A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects[*]

Haibo Hu
Dept. of Computer Science
Hong Kong University of Science & Technology
haibo@cs.ust.hk

Jianliang Xu
Dept. of Computer Science
Hong Kong Baptist University
xujl@comp.hkbu.edu.hk

Dik Lun Lee
Dept. of Computer Science
Hong Kong University of Science & Technology
dlee@cs.ust.hk

ABSTRACT

This paper proposes a generic framework for monitoring continuous spatial queries over moving objects. The framework distinguishes itself from existing work by being the first to address the location update issue and to provide a common interface for monitoring mixed types of queries. Based on the notion of safe region, the client location update strategy is developed based on the queries being monitored. Thus, it significantly reduces the wireless communication and query reevaluation costs required to maintain the up-to-date query results. We propose algorithms for query evaluation/reevaluation and for safe region computation in this framework. Enhancements are also proposed to take advantage of two practical mobility assumptions: maximum speed and steady movement. The experimental results show that our framework substantially outperforms the traditional periodic monitoring scheme in terms of monitoring accuracy and CPU time while achieving a close-to-optimal wireless communication cost. The framework also can scale up to a large monitoring system and is robust under various object mobility patterns.

1. INTRODUCTION

With the advent of mobile and ubiquitous computing, monitoring continuous spatial queries over moving objects has become a necessity for various daily applications, such as fleet management, cargo tracking, child care, and location-aware advertisement. Such a monitoring system typically consists of a database server, some base stations, application servers, and a large number of moving objects (see Figure 1.1).[1] The database server manages the locations of the moving objects. The application servers register spatial queries of interest at the database server, which then continuously updates the query results until the queries are deregistered. There are two predominant costs that determine the system performance: the wireless communication cost for location updates and the query evaluation cost at the database server.

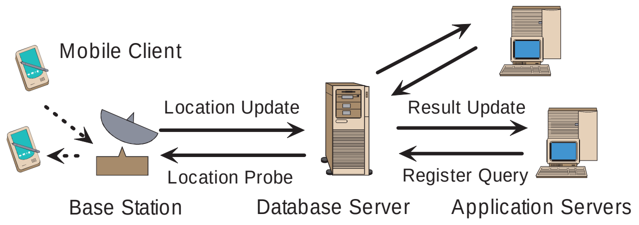


Figure 1.1: The System Architecture

Most existing studies on continuous query monitoring [13, 17, 19, 28] focused on reducing the evaluation cost only. For the location update cost, they simplified the problem by assuming that the update is decided solely by the moving client itself. In other words, each autonomous client reports its new location either periodically or whenever the location (or velocity) changes significantly from the last update. This assumption has several deficiencies. First, location updates are query blind, i.e., location updates are performed regardless of the existence of queries. When there are very few queries, the precious wireless bandwidth and client battery power are wasted as most of the updates do not yield changes to the query results. Second, the monitored query results may deviate from the actual values because the server is unaware of the locations of the objects between location updates. Under the periodic update scheme, the degree of deviation largely depends on the update frequency, which is hard to optimize, because a high frequency may unnecessarily burden both the server and the client whereas a low frequency may result in a significant degree of deviation. Third, the server load is not balanced over time, because location updates from all of the clients must be synchronized in order to produce the correct query results, thus creating high workloads at the synchronization points when the server has to handle numerous location updates and query reevaluations simultaneously (the delayed reevaluation also worsens the result deviation problem). In addition to the simplification resulted from the assumption on autonomous location updates, most of the previous studies were designed to support specific query types only, e.g., [19] for range queries only, [24] for k-nearest neighbor (kNN) queries only, and [12] for distance semi-joins only.

[*]This work is supported by the Research Grants Council, Hong Kong SAR under grants AoE/E-01/99 and HKUST6277/04E.

[1]In this paper, the terms “moving object” and “mobile client” are used interchangeably.

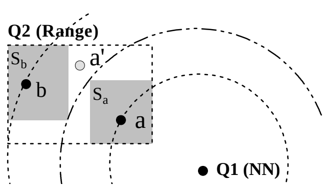


Figure 1.2: Example of Safe Regions

In this paper, we propose an innovative and generic monitoring framework to overcome these problems by taking a systematic approach. More specifically, the clients are aware of the spatial queries being monitored, so the location updates occur only when the results for some queries might change. To guarantee not missing any result change, the database server maintains a view on each moving object, i.e., a rectangular area called a *safe region*,[2] around the object. The safe region of each object is computed at the server side based on the queries in such a way that the current results of all queries are guaranteed to remain valid as long as all objects are residing inside their respective safe regions. Figure 1.2 shows a simple example where there are two moving objects a and b and two registered queries $Q1$ (an NN query) and $Q2$ (a range query). The current results of these two queries are $\{a\}$ and $\{a, b\}$ respectively, which will not change if a and b are in their safe regions S_a and S_b (the shaded boxes).

Each moving object is aware of its safe region and issues location updates to the server only when it moves out of the region (this is called a *source-initiated update*). After the database server receives this update, it finds and incrementally reevaluates the affected queries and computes a new safe region for the object. For a query involving the spatial relations of two or more clients (e.g., a kNN query like $Q1$), the location update from one client may invalidate the safe regions of some other clients because the query result is now undecided based on the safe regions. As an example in Figure 1.2, when a moves out of S_a to a new location a', the result of $Q1$ becomes undecided as either of the two objects could be the nearest neighbor. To resolve the ambiguity, the server has to *probe* some objects (in this example, b) to request an immediate location update (this is called a *server-initiated probe and update*).

Compared to the previous work, the proposed framework exhibits the following advantages:

- To our knowledge, this is the first monitoring scheme that addresses the location update issue. By safe regions, the moving clients are query aware and report location updates to the server only when they are very likely to alter the results. Hence, both the wireless communication cost and the query evaluation cost are greatly reduced. Yet the performance gain is not achieved by sacrificing the simplicity at the client side: the new location update logic is as straightforward as before.
- The framework is generic in the sense that it is not designed for a specific query type. Rather, the framework provides a common interface for monitoring various types of spatial queries such as range queries and kNN queries. Moreover, the framework does not presume any mobility pattern on moving objects.
- Since any object movement that might change the query results is captured by the location update, the framework offers us *accurate* monitoring results *at any time*,[3] as opposed to deviated results as can be observed in the previous periodic monitoring approach.
- In contrast to periodical reevaluation of queries in most existing studies, query reevaluation in this framework is triggered by location updates only. As these location updates are asynchronous, the workload of the server is evenly distributed over time.

[2]We adopt a rectangular shape for the safe region because of its simplicity and efficiency in query processing, as inspired from the use of rectangle to bound index entries in spatial index structures such as the R-tree [9].

Two fundamental issues at the database server need to be addressed in this framework:

1. How to evaluate a newly registered query based on the set of the safe regions and then to update the safe regions of the objects being probed during the evaluation? As these are the only objects leading to ambiguous results, they are the only ones that need to be aware of this new query.
2. Upon receiving a source-initiated location update from a moving object, how to find and incrementally reevaluate the queries whose results are potentially affected and to update the safe regions of this object and any other object being probed during the reevaluation?

In the rest of this paper, we are going to answer these questions with the objective of minimizing the number of location updates, which has a direct impact on the wireless communication cost and power consumption, the most precious resources in mobile environments. The remaining of the paper is structured as follows. Section 2 reviews the related work. We introduce the system model and the framework overview in Section 3. In Sections 4 and 5, we discuss the techniques for query evaluation/reevaluation and safe region computation with respect to range and kNN queries. Two enhancements are proposed by taking advantage of mobility assumptions in Section 6. We conduct extensive simulation experiments to evaluate the performance of the proposed framework in Section 7.

2. RELATED WORK

There is a large body of research work on spatial-temporal query processing. Early work assumed a static dataset and focused on efficient access methods (e.g., R-tree [9]) and query evaluation algorithms (e.g., [10, 21]). Recently, a lot of attention has been paid to moving-object databases, where data objects or queries (or both) move.

Assuming object movement trajectories are known a priori, Saltenis *et al.* [22] proposed the Time-Parameterized R-tree (TPR-tree) for indexing moving objects, in which the location of a moving object is represented by a linear function of time. Benetis *et al.* [3] developed query evaluation algorithms for NN and reverse NN search based on the TPR-tree. Tao *et al.* [25] optimized the performance of the TPR-tree and extended it to the TPR*-tree. Chon *et al.* [7] studied range and kNN queries based on a grid model. Patel *et al.* [18] proposed a novel index structure called STRIPES using a dual transformation technique.

[3]The framework guarantees this accuracy without considering the delay between location updating and query processing.

The work on monitoring continuous spatial queries can be classified into two categories. The first category assumes known movement trajectories. Continuous kNN monitoring has been investigated for moving queries over stationary objects [24] and linearly moving objects [11, 20]. Iwerks *et al.* even extended [11] to monitor distance semijoins for two linearly moving datasets [12]. However, as pointed out in [23], such an assumption does not hold for many application scenarios (e.g., the velocity may change constantly when a car moves on the road).

The second category does not make any assumption on object movement patterns. Xu *et al.* [27] and Zhang *et al.* [29] suggested returning to a moving query the current result as well as its validity scope where the result remains the same. The query is reevaluated only when the query exits the validity scope. However, these proposals work for stationary objects only. For continuous monitoring of moving objects, the prevailing approach is periodic reevaluation of queries [13, 17, 19, 28]. Prabhakar *et al.* [19] proposed the Q-index, which indexes queries using an R-tree-like structure. At each evaluation step, only those objects that have moved since the previous evaluation step are evaluated against the Q-index. While this study is limited to range queries, Mokbel *et al.* [17] proposed a scalable incremental hash-based algorithm (SINA) for range and kNN queries. SINA indexes both queries and objects, and it achieves scalability by employing shared execution and incremental evaluation of continuous queries [17, 26]. Kalashnikov *et al.* and Yu *et al.* suggested grid-based in-memory structures for object and query indexes to speed up reevaluation process of range queries [14] and kNN queries [28]. Access methods to support frequent location updates of moving objects have also been investigated (e.g., [13, 15]). Our study falls into this category and distinguishes itself from these previous studies as discussed in the Introduction.

Distributed approaches for monitoring continuous range queries have been investigated in [5, 8]. The main idea is to shift some load from the server to the mobile clients. Monitoring queries have also been studied for distributed Internet databases [6], data streams [1], and sensor databases [16]. However, these studies are not applicable to monitoring of moving objects, where a two-dimensional space is assumed.

3. THE FRAMEWORK OVERVIEW

In this section, we first make some assumptions on the system model, and then introduce the framework by describing the structure and behaviors of the database server. The next two sections will show the detailed query evaluation/reevaluation and safe region computation algorithms at the database server.

To simplify the system model, we make the following assumptions:

- At the database server, all registered queries can be fit into main memory whereas not all the moving objects can. This is a common and fair assumption in monitoring continuous spatial queries [14, 28].
- The database server handles location updates sequentially. In other words, no location updates take place during the processing of a new query or another location update. Although this is not a prerequisite for this framework, it is a reasonable assumption to relieve us from considering the read/write consistency. In fact, in reality communication incurs a measurable update propagation delay and thus the exact client location and the location maintained at the database server are not always synchronized. The violation of this assumption affects the monitoring accuracy, which will be measured in the experiments.
- The communication cost between every client and the database server is constant. Throughout this paper, C_l denotes the cost for one source-initiated location update and C_p denotes the cost for one server-initiated location probe and update.
- Mobile clients are able to detect their locations, through positioning technologies such as GPS.

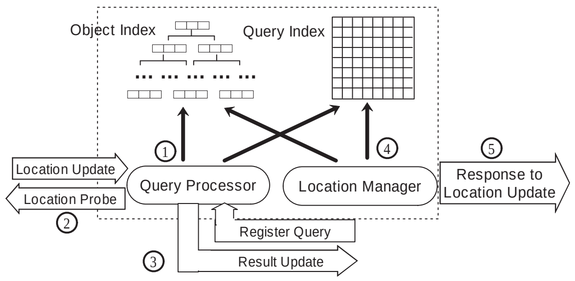


Figure 3.1: The Database Server Structure

3.1 The Database Server

As depicted in Figure 3.1, the database server has four components: the on-disk index for the moving objects, the in-memory index for the queries, the location manager, and the query processor. Upon receiving a location update, the query processor first reevaluates those queries affected by this update (step ①) based on the indexes. During the reevaluation, the query processor might need to probe some objects for server-initiated location updates to determine the query results (step ②). The updated query results are then reported to the application servers which register these queries (step ③). Afterwards, the location manager computes the new safe regions for this object and the probed objects (step ④), also based on the two indexes. Finally, these new safe regions are sent back to the corresponding clients as the responses for their location updates (step ⑤). The server's behaviors upon a new query being registered are similar: it goes through steps ① to ⑤, except that in ①, the query is evaluated from scratch instead of being reevaluated incrementally.

3.2 The Object Index

The object index stores the current safe regions of all the objects. While many spatial index structures can serve this purpose, this paper employs the well-known R-tree based index [2, 9]. Since the safe region changes each time the object updates its location (either client-initiated or server-initiated), the index should be optimized to handle frequent updates. The existing study on this issue (e.g., [15]) can be adopted in this framework.

3.3 The Query Index

For each query, the database server stores: (1) the parameters of the query (e.g., the rectangle of a range query, the query point and the k value of a kNN query), (2) the current query results and, (3) the *quarantine area* of the query. The quarantine area is such an area that as long as all result objects stay inside it and all non-result objects stay outside it, the results of this query do not change. This area is used to identify the affected queries upon a source-initiated location update. For a range query, the quarantine area is simply the query rectangle; for a kNN query, the area can be any circle centered at the query point and completely covering the k-th NN o_k but not the $k+1$-th NN o_{k+1}. In other words, the radius of the circle should be equal to or greater than $\Delta(q, o_k)$, the maximum distance between the query point q and the safe region of o_k, but less than $\delta(q, o_{k+1})$, the minimum distance between q and the safe region of o_{k+1}.[4] In this paper, we set the radius as the midpoint of these two distances.

Through the notion of quarantine area, upon a location update a query Q is affected only if of the new updated location p and the last updated location p_{lst}, one point is in the quarantine area of Q and the other is not. To quickly locate these affected queries, a grid-based index is built on the quarantine areas of all registered queries in main memory. The grid-index partitions the entire space into $M \times M$ uniform cells. The bucket for each cell stores the pointers pointing to those queries whose quarantine areas overlap this cell. Therefore, upon a location update, only those queries pointed by the cell containing p and the cell containing p_{lst} need to be checked for possible reevaluation.

So far we ignore the ordering of the k nearest neighbors for a kNN query, i.e., the query is *order-insensitive*. However, if the kNN query is *order-sensitive* (the result set $\{a, b\}$ is different from $\{b, a\}$), even if both p and p_{lst} are inside the quarantine area, the results might change. Therefore, an order-sensitive kNN query Q is not affected only if p and p_{lst} are both outside its quarantine area.

4. QUERY EVALUATION AND REEVALUATION

Algorithm 1 Overview of Database Behavior

```
1:  while receiving a request do
2:    if the request is to register query $q$ then
3:      evaluate $q$ (probe objects' locations if necessary);
4:      compute $q$'s quarantine area and insert it into the
        query index;
5:      return the results to the application server;
6:    else if the request is to deregister query $q$ then
7:      remove $q$ from the query index;
8:    else if the request is a location update from object $p$
      then
9:      determine the set of affected queries;
10:     for each affected query $q'$ do
11:       reevaluate $q'$ (probe objects' locations if neces-
          sary);
12:       update the results to the application server;
13:       recompute $q'$'s quarantine area and update the
          query index;
14:     update the safe region of $p$;
15:   update the safe region of any probed object;
```

Algorithm 1 shows the skeleton code of the database server handling a query registration/unregistration or a source-initiated location update. When a new query is registered, the query processor evaluates its initial result based on the object index. The evaluation is different from that for traditional spatial queries as the objects are represented by safe regions rather than their exact locations. In case of ambiguity, location probes are necessary. In order to reduce the number of probes, we apply a *lazy probe* technique so that probes occur only when the evaluation cannot continue. Similarly, upon receiving a source-initiated location update, the query processor finds out the affected queries using the query index. It then incrementally reevaluates these queries and updates their results if they change. Finally, the safe regions of the updated object and the probed objects are recomputed by the location manager. We discuss query evaluation and reevaluation algorithms in this section and the computation of safe regions in the next section. Due to space limitations, this paper presents the solutions for two most common spatial queries, namely the range and kNN queries, and the extension to other types of queries follows the same rationale.

4.1 Evaluating New Range Query

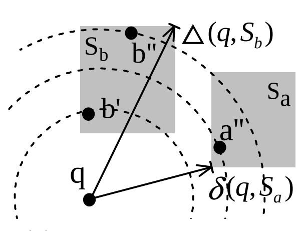
Processing a new range query on safe regions is very similar to that on exact object locations. We start from the index root and recursively traverse down the index entries that overlap the query rectangle until reaching the leaf entries where the safe regions are stored. If the safe region of an object is fully covered by the query rectangle, the object is a result. Otherwise, they overlap and the object must be probed to resolve the ambiguity.

4.2 Evaluating New kNN Query

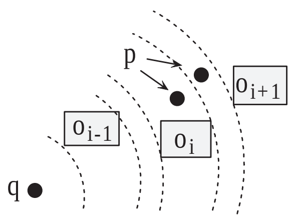
Evaluating an order-sensitive kNN query on safe regions is more complicated than that on exact object locations. We adopt the best-first search (BFS) as the paradigm for this algorithm [10]. Similar to the original BFS, we maintain a priority queue which stores intermediate or leaf index entries (i.e., object locations). The object location is in the form of either a safe region or the exact point (after the server probes it). The key to sort the elements in the queue is the (minimum) distance to the query point q. The searching algorithm (Algorithm 2) is the same as the original BFS except when a leaf entry, object p, is popped from the queue. If p is already represented by a point, it is returned as a result immediately. Otherwise p, represented by a safe region, is held until the next object u is popped. Then, the maximum distance between p and q ($\Delta(q, p)$) is compared with the minimum distance between u and q ($\delta(q, u)$). If the former is shorter, p is guaranteed closer to q than any other objects in the queue. As such, p is returned as a result. However, if the former is longer, the query result is undecided based on the safe regions. Therefore, p is probed and p together with its exact location is inserted back to the priority queue. And u is also inserted back to the queue. The algorithm continues until k objects are returned as results. In addition, in order to find the radius r of the quarantine area for this query, the algorithm pops one more element from the queue and the value of r is the midpoint between the keys of the k-th NN and the last popped element.

It is worthwhile to note that this algorithm guarantees that the object is not probed until it is about to be returned.

[4]Throughout this paper, $d(s, t)$ denotes the distance between two points s and t, $\delta(S, T)$ ($\Delta(S, T)$) denotes the minimum (maximum) distance between a pair of points in areas S and T. Note that S or T could also be a point.



(a) Evaluation of New kNN



(b) Reevaluation of kNN

Figure 4.1: Processing a kNN Query

As such, the lazy probe technique assures all the probes are mandatory. The same technique is also exploited in reevaluating kNN queries in the next subsection.

Continuing on the example shown in Figure 1.2, suppose a new NN query issued from q (see Figure 4.1(a)) is registered. At some stage, the priority queue contains b and a. Then, b is popped from the queue; since it is represented by a safe region S_b, it is held. Then, a is popped. Since $\Delta(q, S_b) > \delta(q, S_a)$, b is probed. If b is at location b', which is shorter than $\delta(q, S_a)$, b is popped again prior to a; this time it is returned as the result and $r = (d(q, b') + \delta(q, S_a))/2$. Otherwise, if b is at location b'', a is popped before b and since $\Delta(q, S_a) > d(q, b'')$, a is also probed. If afterwards it turns out that a is at point a'', a is returned as the result and $r = (d(q, a'') + d(q, b''))/2$.

Algorithm 2 Evaluating a new kNN Query

```
Input: root: root node of object index
       q: the query point
Output: C: the set of kNNs
       r: the radius of the quarantine area
Procedure:
1: initialize the priority queue;
2: enqueue ⟨root, $\delta(q, root)$⟩ into the queue;
3: while $|C| < k$ and queue is not empty do
4:     dequeue the top element to $u$;
5:     if $u$ is an object location then
6:         if there is an object $p$ held then
7:             if $\Delta(q, p) \leq \delta(q, u)$ then
8:                 insert $p$ into $C$;
9:             else
10:                enqueue ⟨$u$, $\delta(q, u)$⟩ back to the queue;
11:                probe $p$;
12:                enqueue ⟨$p$, $d(q, p)$⟩ back to the queue;
13:                continue;
14:        if $u$ is represented by a safe region then
15:            hold $u$;
16:        else
17:            insert $u$ into $C$;
18:    else if $u$ is an index entry then
19:        for each child entry $v$ of $u$ do
20:            enqueue ⟨$v$, $\delta(v, q)$⟩ into the queue;
21: dequeue one more element to $u$;
22: $r = (\Delta(q, C_k) + \delta(q, u))/2$;
23: return $C$ and $r$;
```

The evaluation of an order-insensitive kNN query is the same as Algorithm 2 except that at most k objects can be held at the same time. Lines 7 ∼ 8 in Algorithm 2 are executed for any p that is held and lines 10 ∼ 13 are executed only if there are already k objects held (here p is the first held object). As such, the number of probes is fewer than that in an order-sensitive kNN query.

4.3 Reevaluating Range and kNN Queries

The affected queries to be reevaluated are first identified based on the query index, as discussed in Section 3.3. The incremental reevaluation of an affected range query is straightforward. If the updated object p is now inside the query rectangle Q, p must have moved into this rectangle from outside, so it becomes a new result of Q. Similarly, if p is now outside the rectangle, p is removed from the result set of Q. In either case, the database server reports the update to the application server.

For an order-sensitive kNN query, there are three cases in reevaluation: 1) p is not in the quarantine area, but p_{lst} is; 2) p is in the quarantine area, but p_{lst} is not; and 3) both p and p_{lst} are in the quarantine area. In case 1, p possibly becomes a non-result, so a 1NN query is issued to find the new k-th NN. The algorithm is the same as Algorithm 2 except that the remaining $k-1$ result objects should be excluded from the search. In case 2, before the location update, the set of result objects $\{o_1, o_2, \cdots, o_k\}$ of this query are strictly ordered in terms of their distances to q. In other words, $\delta(q, o_1) \leq \Delta(q, o_1) \leq \delta(q, o_2) \leq \Delta(q, o_2) \leq ... \leq \delta(q, o_k) \leq \Delta(q, o_k)$. When p is updated, $d(q, p)$ is either between $\Delta(q, o_i)$ and $\delta(q, o_{i+1})$ or between $\delta(q, o_i)$ and $\Delta(q, o_i)$ for some $1 \leq i \leq k$ (see Figure 4.1(b)). For the former case, p directly becomes the $(i + 1)$-th NN. For the latter case, o_i must be probed to decide which one is closer, o_i or p. Whichever the case is, the NNs after o_i have their NN sequence numbers increased by 1 and the last NN o_k is dropped. The radius of the new quarantine area is the average of $\Delta(q, o'_k)$ (o'_k is the new k-th NN) and $\delta(q, o_k)$. Case 3 is similar to case 2 except that p is removed from the result set (i.e., $\delta(p, q)$ and $\Delta(p, q)$ are removed from the above sequence) before $d(q, p)$ is located in this sequence as in case 2, and that the last NN is not dropped. The radius of the quarantine area in this case does not change. It is noteworthy that in any case the above reevaluation algorithm needs at most one location probe.

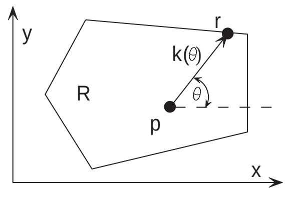
For an order-insensitive kNN query, only the former two cases exist. Since there is no strict ordering among current results, the query must be reevaluated as a new query.

5. SAFE REGION COMPUTATION

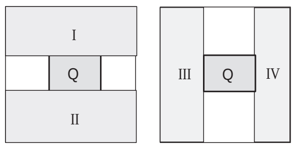
The safe region of a moving object p (denoted as $p.sr$) designates how far p can reach without affecting the results of any registered query. As queries are independent of each other, we define the safe region for a query Q (denoted as $p.sr_Q$) as the rectangular region in which p does not affect Q's result. $p.sr_Q$ is essentially a rectangular approximation of Q's quarantine area or its complement. Obviously, $p.sr$ is the intersection of individual $p.sr_Q$ for all registered queries. To efficiently eliminate those queries whose $p.sr_Q$ do not contribute to $p.sr$, we require $p.sr$ (and $p.sr_Q$) to be fully contained in the grid cell (defined in Section 3.3) in which p currently resides. By this means, we only need to compute $p.sr_Q$ for those queries whose quarantine areas overlap this cell as the $p.sr_Q$ for any rest query is the cell itself. These overlapping queries are called *relevant queries* and are exactly pointed by the bucket of this cell in the query index (see Section 3.3).

Recall in Algorithm 1 that the server needs to recompute the safe region of an object p in three cases: 1) During the evaluation of a new query Q, if p is probed, its safe region needs to be updated. Since none of the existing queries change their quarantine areas, the new safe region $p.sr'$ is simply the intersection of the current safe region $p.sr$ and the safe region for this new query Q, i.e., $p.sr' = p.sr \cap p.sr_Q$. 2) After processing a source-initiated location update of object p, p's safe region needs to be completely recomputed by computing the $p.sr_Q$ for each relevant query. 3) During the processing of a source-initiated location update, if object p is probed, its safe region is also completely recomputed as in case 2. Although it is still a probe as in case 1 and only one $p.sr_Q$ changes (i.e., the query which probes p), we completely recompute $p.sr$ since $p.sr_Q$ could be enlarged by this probe and recomputing it allows such enlargement to contribute to $p.sr$.

As the objective of a safe region is to reduce the number of location updates, we first give a theorem which shows that minimizing the number is equivalent to maximizing the perimeter of the safe region.



(a) Theorem 5.1



(b) The Four Candidates

Figure 5.1: Computing Safe Regions

THEOREM 5.1. *Assume that the object p moves in a randomly chosen direction with a constant speed ϕ (see Figure 5.1(a)). Given a convex safe region R and the updated location p, the amortized location update cost for p over time, $Cost_p$, is*

$$Cost_p = C_l \cdot \Big(\int_0^{2\pi} \frac{k(\theta)d\theta}{2\pi\phi}\Big)^{-1} = \frac{C_l \cdot 2\pi\phi}{Perimeter(R)},$$

where C_l is the cost for one location update, θ is the angle between the moving direction and the positive x-axis, $k(\theta)$ is the length of segment $\overline{pr}$, r is the intersection point of this direction and the boundary of R, in other words, r is the location at which the next location update occurs.

PROOF. First of all, r must be unique for every θ. Otherwise, if there were another r', the points in segment rr' do not belong to R, which contradicts the convex assumption. As such, given θ, the elapsed time before the next location update is $\frac{k(\theta)}{\phi}$. The average elapsed time over all θ is $\frac{\int_0^{2\pi}\frac{k(\theta)}{\phi}d\theta}{\int_0^{2\pi}d\theta} = \int_0^{2\pi}\frac{k(\theta)d\theta}{2\pi\phi}$. Therefore, we have

$$Cost_p = C_l \cdot \Big(\int_0^{2\pi} \frac{k(\theta)d\theta}{2\pi\phi}\Big)^{-1} = \frac{C_l \cdot 2\pi\phi}{Perimeter(R)},$$

because $\int_0^{2\pi} k(\theta)d\theta = Perimeter(R)$. $\square$

Therefore, the problem of computing $p.sr_Q$ is to find the rectangle with the longest perimeter which is inside p's grid cell as well as Q's quarantine area (or its complement) and yet contains the updated location of p. In what follows, we first discuss the computation of $p.sr_Q$ for a range query and a kNN query, and then propose an efficient algorithm to compute the safe region for a batch of range queries.

5.1 Safe Region for Range Query

If Q is a range query and p is in its quarantine area (i.e., the query rectangle), the safe region $p.sr_Q$ is simply the quarantine area itself. Otherwise, p is outside the quarantine area, and there are four possible rectangles within the cell that can serve as $p.sr_Q$, each of which has one of its sides coincide with a side of the cell (see Figure 5.1(b)). $p.sr_Q$ should be the one with the longest perimeter. There is a chance that the quarantine area is not fully contained in the cell; in such cases, only the area within the cell is considered as Q's quarantine area.

5.2 Safe Region for kNN Query

For an order-insensitive kNN query, if p is in its quarantine area (a circle), $p.sr_Q$ should be the $\underline{i}$nscribed $\underline{r}$ectangle of this circle with the $\underline{l}$ongest $\underline{p}$erimeter which contains p, or for short the *Ir-lp* of the circle. Otherwise, $p.sr_Q$ should be the *Ir-lp* of the complement of this circle, which is the grid cell subtracted by the circle. However, for an order-sensitive kNN query, the movement within the quarantine area might still affect the kNN order. Therefore, we need to restrain p from interfering with other kNNs. Suppose p is the i-th NN and q is the query point, we have $\Delta(q, o_{i-1}.sr) \le d(q,p) \le \delta(q, o_{i+1}.sr)$. In other words, p must be inside the ring centered at q with inner radius $\Delta(q, o_{i-1})$ and outer radius $\delta(q, o_{i+1})$. So $p.sr_Q$ should be the *Ir-lp* of this ring. If $i = 1$, the ring degrades to a circle and if $i = k$, the ring degrades to the complement of a circle. In case the safe region of o_{i-1} is invalid (o_{i-1} might be probed and its safe region is not recomputed yet), we replace $\Delta(q, o_{i-1}.sr)$ with $(d(q, o_{i-1}) + d(q,p))/2$ as the inner radius. The similar replacement is made if the safe region of o_{i+1} is invalid.

In the following, we propose the solution to find the *Ir-lp* of a circle, the complement of a circle, and a ring. Similar to range queries, there is a chance that the circle or ring is not fully contained in the cell. To simplify such scenarios, we enlarge the cell to fully contain the circle or ring, and the resultant *Ir-lp* will then be intersected by the original cell to yield the final $p.sr_Q$.

5.2.1 *Ir-lp of a Circle*

Let x denote one of the corners for an inscribed rectangle of the circle centered at q with radius r, and θ denote the angle between $\overline{qx}$ and the y-axis (see Figure 5.2). Here without loss of generality, we suppose p is in the third quadrant from the origin q. The perimeter is $4r(sin\theta + cos\theta)$. The first derivative of θ shows that the perimeter has a maximum at $\pi/4$ and the value monotonously increases (or decreases) before (or after) this point. However, as seen in the figure, θ is only valid within the range between $arcsin\frac{q.x-p.x}{r}$ (denoted as θ_x) and $arccos\frac{q.y-p.y}{r}$ (denoted as θ_y) in order for p to be contained in $p.sr_Q$, where $p.x$ ($p.y$) is the x-coordinate (y-coordinate) of p. As such, we reach the following proposition:

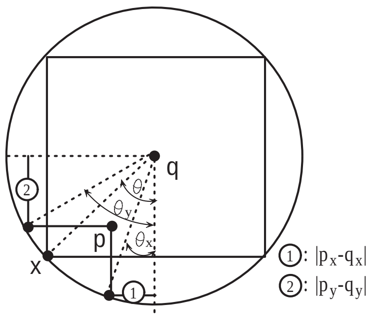


Figure 5.2: *Ir-lp* **of a Circle**

PROPOSITION 5.2. *The Ir-lp of a circle centered at p with radius r is the inscribed rectangle whose θ is set as follows:*

$$\theta = \begin{cases} \pi/4 & if\, \theta_x \le \pi/4 \le \theta_y\,, or \\ \theta_y & if\, \theta_y < \pi/4\,, or \\ \theta_x & if\, \theta_x > \pi/4. \end{cases}$$

5.2.2 Ir-lp of the Complement of a Circle

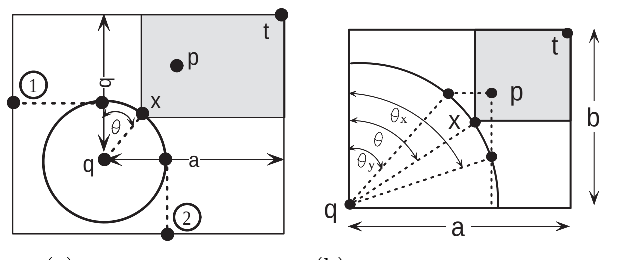
With point q being the origin, the cell has four quadrants; each of the four corners of the cell corresponds to one quadrant. Let t denote the corner that corresponds to the quadrant where p resides (see Figure 5.3(a)). We have the following lemma regarding t and the *Ir-lp* of this complement:

LEMMA 5.3. *t is a corner of the Ir-lp.*

PROOF. Otherwise, the corner of the *Ir-lp* that is closest to t could be extended to t without causing overlap with the circle. The extended rectangle is thus with a longer perimeter than the *Ir-lp*, which contradicts the definition of *Ir-lp*. □

By this lemma, we fix one corner of the *Ir-lp* at t and find the position of its opposite corner (denoted as x in Figure 5.3(a)). Without loss of generality, let p, x, and t be in the first quadrant. There are only three possible positions for x: some point on the 1/4 circle, point ①, and point ②. Let θ still denote the angle between $\overline{qx}$ and the y-axis. If x is on the circle, θ must range between 0 and $\pi/2$, otherwise the *Ir-lp* overlaps the circle, although the actual valid θ range might be smaller (discussed in the next paragraph). Point ① is possible only when the valid range covers $\theta = 0$ and point ② is possible only when it covers $\theta = \pi/2$.

Figure 5.3(b) is a closeup of the first quadrant depicted in Figure 5.3(a). If x is on the 1/4 circle, in order for p to be



(a) Four Subproblems (b) Solution of One Subproblem

Figure 5.3: *Ir-lp* **of the Complement of a Circle**

contained in $p.sr_Q$, θ must range between θ_y ($arccos\frac{p.y-q.y}{r}$ or 0 if $p.y - q.y > r$) and θ_x ($arcsin\frac{p.x-q.x}{r}$ or $\pi/2$ if $p.x - q.x > r$). The perimeter of the *Ir-lp* is: $2(a - r \cdot sin\theta) + 2(b - r \cdot cos\theta)$, where a, b are the width and height of the cell within this quadrant (see Figure 5.3). The first-order derivative of θ shows that the perimeter has a maximum at $\pi/4$ and the value monotonously increases (or decreases) before (or after) this point. Meanwhile, if $\theta_y = 0$ or $\theta_x = \pi/2$, we still need to check the rectangle whose x is at positions ① or ② to see if its perimeter is longer. Therefore, we reach the following proposition:

PROPOSITION 5.4. *The Ir-lp of the complement of a circle centered at q with radius r is the inscribed rectangle with one corner being the cell corner corresponding to p and the opposite corner is x. x is either on the 1/4 circle whose θ is:*

$$\theta = \begin{cases} \pi/4 & if\, \theta_y \le \pi/4 \le \theta_x\,, or \\ \theta_x & if\, \theta_x < \pi/4\,, or \\ \theta_y & if\, \theta_y > \pi/4, \end{cases}$$

or at position ① if $\theta_y = 0$ and at position ② if $\theta_x = \pi/2$, whichever has the longest perimeter.

5.2.3 Ir-lp of a Ring

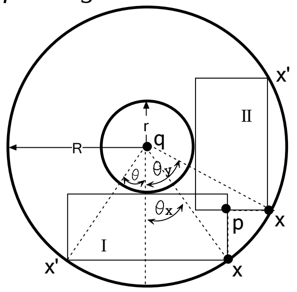


Figure 5.4: *Ir-lp* **of a Ring**

The *Ir-lp* must have one side tangent to the inner circle (with radius r) and two corners (denoted as x and x') on the outer circle (with radius R), as shown in Figure 5.4. For any p in the ring, there are two possible layouts for the *Ir-lp*: tangent to the inner circle horizontally as rectangle I or vertically as rectangle II. Let θ still denote the angle between $\overline{qx}$ and the y-axis. Here without loss of generality, we suppose p in the fourth quadrant from the origin q. In order to contain p in $p.sr_Q$, $\theta \in [\theta_x, \theta_y]$, where $\theta_x = arcsin\frac{p.x-q.x}{R}$ and $\theta_y = arccos\frac{q.y-p.y}{R}$. The perimeter of the first-layout *Ir-lp* is $4Rsin\theta + 2(Rcos\theta - r)$, and that of the second-layout *Ir-lp* is $4Rcos\theta + 2(Rsin\theta - r)$. The first-order derivative shows that for the first (second) layout, the perimeter has a maximum at $\theta = arctg2$ ($\theta = arcctg2$) and the value monotonously increases (or decreases) before (or after) this point. As such, we reach the following preposition:

PROPOSITION 5.5. *The Ir-lp of a ring centered at q with inner radius r and outer radius R is the one of the two Ir-lp which has a longer perimeter. The perimeter of the first horizontal Ir-lp is $4Rsin\theta_1 + 2(Rcos\theta_1 - r)$ where θ_1 is:*

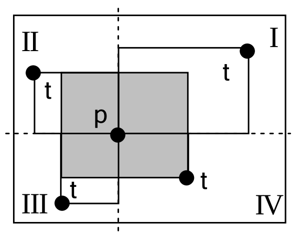
$$\theta_1 = \begin{cases} arctg2 & if\, \theta_x \le arctg2 \le \theta_y\,, or \\ \theta_x & if\, \theta_x < arctg2\,, or \\ \theta_y & if\, \theta_y > arctg2; \end{cases}$$

and the perimeter of the second vertical Ir-lp is $4Rcos\theta_2 + 2(Rsin\theta_2 - r)$ *where* θ_2 *is:*

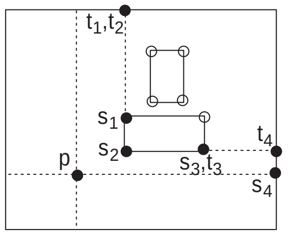
$$\theta_2 = \begin{cases} arcctg2 & if\, \theta_x \leq arcctg2 \leq \theta_y \, , or \\ \theta_x & if\, \theta_x < arcctg2\, , or \\ \theta_y & if\, \theta_y > arcctg2. \end{cases}$$

5.3 Safe Region for a Batch of Range Queries

So far we compute the safe region $p.sr_Q$ separately for each individual query Q. The final $p.sr$ is obtained by intersecting the safe regions for all queries. In this subsection, we devise an algorithm to compute the final safe region for all *range queries* in a single procedure, with the hope to maximize the perimeter of $p.sr$. Since for a range query whose quarantine area contains p, the best safe region is always the quarantine area, this algorithm is only concerns with range queries whose quarantine areas do not contain p. In other words, this algorithm attempts to find the *Ir-lp* of the complement of a set of query rectangles.



(a) Rectangular Union of Component Rectangles



(b) Find Opposite Corners for Component Rectangles

Figure 5.5: Safe Region for Batch Range Queries

With point p being the origin, the algorithm partitions the cell into four quadrants (see Figure 5.5(a)). By definition, the *Ir-lp* of the complement of a set of query rectangles is the rectangular union of four inscribed rectangles in each quadrant which do not overlap any query rectangle and one of whose corners is p. These inscribed rectangles are maximized in terms of their perimeters and are called *component rectangles*. In Figure 5.5(a), the white rectangles defined by p and t are component rectangles and the shaded rectangle is their rectangular union, i.e., the *Ir-lp*.

The first step is to find in each quadrant the set of component rectangles, or more specifically, the t set. Consider the first quadrant without loss of generality; the following proposition shows this problem is relevant to the dominating point problem.

PROPOSITION 5.6. *Consider the set of corners of the query rectangles that do not dominate* [5] *the other corners, plus the intersection point of the x-axis and the cell. Number these points* $(s_1, s_2, \cdots)$ *in increasing order of x-coordinate. The t set* $(\{t_1, t_2, \cdots\})$*, i.e., the opposite corners of the component rectangles, correspond to the s set* $(\{s_1, s_2, \cdots\})$*: each* t_i *has the same x-coordinate as* s_i*, but its y-coordinate is that of* s_{i-1} *or* t_{i-1} *if* t_i *and* t_{i-1} *have the same x-coordinate;* t_1*'s y-coordinate is set to the top bound of the cell.*

PROOF. First, any t_i computed by this proposition is the opposite corner for a component rectangle. It does not overlap any query rectangle because t_i is not dominating any other corners. It is maximized because t_i will dominate s_i if extended along the x-axis and it will dominate s_{i-1} if extended along y-axis. Second, any opposite corner for a component rectangle is a t_i. Otherwise, this corner could be extended along the x- or y-axis without dominating any s_i; thus, the component rectangle is not maximized, which contradicts its definition. □

Figure 5.5(b) illustrates an example with two range queries. While the hollow dots dominate some other corners, s_1, s_2, s_3 are not dominating. So together with s_4, they form the s set. The t set is obtained by setting $t_i.y$ to $s_{i-1}.y$. For t_1, its y-coordinate is set to the top bound of the cell. For t_2, since $t_1.x = t_2.x$, t_2 coincides with t_1 and hence can be omitted. Finally, $\{t_1, t_3, t_4\}$ is the set of opposite corners for the component rectangles. The rest quadrants can be handled by transforming the points to the first quadrant and then applying this proposition.

The next step is to find the best rectangular union of the component rectangles from the four quadrants. The optimal solution requires to enumerate every possible combination, which takes quartic time. As such, the algorithm applies a greedy heuristic. It sets the initial union as the cell and begins with the quadrant which has a component rectangle with the longest perimeter. The t_i of this rectangle trims the union by the horizontal and vertical lines crossing t_i. The algorithm then chooses the next quadrant in a clockwise order. In this quadrant, it greedily chooses the component rectangle which leads to a remaining union with the longest perimeter after being trimmed by its t_i. The algorithm continues until all the four quadrants are processed and the final remaining union is the resultant *Ir-lp*. Although it greedily chooses the component rectangle to maximize the remaining union, it makes only four greedy decisions, as opposed to the number of range queries if $p.sr_Q$ is computed individually. Therefore, the batch computing expectedly renders a safe region with a longer perimeter.

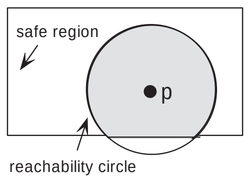
6. ENHANCEMENTS

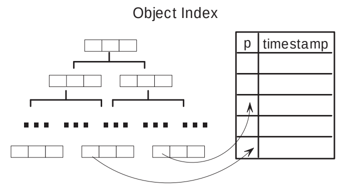
In the previous section, the computation of safe region is not based any mobility assumption. However, in practice the object always moves steadily towards its destination, in other words, it follows a rough direction for a significantly long period of time. This is called the *steady movement* assumption. Another practical assumption is that each object is limited by a maximum moving speed. In this section, we exploit these two assumptions to compute safe regions that could further reduce the number of location updates.

6.1 Maximum Speed for Query Evaluation

In general, a larger safe region with a longer perimeter reduces the number of source-initiated updates, but it might incur more probes during query evaluation/reevaluation. To compensate this side effect, we propose to bound the object location with an additional *reachability circle* (see Figure 6.1(a)). The circle is centered at the object's last reported location p_{lst} and is ever expanding by the rate of V, the maximum speed. The reachability circle shows how far the object can reach at any moment after the last location update. During query evaluation/reevaluation, the circle is

[5] Point a dominates point b iff $a.x > b.x$ and $a.y > b.y$.





(a) Reachability Circle (b) Augmented Object Index

Figure 6.1: Maximum Speed for Query Processing

the constructed before a location probe is issued, with the hope to resolve the ambiguity without probing.

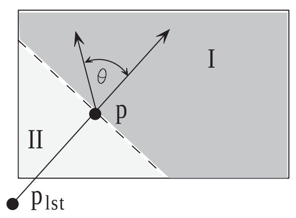
To compute the reachability circle, the object index stores both the last updated position p_{lst} and the timestamp of the update T. These data are pointed by leaf entries (see Figure 6.1(b)). At any moment t, the reachability circle is centered at p_{lst} and with radius $V(t - T)$.

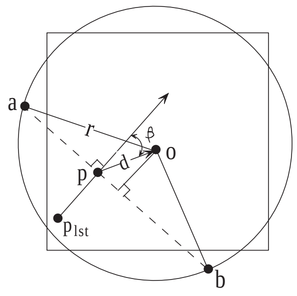
6.2 Steady Movement for Safe Region Computation

The average elapsed time before the next location update can be generally formulated as $\int_0^{2\pi} \frac{k(\theta)p(\theta)d\theta}{\phi}$, where $p(\theta)$ is the probability density function of θ, the moving direction after the location update, $\int_0^{2\pi} p(\theta) = 1$, and the other notations follow the same meanings as in Theorem 5.1. In that theorem, by assuming a random moving direction after the update, $p(\theta)$ is uniformly distributed, i.e., $p(\theta) = 1/2\pi$. However, the steady movement assumption reveals that the object tends to follow a similar θ to the one it followed when moving from p_{lst}, the last updated location, to p. In other words, $p(\theta)$ is practically not uniformly distributed. Thus, we propose a simple density function as follows:

$$p(\theta) = \begin{cases} \frac{1+D}{2\pi} & if\, \theta \in [-\frac{\pi}{2}, \frac{\pi}{2}], or \\ \frac{1-D}{2\pi} & otherwise, \end{cases}$$

where θ is the angle between the moving direction and $\overline{p_{lst}p}$ (see Figure 6.2(a)) and $0 \le D \le 1$ is a parameter of *steadiness*. Under this density function, the integration is pro-





(a) A Simple Density Function (b) A Fast Algorithm for Perimeter Conversion

Figure 6.2: Steady Movement for Safe Region

portional to the weighted sum of the two partial perimeters segmented by the normal (the dotted line in Figure 6.2(a)) of $p_{lst}p$, where the weights of parts I and II are $1 + D$ and $1-D$. We use this *weighted perimeter* instead of the ordinary perimeter for the *Ir-lp* derivation in Section 5. The following is a fast approximation algorithm that does this ordinary-to-weighted transition. Suppose λ is the ordinary perimeter, o is the center of the rectangle, d is the length of $\overline{po}$, and β is the angle between $\overline{po}$ and $\overline{p_{lst}p}$ (see Figure 6.2(b)). We draw a circle $\mathcal{C}$ with the same perimeter (i.e., with radius $r = \lambda/2\pi$). $\mathcal{C}$ is used to approximate the weighted perimeter because when $p = o$ its perimeter equals to the weighted perimeter for any $p(\theta)$ distribution. Let a, b be the intersection points of the normal and $\mathcal{C}$. Angle $\overline{aob} = 2arccos\frac{dcos\beta}{r}$. Therefore, the weighted perimeter λ_w is

$$\lambda_w = (1 + D)\lambda - \frac{2D\lambda}{\pi} arccos\frac{2\pi dcos\beta}{\lambda}.$$

The above formula cannot be directly applied to derive all *Ir-lp* in Section 5 because the optimal θ may not have a closed form under the weighted perimeter definition. For such cases, we propose a binary search strategy to find a suboptimal θ as follows. At each step, it keeps the current search range for θ as $[\theta_b, \theta_e]$. Let $\theta_c = (\theta_b + \theta_e)/2$. Then it drops one of these three values of θ which yields the shortest λ_w. The remaining two values form the search range for the next step. The search terminates if the maximum number of steps are reached or the next search range is the same as the current one. And θ_c in the last step is the suboptimal θ.

7. PERFORMANCE EVALUATION

We have developed a discrete event-driven simulator to evaluate the performance of the proposed safe-region-based query monitoring (denoted as SRB) framework. We compare it with two other schemes, i.e., the *optimal monitoring* (OPT) and the *periodic monitoring* (PRD). In optimal monitoring, every mobile client has the perfect knowledge of the registered queries and the movements of other objects. Therefore, the client knows precisely when its movement changes the results of some queries, and only then does it send a source-initiated location update to the server. Although not feasible in practice, this scheme serves as a lower bound for the number of location updates and as the yardstick from which we measure the monitoring accuracy of other schemes. In periodic monitoring, all clients periodically send out source-initiated updates simultaneously and the server reevaluates all registered queries based on these updates. Obviously, its performance depends on the updating interval t_{prd}. We test PRD with $t_{prd} = 0.1$ and $t_{prd} = 1$, denoted as PRD(0.1) and PRD(1) hereafter. In the following, we first describe the simulation setup and then present the detailed simulation results.

7.1 Simulation Setup

We simulate a mobile environment where N objects move within a unit-square space [0..1, 0..1]. Each object moves according to the random waypoint mobility model: the client chooses a random point in the space as its destination and moves to it at a speed randomly selected from the range $[0, 2\overline{v}]$; upon arrival or expiration of a *constant movement period* (randomly picked from the range $[0, 2\overline{t_v}]$), it chooses a new destination and repeats the same process. This is a well accepted and studied model in the mobile computing literature [4].

The query workload consists of W queries, of which half are range queries and half are order-sensitive kNN queries. For range queries, the query rectangle is a square and its side length is uniformly distributed in a range of $[0.5q_{len}$,

Parameter	Default Value	Parameter	Default Value
N	100,000 objects	W	1,000 queries
$\overline{v}$	0.01 per time unit	$\overline{t_v}$	0.005 time unit
q_{len}	0.005	k_{max}	10
t_{prd}	1, 0.1 time unit	M	50

Table 7.1: Simulation Parameter Settings

$1.5q_{len}]$. For kNN queries, the query points are randomly distributed within the space and k ranges from 1 to k_{max}.

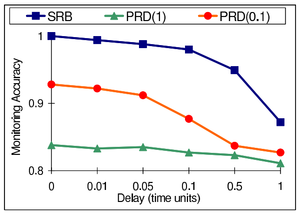
All the behaviors of the N clients are simulated by one process on a Pentium 4 1.8GHz 512MB desktop PC. The database server is implemented on a Pentium 4 2.4GHz PC running WinXP and IIS 5.0. The client process and the server communicate through the standard SOAP/HTTP protocol. The database server maintains an in-memory grid-based ($M \times M$) query index and an R*-tree index [2] on the safe regions (for SRB) or on the object positions (for PRD). To facilitate frequent location updates, we adopt the bottom-up update technique [15] for the R*-tree index. Table 7.1 summarizes the default parameter settings in our simulation.

To remove the effect of hardware configuration, we use logic time units instead of clock time. Each simulation run lasts for 5,000 time units or until the measured value stabilizes (for those simulations take longer than 12 hours). The performance metrics for comparison include:

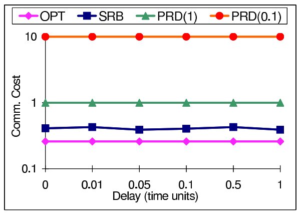
- **Monitoring accuracy:** This is defined as the amortized accuracy of monitored results against its real results over time. More specifically, given a query Q, let $R(Q,t)$ denote the monitored result set at time t and $\mathcal{R}(Q,t)$ denote the real result set monitored by the OPT scheme. The monitoring accuracy is thus defined as $\frac{1}{t_e - t_b}\int_{t_b}^{t_e} ma(Q,t)dt$, where $[t_b, t_e]$ is the monitoring time period of Q, $ma(Q,t) = 1$ if $R(Q,t) = \mathcal{R}(Q,t)$, and $ma(Q,t) = 0$ otherwise.
- **Wireless communication cost:** It is the amortized communication cost for a mobile client to send updates and receive probes in one time unit. We set the uplink channel twice as costly as the downlink channel. Thus, the cost for a source-initiated update $C_l = 1$ and the cost for a server-initiated probe plus update $C_p = 1.5$.
- **Scalability:** This is measured by the amount of CPU time used by the server to monitor the queries per time unit, which includes query evaluation and safe region computation.

7.2 Impact of Communication Delay

The first set of experiments evaluates the impact of communication delay between the client and the server. We have the server receive the location update τ time units after the client sends it. We vary τ from 0 to 1 time unit and plot the monitoring accuracy of the three schemes in Figure 7.1(a). The $\tau = 0$ case reflects the ideal networking environment with zero delay, based on which our SRB framework achieves a 100% accuracy. Even in this case, however, the PRD scheme family only gets a 80%~90% accuracy. As τ increases, all three schemes degrade. PRD(1) seems to be stable under different τ settings simply because it already delays result changes by 0.5 time unit on average due to periodic location updating; the accuracy of PRD(0.1) degrades quickly and close to PRD(1) for a large τ. On the other hand, SRB is stable at 95% even when $\tau = 0.5$, as



(a) Monitoring Accuracy



(b) Communication Cost

Figure 7.1: The Impact of Communication Delay

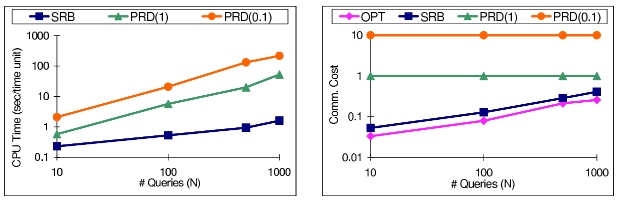
many result changes are detected immediately by using the safe regions.However, as τ keeps increasing, SRB starts to degrade. This is because when the client receives an updated safe region it might have already left the region due to a significant communication delay; the monitoring result is not correct until it updates the location again, which takes another τ time unit. Figure 7.1(b) shows the communication cost of the four schemes under various τ settings. We find that all these schemes show a constant or almost constant cost regardless of the setting of τ.[6] SRB is the closest to the optimal scheme (about 30% higher) whereas the cost of PRD(1) is more than twice higher. The cost of PRD(0.1) is 10, which is even higher. From Figures 7.1(a) and 7.1(b), we conclude that SRB achieves a much higher monitoring accuracy yet with a much lower wireless communication cost than PRD.

7.3 Scalability

This subsection evaluates the scalability of monitoring schemes in terms of server's CPU time and communication cost. Figure 7.2 shows the performance trend when the number of registered queries (W) increases from 10 to 1,000. The CPU time of SRB is sublinear to the increase of W because the grid-based query index filters out a large number of unaffected and irrelevant queries when processing location updates, whereas those of PRD(1) and PRD(0.1) are both linear, as they need to reevaluate every query on each batch of location updates. As an example, the server needs 1.6 CPU seconds to monitor 1,000 queries on 100,000 moving objects for one time unit using SRB, 53 seconds using PRD(1), and 217 seconds using PRD(0.1). The CPU time limits the maximum update frequency of PRD schemes. As in this example, the server cannot afford the PRD(0.1) scheme if one time unit is less than 217 seconds (i.e., update at most once every 21.7 seconds). Nonetheless, SRB has no such limitations. In terms of communication cost, SRB also increases with W as the safe regions shrinks. Nonetheless, the cost increment is sublinear since only a small and sublinear proportion of the queries contributes to the safe region of a specific object. The same reason also explains why SRB is always close to OPT. We also measured the size of the grid-based query index; it increases with W but never exceeds 300KB even when $W = 1,000$.

Similarly, we conduct simulations by varying the number of objects (N) from 100 to 100,000. Figure 7.3(a) shows that the CPU cost increases sublinearly with N, since the queries are evaluated on the R*-tree index which is incrementally maintained. By contrast, PRD(1) and PRD(0.1) both in-

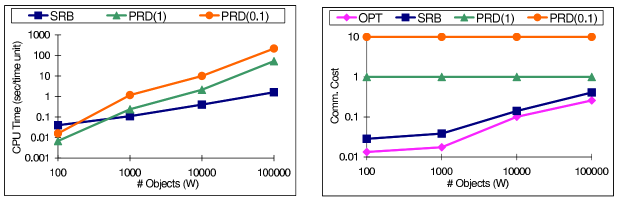
[6]In fact, the monitoring accuracy is the only metric affected by τ. Therefore, in the sequel we consistently use $\tau = 0$ to measure other metrics.



(a) CPU Time (b) Communication Cost

Figure 7.2: Performance vs. Query Numbers (W)

crease linearly or even hyperlinearly, as they need to build a new R*-tree for query reevaluation at each location updating instance. Figure 7.3(b) shows that the communication cost of SRB increases with N as a denser object distribution makes the safe regions shrink. Nonetheless, the increase is sublinear to N, because only a sublinear portion of the objects affects the quarantine area of a kNN query and thus the safe region of a specific object. In summary, SRB is more scalable than PRD in terms of CPU cost while achieving a close to optimal communication cost.



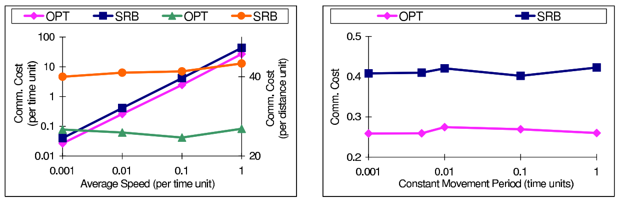
(a) CPU Time (b) Communication Cost

Figure 7.3: Performance vs. Object Numbers (N)

7.4 Sensitivity of SRB

In this subsection, we study the sensitivity of SRB to various affecting factors. The first two factors are the average moving speed ($\overline{v}$) and the average constant movement period ($\overline{t_v}$) for the moving objects. Figure 7.4(a) shows the communication cost when $\overline{v}$ varies from 0.001 to 1 per time unit. It increases linearly as $\overline{v}$ increases, mainly because the elapsed time of an object leaving its safe region is inversely proportional to $\overline{v}$. To eliminate this effect, we also dplot the communication cost per distance unit on the secondary y-axis in the same figure. We observe that this cost is independent of $\overline{v}$. In other words, the number of updates and probes when a client moving along a trajectory is not dependent on the speed, but dependent on the length of this trajectory. The CPU time shows a similar trend and hence is not plotted. We also vary $\overline{t_v}$ from 0.001 to 1 time unit and find it has hardly any effect on the performance of SRB (Figure 7.4(b)). As such, it is safe to conclude that SRB is robust to various moving speeds and steadiness degrees of of movement.

The next affecting factor is the $M \times M$ grid partitioning of the query index. We vary M from 5 to 100 and plot both the communication cost and CPU time in Figure 7.5. The larger is the value of M, the smaller is the grid cell size. The communication cost increases monotonously with M because the grid cell sets the largest possible safe region of an object. Nonetheless, the cost difference for between $M = 5$ and $M = 50$ is not significant as the actual safe regions are determined more by the *relevant queries* than by the grid cell. However, the cost increases sharply when



(a) $\overline{v}$ (b) $\overline{t_v}$

Figure 7.4: Communication Cost vs. Avg. Moving Speed $\overline{v}$ and Avg. Constant Movement Period $\overline{t_v}$

M is changed from 50 to 100, when the safe regions become more restrained by the cell. Meanwhile, the CPU time decreases monotonously because the number of relevant queries in the cell decreases and hence the safe region computation is faster. In this figure, $M = 50$ yields a fairly low communication cost as well as a fairly short CPU time. From this experiment, it is inspiring to adapt the cell size to the server's workload: we first use a large M to partition the grid, and later if the workload turns out to be low, the actual cell for the safe region computation can be the cell where the update occurs plus its neighboring cells within a certain distance. In this way, we can take full advantage of the CPU resource and enlarge the safe region as much as possible.

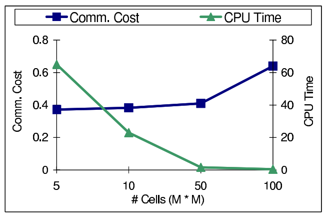
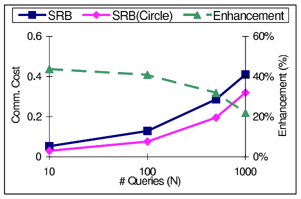


Figure 7.5: Performance vs. Grid Partitioning

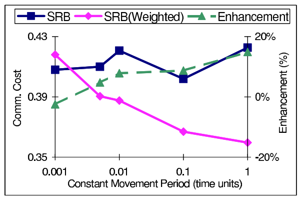
7.5 Performance of the Enhancements

We have proposed the use of reachability circle and weighted perimeter to enhance SRB under the maximum speed and steady movement assumptions in Section 6. We now evaluate their performance. In the first set of experiments, we vary the number of queries (W), which has an effect on the size of the safe region (i.e., the less the W, the larger the safe region size). As shown in Figure 7.6(a), the reachability circle enhances SRB by 20%∼40% (on the secondary y-axis) in terms of communication cost, thanks to a significant reduction of location probes during query evaluation/reevaluation. Nonetheless, the improvement decreases as W increases and the safe region shrinks, because at larger W (smaller safe regions), this ever-expanding circle covers the entire safe region (and becomes useless) more quickly. On the other hand, the enhancement is found to be independent of $\overline{v}$, due to the same reason for why the communication cost per distance unit is independent of $\overline{v}$.

In the second set of experiments, we vary the average constant movement period ($\overline{t_v}$) and plot the performance enhancement of weighted perimeter (with the steadiness parameter D set at 0.5) in Figure 7.6(b). Although the SRB with weighted perimeter is worse than the original SRB at $\overline{t_v} = 0.001$ when the movement speed and direction change rapidly, it outperforms the original SRB by 5%∼15% in all the other settings. As expected, the effectiveness of weighted perimeter is more distinct for larger $\overline{t_v}$ where the object



(a) Reachability Circle



(b) Weighted Perimeter

Figure 7.6: Performance Enhancements of Reachability Circle and Weighted Perimeter

moves in a more steady and predictable pattern. Therefore, it is inspiring to adapt D to $\overline{t_v}$, in other words, the server could adopt different D for different objects according to how frequently the objects change their moving directions.

8. CONCLUSIONS

This paper proposes a generic framework for monitoring continuous spatial queries over moving objects. The framework distinguishes itself from existing work by being the first to address the location update issue and to provide a common interface for monitoring mixed types of queries. Based on the notion of safe region, the location updates are query aware and thus the wireless communication and query reevaluation costs are significantly reduced. We provide detailed algorithms for query evaluation/reevaluation and safe region computation in this framework. Enhancements are also proposed to take advantage of two practical mobility assumptions: maximum speed and steady movement. To evaluate the performance, we thoroughly conduct a series of experiments and compare the proposed framework with the optimal monitoring and the traditional periodic monitoring schemes. The results show that i) our framework substantially outperforms the periodic monitoring in terms of monitoring accuracy and CPU time while achieves a close to optimal communication cost; ii) the framework scales well to the number of monitoring queries and moving objects; iii) the framework is robust to various affecting factors including moving speed, constant movement period, and grid partitioning; iv) if the mobility assumptions hold, the enhancements can reduce the wireless communication cost by up to 40%.

This paper demonstrates the feasibility and performance advantages of the framework. As for future work, we plan to incorporate other types of queries into the framework, such as spatial joins and aggregate queries. We also plan to optimize the performance of the framework. For example, so far the safe regions for kNN queries are computed separately; so the final safe region of the object $p.sr$ may not be optimal. To achieve a larger $p.sr$, we are going to develop algorithms which incrementally update the so-far computed $p.sr$ for each relevant query.

9. REFERENCES

[1] S. Babu and J. Widom. Continuous queries over data streams. In *Proc. SIGMOD*, 2001.

[2] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. SIGMOD*, pages 322–331, 1990.

[3] R. Benetis, C. S. Jensen, G. Karciauskas, and S. Saltenis. Nearest neighbor and reverse nearest neighbor queries for moving objects. In *Proc. IDEAS*, pages 44–53, 2002.

[4] J. Broch, D. A. Maltz, D. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proc. ACM/IEEE MobiCom*, pages 85–97, 1998.

[5] Y. Cai, K. A. Hua, and G. Cao. Processing range-monitoring queries on heterogeneous mobile objects. In *Proc. Mobile Data Management*, 2004.

[6] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. SIGMOD*, 2000.

[7] H. D. Chon, D. Agrawal, and A. E. Abbadi. Range and kNN query processing for moving objects in grid model. *ACM/Kluwer MONET*, 8(4):401–412, 2003.

[8] B. Gedik and L. Liu. MobiEyes: Distributed processing of continuously moving queries on moving objects in a mobile system. In *Proc. EDBT*, 2004.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proc. SIGMOD*, 1984.

[10] G. R. Hjaltason and H. Samet. Distance browsing in spatial databases. *TODS*, 24(2):265–318, 1999.

[11] G. S. Iwerks, H. Samet, and K. Smith. Continuous k-nearest neighbor queries for continuously moving points with updates. In *Proc. VLDB*, 2003.

[12] G. S. Iwerks, H. Samet, and K. Smith. Maintenance of spatial semijoin queries on moving points. In *Proc. VLDB*, 2004.

[13] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-tree based indexing of moving objects. In *Proc. VLDB*, pages 768–779, 2004.

[14] D. V. Kalashnikov, S. Prabhakar, and S. E. Hambrusch. Main memory evaluation of monitoring queries over moving objects. *Distributed Parallel Databases (DPDB)*, 15(2):117–135, 2004.

[15] M.-L. Lee, W. Hsu, C. S. Jensen, B. Cui, and K. L. Teo. Supporting frequent updates in R-trees: A bottom-up approach. In *VLDB Conference, Berlin, Germany*, pages 608–619, 2003.

[16] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proc. USENIX OSDI*, 2002.

[17] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proc. SIGMOD*, 2004.

[18] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An efficient index for predicted trajectories. In *Proc. SIGMOD*, 2004.

[19] 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. on Computers*, 51(10), 2002.

[20] K. Raptopoulou, A. Papadopoulos, and Y. Manolopoulos. Fast nearest-neighbor query processing in moving object databases. *GeoInfomatica*, 7(2):113–137, 2003.

[21] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *Proc. SIGMOD*, 1995.

[22] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *Proc. SIGMOD*, pages 331–342, 2000.

[23] Y. Tao, C. Faloutsos, D. Papadias, and B. Liu. Prediction and indexing of moving objects with unknown motion patterns. In *Proc. SIGMOD*, 2004.

[24] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proc. VLDB*, 2002.

[25] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*, 2003.

[26] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proc. ICDE*, 2005.

[27] J. Xu, X. Tang, and D. L. Lee. Performance analysis of location-dependent cache invalidation schemes for mobile environments. *TKDE*, 15(2):474–488, 2003.

[28] X. Yu, K. Q. Pu, and N. Koudas. Monitoring k-nearest neighbor queries over moving objects. In *Proc. ICDE*, 2005.

[29] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *Proc. SIGMOD*, 2003.