# Efficient Batch Processing of Proximity Queries by Optimized Probing

Seyed Jalal Kazemitabar
Computer Science Department, University of Southern California
kazemita@usc.edu

Farnoush Banaei-Kashani
Computer Science Department, University of Southern California
banaeika@usc.edu

Seyed Jalil Kazemitabar
Economics Department, University of California Berkeley
kazemita@econ.berkeley.edu

Dennis McLeod
Computer Science Department, University of Southern California
mcleod@usc.edu

## ABSTRACT

Many location-based applications are enabled by handling numerous moving queries over mobile objects. Efficient processing of such queries mainly relies on effective probing, i.e., polling the objects to obtain their current locations (required for processing the queries). With effective probing, one can monitor the current location of the objects with sufficient accuracy for the existing queries, by striking a balance between communication cost of probing and accuracy of the knowledge about current location of the objects. In this paper, we focus on location-based applications that reduce to processing a large set of proximity monitoring queries simultaneously, where each query continuously monitors if a pair of objects are within a certain predefined distance. Accordingly, we propose an effective object probing solution for efficient processing of proximity monitoring queries. In particular, with our proposed solution for the first time we formulate optimal probing as a batch processing problem and propose a method to prioritize probing the objects such that the total number of probes required to answer all queries is minimized. Our extensive experiments demonstrate the efficiency of our proposed solution for a wide range of applications involving up to hundreds of millions of queries.

## Categories and Subject Descriptors

H.2.8 [**DATABASE MANAGEMENT**]: Database Applications—*Spatial databases and GIS*; H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Query processing*

## General Terms

DESIGN

## 1. INTRODUCTION

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. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
*SIGSPATIAL* '13, November 5-8, 2013. Orlando, Florida, USA
Copyright 2013 ACM 978-1-4503-2521-9/11/13 ...$15.00.

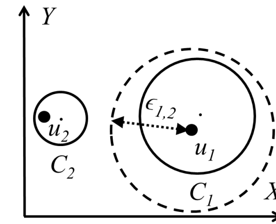


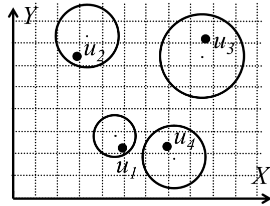

Figure 1: Two objects involved in a proximity query.

One of the required steps toward answering continuous location-based queries over moving objects is to obtain current (approximate and/or exact) locations of the objects from the moving objects. A well-known instance of such location-based queries is *proximity detection query* (or *proximity query*, for short), which continuously monitors the proximity of two given moving objects in order to determine whether their distance is within a certain desired threshold at each time instant $t$. A proximity query notifies two users that share a common interest about their being physically close. The users might or might not know each other beforehand. Proximity queries are used, for example, in large-scale location-based social networking applications (such as foursquare™) to find the user's friends that currently happen to be in proximity of the user; to continuously identify and recommend suppliers of the items that a user may demand as she moves through an extensive urban area (e.g., the city of Los Angeles); and to recognize teammates or opponents of a player who reside in her proximity in massively multi-player online games (MMOGs) with thousands of users (e.g., PlanetSide-2™).

Figure 1 illustrates the need for obtaining approximate and (some times) exact locations of the moving objects to answer proximity queries. In this figure, $u_1$ and $u_2$ (black filled circles) are two query objects shown at a certain time instant $t$, $\epsilon_{1,2}$ is the proximity threshold between $u_1$ and $u_2$ (visualized as a circle with dashed line), and $C_1$ and $C_2$ are the bounding circles for $u_1$ and $u_2$, depicting the current approximate location of the two objects, respectively. In a generic client-server proximity query processing system, the query processor (often a central server with all objects as its clients) is aware of the approximate location of each object (i.e.,

| Query | Pair of objects | $\epsilon_{\mathbf{i,j}}$ |
|---|---|---|
| $q_1$ | $\langle u_1, u_2 \rangle$ | 4 |
| $q_2$ | $\langle u_1, u_3 \rangle$ | 4 |
| $q_3$ | $\langle u_1, u_4 \rangle$ | 4 |
| $q_4$ | $\langle u_3, u_4 \rangle$ | 4 |

(a) Proximity queries




(b) Constellation of objects

Figure 2: A batch of proximity queries. All queries can be answered by probing $u_1, u_3$.

$C_1$ and $C_2$ in Figure 1), and whenever a moving object leaves its approximate location bound, the object sends a so-called *location-update* to the query processor to update its current approximate location. While in some cases knowing the approximate location of the objects is sufficient for the query processor to answer a proximity query at the corresponding time $t$, in many cases the query processor needs to further obtain the current exact location of one or both of the objects to be able to answer the query. For example, in Figure 1, knowing the approximate locations $C_1$ and $C_2$ of the two objects is insufficient for answering the query, because depending on where exactly $u_1$ and $u_2$ are located in their bounding circles, their actual distance might be less or more than the threshold $\epsilon_{1,2}$. In this case, the query processor can obtain the exact location of $u_1$ by so-called *probing* the object $u_1$. As shown in the figure, knowing the exact location of $u_1$ is sufficient for the query processor to deduce that $u_1$ and $u_2$ are not in the $\epsilon_{1,2}$ proximity of each other at time $t$.

Obtaining the location of the moving objects is the most frequent procedure performed to process proximity queries, resulting in a major communication overhead which hinders developing scalable proximity query processing systems. Accordingly, in the literature a variety of communication-efficient proximity query answering solutions have been proposed to reduce the communication cost of obtaining object locations (see Section 2 for a review of these solutions). However, these solutions have almost unanimously focused on reducing the cost of "location-update" and have mainly ignored the cost of "probing". This is true despite the fact that these two operations are often both performed very frequently, while the cost of each probe is twice that of a location-update (because it involves sending an additional request from server to the client/object).

In this paper, we focus on developing an optimized probing technique for efficient proximity query answering. Our query answering solution supports numerous queries at the scale of the aforementioned applications. To the best of our knowledge, this work is the first to consider probe optimization towards communication-efficient proximity query answering. It is important to note that our proposed technique is complementary to the existing solutions that focus on the orthogonal problem of location-update optimization.

The main idea behind our proposed probe optimization technique is considering proximity queries as a *batch* of queries (rather than individually). With this batch processing approach, we can derive the minimum set of probes needed to resolve all queries in the batch, which often count much less than the total number of probes required if queries are considered individually. To realize the benefit of batch processing, one should observe the fact that with typical applications each object is often involved in several proximity queries. To further elaborate, see Figure 2 that illustrates a typical scenario in which none of the proximity queries $q_1$ to $q_4$ can be answered merely based on the approximate location information, and hence, probing is required to resolve the queries. The current state-of-the-art proximity query answering solutions address each query individually and end up probing all involved objects $u_1$ to $u_4$ (i.e., four probes) to resolve the queries. However, one can observe that in this case probing only $u_1$ and $u_3$ (i.e., only two probes) is sufficient to answer all queries.

However, deriving the minimum set of probes needed to answer a batch of queries (possibly containing millions of queries) is challenging. With a naive approach, one may attempt to consider all possible combinations/subsets of potential probes and determine the smallest subset that resolves all queries. However, such a naive approach is not practical considering 1) the tentatively large number of queries in a batch, which leads to humongous number of possible probe combinations, and 2) the real-time requirements of *continuous* proximity query processing, which allows only one time slot for processing of a batch of queries at each time instant $t$.

To address this challenge, we propose a two-phase probe optimization technique that efficiently computes the near-minimal subset of probes which can resolve a given batch of queries. At the first phase, considering the proximity threshold and approximate object locations of each proximity query, we categorize every query in the batch to determine whether each query requires probing, and if so, identify the probe(s) that with highest probability can resolve the query (if queries are considered individually). Successively, at the second phase given the categorization of the queries from the first phase and considering the interdependencies between queries given the shared query objects among them, we use a systematic probe selection approach that intelligently chooses an order of probes that minimizes the total number of probes required to resolve all queries in the batch. Our probe optimization technique is parallelizable, and with extensive experiments we show that it scales well to support hundreds of millions of queries, while on average it incurs 30% less communication cost as compared to the best existing proximity query answering technique. This is a considerable improvement since out of the two users involved in an unanswered query, at least one of them needs to be probed in order to solve the query.

The paper is organized as follows. Section 2 reviews existing algorithms. The assumed system architecture and proximity query processing algorithm are described in Section 3. Section 4 defines the probing problem and presents an overview of our approach. Section 5 covers Phase I where queries are classified and queries with a straightforward optimum probing decision are discovered. Section 6 explains Phase II, where the probing problem over a batch of unanswered queries is modelled as a decision process and an optimal solution as well as a fast near-optimal solution are provided. We also briefly explain the optimal solution which is computationally complex. Experimental results are presented in Section 7 and the paper is concluded in Section 8.

# 2. RELATED WORK

## 2.1 Continuous Range Query

Each proximity detection query can be mapped to a corresponding continuous range query. However, that is a perspective we would like to avoid. Each proximity query, as we will define shortly, has a customized range ($\epsilon-$neighborhood) which together with the two involved objects makes each query unique. Following the paradigm of a range query to answer a single proximity query asks for extra computation to find all matching objects initially, and then filtering out all but at most one of the matching results, making it computationally unjustifiable. While applications generally include a large number of queries, this approach seems not to scale even to a fair number of queries. On the other hand, using our

proximity detection method, hundreds of millions of queries run in a minute on a single computer[1].

SINA [12] utilizes a hash-based approach to incrementally process moving range queries over moving objects. The method focuses on performance aspects and relies on frequent messages sent by objects regarding their exact location, hence not focusing on communication cost. [9] processes static range queries efficiently by computing *safe regions* that minimize location updates. Query results do not change as long as every object stays in its associated safe area. [7] utilizes the safe region concept to process moving range queries. The primary focus there is on computation-efficient processing of static rather than moving objects. None of these work focus on minimizing the communication cost for processing moving range queries over moving objects which –despite the described computational overhead for adaptation– is the closest spatial query to the one we address here.

## 2.2 Proximity Detection Query

**Preamble:** Processing spatial queries over moving objects requires a location tracking policy to locate objects. Location information is transmitted either through a source initiated message, i.e., a location update, or a destination (e.g., server) initiated message, i.e., a probe. Existing proximity detection methods focus and differ on how they send source-initiated messages. As such, we first explain different location update policies and then review current approaches to solve proximity queries. We finally explain how our work fits in the literature.

Various location update policies can be used to help track objects: periodic, distance-based, zone-based position update, and dead reckoning [8, 13]. The first two policies require the object to transmit location update messages at fixed time intervals or upon traversing fixed distance blocks. Such methods are naive and barely communication-efficient since real-world objects follow a dynamic rather than uniform movement pattern. Dead reckoning and subsequent proximity detection methods such as [15] require additional information on the movement patterns of objects. Real movement parameters are typically not available and should be replaced by predefined values, resulting in inefficient communication [18]. Most proximity detection methods, including current state-of-art methods, utilize the zone-based position update policy in which an object notifies the server once it leaves a geographic zone. Such a zone (or region) is defined with regards to the queries and can be either static or moving. For the rest of this section, we explain proximity detection methods that use static and moving geographic zones to track objects.

The proximity detection query was first introduced in [4] where objects actively process mutual queries and communicate in a distributed environment. For each pair involved in a query, a *strip* of width $\epsilon$ divides the space into two static half pages as safe regions. A query result remains unchanged as long as both objects stay in their safe regions. Each object keeps track of all its associated half pages and informs the corresponding object if it leaves a half page. Once informed, the other object then processes the query and sends back a new strip defining new safe regions for the two objects.

Utilizing a distributed architecture does not scale to a fair number of objects and queries as an object needs to multicast its updated location information to fellow objects when it leaves multiple safe regions in a single move, resulting in huge growth in communication cost. In contrast, using a query processing server prevents excessive communication as only the central server and not peer objects need to receive updated location information.

Some proximity detection methods have used static regions together with a client-server architecture to locate objects approximately. In [14], a *dynamic centered circle* surrounds each object to serve as its safe region. Circles for two objects involved in a query are guaranteed to be farther than the associated proximity distance threshold. In a more recent method, the server divides the space into static grid cells [16]. Objects update the server when they leave a cell, helping the server answer many queries based on an object's current cell and not its exact location. All objects involved in unanswered queries will be probed and the grid size would be adaptively updated to reduce server processing time. The method is further extended to road networks in [17]. Keeping static approximate regions for moving objects results in frequent location updates sent by dynamic objects and is not communication-efficient [8].

Moving geographic zones have been recently used to track the location of objects. [8] proposes *vector-based update policy* in which an object moving in a road network is encompassed in a moving circle and propagates a location update message when it leaves this region. Reactive Mobile Detection (RMD), as the current state-of-the-art proximity detection method, utilizes the aforementioned policy and automatically tunes the size of moving regions to reduce the communication cost while processing proximity queries [18]. Specifically, it reacts to a location update message by increasing the radius of the moving region according to a scale factor. Similarly, it reacts to a probe message by decreasing the radius.

Our proximity detection method differentiates itself from the above work in two aspects. First, we focus on probing rather than location update to enable communication-efficient proximity detection. While a location update message is primarily initiated by the movement patterns of the moving object itself, deciding as to which objects to probe is within the control of the query processor and has potentials for optimization. Moreover, unlike a source-initiated location update, a server-initiated probe is always followed by another message from the object, causing more communication cost to both the server and the object, asking for special attention. As the second differentiating feature, while all existing work process queries individually, we consider them as a batch to further optimize probing.

# 3. PROXIMITY QUERY PROCESSING SYSTEM

In this section we provide an overview of our assumed query processing system. We start by defining the proximity query and then explain the assumed system architecture. We describe the query processing routine by first categorizing the transmitted messages and then providing optimizations towards communication-efficient query processing.

## 3.1 Proximity Query Definition

The proximity query $q_{ij}$ is the problem of detecting whether a pair of mobile objects $\langle u_i, u_j \rangle$ are in $\epsilon$-neighbourhood, i.e., closer to each other than the distance threshold $\epsilon_{ij}$. A proximity query is *satisfied* if the two involved objects are closer than $\epsilon_{ij}$ distance units and *unsatisfied* otherwise; resulting in a numeric value of 1 or 0 as the query result. We say that $u_i$ and $u_j$ are in a proximity relationship iff there exists a query $q_{ij}$. Without loss of generality, we assume that any proximity relationship is mutual, meaning that

$$q_{ij} = q_{ji} \equiv q_{i,j} \wedge \epsilon_{ij} = \epsilon_{ji} \equiv \epsilon_{i,j}$$

For brevity of notation, we use $\epsilon$ instead of $\epsilon_{i,j}$ when the pair of

[1] As a secondary difference with continuous range query, in current applications of proximity detection query each object is in relationship with a tiny subset of interest and not all moving objects.

objects are known from the context.

A set of proximity queries is called a *batch*, $\mathcal{Q}$. All the $m = |\mathcal{Q}|$ queries in a batch need to be processed periodically in *epochs*, i.e., intervals of length $\Delta T$. The sequence $\langle 0, \Delta T, 2\Delta T, \ldots \rangle$ shows the start times for consecutive epochs.

*Probing* a mobile object is defined as inquiring it for its exact mobility information. The target mobile object, aka *probee*, always replies back by providing its current location and velocity.

We assume that no prior knowledge is available on movement trajectories of objects. Also, the communication infrastructure guarantees lossless message transmission; meaning that messages issued by a sender are retrieved correctly on the receiver side.

There are $n$ registered objects in the system. Each object is aware of its mobility information. Specifically, $u_i$ knows its exact location $\langle u_i.X(t), u_i.Y(t)\rangle$. All objects measure their positions at the start of each epoch. The velocity of the object, $u_i.V(t)$, is calculated as the average speed between two subsequent positionings (i.e., epochs).

## 3.2 System Architecture and Query Processing

We assume a client-server architecture to address a set of proximity queries. Objects have their exact mobility information and would unveil it to the server if requested to do so. The server processes the set of all queries and informs the objects about the results. For each object, the server stores the corresponding registered queries and a region showing the approximate object location. In order for the query results to be correct, the server needs to process all queries in each epoch and decide their satisfiability. For privacy reasons it is not attractive to involve objects in query processing.

Efficient tracking of moving objects by the server calls for a tracking policy that has specific features. First, the server should maintain approximate rather than exact object locations. Although having exact object location makes query processing trivial, imposing such a requirement makes objects communicate their location very often. Second, similar to a mobile object, its associated region on the server needs to be moving as otherwise, the object moves out of the region frequently. This feature poses considering a separate customized region for each mobile object which by itself enables fine-tuning the region to mobility behaviour and proximity queries of that object.

Approximate information on object locations inserts uncertainty into query processing. Guaranteeing correct query processing in presence of such uncertainty requires objects to contribute and inform the server once they leave their approximate location; as otherwise, the server would follow an incorrect approximation and declare false query results.

Similar to [8, 18], we follow a vector-based update policy to track objects. As shown in Figure 3, $u_i$ is surrounded by a virtual *mobile region* $C_t(u_i)$ which is a moving circle centred at $c_t(u_i)$ with radius $\lambda_t(u_i)$. The circle is moving with a nominal velocity $V_t(u_i)$ which can be different from that of the object, $u_i.V(t)$. For brevity, we omit the time-stamp $t$ when referring to information in a snapshot (thus $\lambda_i$ and $C_i$ instead of $\lambda_t(u_i)$ and $C_t(u_i)$).

Both the object $u_i$ and the server are aware of $C(u_i)$ while regularly, only the object knows its own exact location and velocity. For privacy reasons, the server does not share the mobile region of $u_i$ with any other object (including those who are involved in a query with $u_i$).

Within our client-server architecture, there are three situations in which communication is required to correctly process proximity queries over moving objects. First, when an object leaves its mobile region, the server needs to correct the mobile region as it is no more a valid approximation. The object issues a (source initiated) *location update* message to the server. This message includes the exact location $\langle u_i.X, u_i.Y\rangle$ and velocity $u_i.V$ of the object as well as its id $i$. The server then refreshes $c(u_i)$ and $V(u_i)$ to mirror these values. This event costs one message per each object that leaves its mobile region.

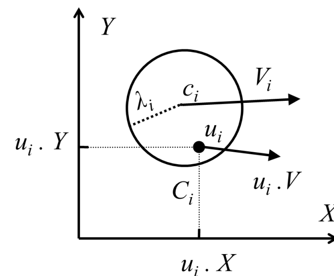



Figure 3: Mobile region for $u_i$ at time $t$.

Second, when the approximate location information of $u_i$ and $u_j$ is not sufficient to decide the satisfiability of $q_{i,j}$, the server makes an *intelligent* selection and probes one of the two objects for its exact location. The object then replies with its exact mobility information. In case the query cannot be solved solely by probing one object, the server probes the other one as well and answers the query based on the exact locations of both objects. We consider a cost of two for a probe message knowing that it always entails a message from the object side. This event costs two or four messages per each uncertain query. The cost though, can be aggregated across queries meaning that an object involved in multiple uncertain queries will still need to be probed and reply once to address all unanswered queries it is involved in. In this paper, we focus on probe optimization and propose an algorithm that selects probees by looking into the whole batch of queries rather than each individual query.

The third situation that initiates client-server communication happens when the server observes that a query result has changed. In this case it *notifies* both objects involved in the query. This event costs one message per each object per each updated query result.

Following [18], we fine-tune the size of mobile region to reduce the communication cost. The mobile region contracts and expands after each location update or probe message. We use a *scale factor*, $\alpha$, to customize this amount. Upon contraction of $C(u_i)$, $\lambda_i \leftarrow \frac{1}{2\alpha} \cdot \lambda_i$, and upon an expansion, $\lambda_i \leftarrow \alpha \cdot \lambda_i$. Both client and the server are aware of this rule and will automatically update the mobile region without having to communicate.

# 4. PROBLEM DEFINITION AND SOLUTION OVERVIEW

*Given a batch of proximity queries, our goal is to minimize the count of probees in each epoch, while continuously monitoring the queries and notifying the involved objects about any result updates.*

Objects measure their location at the start of each epoch and update the server if they leave their mobile regions. The server then analyses the whole set of queries and comes up with a decision as to what objects to probe next so that the expected number of probees for solving all queries is minimized. The server starts to probe selected objects until all queries are solved.

Each probe message initiated during query processing involves twice communication cost compared to a source initiated location update message. Moreover, as its name suggests, a source initiated message is mainly caused by movement patterns of the client, e.g.

because the client took an exit on the right in the highway. This is while an intelligent server-side algorithm can solve all queries with less probes. Therefore, probing deserves special attention and would be our main focus in this paper.

Our probing algorithm consists of two phases. Having a set of queries, a fundamental question is what subset of queries (and their corresponding objects) have a chance to affect the total number of probees. This question is answered in Phase I, where we identify and analyse all possible categories for a query in terms of mutual object locations. In fact, these categories altogether form the *state space* for any proximity detection query at any time. We discover categories that are solvable without probing any objects and further exclude them from the batch. All queries classified as other categories will be further processed in Phase II. One interesting category includes those queries for which an optimal probing decision can be made definitely rather than probabilistically. For such queries, probing a specific object first would have no contribution in solving the query. This conclusion is made without considering any object movement patterns or location distribution. There are also categories of queries for which no trivial optimal probe choice is available. Phase I is explained in Section 5.

Analysing each query individually as done in Phase I brings valuable insights into query processing. However, as illustrated in Figure 2, more optimized probing decisions can be made by considering all unanswered queries globally rather than individually.

Phase II deals with queries that might affect other queries in the batch; meaning that probing some objects involved in some of these queries earlier than some others *might* help reduce the number of total probees. Two questions rise here. First, what are the exact queries that have a tied destiny in terms of affecting each other's probing decision. Second, assuming a set of related queries whose answers are not known to the server, what is the best ordering to probe the involved objects so that the number of probees is minimized. These questions are addressed in Section 6.

Ideally, a probing strategy would start with simultaneously probing one of the two objects involved in every query since only the location of those two objects and no one else's is important to answer that query. This way, all queries would be answered in at most two probe iterations, leaving no space for an ordering on probees. However, in the absence of knowledge on exact object locations, there is a potential to make better probing decisions by interacting with the objects and dividing the original problem into smaller ones that include smaller number of queries, thus dealing with less uncertainty. While our algorithm enables making a one-time decision to order all objects and apply all probe decisions in two iterations, we prefer to follow a longer iterative probing process that fits the criteria of $\Delta T$ time units before the next epoch starts.

## 5. PHASE I: QUERY CATEGORIZATION

For any query $q_{i,j}$, we assume without loss of generality that $\lambda_i > \lambda_j$ meaning that the mobile region surrounding $u_i$ is larger than that of $u_j$. Figure 4 shows that Real line can be partitioned into five non-overlapping intervals. For any query at any time, the real valued distance $d = dist(c_i, c_j)$ settles along one of these intervals; therefore the query would be classified under the corresponding category. In this section we analyse each category in detail.

For each $u_i$, consider two important circles with radii $\epsilon - \lambda_i$ and $\epsilon + \lambda_i$ that have a common center $c_i$ as defined in Section 3.2. Let $I_i$ and $O_i$ denote the inner and outer areas of the smaller and the larger circle, respectively. In Figures 5 to 8, we illustrate these areas by gray color and accordingly they will be referred to as *gray*

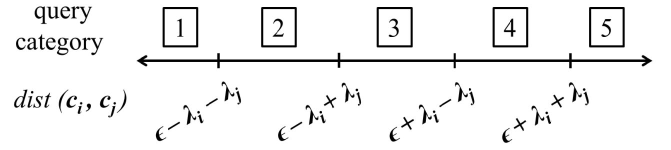
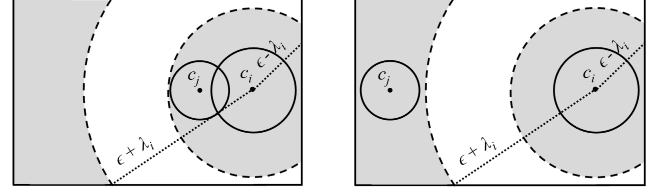



Figure 4: Partitions of the real line and their analogous categories for proximity query.



(a) Category 1: too close mobile regions.

(b) Category 5: too far mobile regions.

Figure 5: Solving queries based on the distance between their mobile regions. The status of a proximity query is determined if an object (e.g. $u_j$) is in the gray area formed by the other one (ie. $u_i$).

*areas* (related to $C_i$). What is in between, the annulus, is named the *blind area* (related to $C_i$). $I_i$ and $O_i$ have an interesting geometric characteristic as described in the following lemma:

LEMMA 1. *Let $I_i$ and $O_i$ denote the inner and outer gray areas of the smaller and the larger circle, respectively. The $\epsilon$-neighborhood of any point in $I_i$ ($O_i$) includes (excludes) the entire $C_i$. Hence if it occurs that another object $u_j$ falls in a corresponding gray area of $C_i$, the server can solve $q_{i,j}$ by merely probing $u_j$, deducing $q_{i,j}$ to be true if $u_j$ is located in $I_i$ and false if it happens to be in $O_i$. Conversely, when $u_j$ resides in the annulus white area of $C_i$, i.e., the blind area, having the location information of $u_i$ is a prerequisite to determine the status of the query.*

There are cases where the server can certainly solve $q_{i,j}$ solely based on the mobile regions $C(u_i)$ and $C(u_j)$ without requiring the exact object locations. Specifically, Figure 5 shows the two categories that share this property (Categories 1 and 5). All queries falling under these categories will be filtered in Phase I. The remaining subset of queries belong to Categories 2 to 4 and will be further analysed in Phase II. As we will see, a straightforward least cost probing decision cannot be made in Categories 2 and 4. The optimum decision for category 3 can be discovered definitely rather than probabilistically. However, taking the probing action is postponed until Phase II to further manage the effect of probe on the whole batch.

**Category 1:** $d < \epsilon - \lambda_i - \lambda_j$

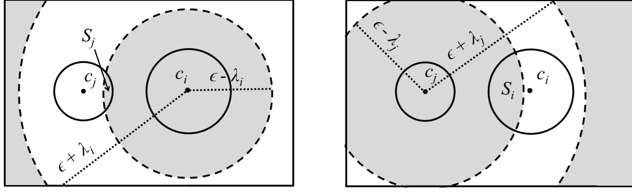
Figure 5a depicts the case where any pair of points in the two mobile regions are inside $\epsilon$-neighbourhood of each other. In other words, the maximum distance between any two points in the circles is smaller than the proximity threshold:

$$maxDist\big(C(u_i), C(u_j)\big) < \epsilon_{i,j}$$

In this case, $q_{i,j}$ is satisfied and the query result is respectively 1.

**Category 2:** $\epsilon - \lambda_i - \lambda_j < d < \epsilon - \lambda_i + \lambda_j$

Figure 6 shows the case where both objects have a chance to solely solve the query when probed and neither can be simply crossed out from probing choices. There are some points in $C(u_j)$ that are inside $\epsilon$-neighborhood of *all* points in $C(u_i)$ and vice versa (thus



(a) Only if $u_j$ is located in $S_j$, probing this object can solve the query.

(b) Only if $u_i$ is located in $S_i$, probing this object can solve the query.

Figure 6: There are chances the query is satisfied only by probing one object. $u_i$ has a higher chance to solve the query.

$q_{i,j} = 1$ for these points). These points together form the gray subarea $\mathcal{S}_j$, i.e., the intersection of $C(u_j)$ and the gray area of $u_i$ (Figure 6a).

If the object $u_j$ is in $S_j$, probing it would be sufficient to solve the query and there is no need to probe the other object. The probability of an object residing in this subarea can be quantified. Assuming a uniform distribution for the location of an object in its mobile region, $p_{ij}$, the probability of successfully solving query $q_{i,j}$ given that only $u_i$ is probed, can be calculated as:

$$p_{ij} = \frac{|\mathcal{S}_i|}{\pi\lambda_i^2}$$

where $|\mathcal{S}_i|$ represents the area for the gray subarea $\mathcal{S}_i$ in Figure 6b. However, $\lambda_i > \lambda_j$ entails that $p_{ij} > p_{ji}$ which means probing $u_i$ is more likely to solve the uncertain query $q_{i,j}$ than $u_j$. In other words, by hiding in the larger circle $C(u_i)$, $u_i$ inserts more ambiguity into the problem than does $u_j$ and disclosing that information can contribute more (in probability) to solve the query. Thus, higher chances are that probing $u_i$ will solve the query.

**Category 3:** $\epsilon - \lambda_i + \lambda_j < d < \epsilon + \lambda_i - \lambda_j$

Figure 7 shows the case where $C(u_j)$ completely sits in the blind area; meaning that no point in $C(u_j)$, including $\langle u_j.X, u_j.Y\rangle$, can solely solve the query $q_{i,j}$. More specifically, for any point $x \in C(u_j)$, there is a $y \in C(u_i)$ that is inside the $\epsilon$-neighborhood of $x$ and there is also a $z \in C(u_i)$ that is outside of it. On the other hand, $C(u_i)$ intersects both gray areas in Figure 7b, implying that there are some points in $C(u_i)$ that are inside the $\epsilon$-neighborhood of *all* points in $C(u_j)$ (thus $q_{i,j} = 1$) and some other points in $C(u_i)$ that are outside the $\epsilon$-neighborhood of *all* points in $C(u_j)$ (thus $q_{i,j} = 0$). Therefore, the server should definitely probe $u_i$ first since unlike for $u_j$, there is a positive chance of finding the answer immediately. The server might later need to probe the second object if the exact location of the first one cannot solve the query (thus, another definite decision).

**Category 4:** $\epsilon + \lambda_i - \lambda_j < d < \epsilon + \lambda_i + \lambda_j$

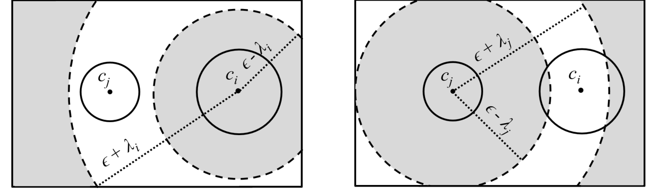
Similar to category 2, there is no definite probing strategy for this case. As shown in Figure 8, there are some points in $C(u_j)$ that are outside of $\epsilon-$neighbourhood of *all* points in $C(u_i)$ and vice versa (thus $q_{i,j} = 0$ for these points). Similar to our discussion under category 2, $\lambda_i > \lambda_j$ entails that $p_{ij} > p_{ji}$.

**Category 5:** $d > \epsilon + \lambda_i + \lambda_j$

Figure 5b represents the case where any pair of points in the two circles are outside $\epsilon$-neighbourhood of each other.In other words, the minimum distance between mobile regions is larger than the proximity threshold:

$$minDist\big((C(u_i), C(u_j)\big) > \epsilon_{i,j}$$

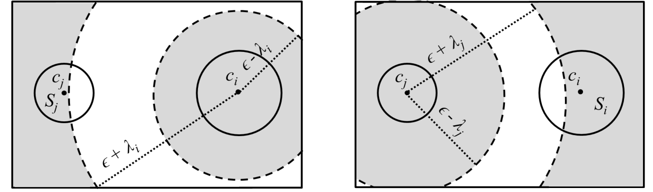
In this case, $q_{i,j}$ is unsatisfied and the query result is respectively



(a) $C_j$ completely falls in the blind area. Thus, probing $u_j$ cannot solely solve the query.

(b) Parts of $C_i$ fall under gray area. So there are chances that probing $u_i$ alone can solve the query.

Figure 7: $C_j$ completely falls in the blind area. $u_i$ must be probed first.



(a) Only if $u_j$ is located in $S_j$, probing this object can solve the query.

(b) Only if $u_i$ is located in $S_i$, probing this object can solve the query.

Figure 8: There are chances the query is unsatisfied only by probing one object. $u_i$ has a higher chance to solve the query.

0.

All queries in Figure 2 belong to Categories 2 and 4. The provided probabilistic metric in this section minimizes the probing cost for solving an individual query. However, our main objective of minimizing the aggregated probing cost among all queries still remains unsolved as the analysis is missing a broad perspective on the whole batch of queries. In the next section we provide optimized probing algorithms for processing queries in Categories 2 to 4.

# 6. PHASE II: PROBE ORDERING

Solving any of the queries forwarded to this phase requires probing one or both involved objects. Given a batch of unanswered queries in an epoch, our algorithm suggests an ordering for probing objects. We first model the received batch of queries as a graph in Section 6.1. Next in Section 6.2, we discover related queries by partitioning the batch (i.e., the graph) into subsets where probing objects in each subset affects the total number of probees only in the affiliated subset. Subsets are independently processed and new probees are introduced iteratively by applying our algorithm to each individual subset.

In Section 6.3, we suggest two approaches for selecting objects as probees in each iteration. We first briefly describe a baseline approach that yields to the "optimal solution" which we then leave for its high computational complexity. We then sacrifice the optimality of that approach in favour of the performance gained by a greedy alternative. In our Enhanced Probe Selection method, we avoid complexity by assigning each object a value and picking the most valuable objects in order to reduce the probe cost (Section 6.3).

## 6.1 Data Structure

In order to capture how queries in Phase II affect each other, we

need to maintain a global view on the whole batch of unanswered queries. We model the batch as a graph $\Gamma = \langle \mathcal{U}, E, W \rangle$, where $\mathcal{U} = \{u_1, \ldots, u_n\}$ represents the set of $n$ mobile objects as nodes. An unanswered query $q_{i,j}$, is represented by an edge $e_{i,j} \in E$ between $u_i$ and $u_j$. $W : \mathcal{U} \mapsto \{0, 1\}$ tags each node with a binary value showing whether the server has decided to probe the object, i.e., a *must-probe* node, or this is not yet decided, i.e., a regular node of the graph.

## 6.2 Probe Ordering Algorithm

Algorithm 1 depicts our batch proximity query processing algorithm run in each epoch. The server first receives the location update messages from objects. It then classifies queries in Phase I and filters out those that can be solved without issuing any probe messages. Next in Phase II, the yet unanswered set of queries is modelled as a graph $\Gamma$.

Deciding to probe objects in some queries might affect the probing decision for objects in some other queries. One important question is what exact subset of queries have a potential to affect each other in terms of probing the involved objects. After probing an object, there might be affiliated queries that remain unsolved, mandating to probe other fellow objects, hence introducing them as new must-probe objects. Consecutive occurrence of this situation is what we call the *cascade effect*. In order to find the exact group of queries with tied destinies, we start from an arbitrary query and imagine the most extensive cascade. A cascade effect can cover a connected component of the graph but not beyond that.

Recognizing the connected components of (a large) $\Gamma$ reduces the probe decision problem to selecting from nodes of a (smaller) single component. This enables more optimized probing decision since all unrelated objects are excluded from available decision choices. Moreover, concurrent probes in different components logistically conform to the limitations present in real-world applications: considering the network turnaround time, a fully sequential probing scenario involving a large number of objects is not operational within a short interval bounded by $\Delta T$ time units.

We now explain Algorithm 2, our iterative algorithm to address unanswered queries in one connected component. At each moment, the server needs to make a decision to select and probe objects from a component. Decision making is either trivial or non-trivial depending on the tags of nodes in the connected component.

Given a connected component, the server needs to decide which node to probe first. This is a non-trivial decision as probing a node has consequences: it might not solve some affiliated queries and also affect what nodes have to be probed next. We suggest two approaches in Section 6.3 that addresses the probe selection problem. After probing a node $u_i$, if an affiliated query $q_{i,j}$ is yet unanswered, it means that exact locations of both objects are required to solve the query. Consequently, $u_j$ will be tagged as must-probe and the server trivially schedules to probe it during the next iteration, i.e., recursive call to Algorithm 2 (line 25).

Ideally, the server continues to probe must-probe nodes in consecutive iterations until no remaining node has a must-probe tag. It is only after these iterations when the server resumes making a non-trivial probe selection (Section 6.3). However, following this method in large real-world applications may not be operational as the network delay caused by consecutive probes may exceed the short duration of an epoch. We relax this problem by making two modifications to our original algorithm. First, we merge the trivial and non-trivial decision iterations and probe new must-probe objects together with those selected by a non-trivial decision in every iteration. The list of probees in iteration $i$ is represented as $l_i$ in Algorithm 2. Second, when making a non-trivial probing decision for a component, we select multiple rather than only one object. The process can be considered as selecting the TopK most appropriate objects. The number of selected objects in each iteration is a system defined parameter which is proportional to the number of objects in the component and length of an epoch, $\Delta T$. These modifications are reflected in Algorithm 2 and we use this algorithm in our experiments.

After probing a node (line 7), we remove all its incident edges from the component and evaluate the affiliated queries (lines 10-18). Newly detected must-probe nodes are scheduled to be probed during the next iteration (line 20). Removing incident edges after probe can break a component into smaller ones. The server detects the smaller connected components and independently processes them during the next iteration (lines 21-25).

The initial call to Algorithm 2 is made through Algorithm 1 where it initializes $l_1$ to all nodes in Category 3. These nodes are then trivially probed in the first iteration.

**Algorithm 1** Batch Query Processing

1: **Algorithm** Batch Query Processor ($\mathcal{Q}$)
2: **Input:** A batch of proximity queries $\mathcal{Q}$
3: Generates ordered list $L = \{l_1, l_2, \ldots, l_s\}$ where $l_{1..s}$ are disjoint sets of probees selected during $s$ iterations. $\bigcup_{i=1..s} l_i \subseteq \mathcal{U}$
4: **Receive** location updates from objects and initialize the set $U \subseteq \mathcal{U}$ of objects whose exact locations are available

▷ Phase I
5: **Classify** queries according to categories provided in Section 5
6: **Process** queries in Categories 1 and 5 and exclude them from $\mathcal{Q}$

▷ Phase II
7: Create the graph $\Gamma = \langle \mathcal{U}, E \rangle$ from the batch $\mathcal{Q}$
8: $i \leftarrow 1$
9: Find connected components of $\Gamma$

▷ Parallel execution for every component $\Gamma_j \subset \Gamma$
10: **for** $\Gamma_j \subset \Gamma$ **do**
11: **if** $|\Gamma_j| > 1$ **then**
12: **Insert** into $l_{1j}$ any object from $\Gamma_j$ that satisfies the condition in Category 3
13: ProcessSubgraph($\Gamma_j$, $i$, $l_{ij}$)
14: Notify objects

## 6.3 Probe Selection Approach

Deciding to select a user for probe in presence of uncertainty about exact user locations can be well modelled by a Markov Decision Process (MDP) [10]. We now briefly explain how the MDP framework maps to our problem but cannot be practically applied to it. The ($\mathbb{A}$) decision maker taking ($\mathbb{B}$) actions to interact with the ($\mathbb{C}$) environment translates to the ($\mathbb{A}'$) server ($\mathbb{B}'$) probing objects to address the ($\mathbb{C}'$) set of unanswered queries over users. A *state* of the environment is represented as a graph $\Gamma$ as explained in Section 6.1. The number of iterations $h$ in a *finite-horizon* optimal behaviour maps to the total number of objects that is $n$. Using available learning algorithms such as backward induction, the server automatically finds the optimal behavioural strategy, dubbed *policy*, probabilistically and selects proper probees. An optimal policy is guaranteed to exist since both the state and action spaces are finite in our representation [5]. Iterative policy learning algorithms for this non-deterministic environment take polynomial time in each iteration in terms of the count of states. As the encoding of a state, i.e., $\Gamma$ is exponential in terms of the number of objects, each iteration of these algorithms runs in pseudopolynomial time, thus not polynomial in terms of the number of objects. Hence, this approach

**Algorithm 2** Processing an Independent Batch in Parallel

1: **Algorithm** ProcessSubgraph $(\Gamma_j, i, l_{ij})$
2: **Input:** Connected component $\Gamma_j \subset \Gamma$, probing iteration $i$, confirmed list of objects to be probed in this iteration.
3: Updates sublist $l_{ij}$ of probees and initiates the next iteration
4: **repeat**
5: Use a Probe Selection Approach (Section 6.3) to rank objects and select one to be inserted into $l_{ij}$
6: **until** a system defined percentage of objects are selected
7: **Probe** all objects in $l_{ij}$
8: $l_i \leftarrow l_i \bigcup l_{ij}$
9: **Receive** exact locations. $U \leftarrow U \bigcup l_{ij}$

▷ Post-probe query evaluation
10: $mustProbeList = \emptyset$
11: **for** $u_p \in l_{ij}$ **do**
12: **for** $\langle u_p, u_q \rangle \in \Gamma_j$ **do**
13: **if** the query can be solved using exact locations
14: in $U$ **then**
15: Remove $\langle u_p, u_q \rangle$ from $\Gamma_j$
16: **else**
17: **Insert** $u_q$ into $mustProbeList_j$
18: Remove edgeless nodes (objects) from $\Gamma_j$
19: $i \leftarrow i + 1$
20: $l_i \leftarrow l_i \bigcup mustProbeList_j$
21: Find connected components of $\Gamma_j$

▷ Parallel execution for every component in $\Gamma_j$ that includes unanswered queries
22: **for** $\Gamma_{jk} \subset \Gamma_j$ **do**
23: **if** $|\Gamma_{jk}| > 1$ **then**
24: **Insert** into $l'_{ik}$ any object from $\Gamma_{jk}$ that are in $mustProbeList_j$
25: ProcessSubgraph($\Gamma_{jk}, i, l'_{ik}$)

does not scale to even a fair number of users and queries.

The above baseline approach follows a dynamic programming approach and regards immediate as well as *delayed* (i.e., future) consequences of a probe. To avoid the complexity of the optimal solution, our Enhanced Probe Selection (EPS) method follows a greedy approach and adheres to the immediate effect of probing each object. To ease computation, EPS myopically approximates the immediate number of solved queries and ignores the subsequent consequences. This comes at the cost of loosing the optimality of our probing algorithm. We prioritize the objects according to their value assigned by the $Value$ function. Put in the context of our MDP approach, EPS approximates the optimal value functions by roughly calculating immediate rewards.

We prioritize objects by calculating their value:

$$Value(u_i) = \frac{\lambda_i}{\lambda_v} + \ldots + \frac{\lambda_i}{\lambda_w}$$

where $u_v, \ldots, u_w$ are the objects in $\Gamma$ that are involved in a query with $u_i$. This function considers the number of unanswered queries as well as the relative size of mobile regions. An object involved in many unanswered queries will become a good probe candidate according to our proposed function. The rationale is that the more unanswered queries an object is involved in, the higher are the chances that at least one such query demands probing both objects. Thus, probing such an object might prevent probing some other objects. Another case is when the server's knowledge of an object location is too approximate. An object having a larger mobile region compared to its fellow objects causes more ambiguity as we discussed in Phase I and is thus a better choice for probing.

**Analytical Evaluation:** The EPS approach is proposed due to computational complexity of the baseline solution. The main task here is to sort objects based on their calculated values. This takes $O$-$(nlgn + m)$ in the worst case making it perform much faster compared to its optimal counterpart.

Table 1: Summary of Parameters.

| Parameter | Default | Range |
|---|---|---|
| Number of users | 400K | 50K- 800K |
| Speed limit (meter/sec.) | 8 | 1.5- 27 |
| Avg. queries/user | 10 | 5- 800 |
| Proximity distance $\epsilon$ (miles) | 2 | 0.1- 40 |
| Initial mobile region radius $\lambda$ | 0.4 | 0.01- 10 |
| Scale factor $\alpha$ | 1.6 | 1.01- 16 |
| Algorithm | EPS | RMD, EPS |

# 7. EMPIRICAL EVALUATION

## 7.1 Experiment Methodology

Experiments were run on Dualcore AMD Opteron nodes with 2.0 GHz CPU and 4GB RAM. For scalability tests including large numbers of queries, Dualcore Intel Xeon nodes with 2.0 GHz CPU and 64GB RAM were used. Each experiment was run on a single node operated by Linux.

We used the network-based moving object data generator [6] to generate the movement of users during 100 epoch. The duration of each epoch, $\Delta T$, equals one minute. The data generator allows us to set the maximum speed limit for users and adjusts the speed based on the density of users as well as the speed limits of the underlying street or highway.

Experiments were run on San Francisco Bay Area road network. This region has an approximate area of 7000 sq. miles with an original population of 7 million people [1]. The network includes a spatial domain of $[0, 98]^2$ miles including highways and streets of nine counties in Bay Area. According to a recent estimate by Department of Motor Vehicles, more than 22 million automobiles have been registered in this area during 2011 [2]. All this information makes this area a good candidate for testing location-based applications.

We focused on a friend-finder application as the mainstream application in our experiments and set the parameters accordingly. Table 1 shows the default values for the parameters. $400,000$ users are in relationship with an average of 10 close friends or family members while commuting with a speed of 8 meters/sec. which is approximately 17.7 MPH i.e., the average speed in streets of Bay Area [3]. The users would like to get notified if their close friends enter or leave the $\epsilon$-neighborhood of 2 miles.

The communication cost of our proximity detection algorithm was evaluated against a wide range of potential applications and audiences by varying different parameters. We also implemented the current state-of-the-art method, RMD, and compared it to our method [18]. For fairness, we ran experiments to find proper equivalent internal parameters for their method in the given road network. Results show that our method outperforms in all settings and saves more than 30% in location tracking cost compared to its closest counterpart.

## 7.2 Results

### 7.2.1 *Location Tracking Cost Evaluation*

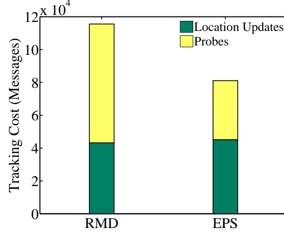



Figure 9: Cost breakdown at the default settings

Tracking cost, as the major communication overhead, is composed of location update messages initiated by users' moving dynamics, and server-side probe messages which are followed by a response from users providing their exact location. Figure 9 shows the tracking cost breakdown at the default setting. With almost similar location update costs, our Enhanced Probe Selection approach, EPS, incurs much less probing cost in each epoch compared to RMD.

Figure 10 shows the average tracking cost with respect to various parameters. Figure 10a shows the effect of proximity distance $\epsilon$ on the tracking cost. For very small and very large $\epsilon$ values, most queries are easily filtered since most pairs are relatively far or close enough not to trouble the server. However, for intermediate $\epsilon$ values, chances increase that a pair falls in the fuzzy margin within the two bounds, thus increasing the communication cost.

The scalability of our algorithm has been tested against large numbers of users and queries. We modelled applications with hundreds of millions of queries in Figure 10b in which each user shares interests with hundreds of other users in the city. Figure 10c shows the tracking cost versus number of users. In all cases our method incurs at least 30% fewer messages to solve proximity queries compared to RMD.

The effect of commute speed has been analysed in Figure 10d. Walking users, and those driving with average and maximum speed in streets and highways of Bay Area have been modelled in our experiments according to available speed statistics [3, 11].

Sensitivity of our approach to internal parameters was analysed by varying the initial mobile region radius, $\lambda$, and the scale factor, $\alpha$. Figure 10e depicts that the tracking cost is barely affected by significant changes in initial $\lambda$. This stability is because our method fine-tunes the mobile region for each user separately: it abruptly reacts to the probes and location updates caused by the user's movement behaviour and the queries it is involved in. Figure 10f shows the effect of scale factor on tracking cost. It can be observed that for a wide range of $\alpha$ values, the communication cost of EPS is less than that of RMD in its best case.

### 7.2.2 *Effect of Category 3*

Optimal decision can be made in a definite way when a query belongs to this category. Thus, the more unanswered queries belong to this case, the closer the algorithm is to the minimum probing cost. Table 2 shows the average percentage of users that satisfied this situation in an epoch compared to the total number of users involved in unanswered queries (categories 2 to 4). Even for small numbers of queries (e.g. a limited number of friends in a friend-finder application), a considerable portion of users were involved in category 3. Probing these users early in query processing addresses many other unanswered queries respectively.

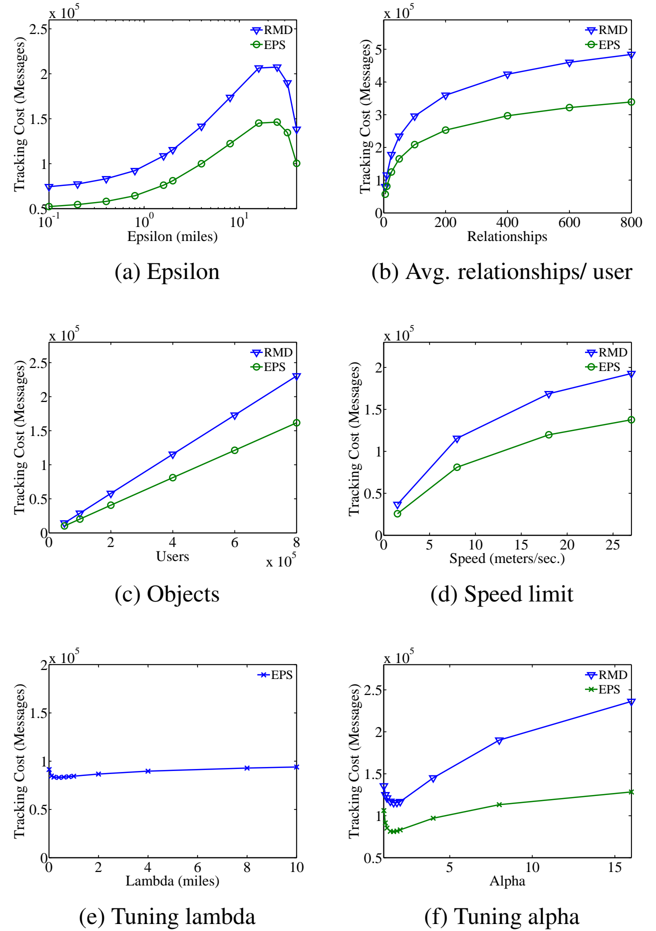



Figure 10: Object tracking cost (location update + probe messages)

### 7.2.3 *CPU Usage*

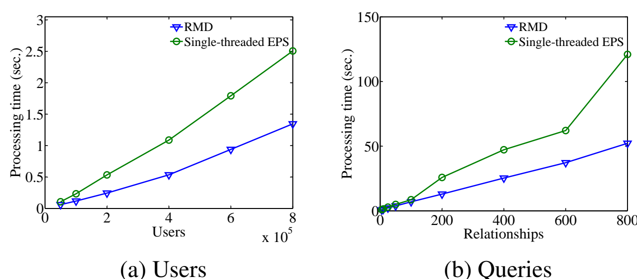
Figure 11 shows the computation cost of our algorithm. Processing queries for a large number of users in a friend-finder application takes no more than a couple of seconds in each epoch (Figure 11a). Computation cost for scalability tests is shown in Figure 11b. We simulated an application with 240 million queries and the single-threaded implementation of our algorithm managed to process all queries within the one minute $\Delta T$ duration. For processing larger number of queries we hit hardware limitations and had to sacrifice CPU cycles to make experiments feasible within the given memory limits.

### 7.2.4 *Algorithm Iterations*

According to Algorithm 2, users having the maximum values in each connected component are probed for their exact location. In order to expedite query processing, a larger number of users can be chosen from each connected component (line 6). For experiments including large numbers of queries, in each iteration we selected from each component the top 1% of users having the highest values. Our experiments show that even with large number of queries, the algorithm ran no more than seven iterations to finish query processing in each epoch. This is due to the fact that after each iteration, some edges of a component are removed (i.e., queries are solved), being replaced by multiple smaller components that would unleash more potentials for parallel probing and processing.

Table 2: Effect of Category 3 on Probing Users.

| Queries/ user | Involved users probed through Category 3 |
|---|---|
| 5 | 10.70% |
| 10 | 13.29% |
| 25 | 17.42% |
| 50 | 20.31% |
| 100 | 22.6% |
| 200 | 24.44% |
| 400 | 26.08% |
| 600 | 27% |
| 800 | 27.62% |




(a) Users (b) Queries

Figure 11: Server-side computation cost

# 8. CONCLUSION AND FUTURE WORK

In this paper, for the first time we argued that in addition to efficient "location update", efficient "probing" can also improve the communication-efficiency of proximity query answering significantly. Accordingly, we proposed a probe optimization technique that considers proximity queries in batch and minimizes the number of probes required to answer the entire batch of queries. Furthermore, we showed that our proposed probe optimization technique is parallelizable, and hence, allows for scale-out. Our experiments demonstrate that our proposed proximity query answering approach enables continuous processing of hundreds of millions of proximity queries.

We plan to pursue this study in two directions. First, we believe that our proposed probe optimization approach can be generalized to enhance the efficiency of other spatial queries on moving objects. Accordingly, we intend to investigate and address specific challenges of probe optimization in the context of this family of queries. Second, given that our proposed probe optimization approach is parallelizable, we will seek enhancements of our proposed technique by leveraging the features offered by cloud computing.

# 9. ACKNOWLEDGEMENTS

Computation for the work described in this paper was supported by the University of Southern California Center for High-Performance Computing and Communications (www.usc.edu/hpcc).

# 10. REFERENCES

[1] Bay Area Census. http://www.bayareacensus.ca.gov/.

[2] Department of Motor Vehicles statistics for 2011. http://apps.dmv.ca.gov/about/profile/official.pdf.

[3] San Francisco commute speeds drop dramatically. http://sanfrancisco.cbslocal.com/2011/12/06/san-francisco-commute-speeds-drop-dramatically/.

[4] A. Amir, A. Efrat, J. Myllymaki, L. Palaniappan, and K. Wampler. Buddy tracking - efficient proximity detection among mobile friends. In *INFOCOM*, 2004.

[5] N. Bäuerle and U. Rieder. *Markov Decision Processes with Applications to Finance*. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

[6] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, June 2002.

[7] M. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous monitoring of distance-based range queries. *Knowledge and Data Engineering, IEEE Transactions on*, 23(8):1182 –1199, Aug. 2011.

[8] A. Civilis, C. S. Jensen, and S. Pakalnis. Techniques for efficient road-network-based tracking of moving objects. *IEEE Trans. on Knowl. and Data Eng.*, 17(5):698–712, May 2005.

[9] H. Hu, J. Xu, and D. L. Lee. A generic framework for monitoring continuous spatial queries over moving objects. In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data*, SIGMOD '05, pages 479–490, New York, NY, USA, 2005. ACM.

[10] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: a survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[11] B. J. Mohler, W. B. Thompson, S. H. Creem-Regehr, H. L. Pick, and W. H. Warren. Visual flow influences gait transition speed and preferred walking speed. *Experimental Brain Research*, 181(2):221–228, 2007.

[12] M. F. Mokbel, X. Xiong, and W. G. Aref. Sina: scalable incremental processing of continuous queries in spatio-temporal databases. In *Proceedings of the 2004 ACM SIGMOD international conference on Management of data*, SIGMOD '04, pages 623–634, New York, NY, USA, 2004. ACM.

[13] P. Pesti, L. Liu, B. Bamba, A. Iyengar, and M. Weber. Roadtrack: scaling location updates for mobile clients on road networks with query awareness. *Proc. VLDB Endow.*, 3(1-2):1493–1504, Sept. 2010.

[14] G. Treu and A. Küpper. Efficient proximity detection for location based services. In *Proceedings of the 2nd Workshop on Positioning, Navigation and Communication (WPNC)*, 2005.

[15] G. Treu, T. Wilder, and A. Küpper. Efficient proximity detection among mobile targets with dead reckoning. In *Proceedings of the 4th ACM international workshop on Mobility management and wireless access*, MobiWac '06, pages 75–83, New York, NY, USA, 2006. ACM.

[16] Z. Xu and A. Jacobsen. Adaptive location constraint processing. In *Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, SIGMOD '07, pages 581–592, New York, NY, USA, 2007. ACM.

[17] Z. Xu and H.-A. Jacobsen. Processing proximity relations in road networks. In *Proceedings of the 2010 international conference on Management of data*, SIGMOD '10, pages 243–254, New York, NY, USA, 2010. ACM.

[18] M. L. Yiu, L. H. U, S. Šaltenis, and K. Tzoumas. Efficient proximity detection among mobile users via self-tuning policies. *Proc. VLDB Endow.*, 3(1-2):985–996, Sept. 2010.