this class of queries as *range-monitoring queries* in this paper. Unlike the conventional range query, a range-monitoring query is removed from the system only when the user explicitly ends the query. Clearly, the aforementioned techniques for indexing mobile objects are not suited to processing such queries:

- First, these schemes compute query results that are valid only for a certain point in time and quickly become obsolete as the objects continue to move. To keep the query result current over a period of time, the same query would have to be issued repeatedly at a very high rate. This brute-force approach would exhaust the server resources, and still cannot provide information in real time.
- Second, the location updates of these techniques are based on velocity models and therefore, can provide only approximated query results. In other words, a given query result does not guarantee that all objects within the query range are reported; nor all the objects reported are indeed in the query region.

In this paper, we focus on efficient techniques for real-time processing of range-monitoring queries. In particular, we consider a location-based service capable of supporting a large number of mobile objects and monitoring queries. We want to be able to monitor each query region, and continuously update the query results in real time. To achieve this goal, we propose an efficient SQM (*Spatial Query Management*) technique. It gets its name from the fact 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 the real-time query management capability in order to support range-monitoring queries. Under SQM, the database map is partitioned into many disjoint subdomains and each mobile object is assigned a subdomain as its *resident domain*. As an object moves, it monitors its position against the monitoring areas inside its resident domain. If it crosses any query boundaries, it reports server to update the affected query results. When an object moves out of its current resident domain, the server will determine a new resident domain for this object. This operation is supported efficiently by organizing the domain decomposition hierarchy using D-Tree (Domain Tree). Our simulation study shows that this technique can be used to provide accurate query results and real-time monitoring updates in a large scale mobile database system.

The remainder of this paper proceeds as follows. We discuss more related techniques in Section 2. Our technique, SQM, for real-time monitoring queries is presented in Section 3. In Section 4, we introduce the D-tree indexing technique. The performance results are examined in Section 5. Finally we give our concluding remarks in Section 6.

2 Related Works

The work most related to our research is the Q-index technique presented in [12]. This method indexes the rectangular-shaped queries, at the server side, using an R-tree [13] or some other spatial indexing technique. When an object moves, its new location is used to search this access structure to find the affected queries, and revise their results accordingly. To avoid excessive location updates, the authors proposed to assign a *safe region* for each mobile object. A safe region is either a circular or rectangular region that does not overlap with any query boundaries. Figure 1 shows mobile object A with its rectangular and circular safe regions. The latter is centered at the current location of object A. This approach allows an object not to report server its location as long as it moves inside its safe region. Unfortunately, determining a safe region requires intensive computation. For example, computing a rectangular safe region takes from $O(n)$ to $O(nlog^3n)$, where n is the number of queries [12]. We note that a new safe region has to be computed when a mobile object moves out of its current safe region. This problem is aggravated considering that adding a new monitoring query could affect all existing safe regions and as a result, the safe regions for all objects have to be re-computed. Therefore, it is unlikely that this approach can be used in a large

scale real-time mobile system.

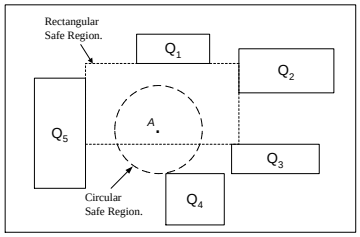


Figure 1: Examples of Safe Regions

3 Proposed Technique: SQM

In this section, we present the *Spatial Query Management* (SQM) technique for scalable processing of range-monitoring queries. In SQM, the entire domain space is partitioned into a set of disjoint subdomains. Figure 2 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. Since a query makes one monitoring region for each subdomain it overlaps, a spanning query has more than one monitoring region. For example, Q_1 makes 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 the monitoring region R_{32}. At any one time, a mobile object is in some subdomain known as its *resident domain*. When an object exits its resident domain, the object reports its new location to the server. In response, the server informs the mobile object of its new resident domain and the monitoring regions inside it. As an object moves about its resident domain, it monitors its spatial relationships with its cached monitoring regions and updates server its location when the object enters or exits these regions. The server will then updates the affected query results accordingly.

3.1 Server Design

At the server side, the subdomains and the monitoring regions are maintained using an index structure called D-tree (*Domain tree*), which will be presented later. We designed this spatial index structure to support continuous range queries. In addition, a binary relation, called *relevance table*, is used to track monitoring regions and their relevant queries. Each tuple of the table stores one monitoring region and one relevant query. Many access structures can be used to retrieve the relevant queries efficiently given a monitoring region. For instances, we can hash or build a B^+-tree index on the monitoring-region field. Alternatively, we can also store them in an adjacency matrix instead of a relational table. Thus, we will refer to this structure as a table and will not concern ourselves with the implementation details.

When a new range query q is submitted, the server searches the D-tree for the subdomains it overlaps. For each such subdomain, it determines their overlapping area, i.e., the monitoring region of this query in this subdomain. The server then inserts a new tuple, (r, q), to the relevance table, where r is the monitoring region. If this is a new distinct monitoring region, it is also inserted to the D-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. Since a typical mobile device has very limited computing resources, the number of the monitoring regions in a subdomain should be kept small. When this size exceeds a predetermined *split threshold*, the corresponding 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 D-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 size 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 list of monitoring regions in the merged domain.

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 accordingly as follows:

- When an object oid enters or exits a monitoring region r, the mobile device sends a message $UpdateQueryResult(r, oid, p)$ 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 re-

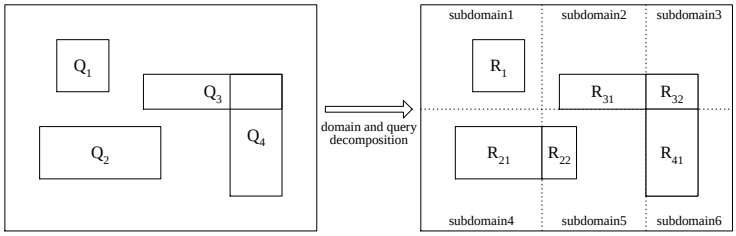


Figure 2: An Example of Domain Decomposition

gion. If a relevant query contains position p, then the object should be in its query result. Otherwise, delete *oid* from its query results.

- When a mobile object *oid* initializes itself or exits its current resident domain, the mobile unit sends the message $RequestResidentDomain(oid, p)$, where p is the current position of the mobile object. In response, the server searches the D-tree to look up the subdomain that contains location p. The server then broadcasts the message $SetResidentDomain(oid, d, l)$, where d and l denote the new resident domain of object *oid* and its new list of monitoring regions, respectively.

3.2 Mobile Unit Design

The following notations are used in the discussion of the mobile object:

myID : the unique identifier of this object.

myPos : the current position of this object.

myDomain : the current resident domain of this object.

myMRs : the list of monitoring regions inside $myDomain$.

The design of a mobile device consists of three main components: *Initialization*, *MessageListener*, and *RegionMonitor*. They are described as follows.

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 $RequestResidentDomain(myID, myPos)$ message to the server.
4. Spawn thread $RegionMonitor$.

MessageListener: The mobile unit listens to these messages and processes accordingly:

- $SetResidentDomain(oid, d, l)$: 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), check each monitoring region r in $myMRs$ and send $UpdateQueryResult(r, myID, myPos)$ message to server if $myPos$ is inside r.
- $AddMonitoringRegion(r)$: If monitoring region r is inside $myDomain$, then do the following:
  - Add r to $myMRs$.
  - Send $UpdateQueryResult(r, myID, myPos)$ message to server if r contains $myPos$.
- $DeleteMonitoringRegion(r)$: Delete monitoring region r from $myMRs$ if r is inside $myDomain$.
- $SplitDomain(d, d_1, d_2)$: If $myDomain == d$, then do the following steps:
  - If subdomain d_1 contains $myPos$, set $myDomain = d_1$; otherwise, set $myDomain = d_2$.
  - For each monitoring region r in $myMRs$, 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 merge of d_1 and d_2.
  - Set $myMRs = l$.

RegionMonitor : The mobile unit constantly performs the following steps when it moves:

- If the object enters or exits any monitoring region r in $myMRs$, then send message $UpdateQueryResult$ $(r, myID, myPos)$ to update the server.
- If the object exits $myDomain$, then send $RequestResidentDomain(myID, myPos)$ message to the server.

4 D-Tree: Domain Tree

A D-tree consists of two types of node: *domain node* and *data node*. All internal nodes are domain nodes while all external nodes are data nodes. The data structure for a domain node is an array of entries, each has the form (R, P). R holds the upper-left and lower-right coordinates of a rectangular subdomain; and P links to either another domain node or a data node. Each domain node represents a decomposition of a parent subdomain. As illustrated in Figure 3, the decomposition of the parent subdomain d_1 consists of the subdomains d_{11}, d_{121}, and d_{122}. A data node stores the monitoring regions that are inside its parent subdomain. A data node also contains an array of entry $(R, null)$, where R holds a monitoring region. As an example, the data node pointed at by the subdomain d_{11}, as shown in Figure 3, is used to record all monitoring regions within d_{11}. Thus, the domain decomposition hierarchy is captured in the internal nodes while each external node represents the monitoring regions inside a subdomain.

The size of the domain nodes can be determined based on the paging system of the host computer to optimize its performance. However, the size of the data nodes should depend on the computing capability of the mobile devices, i.e., a mobile device can handle only limited number of monitoring regions at one time. We can set this parameter by specifying the desired split threshold, i.e., the maximum number of monitoring regions allowed in any subdomain.

In the following subsections, we present the D-tree operations in detail and discuss how they are used with the relevance table to support range-monitoring query. The following notations are used in the discussion:

- Given an entry (R, P) in a D-tree node, $R.child\_node$ denotes the child node pointed at by P.
- Given a D-tree node D, $D.parent$ refers to the parent node who has an entry pointing to D.
- Given a D-tree node D, $D.domain$ is the decomposed domain represented by the node.
- Given two rectangles, R_1 and R_2, $R_1 \cap R_2$ represents their overlapping area.

4.1 Search

Upon receiving $RequestResidentDomain(oid, p)$ message, the server needs to search the D-tree for the resident domain and its contained monitoring regions for the mobile object oid. This is done by calling $Search(root, p)$, where $root$ is the root of the D-tree and p is the current position of the mobile object. The search algorithm is given below:

Search(D_tree_node, p): Search for the subdomain that contains position p, and the monitoring regions inside the subdomain

1. Search for the entry, say (R, P), in D_tree_node such that rectangle R contains position p.
2. If $R.child\_node$ is a data node, then return R and the monitoring regions stored in $R.child\_node$.
3. Otherwise, call $Search(R.child\_node, p)$.

4.2 Insert

A D-tree is initialized as a root 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. When a new query arrives, the server descends the D-tree to look for the data nodes whose subdomains overlap with the query area. For each monitoring region, say r, created by the query, a new tuple (r, q) is added to the relevance table. The monitoring region is also inserted to the D-tree if it is a new distinct one.

An insert might cause a data node to overflow and its domain is then split. The monitoring regions spanning over the new subdomains are decomposed so that each new monitoring region is contained entirely by only one subdomain. When this happens, the relevance table is updated accordingly. We note that a split of a D-tree node may cause its parent node to overflow and split. This effect can percolates up and causes the root node to split. Each time a data node is split, the server broadcasts the message $SplitDomain(d, d_1, d_2)$ to notify mobile units that some domain d has been decomposed into d_1 and d_2. We have discussed how a mobile unit reacts to such a message.

A number of decomposition schemes can be used to split a domain. A simple approach is *center split*,

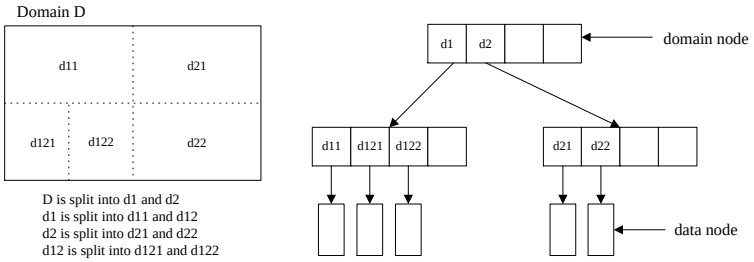


Figure 3: A D-tree Example

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 instance, we can split on the longer dimension to avoid having long and narrow subdomains.

When a new query q arrives, we call $Insert(root, q)$, where *root* is the root of the D-tree. We present the algorithm for this operation as follows.

Insert(D, q): Insert a query q into a D-tree rooted at node D

1. If D is a data node, then do the following:
   - Insert a new tuple, $(q \cap D.domain, q)$, to the relevance table.
   - If no monitoring region in D is equal to $q \cap D.domain$, then do the following:
     - Allocate a new entry in D and set it equal to $(q \cap D.domain, null)$.
     - If D is full, call $SplitNode(D)$.
2. Otherwise, for each entry (R, P) in D, if R overlaps with q, then call $Insert(R.child\_node, q)$.

SplitNode(D): Split D-tree node D

1. If D is the root, create a new root and make D its only child.
2. Look for the entry in $D.parent$, say (R, P), such that P points to D.
3. Split domain R into two subdomains, R_l and R_r, then do the following:
   - Broadcast message $SplitDomain(R, R_l, R_r)$ if D is a data node.
   - Create two new D-tree nodes, $left$ and $right$.
   - Replace the entry (R, P) in $D.parent$ by (R_l, P_l), and direct P_l to point to $left$.
   - Allocate a new entry (R_r, P_r) in $D.parent$, and direct P_r to point to $right$.
4. For each valid entry (R_i, P_i) in D, do the following:
   - If R_i spatially overlaps with R_l and rectangle $R_i \cap R_l$ does not exist in $left$, then allocate a new entry in $left$ and set it equal to $(R_i \cap R_l, P_i)$.
   - If R_i spatially overlaps with R_r and rectangle $R_i \cap R_r$ does not exist in $right$, then allocate a new entry in $right$ and set it equal to $(R_i \cap R_r, P_i)$.
   - If D is a data node and R_i spans over both R_l and R_r, then for each tuple (R, Q) in the relevance table, do the following if $R == R_i$:
     - Replace tuple (R_i, Q) with $(R_i \cap R_l, Q)$.
     - Add a new tuple, $(R_i \cap R_r, Q)$, to the table.
5. If $D.parent$ is full, call $SplitNode(D.parent)$.
6. Discard D.

4.3 Delete

The *Delete* operation is used when a range-monitoring query is terminated. A delete may cause some data nodes to underflow; and the corresponding subdomains and their split counterparts need to be merged. This is a reverse function of splitting a node in the insert operation. When two data nodes are merged, the mobile objects are notified by a broadcast message $MergeDomain(d_1, d_2, l)$, where d_1 and d_2 are the subdomains to be merged, and l is the new list of monitoring queries inside the merged domain. Deleting a q is done by calling $Delete(root, q)$, where *root* is the root of the D-tree. Due to the space limitation, we omit the detail algorithms for this operation.

5 Performance Study

To evaluate the performance of the proposed SQM approach, we implemented simulators to compare it with the Q-index technique. 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. 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 quries 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).

Mobile Communication Cost: This cost is measured as the total number of messages sent by the mobile objects to the server.

5.1 Simulation Model

We implemented the D-tree using a center-split strategy as discussed in Section 4 - splitting at the middle of the longer dimension. The maximum number of entries in both data node and domain node is set to be 50. D-tree is used in both SQM and Q-index to index the monitoring regions. This allows us to compare the server computation costs of the two techniques fairly. Under Q-index scheme, we compute the largest circular region within the resident domain of each mobile object as its *safe region*, such that the safe region does not overlap with the boundaries of any query. We choose not to use rectangular safe region because its algorithm is much more complicated while the resulted performance is similar to that of using circular safe region, as indicated in [12]. We note that we actually compare SQM with an improved version of Q-index in this study. Although the safe region determined by this approach is suboptimal, the D-tree approach limits the consideration to only the monitoring regions inside a subdomain. It would have required the original Q-index technique 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 [12] 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.

Other simulation parameters are as follows. We generate 1,000 mobile objects and place them in a uniform distribution over a rectangular domain of [0...10K, 0...10K]. 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 constant through out each simulation run. Their initial moving directions are set randomly. Each object moves linearly until it reaches the boundary of a subdomain, it then changes its direction and continues to move at the same speed. This process is repeated until the simulation is ended.

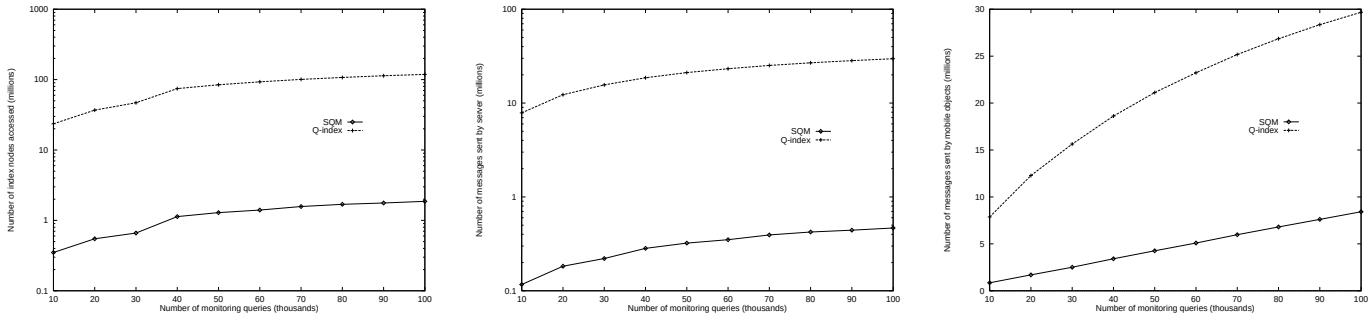
For each simulation run, we generate a certain number of square range-monitoring queries, from 10,000 to 100,000. The sizes of these monitoring squares range from 1×1 to 100×100 and they are placed in the domain space following a uniform distribution. Each simulation run consists of two phases. During the first phase, a new query is inserted every time unit until we have inserted the desired number of queries. During the second phase, the objects continue to move around, but we do not add any more query. This phase lasts 10,000 time units. The reason for separating these two phases is due to the fact that Q-index performs poorly whenever the system experiences new queries because it has to re-compute the safe regions for all mobile objects. It is not interesting to compare SQM with Q-index under such circumstances.

5.2 Simulation Results

The performance data collected in the second phase are plotted in Figure 4(a), (b), and (c). All three figures indicate that SQM outperforms Q-index by a very wide margin. We explain them briefly as follows. First, when a mobile object exits its safe region, Q-index needs to search the D-tree to determine if it enters or exits any monitoring region for potential update of query results. In contrast, a mobile object under SQM provides the affected monitoring region directly, allowing the server to update query results without looking up the D-tree. Second, since a safe region cannot overlap with any query boundaries, it is generally many times smaller than the containing subdomain. Therefore, a mobile object under Q-index needs to communicate with the server much more frequently for its new safe region. This presents a signifant more workload to the server. We observe that both the communication costs and server processing cost increase with the increases in the number of queries. This is due to the fact that increasing the number of queries reduces the average size of the safe regions, and therefore the chance of requesting a new safe region becomes higher.

6 Concluding Remarks

Technologies such as Global Positioning Systems, as well as technologies embedded in the wireless connectivity infrastructures, will enable the positioning of



(a) Server processing cost (b) Server communication cost (c) Mobile communication cost

Figure 4: Effect of monitoring query number

a substantial portion of the appliances and their users. In this paper, we address the challenges of providing region-based monitoring services that require continuous real-time updates of the query results. Our technique, called SQM (*Spatial Query Management*), has the following advantages:

Power Conservation : Since a mobile device does not need to constantly report its position, the power conservation is excellent.

Scalability : Since the server does not need to monitor the mobile objects and track the query results, SQM is highly scalable to support a very large user community.

Reliability : SQM is more reliable than techniques based on position estimation because it is not affected by estimation errors.

In addition to the above benefits, SQM is simple to implement. To assess the performance of SQM, we implemented simulators to compare it with an improved version of Q-index. Our simulation results, under various workloads, indicate that SQM is substantially more scalable and incurs significantly less communication cost and server processing cost.

References

[1] V. Gaede and O. Gunther. Multidimensional access methods. *Computing Surveys*, 30:170–231, 1998.

[2] 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.

[3] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. In *Proc. of ACM PODS'99*, pages 261–272, 1999.

[4] 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.

[5] 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.

[6] P. K. Argarwal, L. Arge, and J. Erickson. Indexing moving points. In *Proc. of ACM PODS'00*, pages 175–186, 2000.

[7] D. Pfoser, Y. Theodoidis, and C. S. Jensen. Indexing trajectories of moving point objects. In *Chorochronos Technical Report, CH-99-3*, 1999.

[8] D. Pfoser, C. S. Jensen, and Y. Theodoidis. Novel approaches in query processing for moving objects. In *Proc. of VLDB'00*, 2000.

[9] J. Tayeb, O. Ulusoy, and O. Wolfson. A quadtree-based dynamic attribute indexing method. *The Computer Journal*, 41(3):185–200, 1998.

[10] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE TKDE*, 10(1):1–20, 1998.

[11] 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.

[12] 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.

[13] A. Guttman. R-tree: A dynamic index structure for spatial searching. In *Proc. of ACM SIGMOD'84*, pages 47–57, 1984.