

# Partition-Based Lazy Updates for Continuous Queries over Moving Objects

Yu-Ling Hsueh<sup>†</sup> Roger Zimmermann<sup>‡</sup> Haojun Wang<sup>†</sup> Wei-Shinn Ku<sup>§</sup>

<sup>†</sup>Dept. of Computer Science, University of Southern California, Los Angeles, CA 90089

<sup>‡</sup>Computer Science Department, National University of Singapore, Singapore 117543

<sup>§</sup>Dept. of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849

{hsueh@usc.edu, rogerz@comp.nus.edu.sg, haojunwa@usc.edu, weishinn@eng.auburn.edu}

## Abstract

*Continuous spatial queries posted within an environment of moving objects produce as their results a time-varying set of objects. In the most ambitious case both queries and data objects are dynamic, making it very challenging to find an efficient query evaluation strategy. The significant overhead related to frequent location updates from moving objects often results in poor performance. The most advanced existing techniques use the concept of simple geometric safe regions to delay or avoid location updates. We introduce a Partition-based Lazy Update (PLU) algorithm that elevates this idea further by adopting Location Information Tables (LIT) which (a) allow each moving object to estimate possible query movements and issue a location update only when it may affect any query results and (b) enable smart server probing that results in fewer messages. Among the significant advantages, our technique performs well even in very highly dynamic environments (with up to 100% mobility) where many other techniques deteriorate. PLU can be efficiently implemented and we demonstrate its query performance improvement of up to 28% over the current state-of-the-art.*

## 1. Introduction

The impressive advancement of mobile communication technologies, such as IEEE 802.11 and cellular networks, together with ever more capable handheld devices has sparked intense interest in location-aware services. One of the most challenging core functions is the efficient evaluation of *continuous queries over moving objects*, given an environment with a large number of objects and concurrent queries.

With the mobility introduced by portable and handheld devices, the performance bottleneck for continuous spatial query processing is often concentrated in the handling of the frequent location updates at the server and the utilization of the communication chan-

nel between the moving client objects (also called *mobiles*) and the server. Wireless bandwidth is generally still much more scarce than wired bandwidth and – adding to the challenge – the movement dynamics of such an environment require frequent mobile-server message exchanges that contain location information for the database engine to maintain an up-to-date view of the world.

Therefore, one of the key challenges is minimizing the number of wireless location update messages. A number of previously proposed techniques have provided significant insight into this issue. In the simplest case, whenever an object moves it sends its new location to the server. Obviously this can be very wasteful, for example if the moving object is located in an area where it does not affect any query results. Making informed decisions when to communicate update messages becomes a key design issue to improve scalability. The message count can be reduced through the following optimizations. The mobile client may be equipped with computation capabilities to maintain a *safe region* [8] with the purpose that movements within the safe region will not affect any query results (hence no location updates must be sent to the server). Safe regions are bounded by the nearest query rectangles around a mobile client and must be recomputed when certain events take place such as a new query is inserted or a moving object moves beyond its safe region boundary. In some cases (e.g., query insertion) a moving object is initially unaware of the event and the server must *probe* its current location. Because of the usually simple shape of safe regions (e.g., rectangles or spheres) they can only help to avoid a fraction of unnecessary location updates.

In this paper, we propose a novel partition-based lazy update approach that significantly reduces unnecessary location updates by maintaining a *Location Information Table* (LIT) on each moving object. A LIT represents the detailed query boundaries and distances across the terrain locally around the object. We build a mobile-side LIT from  $m \times m$  tiles where each tile corresponds to a LIT *cell* that stores the distance to the closest query boundary. The representation is easy to maintain and can be built on top of existing indexing methods for tracking moving objects, such as grids or R-trees. The server maintains its own vector of LITs where each vector element is of size  $n \times n$  (with  $n \geq m$ ) and represents the complete service space at the time instance of a specific event. Significantly, a LIT encapsulates surrounding query information in much more detail, yet compactly, as compared with geometric safe regions. This concept (plus other enhancements) allows our Partition-based Lazy Update (PLU) technique to reduce unnecessary update messages and perform significantly better than

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

GIS'07, November 7-9, 2007, Seattle, Washington, USA.

Copyright 2007 ACM 978-1-59593-701-8/07/0009 ...\$5.00.

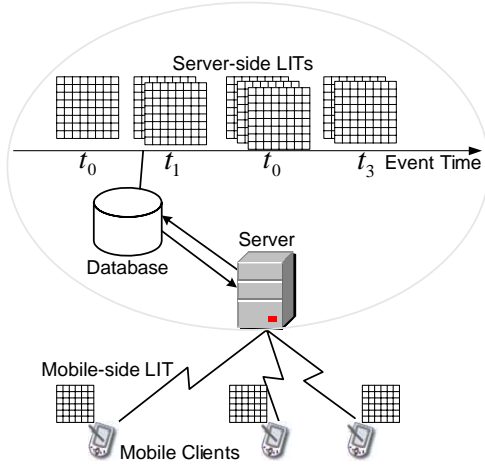


Figure 1. System framework

traditional techniques.

The remainder of this paper is organized as follows. Section 2 describes the related work. Section 3 presents and details our continuous query processing design illustrated in Figure 1. We extensively verify the performance of our technique in Section 4 and finally conclude with Section 5.

## 2. Related Work

A number of pioneering techniques have been designed for processing of continuous queries over moving objects. Prabhakar et al. [8] first proposed two elementary techniques called *Query Indexing* and *Velocity Constrained Indexing (VCI)* and also introduced the important concept of *safe regions*. Cai et al. [1] proposed the *Monitoring Query Management (MQM)* approach to leverage the computational capabilities of moving objects for efficient processing of continuous range queries. Subsequently, Hu et al. [3] proposed a generic framework to handle continuous queries by leveraging the concept of safe regions through which the location updates from mobile clients can be further reduced. However, these methods only address part of the mobility challenge since they are based on the assumption that queries are static. Nowadays, an extensive number of spatial applications require the capability to process moving objects in conjunction with dynamic continuous queries.

The two main challenges in supporting both a large number of objects and continuous queries in a highly dynamic environment are to (a) reduce the object tracking and query evaluation costs and (b) minimize the communication costs of location updates from objects and queries. To address these two issues, the design of an efficient index structure that can manage the locations of moving objects has been intensely studied. For example, predicting the movement of objects (i.e., their trajectory) has been used with R-tree-based structures (e.g., the *TPR-tree* and its variants [9, 10]) and B-tree-based structures (e.g., the *B<sup>x</sup>* tree [4]). However, these index structures require knowledge of object trajectories, which is not available in many environments. In order to handle arbitrary object movements, periodic position updates from moving objects are widely used. However, with such a paradigm tree-based indices suffer from excessive node reconstructions when tracking object locations. As an alternative, grid-based indices (e.g., LUGrid [12]) have attracted much attention because of their simplicity and scalability in handling position updates.

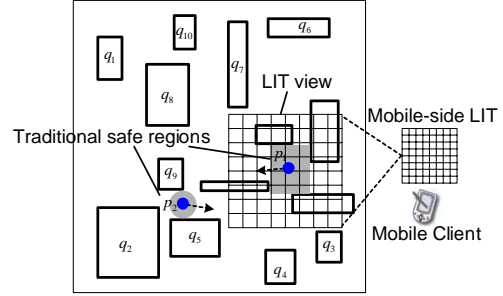


Figure 2. Illustration of concepts

Utilizing a grid index, some methods have been proposed to process dynamic continuous queries over moving objects. For instance, *MobiEyes* [2] introduced a distributed infrastructure to process dynamic range queries where the server is acting as a mediator to coordinate query processing on both the server and moving object sides. The main drawback of this method is that an object will repeatedly send location updates to the server when it is enclosed by a query, which consumes a large amount of bandwidth when the query density is high. SINA [6] has been introduced as centralized solution to process continuous range and  $k$  nearest neighbor ( $k$ NN) queries over moving objects. Yu et al. [13] proposed an algorithm that computes the query results by defining a search region based on the maximum distance between the query point and the current locations of previous  $k$ NNs. However, the algorithm results in high re-computation costs when the query point is highly dynamic. Similarly, Xiong et al. [11] suggested the SEA-CNN framework which uses the concept of shared execution. SEA-CNN continuously maintains the *search radius* of the query point to avoid rebuilding the query result once the query point changes its location. As an enhancement, Mouratidis et al. [7] presented a technique called CPM that defines a conceptual partitioning of the space by organizing grid cells into rectangles. Location updates are handled only when objects fall into the vicinity of queries, hence improving system throughput.

Our PLU approach is inspired by some of these prior techniques. Its main contribution lies in the development of “smarter” safe regions represented via location information tables that enable enhanced (i.e., more independent) mobile-side decision making for location updates. The technique does not deteriorate when faced with high mobility rates as demonstrated by our simulation results.

## 3. The System Overview

To describe what motivates our work, let us first illustrate how the best current techniques operate with Figure 2 serving as an example. The gray areas represent the safe regions of two moving objects  $p_1$  and  $p_2$ . A traditional safe region is either a rectangle or a sphere which is determined by the set of surrounding queries [8]. When an object moves outside of its safe region, it incurs a location update. From the example we can observe that, as  $p_1$  moves out of its safe region (in the direction of the arrow), it issues an unnecessary update because of the limited safe region information. Furthermore, the safe region of a moving object is determined based on its current location. When a query moves to a new location or a new query is inserted, the server triggers a location probe to the affected moving objects and re-calculates new safe regions for them. When receiving the location probes (downstream) from the

server, the moving objects need to send their locations (upstream) back to the server. Once the server completes the safe region computations, it sends the safe regions (downstream) to those moving objects. Hence a total of three network messages are sent back and forth between the server and each mobile client. As illustrated, the safe region approach incurs significant network traffic in this scenario.

In contrast, we propose a partition-based technique by defining a grid-like LIT (also shown in Figure 2) which provides a moving object with a detailed view of the surrounding query locations across the terrain. As an additional advantage, a LIT is determined without referring to the locations of moving objects. If a query is inserted, the server can send the new LIT with the added query information (downstream) to the affected moving objects directly, and only a fraction of the mobile clients that receive the updated LIT must issue location updates (upstream) back to the server (–namely if they are part of the new query result). Therefore, the number of network messages is reduced to at most two. The overall PLU process is discussed in detail in the subsequent sections.

To enable a focused discussion we make some explicit assumptions. A centralized server is assumed in the environment to process continuous queries. The mobile units consist of a set of dynamic query objects  $Q$  and a set of moving objects  $P$ . Both queries and moving objects are identified by a unique identifier to distinguish their types. Each mobile unit can move arbitrarily without exceeding a given maximum speed  $\lambda$ . The communication between the server and moving objects is through wireless messages. We use a main-memory grid  $G$  as the underlying structure to index moving objects because of its simplicity and ease-of-maintainance in a highly dynamic environment. Because of space constraints we focus on range queries in this paper. For high performance an event-driven approach is adopted to evaluate continuous queries. To maintain the correctness of the query results, the server monitors registered query objects and requires that each query object update the server when changing its location. Thus, the server can evaluate the query based on its new location. We assume that each mobile unit has enough computational capabilities and memory to carry out the required tasks. We describe the details of the data structures used in this paper in Section 3.1. The LIT design and the procedure for generating a LIT are described in Section 3.2. Mobile-side and server-side processing are discussed in Sections 3.3 and 3.4, respectively. Finally, we analyze a spatial data compression scheme for LITs to reduce the number of messages sent from the server in Section 3.5.

### 3.1 Data Structures

We use the following data structures in the system, also shown in Figure 3.

- Object Grid ( $G$ ): A  $w \times w$  object grid  $G$  is used to index the moving objects. We adopt a grid structure that is similar to the one proposed in [7]. During an execution interval, each cell  $c_{(i,j)}$  maintains a moving object list denoted by  $\mathbb{M}_{(i,j)}$ . Every moving object is represented by a tuple  $\langle ID, LocXY, LIT_{id} \rangle$  where  $ID$  is the object identifier,  $LocXY$  is the latest reported location information and  $LIT_{id}$  is an identifier of the server-side LIT where the mobile-side LIT of the moving objects was retrieved.
- Query List ( $Q$ ): Queries are organized via an in-memory sequential list  $Q$ . Each query entry is of the form  $\langle ID, LocXY, ANS \rangle$ , where  $ANS$  is a set of the query results.

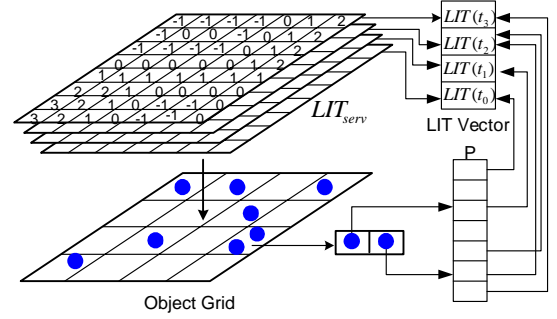


Figure 3. Data Structures

- $LIT_{serv}$ : A server-side location information table consists of  $n \times n$  cells of uniform side length  $\delta$ . A  $LIT_{serv}$  has an identifier  $LIT_{serv}.id$  and each  $LIT_{serv}$  cell maintains a pair of values  $\langle LIT_{serv}.value, TBound \rangle$  where  $LIT_{serv}.value$  stores an integer value to indicate the query boundaries and  $TBound$  is the actual coordinates of the boundary that the  $LIT_{serv}$  cell corresponds to on the terrain. Locating a  $LIT_{serv}$  cell for a moving object  $p$  with coordinates  $(x,y)$  can be done by calculating the index  $LIT_{serv}i, j$  where  $i = \lfloor \frac{x}{\delta} \rfloor$  and  $j = \lfloor \frac{y}{\delta} \rfloor$ .
- LIT Vector ( $\mathbb{LIT}$ ): A LIT vector stores a list of LITs generated for every specific event time.  $\mathbb{LIT}(i)$  returns a LIT created at time instance  $t_i$ . When there is no moving object engaging a LIT, it is removed from the LIT Vector.
- $LIT_{mob}$ : A mobile-side LIT is a subset  $m \times m$  table extracted from the latest server-side LIT with  $m \leq n$ . When a mobile object  $p$  obtains its mobile-side LIT which is extracted from  $LIT_{serv}$ , we term that  $p$  references to  $LIT_{serv}$ . Let  $(i, j)$  be the residing cell index of  $p$  on the server-side LIT. The mobile-side LIT of  $p$  is extracted from the area of  $[i - \ell \dots i + \ell, j - \ell \dots j + \ell]$  on the server-side LIT where  $\ell$  is defined as a level of the mobile-side LIT. Therefore, the size of a mobile-side LIT  $m$  is  $2\ell + 1$ .

The five data structures presented above support the operation of our PLU technique as follows. Initially, startup mobile-side LITs are generated by the server and sent to each registered moving object. With the possession of an LIT, each moving object can locally determine a location update and send it only when the new location may affect a query result. When an object issues a location update, it receives the latest mobile LIT in response with newly reported positions of query boundaries. The previous server-side LITs are referenced later during the processing of query insertions and movements. To achieve memory efficiency the system removes the old server-side LITs when there are no more objects referencing it. A flushing mechanism is also used to force objects which have not reported their location for a certain period of time to issue an update, hence allowing the old server-side LITs to be removed from memory. An  $LIT_{mob}$  assigned to a moving object is in general a subset table of the server-side LIT due to memory limitations of moving objects and to reduce communication costs. Table 1 summarizes the symbols and functions we use throughout the following sections.

### 3.2 LIT Details

Symbol	Description
$G$	A $w \times w$ object grid where objects are hashed based on their locations to the grid cells.
$Q$	A set of query objects
$P$	A set of moving objects
$LIT_{serv}$	A $n \times n$ server-side LIT
$LIT_{mob}$	A $m \times m$ client-side LIT
$LIT$	A server-side LIT vector
$\lambda$	Maximum speed for any object
$\chi$	Unit maximum moving distance

Table 1. Symbols used in this paper

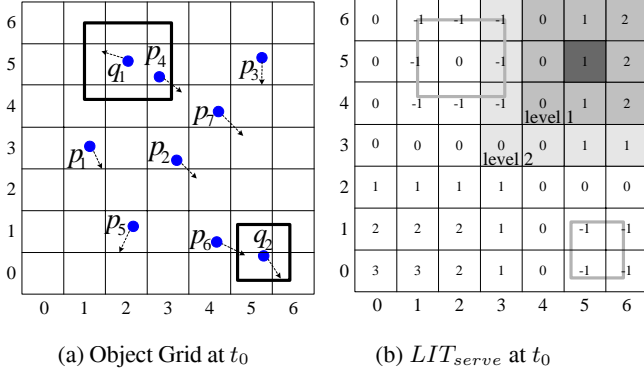


Figure 4. The object grid and a server-side LIT example

We first describe the server-side LIT details. A  $LIT_{serv}$  is generated when one of the following two events happen: (1) an existing query changes its location or (2) a new query is registered with the system. The general attributes described in this section for the server-side LIT are also applicable to the mobile-side LITs extracted from it. A mobile-side LIT simply inherits all the attributes and query boundary information from the server-side LIT. However, each moving object maintains (i.e., updates) the mobile-side LIT locally after receiving it from the server based on a specific event.

A  $LIT.value$  for  $LIT_{serv}(i, j)$  stores an integer number that represents a *safe distance*. The safe distance for  $LIT_{serv}(i, j)$  is defined as the minimal linear distance in cells from the  $LIT_{serv}(i, j)$  cell to the nearest query boundary. We distinguish two cases when assigning a value to  $LIT_{serv}(i, j)$ :  $LIT.value \geq 0$ , if  $LIT_{serv}(i, j)$  does not overlap a query boundary; and  $LIT.value = -1$ , if  $LIT_{serv}(i, j)$  is covered by a query boundary. Figure 4(a) shows an object grid with a set of registered queries and moving objects on the terrain at time  $t_0$ . The corresponding server-side LIT created at  $t_0$  is illustrated in Figure 4(b). In this example we assume that the server-side LIT size is the same as the object grid. The LIT values of the cells that overlap the boundaries of query  $q_1$  and  $q_2$  are set to -1. We define two types of cell zones: a *border zone* (LIT value = -1) and a *zero zone* (LIT value = 0). A *border zone* consists of cells that overlap with the boundaries of some queries. A *zero zone* is essentially a prediction zone which might be covered by nearby moving queries as time proceeds. Since a zero zone has a safe distance equal to zero, it is more likely to be covered by a moving query, say  $q_1$ , soon. Both border and zero zones are important indicators for a moving object to decide on a location update. In order to predict the moving query locations, each moving object updates its local LIT and marks the new prediction cells as

zero zones. The detailed update mechanism for mobile-side LITs will be described later. Algorithm 1 presents the pseudo code of generating a server-side LIT. Lines 2-4 assign -1 to the cells which are covered by a query boundary first and lines 5-11 calculate the LIT value for the rest of the cells. To compute a LIT value for  $LIT_{serv}(i, j)$ , the algorithm checks its surrounding cells level by level by calling  $GetCellsAtLevel(LIT_{serv}(i, j), l)$ , where  $l$  is a level number. The procedure terminates the loop when a cell with a LIT value equal to -1 appears. Figure 4(b) shows such levels of  $LIT_{5,5}$  in different gray scales. Since -1s appear at level 2, the loop is terminated and we obtain 1 as the LIT value for  $LIT_{5,5}$ .

#### Algorithm 1 CreateLIT( $G, Q$ )

```

1: Let  $LIT$  be a  $n \times n$  table and initialize the value of each cell to  $\infty$ 
2: for (every  $q \in Q$ ) do
3:   Set the value equal to -1 for each LIT cell that are covered by the
   query boundary of  $q$ 
4: end for
5: for (every cell  $LIT_{serv}(i, j)$ , which has the LIT value  $\neq -1$ ) do
6:   Let  $LIT_{serv}(i, j).value = 0$  and  $l = 1$ 
7:   while ( $GetCellsAtLevel(LIT_{serv}(i, j), l)$  returns  $C$  AND  $C \neq \emptyset$ )
   do
8:     if (any cell in  $C$  has LIT value equal to -1) then break the loop
9:     else Increment  $LIT_{serv}(i, j).value$  and  $l$  by one
10:    end while
11: end for

```

### 3.3 Mobile-Side Processing

Each moving object independently performs the following two major tasks to achieve the desired location update traffic reduction: progressive revision of the mobile-side LIT and determination of when to send location updates. Each time a moving object transmits its location to the server, an up-to-date mobile-side LIT will be sent to the moving object. However, since we consider dynamic queries in this paper, the LITs are subject to change whenever the queries change their locations during the course of the execution. Instead of sending a new mobile-side LIT with the latest query locations to each moving object repeatedly, we propose a periodic LIT update method to independently adjust the mobile-side LIT to reflect all the possible query movements while ensuring the correctness of the query results. We first discuss how a moving object updates its local LIT and then describe the mechanisms for triggering a location update based on the mobile-side LIT.

#### 3.3.1 Mobile-side LIT Updates

Since the maximum speed  $\lambda$  for mobile units is limited, we can estimate the possible query locations in the mobile-side LIT. Continuing the example shown in Figure 4, each moving object  $p$  is given a mobile-side LIT by the server as shown in Figure 4(b). Without loss of generality we assume that the size for an object grid, server-side LIT and mobile-side LIT are the same in this example. Figure 5(a) illustrates the current locations of mobile units at time  $t_1$  and Figure 5(b) shows the mobile-side LIT updated by  $p$  at  $t_1$ . One can observe that in the worst case, by considering that a query may move to its surrounding cells in any direction, the area between two dashed rectangles shows all the possible coverage of the query boundary with such movements. Since a border zone may overlap more than one query boundary anywhere within the zone, the two solid rectangles represent the outermost query boundaries of the zone. For simplicity, we draw two dashed rectangles inwardly (shrunk) and outwardly (expanded) by extending

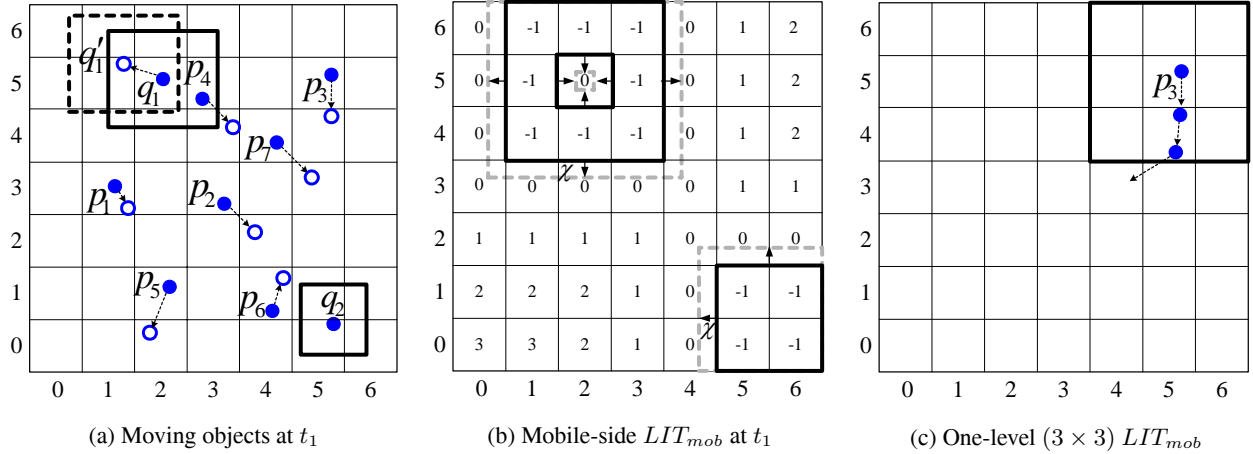


Figure 5. Object grid and mobile LIT

the solid rectangle by the length of the maximum moving distance  $\chi (= \lambda \times \Delta t)$  for every time instance. The cells that are newly covered by the area between the dashed rectangles become zero zones. As a final step, the LIT values of the remaining cells need to be updated. The procedure for computing the new LIT values for each cell can be found in the lines 2-11 of Algorithm 1.

### 3.3.2 Location Update Check

The event-driven procedure for deciding on a location update is performed by the moving object only when it moves to a new location. We continue with the example of Figure 5(a) that shows the new locations of queries and moving objects at time  $t_1$ . Referring to the mobile-side LIT in Figure 5(b),  $p_2$  in  $LIT_{mob}(3, 3)$  steps into a zero zone in  $LIT_{mob}(4, 2)$  which is a zero zone, so  $p_2$  might overlap with a query at this moment.  $p_4$  was in a border zone and it changed its location since the latest update to the server, so it may exit or enter a query boundary. Therefore, both  $p_2$  and  $p_4$  have to issue a location update at  $t_1$ . We describe the location update checking procedure in this section. Let  $LIT_{mob}(i, j)$  be the cell in which an object originally resided since the last update to the server, and  $LIT_{mob}(i', j') (\neq LIT_{mob}(i, j))$  be the cell where the object is currently located some time after it updated the server. To determine a location update request for  $p$ , we perform the following steps:

- Step 1:** Check if  $p$  was originally located in the border zone  $LIT_{mob}(i, j)$  since its last update to the server. If this is the case,  $p$  issues a location update and exits the checking steps. Otherwise, continue to Step 2.
- Step 2:** Check if  $p$  currently stepped into a border zone or zero zone  $LIT_{mob}(i', j')$ . If this is the case,  $p$  issues a location update and exits the checking steps. Otherwise, continue to Step 3.
- Step 3:** Check if  $p$  currently stepped into a cell  $LIT_{mob}(i', j')$  which is outside of the mobile-side LIT boundary;  $p$  issues a location update and terminates the checking steps.

In Step 1, we monitor the objects that are in the border zones. Since these moving objects are close to some query boundaries,

they are more likely to alter any query results. If a moving object moves into a border or zero zone as described in Step 2, it may enter a range query. Therefore, it has to report its current location to the server. In Step 3, when the moving object moves out of its local LIT boundary, it issues a location update. Figure 5(c) shows a one-level  $3 \times 3$  LIT for  $p_3$  and after some time instances at  $t_3$ ,  $p_3$  moves out of the LIT boundary to  $LIT_{mob}(4, 3)$ , and therefore it must issue a location update at  $t_3$ .

The above procedures is called push mode, since the location updates are issued from the moving objects. We also propose a pull mode that is executed by the server to request a location update from the mobile clients. To take advantages of the pull mode approach, we describe the following scenario first. In Figure 5(a),  $p_7$  was located in a zero zone  $LIT_{mob}(4, 4)$  at  $t_0$  and moves to  $LIT_{mob}(5, 3)$  at  $t_1$ . At this time  $p_7$  does not issue a location update since  $LIT_{mob}(5, 3)$  is neither a border nor a zero zone. However, the latest reported location of  $p_7$  is still in  $LIT_{mob}(4, 4)$  at the server and it might be evaluated as a query result because  $LIT_{mob}(4, 4)$  is a zero zone indicating a nearby moving query boundary might cover it. Therefore,  $p_7$  needs to inform the server about its current location for further validation. As a result, it incurs many unnecessary location updates from the moving objects as time proceeds (many zero zones appear due to the mobile-side LIT updates). We propose a server-side detection procedure *DecObj* to handle this case. Since the server has the latest locations of the queries, it is able to determine whether  $LIT_{mob}(4, 4)$  is actually covered by some query. If this is true, the server sends the latest LIT to  $p$  and requests for its current location. In this example, since the actual new position of  $q_1$  at  $t_1$  does not cover  $LIT_{mob}(4, 4)$ , the server does not need to send the latest LIT to  $p$ .

### 3.4 Server-Side Processing

We implement an event-driven approach to handle the requests from mobile units. There are two major server-side procedures: *DecObj*, which was discussed in the previous section for handling query updates and *QurIns* for processing query insertions. We focus on the *QurIns* procedure in this section.

When a new query  $q$  is inserted, instead of informing the entire registered moving objects population (that lack this new query boundary information) of the changes, the server performs the *QurIns*

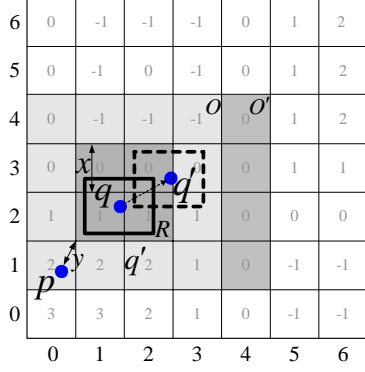


Figure 6.  $LIT_{serv}$  generated at  $t_0$

procedure to determine a set of moving objects  $P'$  that may enter the new query boundary. Then it sends the latest mobile-side LIT to these objects only. In order to determine  $P'$ , the procedure needs to utilize the old server-side LITs which are referenced by some of the moving objects. First, for every old server-side LIT,  $Qurlns$  checks each moving object  $p$  in the set of LIT cells  $O$ , where the objects have the mobile-side LIT overlapping with  $R$  (the set of LIT cells covered by the new query boundary). Let  $c \in R$  be the nearest LIT cell of  $p$ . In the worst case, the procedure computes the minimum distance between  $p$  and  $c$  and the distance between  $c$  and the nearest border zone  $R$ . If the latter is smaller, the server does not have to inform  $p$  of the query insertion. This is because through the mobile-side LIT updates on  $p$ , the area covered by  $R$  will become zero zones before  $p$  moves into that area. Therefore, the query insertion will not cause any missed location updates. When  $q$  moves to a new location later, the system needs to perform the  $Qurlns$  procedure again to inform the moving objects that have neither been informed of the query insertion by  $Qurlns$  previously nor been assigned a new LIT with  $q$ 's location. The pseudo code of the  $Qurlns$  procedure is shown in Algorithm 2. In line 7, the procedure checks  $c$ 's LIT value because it represents the nearest distance (in cells) to the border zone.

---

#### Algorithm 2 $Qurlns(i, LIT)$

---

```

1:  $P' = \phi$ 
2: for (every old server-side LIT  $K$ ) do
3:   Let  $R$  be the set of LIT cells covering by the range of query  $i$ .
4:   Let  $O$  be the set of LIT cells where the objects' mobile-side LIT overlapping with  $R$ 
5:   for (each object  $p \in O$ ) AND  $p$  has not noticed yet do
6:     Let  $c \in R$  be the closest cell to  $p$ 
7:     Let  $v$  be the LIT value of  $c$  by checking the server-side LIT  $K$ 
8:     Let  $x = v \times \delta$ 
9:     Let  $y = mindist(p, c)$ 
10:    if ( $x \geq y$ ) then
11:      Insert  $p$  into  $P'$ 
12:    end if
13:  end for
14: end for

```

---

Consider the following example. A new query  $q$  registers with the system at  $t_1$ . Assume that  $p$  references to  $LIT_{serv}$  as shown in Figure 6, and each moving object is assigned a  $3 \times 3$  (level  $\ell = 1$ ) mobile-side LIT. The new query  $q$  covers the gray area  $R$ . Since a one-level mobile-side LIT is assigned to each moving object, the area  $O$  is one-level larger than  $R$ . Since  $x > y$ ,  $p$  may reach the closest cell  $c_{1,2}$  earlier than  $c_{1,2}$  becomes a zero zone through  $p$ 's mobile-side LIT updates. Therefore, the server needs to inform  $p$

of the new insertion. Now let us assume that  $q$  moves to  $q'$  later at  $t_2$ . The dark gray area  $O'$  represents the area where the objects have not been checked by  $Qurlns$  at  $t_1$ , so the  $Qurlns$  procedure only checks the moving objects in this area at  $t_2$ .

The pseudo code of Algorithm 3 represents the complete server-side event-driven system function. In line 7, the  $Qurlns$  procedure determines a set of objects that may be affected by newly inserted queries. Line 9 sends mobile-side LITs directly to objects  $O$  (downstream), because a mobile-side LIT is extracted from a server-side LIT, which is obtained based on the query locations only. The system can send the latest mobile-side LITs to  $O$  without probing their locations first. The mobile-side LITs here are extracted based their latest reported locations on the server. Therefore, each object  $o \in O$  can determine a location update based on the new mobile-side LIT. Any object  $o$  in  $O$  issues a location update (upstream) only when it is part of the result of the new query. Likewise,  $DectObj$  in lines 13-15 obtains a set of objects whose last reported locations at the server are covered by existing queries. Line 15 sends the latest mobile-side LIT to those objects and at the same time requests for the current locations (downstream). The moving objects then send back their current locations to the server (upstream). By applying these techniques our goal of reducing the overall network traffic is achieved.

---

#### Algorithm 3 System Overview

---

```

1: while (there is a request from a mobile unit  $r$ ) do
2:   Let  $B$  be a buffer
3:   if (the request is a moving object insertion) then
4:     Insert  $r$  to  $B$ 
5:   else if (the request is a query insertion) then
6:     Insert  $r$  into the query index
7:     Call  $Qurlns(r, LIT)$  that returns a set of objects  $O$ 
8:     Perform  $CreateLIT(G, Q)$  to generate a new server-side LIT.
9:     Send each  $o \in O$  the latest mobile-side LIT
10:  else if (the request is a moving object or query deletion) then
11:    Remove  $r$  from the system
12:  else if (the request is a query update) then
13:    Call  $DectObj(r)$  that returns a set of objects  $Z$ 
14:    Perform  $CreateLIT(G, Q)$  to generate a new server-side LIT.
15:    Send each  $z \in Z$  the latest mobile-side LIT and request for  $z$ 's current location
16:    Insert  $Z$  to  $B$ 
17:  if ( $r$  is an inserted query during the execution) then
18:    Call  $Qurlns(r, LIT)$  that returns a set of objects  $O$ 
19:    Send each  $o \in O$  the latest LIT
20:  end if
21:  else if (the request is a moving object update) then
22:    Insert  $r$  to  $B$ 
23:  end if
24: end while
25: Update/insert  $b$ 's location and cell index,  $\forall b \in B$ 
26: Call  $RangeQuery(Q)$  to retrieve query results
27: For each  $b \in B$ , Sent the latest mobile-side LIT to  $b$  if it does not have the latest LIT

```

---

### 3.5 Spatial Data Compression for Local LITs

Each moving object obtains a  $m \times m$  mobile-side LIT. While a large value for  $m$  provides more location information to a moving object, the data streams for the mobile-side LIT need to be broken up into more packets that adversely affect performance. In this paper, we use the Internet standard for the largest amount of data packet payload size (MTU) equal to 1500 bytes. In some prior techniques, such as the safe region approach, basically only a pair of  $x$ - and  $y$ -coordinates must be sent to the each moving object.

Hence such information can easily be fitted into a single packet. Our goal here is to use a data compression method to compress a LIT small enough to fit into one packet to perform on par with existing techniques. We apply three lossless data compression methods: quantization, run-length encoding (RLE) and Huffman encoding. Firstly, we de-correlate the LIT values by subtracting pairs of adjacent LIT numbers. This is effective since the LIT values are highly correlated or spatially redundant. As a result, the difference values are repetitive and smaller than the original LIT values. Secondly, RLE is utilized to take advantage of the large amount of spatial redundancy in a LIT and we use a Hilbert curve as the data scanning path along which we count repeated numbers. An extra marker bit is used to distinguish the count bit and number values. Finally, we performed Huffman encoding which is based on the frequency of occurrence of a data item and uses a lower number of bits to encode the data that occur more frequently. Huffman encoding following RLE is a natural choice since the result of RLE is a set of symbols representing run lengths that occur with varying frequencies. Overall, our experimental result shows the combination of these methods can reduce the size of a LIT by up to 79% from its original size. To pack a mobile-side LIT within a 1500 bytes packet, we suggest a maximum table size of  $64 \times 64$  for a mobile-side LIT.

## 4. Experimental Evaluation

We evaluated the performance of the partition-based lazy update algorithm (PLU) by comparing it with the safe region approach [3, 8] and the traditional periodical update approach (PER). For a fair comparison, we extended the safe region approach to handle dynamic queries. To process a query insertion, the safe-region approach probes a set of objects whose safe regions overlap with the boundary of the new query. In the case when a query changes its location, it is treated as a query insertion. We implement the extended safe region approach with safe rectangles (SR\*-Rec) and safe spheres (SR\*-SP).

We use a main memory  $100 \times 100$  grid as the underlying index structure for all the approaches. Our data sets are generated on a terrain service space of  $[0, 1000]$  based on the random walk mobility model [5]. Each object moves with a constant velocity until an expiration time. The velocity is then replaced by a new velocity with a new expiration time. For the range query, the query boundary is a square and the side length  $q_{len}$  is in the range of  $[1, 10]$ . The number of queries is up to 1k. The maximum moving distance per time step for any moving object is in the range  $[0.48, 1.25]$ , which corresponds to a speed of 35 to 90 miles per hour. The mobility  $f_{move}$  (the percentage of objects that move within a time step) for the objects is set in a range from 30% to 100%. We select an optimal size  $n$  for the sever-side LIT from  $[100, 500]$  per side and choose the level  $\ell$  from  $[1, 10]$  for a mobile-side LIT. The main measurement in the following simulations is the number of network messages sent between the server and moving objects. At the server side, we count the number of messages (downstream) from probing an object's location and sending an LIT to an object; at the mobile client side, we count the number of messages (upstream) from issuing a location update to the server. Experiments are conducted with a Pentium 2.4 GHz CPU and 1 GByte of memory. The query results are evaluated in an even-driven approach. Our experiments use several metrics to compare these algorithms. Table 2 summarizes the default parameter settings in the following simulations.

Parameter	Default	Range
$P$	100k	-
$Q$	1000	300, 500, 1000
$f_{move}$	50%	30%, 50%, 70%, 100%
$\lambda$	1.25	0.48(35mph)-1.25(90mph)
$q_{len}$	5	1, 5, 10
$m$	250	100, 200, 250, 500
$\ell$	5	1, 5, 10

Table 2. Simulation parameters

### 4.1 LIT Size

First, we measured the overall number of network messages including upstream and downstream directions of the PLU algorithm by varying the server-side LIT size. The choice of the server-side LIT size is a trade-off between the number of network messages and the server performance. Figures 7(a) and (b) show the number of overall network messages and CPU overheads v.s. the LIT sizes ranging from  $100 \times 100$  to  $500 \times 500$ , respectively. When the LIT size is set to more than 500 per side, the performance of PLU is degraded in terms of the number of network messages and CPU time because it incurs more LIT value calculations for all the LIT cells. The LIT size  $250 \times 250$  constitutes a good tradeoff between the number of network messages and CPU time. Therefore,  $250 \times 250$  is chosen as the server-side LIT size for the rest of our experiments.

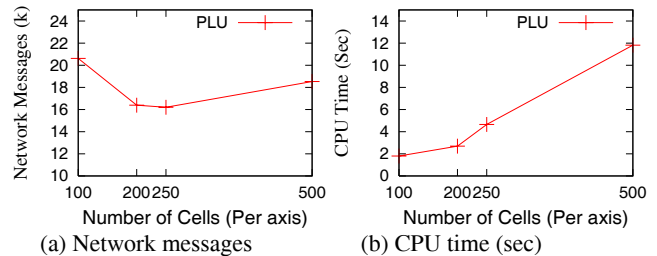


Figure 7. Performance v.s. server-side LIT size

Next we examine the size for a mobile-side LIT. Figures 8 measures the effect of varying the size of the mobile-side LIT from level 1 ( $3 \times 3$ ) to level 5 ( $11 \times 11$ ) in terms of network messages. The size of a mobile-side LIT significantly affects the number of network messages. When a mobile-side LIT is small, a moving object issues more network messages because it has more chance to move out of the LIT boundary; when a mobile-side LIT is large, it also incurs more network messages from the query insertion process since the procedure needs to check more objects from a larger area where the moving objects have the mobile-side LITs overlapping with the new query boundary. We choose  $\ell = 5$  as the mobile-side LIT size for the remaining experiments, because it achieves better performance in terms of the network messages.

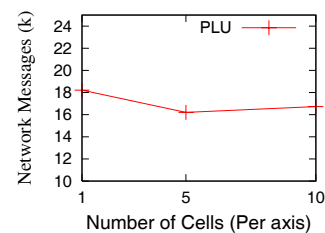


Figure 8. Performance v.s. mobile-side LIT size

## 4.2 Query Coverage

The query coverage on the terrain is a crucial factor in the performance of continuous query algorithms. The query coverage varies with the number and side length of the queries. Figure 9(a) shows the network messages as a function of the number of queries. Overall, the number of network messages increases as a function of the number of queries, because the chance of moving into the query boundaries for a moving object is high. PLU achieves a significant reduction in the number of updates compared to the other techniques. For the PER approach, since the server does not perform any computations regarding the location update reduction, we only count the number of network messages sent from the mobile clients. PER approach is independent of the query coverage, because the number of updates depends on the mobility only in PER approach. Therefore, the network messages remain the same in this simulation. In Figure 9(b), we evaluate the side length of queries with the values [1, 5, 10]. Obviously, when the length of queries increases, SR\*-Rec and SR\*-SP incur more updates, because SR\*-Rec and SR\*-SP perform server-side probes to those objects which have the safe regions overlapping with the queries. When the length of the queries increases, the server needs to probe more moving objects when queries change to new locations or when new queries are inserted. When  $q_{len} = 10$ , SR\*-SP has the same network messages as the baseline PER approach. The simulation results confirm the importance of adopting PLU approach which significantly reduces the network messages and hence decreases the communication cost.

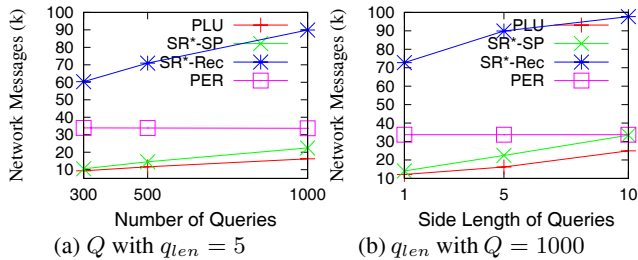


Figure 9. Effect of query coverage with  $Q$  and  $q_{len}$

## 4.3 Mobility

Finally, we evaluated the impact of the mobility rate. Figure 10(a) shows the number of network messages as a function of the object mobility. The PLU approach achieves a higher location update reduction than the other three approaches for all mobility rates. Figure 10(b) illustrates the CPU time v.s. the object mobility. Although PLU applies more server-side procedures (e.g. *DectObj* and *Qurlns*) to reduce the network messages, PLU still has a competitive CPU performance with SR\*-SP. However, SR\*-Rec has the worst performance in terms of network messages and CPU overheads. The degradation is caused by the expensive calculations of safe rectangles. SR\*-Rec in general computes larger safe regions for moving objects than SR\*-SP, so SR\*-Rec incurs many server-side probes to the moving objects when queries change their locations.

## 5. Conclusions

We have designed a partition-based lazy update approach for highly dynamic environments where mobile units (including queries) may freely change their locations. The novel concept of a Location Information Table is introduced to provide a mobile object

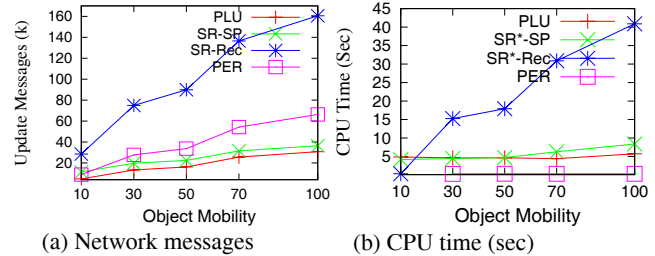


Figure 10. Performance v.s. object mobility

with information about queries, hence enabling it to estimate query movements and transmit a location update to the server only when it affects the query results. To further reduce network traffic the server uses smart on-demand location probes. Finally, the proposed mechanism efficiently determines the set of objects that are affected by a query insertion, improving scalability. The data structure for a LIT is larger than for a safe region and we present a spatial data compression method to reduce its size and fit it into an Internet packet. Experimental results demonstrate that our approach scales better than existing techniques as it reduces the update message traffic by 10% to 28% compared with the extended safe sphere method, and from 10% to 70% for the periodic approach. These results are even more noteworthy as we compare our technique with an enhanced version of the safe region approach that we improved to handle moving queries.

## 6. Acknowledgments

This research has been funded in part by NSF grants EEC-9529152 (IMSC ERC), CMS-0219463 (ITR), IIS-0534761, NUS AcRF grant WBS R-252-050-280-101/133 and equipment gifts from the Intel Corporation, Hewlett-Packard, Sun Microsystems and Raptor Networks Technology. We also acknowledge the support of the NUS Interactive and Digital Media Institute (IDMI).

## 7. References

- [1] Y. Cai, K. Hua, and G. Cao. Processing Range-Monitoring Queries on Heterogeneous Mobile Objects. In *MDM*, 2004.
- [2] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT*, 2004.
- [3] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *SIGMOD Conference*, pages 479–490, 2005.
- [4] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In *VLDB*, 2004.
- [5] A. B. McDonald. A mobility-based framework for adaptive dynamic cluster-based hybrid routing in wireless ad-hoc networks, 1999.
- [6] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *SIGMOD Conference*, pages 623–634, 2004.
- [7] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD Conference*, 2005.
- [8] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Trans. Computers*, 51(10):1124–1140, 2002.
- [9] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD Conference*, 2000.
- [10] Y. Tao, D. Papadias, and J. Sun. The TPR\*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *VLDB*, pages 790–801, 2003.
- [11] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *ICDE*, pages 643–654, 2005.
- [12] X. Xiong, M. F. Mokbel, and W. G. Aref. LUGrid: Update-tolerant grid-based indexing for moving objects. In *MDM*, 2006.
- [13] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *ICDE*, pages 631–642, 2005.