TRANSACTIONS ON DATA PRIVACY 2 (2009) 21–46

Location Privacy in Moving-Object Environments

Dan Lin[†], Elisa Bertino[‡], Reynold Cheng[§], Sunil Prabhakar[‡]

[†]Department of Computer Science, Missouri University of Science and Technology

500 w. 15th St., Rolla, MO 65409, USA

E-mail: lindan@mst.edu

[‡]Department of Computer Science, Purdue University

305 N. University St., West Lafayette, IN 47907, USA

E-mail: {bertino,sunil}@cs.purdue.edu

[§]Department of Computer Science, Hong Kong University

Pokfulam Road, Hong Kong

E-mail: ckcheng@cs.hku.hk

Abstract. The expanding use of location-based services has profound implications on the privacy of personal information. If no adequate protection is adopted, information about movements of specific individuals could be disclosed to unauthorized subjects or organizations, thus resulting in privacy breaches. In this paper, we propose a framework for preserving location privacy in moving-object environments. Our approach is based on the idea of sending to the service provider suitably modified location information. Such modifications, that include transformations like scaling, are performed by agents interposed between users and service providers. Agents execute data transformation and the service provider directly processes the transformed dataset. Our technique not only prevents the service provider from knowing the exact locations of users, but also protects information about user movements and locations from being disclosed to other users who are not authorized to access this information. A key characteristic of our approach is that it achieves privacy without degrading service quality. We also define a privacy model to analyze our framework, and examine our approach experimentally.

1 Introduction

The expanding use of spatial, mobile and context-aware technologies, the deployment of integrated spatial data infrastructures and sensor-networks, and the use of location data as the foundation for many current and future information systems have profound implications on the privacy of personal information. Today people are increasingly aware of privacy issues and do not want to expose their personal information to unauthorized subjects or organizations. An important problem is represented by the possibility that a piece of personal information released by an individual to a party be combined by this party, or other parties, with other information, leading to the disclosure of sensitive personal information. In other cases, even if an individual does not directly release personal information to another party, this party may still become aware of this information if it has to provide a service to such an individual. This is in particular the case of location-based service providers that, because of the very nature of the services they provide, need to track user movements and locations. It is then easy, based on this information, to discover user habits and other personal information. There is therefore an important concern for *location privacy* in location-based services,

that is: "how can we prevent other parties from learning one's current or past location? [1]". By looking more closely at the privacy problem in such a context, we can see that there are at least two important requirements, that is, keeping movement and location information private from service providers and from other users. For example, GPS users who do not want to disclose their locations to the system may still require service such as "is there any of my friends close to me now?" There are two privacy requirements for this query. First, service providers are not allowed to know the real locations of users. Second, users can only query an authorized dataset (e.g. a list of their friends).

In this paper, we address such a problem by developing a framework to preserve location privacy in moving-object environments. The basic idea of our approach is to send transformed user location data to the service provider. We support a number of different types of transformations, such as scaling, translation, rotation, to cloak user information. These transformations are performed by agents interposed between users and service providers. Agents are only responsible for transforming information either received from the users or the server. They serve as intermediaries and do not store user information. The service providers receive the transformed data and compute answers to queries on these transformed data.

An important feature of our approach, which is critical for privacy assurance, is the use of multiple agents. The user can randomly choose the agent to receive his information each time he issues an update. Thus, each agent only has a part of the information concerning the user. Such an approach is crucial for enhancing privacy. For example, if an adversary hacks one agent, it is still unable to track the user; if some agents illegally store user information, they cannot determine the trajectories of users without colluding with other entities. Here our approach closely adheres to an important security principle, dictating that sensitive information should not be entrusted to a single entity; rather such information should be spread among several entities.

In our framework, the server stores for each agent a sub-dataset specific to the agent. A query is thus executed by the server separately on the sub-dataset of each agent. It is important to notice that location-based queries require that relative distance among users through the same agent be maintained after the transformation. The transformations we adopt have such a property. Specifically, we employ a combination of the three basic types of transformations, that is, scaling, rotation, translation, as our transformation functions. It is however important to notice that maintaining the relative distance after the transformation may reveal the map topology. Therefore, we introduce the concept of *multiple transformation* that applies slightly different transformation functions to users' positions updated at different time instants. This makes the relative distance hard to be inferred. Correspondingly, the multiple transformation also needs to be applied to queries. To avoid handling the increased number of queries, a *super query* is then proposed, which covers all queries after the multiple transformation. As explained later, a super query is essentially an approximate version of the original query, which facilitates efficient evaluation of this framework with additional filtering costs.

Our technique not only prevents service providers from inferring the exact locations of users, but also keeps information about the location of an individual private from other individuals not authorized to access such information. Specifically, users have a list of group IDs that indicate which groups they belong to. Based on these group IDs, the server can remove the query answers that are not in the qualified groups, so that users can avoid their privacy leaked to other users not belonging to the same group. A key characteristic of our approach is that privacy is achieved without degrading service quality. Based on the experiments that we have carried out, our approach is particularly efficient for update operations, in that it also reduces the number of disk accesses compared to conventional algorithms. Such improvement is very attractive in moving-object environments where update frequency is always high.

Finally, we develop a privacy model to analyze the privacy level achieved in our framework. In particular, we investigate the threats posted by the query server from discovering the users' true lo-

cations and movement pattern. We then propose intuitive methods to quantify the level of protection against these threats in our system.

To the best of our knowledge, this is the first framework that protects location privacy in moving-object environments without sacrificing accuracy, and which is also scalable and supports a large variety of queries.

A preliminary version of this paper appears in [14], where we presented the basic idea. In this paper, we make the following additional contributions. First, we provided more detailed description of the framework. Second, we developed a new algorithm for k nearest neighbor queries. Third, we extended the discussion in the section of system analysis. Furthermore, we run a more comprehensive set of experiments to demonstrate the efficiency of the system.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the system architecture. Section 4 presents the detailed algorithms for location updates and queries. Section 5 presents the system analysis. Section 6 covers comprehensive performance experiments. Finally, Section 7 concludes and gives future research directions.

2 Related Work

Privacy issues in location-aware mobile devices [15] have recently attracted considerable research interest. Some early works on location privacy protection suggest the use of policies, which serve as a contractual agreement about how user's location information can be used by service providers [9, 20]. Typically, users have to trust the service providers. However, such a trusted relationship is hard and costly to establish especially for small or temporary service providers.

Therefore, more recent works focus on the development of anonymization techniques specific to location-based service environments. A common technique is based on the notion of spatial-temporal cloaking. The idea is firstly introduced by Gruteser et al. [8]. They propose the application of the k-anonymity technique to cloak location information in order to support anonymous applications. Specifically, a user's location is represented by a region in which other $k-1$ users are also present. This model has later been improved by Gedik et al. [6]. Their approach supports the assignment of different values for different users to the k parameter in a system. Also as part of their work, they investigate the tradeoff between anonymity and accuracy requirements. In [1], Beresford et al. use the k-anonymity metric in pseudonymous applications. The idea is to rename user's identity when there are at least k users in the same zone. When there are less than k users in the same zone, a user may refuse to disclose his location. Recently, Cheng et al. [4] invetigated the trade-off of location cloaking, privacy and quality of service. They developed queries that evaluate cloaked data and provide probabilistic answers. They also presented quality metrics in order to quantify the effect of cloaking on service quality. Based on the similar idea, Mokbel et al.[16] propose a framework to protect mobile users in location-based services, which adopts the cloaking idea and supports various k parameters.

However, the above k-anonymity model based approaches have at least one of the following drawbacks. First, some approaches cannot guarantee the accuracy of the query answers. Second, some approaches cannot be applied when there are less than k users in a specific area. Third, they trust agents and allow agents to store information about users, which may make agents the target of attacks by malicious parties. Finally, such a k-anonymity model may not be able to support anonymization around sensitive areas such as home addresses in non-anonymous applications. For example, if a user's ID is known, the cloaking region around his home address will tell attackers that the user is probably at his home.

Some other approaches are based on cryptographic techniques. Hore et al. [10] suggest encrypting location data and using a privacy-preserving index for executing range queries over encrypted data.

However, this technique only works for specific query operators and is unable to provide accurate query answers. Similarly, Khoshgozaran et al. [13] also propose a one-way transformation to encode all static and dynamic objects and resolve the query blindly in the encoded space. Again, they are not able to generate the exact query answers. To rectify the shortcomings of previous work, Yiu et al. [24] have proposed a client-side query processing technique that retrieves points of interest from the server incrementally until accurate query answers are obtained. The main problem of this approach is the expensive communication cost since users need to receive much more data than just query answers. Ghinita et al. [7] propose a framework to support private nearest neighbor queries based on Private Information Retrieval (PIR). Their approach does not require users to trust any third-party anonymizer and can return exact answers. However, PIR may be too costly to be applied in practice.

Regarding the data transformation that we use in our system, there is one related work by Chen et al. [3]. They also apply geometric transformations to data but with a different purpose which is to preserve privacy in data classification.

In recent years, researchers have developed a number of indexing techniques for moving objects. As we will explain in Section 3, any index for moving objects can be used in our framework. Representative indexes include the TPR-tree (Time-Parameterized R-tree) family of indexes (e.g., [19, 21]), transformation-based indexes such as STRIPES [17], B^+-tree-based indexes such as the B^x-tree [12] and the B^{dual}-tree [23]. In this paper, we employ the TPR*-tree [21] to manage data at the server side. Unlike existing approaches to the problem of location privacy protection, our approach can be applied to anonymous, pseudonymous and non-anonymous applications, and guarantees 100% correct query answers without information leaking.

3 The Strategies and the Architecture of the Location Privacy Protection System

In this section, we describe the strategies and the architecture of our Location Privacy Protection (LPP) system. Figure 1 illustrates this architecture. The basic strategy underlying our approach is to reduce the leaking of private information by using data transformation and employing m agents in-between users and servers. Each time a user [1] needs to update his position, he does not directly contact the server; instead, he *randomly* selects an agent to which he sends his data. When querying, the user has to send the query to all agents. Then the agents will execute a transformation on the user data or queries and pass the transformed data to the server. The server handles the data processing and returns the query results to the agents. After receiving the results from the server, the agents perform a reverse transformation before returning the results to the user. We now proceed to describe in details how each component of our system works.

- **User**

  Users are position providers or query issuers. Users' positions are assumed to be unchanged until next update, that is, the *location database* at the service provider keeps the latest position of each user. Users may have a list of qualified agents, and they are assumed to have the ability to randomly choose agents and perform some postprocessing.

  Different policies can be adopted to protect information about a given user from other users. One policy is a global ranking, which allows users with high ranks to query location and movement information about users with equal or lower ranks. Another policy is a group policy, under which users can query location and movement information about users in the same group. A user can be

[1] We use the term 'user' in the discussion. In reality the described activities are carried out by some client software residing at the user's device.

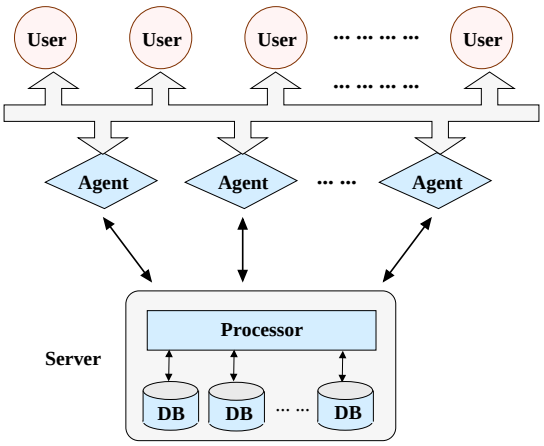


Figure 1: LPP System Overview

a member of multiple groups and hence he may have a list of group IDs. In our system, we adopt the latter policy. Hence, in location databases users are represented by records of the form $\langle uid, gids, loc, t_{up}\rangle$, where uid is the user ID, $gids$ is a list of group IDs, and loc is the user's location at time t_{up}.

- **Agent**

  Agents are a critical component in our approach. An agent transforms the data received from the users and sends them to the server. It also executes reverse transformations on the data obtained from the server and then forwards them to the users.

  The types of transformations supported by an agent includes the transformation of the user ID, the group IDs, and the user locations. Agents periodically change their transformation functions in order to prevent the server from analyzing the data from the same agent. Thus, agents need to maintain transformation tables for each type of data. Such tables store records of the form $\langle\ t_{id}, f_{id}, count_{id}\rangle$ and $\langle t_{loc}, f_{loc}, count_{loc}\rangle$, where t records the time instant at which the transformation function f has started to be used, and $count$ is the number of objects being transformed by f.

  There are three important features about our agents. First, for the security purpose, agents are independent of the main server, which means they are not under control of the server. Second, agents do not store any user data and hence they are lightweight computers. Therefore, it is possible to verify their code in order to provide assurances about their correct behavior. Third, transformation functions for different types of data do not need to be changed at the same time.

- **Server**

  The server is responsible for data storage, maintenance and query processing. It also maintains datasets transferred by various agents separately. Any index for moving objects that supports efficient updates and queries can be adopted to manage the datasets in the server.

The main advantage of our approach is that no single entity (m agents or server) is able to track the movement of any user without colluding with other entities in the system. Because each agent only collects a subset of the locations of each user in the system, the level of trust required from each agent does not need to be high. Moreover, the use of m agents allows multiple transformations to

be applied to the data by the same user. This makes it much harder for the server to keep track of the relative distance among users. In essence, the server is only a computing engine for the various agents.

Finally, we would like to mention that we focus on queries over moving objects in this paper. For queries over static objects (e.g. restaurants, gas stations), our framework can be extended in the following way. We can store the static objects in a separate database in the server since such objects may not have any concern over location privacy, and then we use slightly modified query algorithms (which will be explained later). Unless specified otherwise, we assume the data of interest are moving objects in subsequent discussions.

4 Algorithms

In this section, we present the detailed algorithms for data transformation, queries and updates in the LPP system.

4.1 Data Transformation

Data transformation includes transformation of user IDs, group IDs, user locations, and queries. We address each of them respectively in the following sections.

4.1.1 ID Transformation

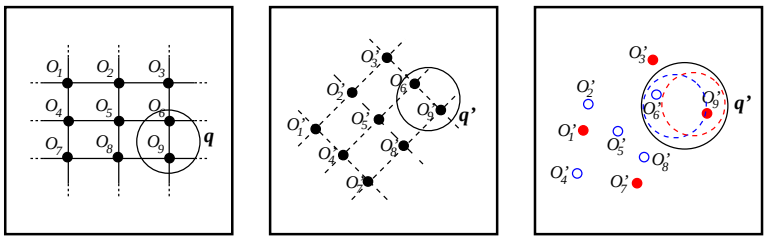
The main purpose of ID transformation is two-fold. First, we need to prevent the server from identifying the same users through different agents. This can be easily achieved by choosing different transformation functions for different agents. There are no restrictions on the transformation function itself. It could be a simple encryption. Also, we need to prevent the server from tracking the positions of the same user from one agent. We thus propose to periodically change the transformation functions for each agent, which can assign different pseudo-IDs to the same user who sends data at different time instants. A transformation table is then maintained for each agent. As mentioned previously, the transformation table consists of records of the form $\langle t_{id}, f_{id}, count_{id} \rangle$. Algorithms for its maintenance are covered in section 4.2.

4.1.2 Location Transformation

Just transforming IDs is not enough to provide location privacy for users because some locations (e.g. homes) are strong-ly associated with user IDs and may thus cause information leak. Therefore, we introduce the notion of location transformation, which is a crucial feature of our system.

The main challenge in the development of suitable functions for location transformation is to keep the relative distance in each sub-dataset (the dataset obtained from the same agent) unaltered by the transformation in order to support location based services (e.g. nearest neighbor queries). Possible transformation functions include scaling, rotating, translation, and their combinations. In our system, we employ a combination of scaling, rotation and translation. We represent the transformation function through its parameters denoted by the tuple $[s, \theta, (t_x, t_y)]$, where s is the scaling factor, θ is the rotation angle, and t_x, t_y are the translation distance along the x and y axes respectively.

However, the preservation of the relative distance among objects could disclose the map topology. For example, if the server tries to connect objects close to one another, it may be able to discover the joint distribution of objects and then determine the road network. Figure 2 gives a simple example. Suppose that the original data lie on a grid-like road network. If they are transformed by a single transformation function, the server may discover the grid by connecting objects on the same lines



(a) Original Data **(b) Single Transformation** **(c) Multiple Transformation**

Figure 2: An Example of Position Transformation

(dashed lines in the figure). To address such a problem, our approach is to make the relative distance hard to be inferred. We thus adopt a strategy that requires each agent to periodically change the location transformation function. We refer to such strategy as *multiple transformation*. The bottom-right part of Figure 2 shows the effect of the multiple transformation strategy. Assume that objects O_1, O_3, O_7 and O_9 are transformed by a function f_1; O_2, O_4, O_5, O_6 and O_8 are transformed by another function f_2 that is only a little bit different from f_1. From the transformed objects, it is hard to discover the original data distribution.

We now proceed to present the generation of the multiple transformation. The first transformation function can be an arbitrary one, while the following transformation functions need to fulfill some constraints. The differences among the transformed positions obtained by various transformation functions should be kept within a small range. Such a constraint is crucial in order to provide good quality answers to queries based on the relative distance among objects.

A simple strategy to satisfy the above constraints is to apply the translation operations with different parameters to the first transformation function. Moreover, to achieve efficient queries, multiple transformation should preserve the following property.

Property 1. Let $\langle x, y\rangle$ be a point, $\langle x_0, y_0\rangle$ be the position obtained by applying the initial transformation function to $\langle x, y\rangle$, and $\langle x_1, y_1\rangle$, $\langle x_2, y_2\rangle$,..., $\langle x_{n-1}, y_{n-1}\rangle$ be the positions obtained from subsequent multiple transformation functions. The distance between $\langle x_0, y_0\rangle$ and $\langle x_i, y_i\rangle$ ($1 \leq i \leq n-1$) must be less than or equal to a threshold λ.

The detailed algorithm for multiple transformation is summarized in Figure 3. The first step selects an initial transformation function $[s_0, \theta_0, (t_{x0}, t_{y0})]$, sets its counter $count_0$ to 0, and stores the values in the transformation table. After a period of time t_{int}, we generate a new transformation function. We first randomly choose a value d in $(0, \lambda)$, and then randomly generate the parameter d_x (the translation distance of x axis) in the range of $(-d, d)$. The parameter for the y axis d_y can be computed by $d_y = \pm(d^2 - d_x^2)^{\frac{1}{2}}$. Then we insert a new tuple $\langle t_1, [s, \theta, (t_{x0} + d_x, t_{y0} + d_y)], 0\rangle$ in the transformation table. This process is repeated every t_{int} time interval. There are two things worth noting. First, each agent can choose his own λ. Second, the transformation table will not keep growing. Functions that are no longer used by users will be removed during the update operations (as addressed in Section 4.2).

4.1.3 Query Transformation

We now address how to transform queries. In the discussion we focus on snapshot range queries. A range query retrieves all objects the location of which falls within the circular range $q = (c(x, y), r)$ at a given query timestamp, where $c(x, y)$ is the center and r is the radius of the query.

Algorithm Multiple_Transformation($Ttable$**,** t_c**)**
Input: $Ttable$ is a transformation table, t_c is current time

1. **if** (t_c = 0) **then**
   // select the first transformation function
2. randomly generate s_0, θ_0, t_{x0}, t_{y0}
3. insert $\langle 0, [s_0, \theta_0, (t_{x0}, t_{y0})], 0\rangle$ into $Ttable$
4. **else**
5. randomly generate d in the range of $(0, \lambda)$
6. randomly generate d_x in the range of $(-d, d)$
7. randomly select d_y from $\{-(d^2 - d_x^2)^{\frac{1}{2}}, (d^2 - d_x^2)^{\frac{1}{2}}\}$
8. insert $\langle t_c, [s_0, \theta_0, (t_{x0} + d_x, t_{y0} + d_y)], 0\rangle$ to $Ttable$

end Multiple_Transformation.

Figure 3: Multiple Transformation Generation Algorithm

Due to the multiple transformation on the users' positions, a query has to handle data from different transformations. One solution is to transform the query using all transformation functions, and then execute multiple queries. However, this is not efficient and may disclose the relationship among transformation functions. Therefore, we introduce the concept of *super query*, which covers all queries after multiple transformations. For example, in Figure 2, a range query q is first transformed into two queries (represented in the figure as dashed circles) by function f_1 and f_2. Instead of answering these two queries, we propose answering a super query q' that covers the regions of these two queries. In this case, the query efficiency mainly depends on the extra area covered by the super query. In the following, we first describe how to generate the super query, and then analyze the characteristics of the super query.

Given a query $q = (c(x, y), r)$, we can obtain a set of transformed queries by using the multiple transformation functions. Since the transformation functions change with time, to compute a super query that tightly bounds all transformed queries requires the checking of all the transformation functions and thus involves extensive computations. We propose to use an easily-computed super query (denoted as q_s) which is always a superset of the transformed queries unless the parameter λ changes. Specifically, q_s is computed as: $q_s = (c(f_0(x)\ ,\ f_0(y)),\ f_0(r) + \lambda)$, where f_0 is the first (initial) transformation function. Figure 4 illustrates an example, where the black point is the transformed query center by using the first transformation function, white circles are positions after other transformations, and the transformed radius of the query is r'.

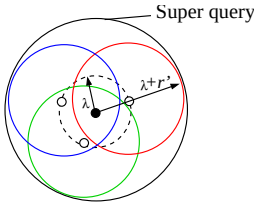


Figure 4: Super Query

The generation of the easily-computed super query is based on Property 1 (see previous section). Property 1 prevents the super query from growing arbitrarily large. It guarantees that the radius of the

super range query is at most λ larger than that of any transformed query. It is true that the super query may incur some overhead due to the search of a larger space compared to the query transformed by any one of the transformation functions. To characterize the super query, we define its *false negative rate* as the number of missing query answers divided by the number of correct query answers, and define its *false positive rate* as the number of false query answers divided by the number of correct query answers. Esstimates for false positive and false negative rates are established by the following theorem.

Theorem 1. *Let $q = (c, r)$ be a query, and f_0, f_1, ..., f_{n-1} be a set of transformation functions, where f_0 is the initial transformation function. Its super query $q_s = (c_s, r_s)$ satisfies the following properties*

(i) false negative rate fn is 0;

(ii) false positive rate fp is approximately $2\lambda/f_i(r)$ $(0 \leq i \leq n-1)$.

Proof. We denote the query transformed by f_i as $q_i = (c_i, r_i)$ $(0 \leq i \leq n-1)$. We denote the correct answer set as A.

(i) To prove the false negative rate is 0, we need to prove that for any $a \in A$, a can be captured by q_s.

We know that a is transformed by one of the transformation functions, say f_i $(0 \leq i \leq n-1)$. Then, a can be captured by the query q_i which is transformed by the same transformation function.

According to Property 1, the distance between the centers of q_i and q_0 (transformed by f_0) is less than λ. According to the generation algorithm of the super query, the center of q_0 is the same as the center of q_s, and the radius of the q_s is λ more than that of the q_i. Consequently, we have $r_s - r_i \geq distance(c_s, c_i)$, which indicates that q_s covers q_i. Hence a can be captured by q_s.

(ii) Assume the data points are evenly distributed, then we may use the areas to see how more points can be covered by the super query compared with the query by a single transformation (i.e., the number of false positives is proportional to the extra area).

The area S_i covered by a query q_i is πr_i^2 . The area S_s covered by the super query is $\pi(r_i + \lambda)^2$. Then the percentage of increase in the area of the super query is:

$fp = \frac{S_s - S_i}{S_i} = \frac{\pi(r_i+\lambda)^2 - \pi r_i^2}{\pi r_i^2} = \frac{\lambda(2r_i+\lambda)}{r_i^2}$

When $\lambda \ll r_i$, $fp \simeq 2\lambda/r_i$. □

Theorem 1 demonstrates the correctness of the super query (no false negatives) and points out a way to tune the performance of the query. Given a false positive rate, we can choose a proper λ.

Note that from the users' point of view, there will be no false positive because the agent will filter the data rerturned by the server in order to eliminate the false positives.

4.2 Updates

Generally, an update is interpreted as a deletion followed by an insertion. Figure 5 shows the detailed update algorithm.

To insert a tuple $\langle uid, gids, loc, t_{up} \rangle$ of a user, three steps are executed. First, the user randomly selects an agent and sends his information to the agent. Second, the agent transforms the user ID, the group ID list and the location, and then sends the transformed data to the server. During the transformation, the agents will adjust the counters of the transformation functions, and remove the ones with counters equal to 0 which will not be used in the future. Finally, the server tags the data with the agent ID and stores them.

For the deletion, the user needs to submit his old information to the same agent which handled the insertion of this information. The agent will check the transformation table and look for the

Algorithm Update

User:
Insertion:
1. randomly select an agent with ID aid
2. send $\langle uid, gids, loc, t_{up}, ' i' \rangle$ to the agent aid

Deletion:
1. send $\langle uid, gids, loc, t_{up}, ' d' \rangle$ to the agent aid

Agent:
1. receive $\langle uid, gids, loc, t_{up}, op \rangle$ from the user
2. $f_{id} \leftarrow$ ID transformation function of time t_{up}
3. $(uid', gids') \leftarrow f_{id}(uid, gids)$
4. $f_{loc} \leftarrow$ location transformation function of time t_{up}
5. $loc' \leftarrow f_{loc}(loc)$
6. send $\langle uid', gids', loc', t_{up}, op, aid \rangle$ to the server
7. **if** ($op ==' i'$) **then** // this is an insertion
8. $\quad count_{id} \leftarrow count_{id} + 1$
9. $\quad count_{loc} \leftarrow count_{loc} + 1$
10. **else** // this is a deletion
11. $\quad count_{id} \leftarrow count_{id} - 1$
12. $\quad count_{loc} \leftarrow count_{loc} - 1$
13. $\quad$ **if** ($count_{loc}$ is 0 and f_{loc} is not 1^{st} function)
14. $\quad\quad$ delete the tuple of f_{loc} from transformation table
15. invoke Multiple_Transformation every t_{int}

Server:
1. receive $\langle uid', gids', loc', t_{up}, op, aid \rangle$ from the agent
2. **if** ($op ==' i'$) **then** // this is an insertion
3. $\quad$ insert $\langle uid', gids', loc', t_{up}, aid \rangle$
4. **else** // this is a deletion
5. $\quad$ delete $\langle uid', gids', loc', t_{up}, aid \rangle$

end Update.

Figure 5: Update Algorithm

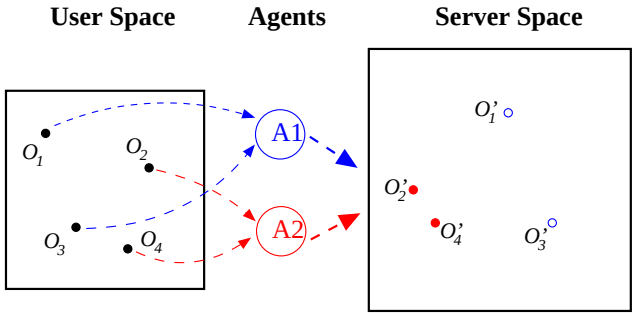


Figure 6: An Example of Update Operation

corresponding function at the update time. Then, the agent will use this function to transform user information, and decrease the counter of this function by one. If the counter is 0, the function (except for the first one) will be removed from the transformation table. The remaining process for deletion is similar to the insertion.

It is worth noting that users can send deletion message to the old agents and insertion message to the new agents.

Consider the example shown in Figure 6. Suppose there are four users O_1, O_2, O_3 and O_4, and two agents A_1 and A_2. O_1 and O_3 select agent A_1, and O_2 and O_4 select agent A_2. The transformed data of O_1 and O_3 is O'_1 and O'_3, and the transformed data of O_2 and O_4 is O'_2 and O'_4, respectively.

We also consider the situation when an object disappears accidentally without being able to notify the server. The information of such objects will soon be outdated. We define that an object is outdated if difference between its latest update time and current time is larger than a given threshold. During each insertion or deletion, we identify and delete outdated entries in accessed nodes.

4.3 Queries

Our model supports various types of snapshot queries. In the following, we outline the query execution strategies for two popular types of queries, range queries and k nearest neighbor queries.

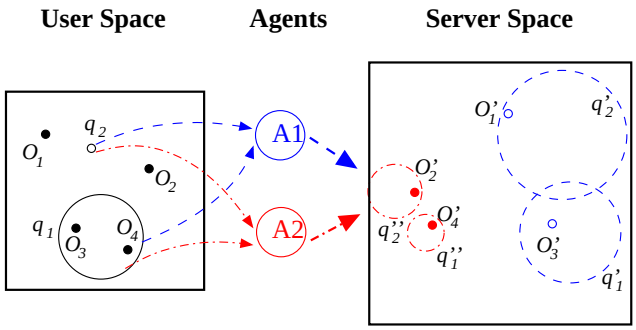


Figure 7: An Example of Query Operation

4.3.1 Range Query

A range query retrieves all objects whose location falls within the circular range $q = (c(x, y), r)$ at a given query timestamp, where $c(x, y)$ is the center and r is the radius of the query.

As object positions are transformed in different ways through different agents, we have to send a query to all agents. Each agent will generate and send a super query to the server. After receiving the query answers from the server, the agent needs to transform them back and to check whether they are the correct answers to the original query. Finally, users will aggregate the partial results obtained from the agents. If user ranks or group IDs are to be taken into by the query, one more filtering step will be carried out by the server in order to prune unqualified answers. Note that the server can filter the results based on transformed IDs before sending any results to agents. Figure 8 shows the outline of the algorithm.

Figure 7 gives a simple query example, where q_1 is a current circular range query and the dataset in Figure 6 is reused. We can see from the user space that O_3 and O_4 are the query answers. Since O_3 and O_4 are transformed by different agents, in order to capture their transformed positions in the server space, q_1 needs to be transformed through all agents. The transformation generates queries q'_1

Algorithm Range_Query

User:
1. **for** ($i \leftarrow 0$) **to** ($i < m$) **do**
2. send $\langle uid, gids, c(x, y), r\rangle$ to the i^{th} agent

Agent:
1. receive $\langle uid, gids, c(x, y), r\rangle$ from the user
2. $gids' \leftarrow f_{id}(gids)$
3. $\langle c'(x', y'), r'\rangle \leftarrow \langle c(f_0(x), f_0(y)), f_0(r) + \lambda\rangle$
4. send $\langle c'(x', y'), r', gids', aid\rangle$ to the server

Server:
1. receive $\langle c'(x', y'), r', gids', aid\rangle$ from the agent
2. find users in the query range
3. remove users that not in any group of $gids'$
4. return query result $\langle qresult\rangle$ to agent aid

Agent:
1. receive the query result $\langle qid', qresult\rangle$ from the server
2. $qresult' \leftarrow$ reverse transform $qresult$
3. **for** each result qr in $qresult'$ **do**
4. **if** (qr not an answer of the original query) **then**
5. remove qr from $qresult'$
6. return $qresult'$ to user uid

User:
1. **for** ($i \leftarrow 0$) **to** ($i < m$) **do**
2. receive $qresult$ from the i^{th} agent
3. aggregate all the query results

end Range_Query.

Figure 8: Range Query Algorithm

and q_1''. Then q_1' will return the answer O_3' to agent A_1, and q_1'' will return the answer O_4' to agent A_2. Agents execute reverse transformations on the obtained answers and send the final answer O_3 and O_4 back to the user.

If a range query about static objects that have no privacy (e.g., restaurants) is submitted by the user, the algorithm in Figure 8 is simplified as follows. First, the query does not contain any user group information. Second, the user only sends it to any one of the agents. The agent does not need to do any transformation (i.e., steps 2 and 3 are skipped). The server then evaluates the query as usual, but this time using the static object database. Finally, the agent simply passes back the result obtained from the server to the user without doing any transformation.

4.3.2 K Nearest Neighbor Query

Given a query object with position (qx, qy), the k nearest neighbor query (kNN query) retrieves k objects for which no other objects are nearer to the query object at a given query timestamp.

One way to compute this kind of query is to transform the position of the query object using all the functions in the agent's transformation table. And the server needs to consider kNN for each transformed query position. For simplicity, we propose to compute the kNN query by iteratively performing range queries with an incrementally expanded search region until k answers are obtained.

The conversion from a kNN query to a range query is as follows. The first range query q_0 is centered at (qx, qy) with radius $r_0 = D_k$, where D_k is the estimated distance between the query object and its k'th nearest neighbor; D_k can be estimated by the equation [22]:

$$D_k = \frac{2}{\sqrt{\pi}} \left[1 - \sqrt{1 - \left(\frac{k}{N} \right)^{\frac{1}{2}}} \right]$$

where N is the number of objects. The radius will be enlarged by $r_q = D_k/k$ at each iteration in query processing, until k answers are found.

Like the range query, a kNN query also needs to be sent to all agents. The main difference is that each agent needs to convert the kNN query to a range query first. Then the agent transforms the range query and the expansion parameter r_q, and sends them to the server (the transformed query and r_q are denoted as q' and r'_q respectively). The server will keep processing the range query q' with the radius extended by r'_q each time, and return the query result to the agent once it obtains k qualified answers. Finally, each agent computes the correct distance, and sends the distance along with the user IDs to the user that issued the query. The user then combines these to find his true k nearest neighbors.

For example (see Figure 7), q_2 is a nearest neighbor query, and q'_2 and q''_2 are corresponding queries in the server space after the transformation. From q'_2, agent A_1 gets a candidate nearest neighbor O'_1. From q''_2, agent A_2 gets a candidate nearest neighbor O'_2. Then the user will receive two candidates O_1 and O_2. After comparing the real distance between candidates and the query object, the user finally obtain its nearest neighbor O_1.

If a kNN query is executed over non-private static objects, the query just needs to be submitted to one of the agent, which does not do any transformation and forward the query to the server. The server executes the kNN query over the static object database and returns the result to the user through the help of the agent. If the query object of the kNN query is a private property (e.g., it is the current location of the user), then the kNN query can be converted to a range query in order to hide the actual position of the query object.

5 System Analysis

This section analyzes the privacy protection, communication costs and concurrent processing in the LPP system.

5.1 Privacy

For the privacy analysis, we provide a formal model for better understanding and evaluation of the LPP system. We focus on location breach rather than ID protection in the following discussion. Several assumptions are adopted in the model. First, we assume that agents are trustable since they are lightweight systems and may be easily verified. This assumption is commonly used in many other location privacy protection methods (e.g. [4, 16]). Second, we assume that the server knows the overall architecture of the LPP system, which means the server knows from which agent an update or a query is sent. Based on these assumptions, we define our privacy model, *Spatial* $\Gamma-anonymity$, as below.

Definition 2. *Spatial* $\Gamma-anonymity$

Given a user U, U is said to satisfy *Spatial* $\Gamma-anonymity$ if the probability that the server can infer the position of this user is less than or equal to Γ.

In the LPP system, a global privacy threshold Γ is guaranteed for all users by properly setting system parameters. Given a privacy requirement, there could be more than one applicable system settings. An important step of the system configuration is to define an analytical model of the privacy achieved by our approach. In what follows, let Γ_{LPP} denote the spatial anonymity achieved by the LPP system. We describe how Γ_{LPP} is formulated.

First, let us review the multiple transformation strategy. At each agent, the first transformation function is randomly selected and the following transformation functions are developed from the first function by using λ. We define Γ_{tr_i} as the probability that the first transformation function of agent i is disclosed, and Γ_{λ_i} as the probability that the λ value of agent i is disclosed. To guess one location of a user, the server needs to know the reverse transformation function of the corresponding agent, of which the probability is $\Gamma_{tr_i} \cdot \Gamma_{\lambda_i}$. Then for any user, we have Γ_{LPP} as follows:

$$\Gamma_{LPP} = max_{i=1}^{m}(\Gamma_{tr_i} \cdot \Gamma_{\lambda_i}) \tag{1}$$

where m is the number of agents. We now proceed to present how to obtain Γ_{tr} and Γ_{λ} and analyze possible threats in the LPP system (for convenience, we drop the subscript i from Γ_{tr_i} and Γ_{λ_i}). We will mainly introduce two types of privacy issues: privacy against location discovery and privacy against pattern discovery.

Γ_{tr} largely determines the **privacy against location discovery** since the data transformation is dominated by the first transformation function. To compute Γ_{tr}, we classify the servers into three categories: (i) Servers without any prior knowledge; (ii) Servers with weak prior knowledge; and (iii) Servers with strong prior knowledge.

We denote the user's original position as (x, y). After applying a combination of translation, scaling and rotation (i.e., the first transformation function), we obtain the transformed position (x', y'). The transformation process is formalized as follows:

$$\begin{cases} x' = R_\theta(d_x + s \cdot x) \\ y' = R_\theta(d_y + s \cdot y) \end{cases} \tag{2}$$

where R_θ denotes the rotation (θ is the angle), d_x and d_y are translation parameters, and s is the scaling parameter. The original domains of θ, d and s are denoted as $\mathbb{R}_0$, $\mathbb{D}_0$ and $\mathbb{S}_0$.

If the server does not have any prior knowledge, and in particular it does not even know the type of applied transformation, it is unable to determine (x, y) from (x', y') because the right side of the equation 2 is totally unknown to it. In this case, the probability Γ_{tr} that the server can infer the user's location at this agent is close to 0, which means that user locations have the maximum degree of privacy.

If the server has some weak prior knowledge, for example it knows the type of transformation and some constraints on the application, the original domain of the parameter can be narrowed to some extent. Let $\mathbb{R}$, $\mathbb{D}$ and $\mathbb{S}$ denote the new domains. To find the original location (x, y), the server needs to try all the combinations of the three transformation parameters in the new domains. Here, Γ_{tr} represents an estimate of the possibility of determining the original position. If the values in the domain are discrete, Γ_{tr} can be evaluated by equation 3, where $|\mathbb{R}|$, $|\mathbb{D}|$ and $|\mathbb{S}|$ are the cardinalities of the domains.

$$\Gamma_{tr} = \frac{1}{|\mathbb{R}| \cdot |\mathbb{D}| \cdot |\mathbb{S}|} \tag{3}$$

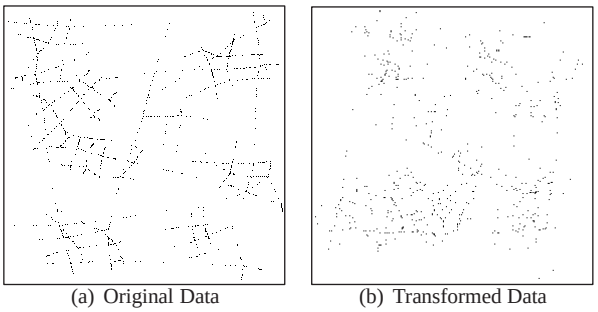
If the values in the domain are continuous, Γ_{tr} can be estimated by the volume of the three domains. Given the range of each domain to be $\mathbb{R} = [R^-, R^+]$, $\mathbb{D} = [D^-, D^+]$ and $\mathbb{S} = [S^-, S^+]$, and the granularity that an application requires to be G, we measure Γ_{tr} by equation 4.

$$\Gamma_{tr} = \frac{1}{(\frac{|R^+ - R^-|}{G} + 1)(\frac{|D^+ - D^-|}{G} + 1)(\frac{|S^+ - S^-|}{G} + 1)} \tag{4}$$

Just having the knowledge of the first transformation function, the server can only infer that the user location is within a certain circle with radius λ. λ is the value that indicates how much a transformed location will deviate from the one strictly preserving the relative distance. Therefore, we define Γ_λ in equation 5. The larger the λ, the harder it is to discover the real location of the user, and privacy is thus better protected.

$$\Gamma_\lambda = \frac{1}{\pi\lambda^2/G} \tag{5}$$

On the other hand, λ also protects **privacy against pattern discovery**. If the server has strong prior knowledge, such information may not only provide information on parameter constraints of the transformation functions, but may also indicate the pattern of distribution of users' locations. However, the identification of such patterns is still a difficult problem for both statisticians and computer scientists [5, 11], and after using our proposed multiple transformation strategy, the problem could become even harder as illustrated in Figure 9. Figure 9(a) shows the original data (about 1K user locations), from which we can clearly observe the road topology. Figure 9(b) shows the transformed data from one agent (3 agents in total), which is transformed by the combination of scaling, rotation and translation. We can see that after transformation, it is hard to identify the pattern; only some dense regions can be seen.



(a) Original Data (b) Transformed Data

Figure 9: Original Data vs. Transformed Data

To sum up, Γ_{tr} gives the probability that the server discover the single transformation function at each agent; and $\Gamma_{tr} \cdot \Gamma_\lambda$ gives the probability that the server discover the multiple transformation strategy. Then, the final Γ_{LPP} is the maximum value of $\Gamma_{tr} \cdot \Gamma_\lambda$ of m agents, which is the probability that the server knows about data transformation at any agent. We would like to mention that Γ_{LPP} is generally very small and can satisfy most privacy requirements. To have some idea of how small this Γ_{LPP} could be, let us look at the following example. Suppose at the agent with the most prior knowledge, the rotation domain has been constrained within 0 degree to 60 degree, the translation domain is [0..10], the scaling value is chosen from 1 to 3, $\pi\lambda^2$ is 10, and the granularity G is 10^{-6}. We can compute that Γ_{tr} is 5.6×10^{-10} and Γ_λ is 0.1. The Γ_{LPP} is about only 6×10^{-11}. On the other hand, we can also see that by adjusting domain size or λ value, the LPP system can achieve a given privacy requirement. The detailed configuration is left to the future work.

Another common threat in network services is eavesdropping during communications. However, we do not consider it in our paper since this type of threat can be mitigated or avoided by data encryption.

5.2 Communication Cost

In our system, there are two types of operations: update and query operations. An update needs one round of communication between a user (agent) and an agent (server). Its communication cost is independent of the number of agents. A query needs one round between a user and m agents, and a server and the m agents. The server returns subquery results to each agent. Suppose the message sizes of a query and a query result set are S_q and S_r respectively. The subquery result size is S_r in the worst case. Then the communication cost of a query is $2m(S_q + S_r)$. Since m determines the privacy level ($m = 1$ i.e. no privacy), the larger the value of m, the higher the privacy level would be. Therefore, a trade-off exists between the communication costs and the privacy level.

The trade-off issues between privacy and communication costs have been widely studied in context of network-level privacy protection. In particular, techniques have been devised to enhance network privacy by increasing the communication costs. For example, in [2, 18], in order to conceal the IP address, network packets have to go through m agents before reaching the receiver. In this case, a complexity of $O(m)$ for communication costs is required.

5.3 Concurrent Processing

We now discuss the effect of concurrent processing in our system compared with systems that do not use any agents. As mentioned in the previous sections, our method converts one query from a user into several small queries. From a user's view, the performance difference lies in "one server handling one big query" versus "one server handling several small queries". The processing time for a small query is obviously short. Due to the limited thread pool, all small queries may not be executed exactly simultaneously. So the time T to get results from all small queries may be a little longer than the time for executing a single small query. We cannot say T is always longer than the time to process a big query. There should be a balancing point. In our case, the balancing point may be found by varying the number of agents (i.e. the number of small queries) when the system configuration is known. In the worst case, small queries are executed in sequence, the query performance is still comparable to that of traditional methods as shown in our results. Further, our approach can be easily applied to multiple-server environments as the sub-databases in the server are relatively independent of one another. If so, the software contention may be reduced to more extent than traditional approaches. In fact, our approach provides increased opportunities for parallel execution.

6 Performance Study

6.1 Experimental Settings

All the experiments were run on a 2.6G Pentium IV desktop with 1Gbyte of memory. The page size is 4K. At the server side we employ the TPR*-tree [21] to index moving objects. The original range query algorithm for the TPR*-tree only supports rectangle ranges. We modified it to support the circle ranges by executing a regular rectangle range query which tightly covers the circle range, and then filtering the extra results. We compare both query and update performance of our model against the pure TPR*-tree. Performance is measured in terms of disk page I/O and CPU time.

We use synthetic datasets of users with positions in a space domain of 1000×1000. One may think of the unit of space being the kilometer. In most experiments, we use uniform data, where users' positions are chosen randomly. We have also run experiments on skewed datasets that follow the exponential distribution. The maximum interval between two successive updates by a user is 120 time units. Unless noted otherwise, we create the initial dataset for all users at time 0, and

Parameter	Setting
Page size	4K
Buffer pages	100
Number of agents	2, **3**, 4, 5, ... , 20
$\lambda/f(r)$	0.01, ..., **0.05**, ... , 0.1
Time interval of changing function	0, 5, **10**, 15, ... , 50
Max update interval	120
Query size (diameter)	10, ..., **50**, ... , 100
Number of neighbors, k	10, 20, 30, ... , 100
Number of queries	100
Data size	**100K**, ..., 1M
Data distribution	**uniform**, exponential

Table 1: Parameters and Their Settings

then evaluate the system performance after the maximum update interval during which each user has issued at least one update.

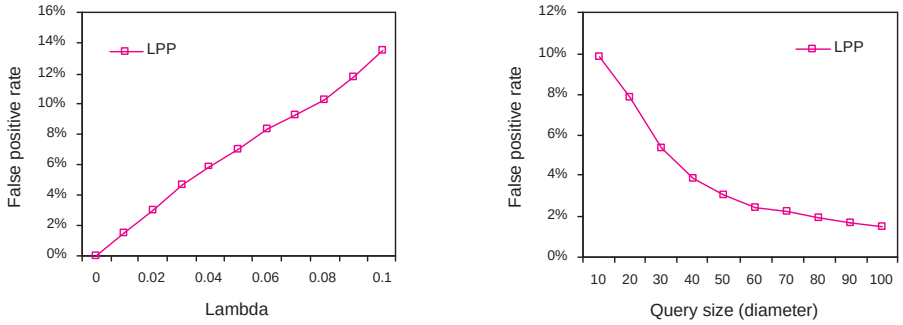
The parameters used in the experiments are summarized in Table 1, where values in bold denote the default values.

6.2 Range Query Performance

6.2.1 Impact of Super Queries

The notion of super query is an important component of our approach with respect to the protection of the map topology. However, super queries may introduce some false positives that may adversely affect performance. In the experiments reported here, we thus investigate the performance impact of the super query by examining the false positive rate. Recall that the false positive rate is the number of query answers filtered by the agent divided by the number of query answers received from the server. The smaller the false positive rate, the less additional work the server and the agent have to carry out.

First, we use the same size of range queries in a 100K dataset, and test the false positive rate when



(a) Varying λ (b) Varying Query Size

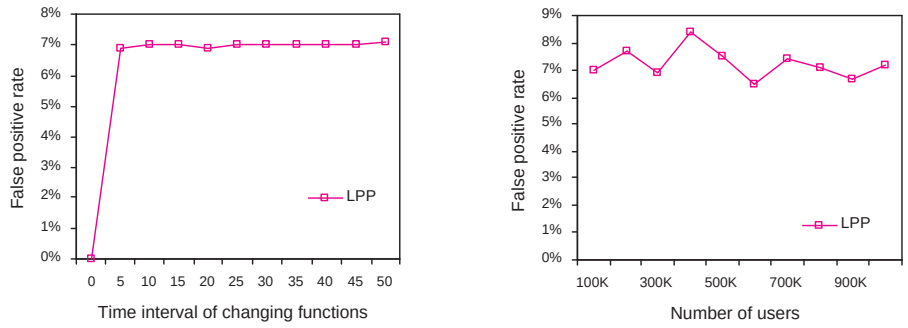
Figure 10: False Positive Rate

varying the values of λ. Figure 10(a) shows the results, where the x-axis is the rate of $\lambda/f(r)$ ($f(r)$ is the query radius after transformation). As expected, the false positive rate increases linearly with $\lambda/f(r)$; a larger λ results in a larger searching space.

Then, we fix the value of λ to 0.5 and vary the query range diameter from 10 to 100. Figure 10(b) shows the corresponding false positive rate. We can observe that the false positive rate decreases when the query size increases. As we know, the higher the value of λ is, the more obscure the transformed data pattern would be. This indicates that the LPP system provides higher privacy and with smaller performance overhead when the query size is large.

Next, we vary the time interval t_{int} between each pair of consecutive transformation functions. As shown in Figure 11(a), the false positive rate for different t_{int} is almost the same. The reason is that the super query is computed based on the first transformation function and the value of λ, and hence the frequency of the transformation function changes does not affect performance.

We also evaluate the false positive rate for values of data size ranging from 100K to 1M. Figure 11(b) shows that the false positive rate oscillates around 7% for different sizes of dataset. This again shows that the false positive rate is dominated by the rate of $\lambda/f(r)$ as stated in Theorem 1.



(a) Varying Time Interval of Changing Functions

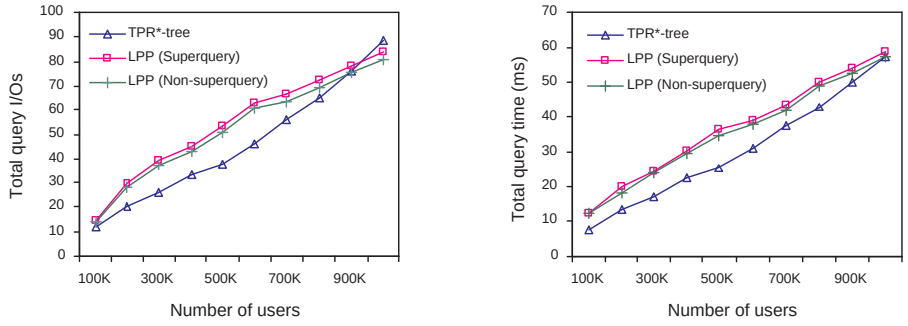
(b) Varying the Number of Users

Figure 11: False Positive Rate

6.2.2 Impact of Data size

In this set of experiments, we vary the data size and analyze the range query performance of the single TPR*-tree and two versions of our model. "LPP (superquery)" denotes the version that uses the concept of the super query; "LPP (non-superquery)" denotes the version that uses the single transformation. The reason for comparing these two versions is to investigate the possible performance degradation incurred by the super query.

Figure 12 compares the query cost of the TPR*-tree and the sum of query cost of all agents in our model. Based on the results reported in the figure, we can make the following observations. First, the performance of the approach based on the super query is quite similar to that of the approach based on the single transformation. The difference between them is less than 3%, which indicates that the use of the super query provides increased privacy protection without compromising query performance. This is an important experimental result that validates a key idea of our approach. In the experiments reported in what follows, we thus only consider the version of our techniques that uses the super query. Second, given m agents, the total query cost of our approach is sometimes a little bit higher but not m times more than that of the TPR*-tree. This is because one query will be sent to all agents according to our schema, and the server needs to compute the transformed queries



(a) Total I/Os (b) Total CPU time

Figure 12: Impact of Data Sizes on Range Query Performance

from all agents. The cost of computing a query from an agent is less than that of evaluating a query in the single TPR*-tree since the query from an agent is executed on a smaller dataset that maintains transformed data from the same agent.

Although our model may incur a little bit higher total query costs, the query response time of our model could be better given that the server supports multi-tasks or there are multiple servers; it can run multiple queries in parallel since each sub-dataset is relatively independent. As shown in Figure 13, the response time of our approach is much smaller than that of the TPR*-tree, and the difference increases with growing data size. This behavior is not surprising. The response time of our approach corresponds to the time required to execute a query from an agent because the server can compute queries from all agents simultaneously. As mentioned previously, the cost to compute a query from an agent is smaller because it is executed on a small sub-dataset.

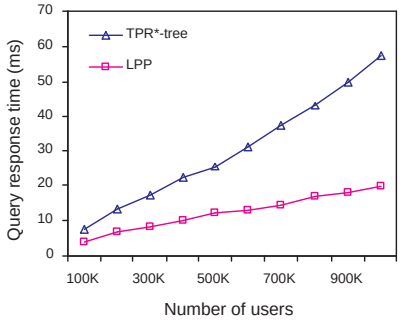
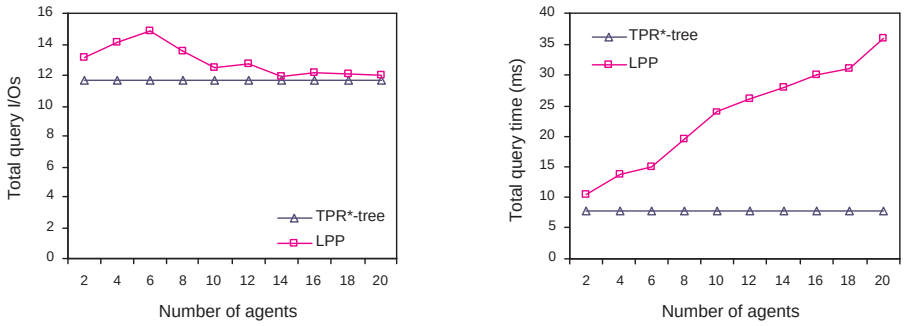


Figure 13: Query Response Time with Varying the Data Size

6.2.3 Impact of Number of Agents

We next study the impact the number of agents has on the query performance. The TPR*-tree is used as the baseline for comparison.

Figure 14 shows the total query cost as a function of the number of agents. We observe that, for our model, the total query I/Os first increases until a point before it decreases and then remains almost constant. Specifically, in the 100K dataset, the total query I/Os starts to decrease when more than 6 agents are used. This behavior can be explained as follows. The total query cost is determined



(a) Total I/Os (b) Total CPU time

Figure 14: Impact of Number of Agents on Range Query Performance

by two factors: the query cost of one agent and the number of agents. When the number of agents increases, the query cost for one agent decreases due to the decreased dataset size with respect to one agent. Therefore, the results can be seen as a combination of the two effects. From the Figure, we observe that their product reach a maximum point, which is 6 in this case.

However, the total query time of our model always increases with the number of agents. This can be explained by observing that the query time does not decrease as fast as the increase of the number of agents. In the TPR*-tree, the number of node accesses can be reduced to a greater extent when the dataset becomes small, while the CPU time decreases much slower as shown in the Figure 15. Figure 15 also indicates that our model may achieve better response time compared with the TPR*-tree. The reason is similar to that we discussed for the previous experiments (Section 6.2.2).

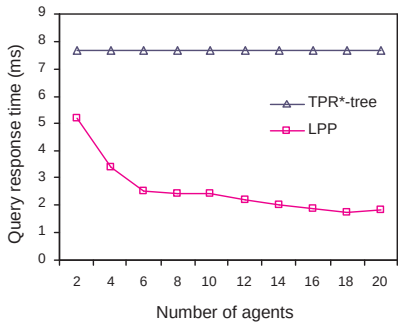
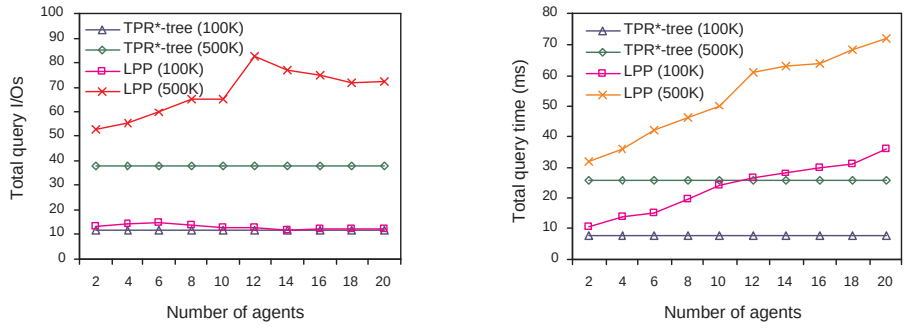


Figure 15: Query response time for varying number of agents

In the following experiments, we explore the combined effect of the number of agents and data sizes. Figure 16 shows the results in the 100K and 500K dataset by using up to 20 agents. We can observe that the performance of 100K and 500K dataset demonstrates similar patterns, while the point at which the query I/O cost starts to decrease is a little bit different, namely 6 agents for 100K dataset, and 12 agents for 500K dataset. This implies that larger datasets may need more agents.

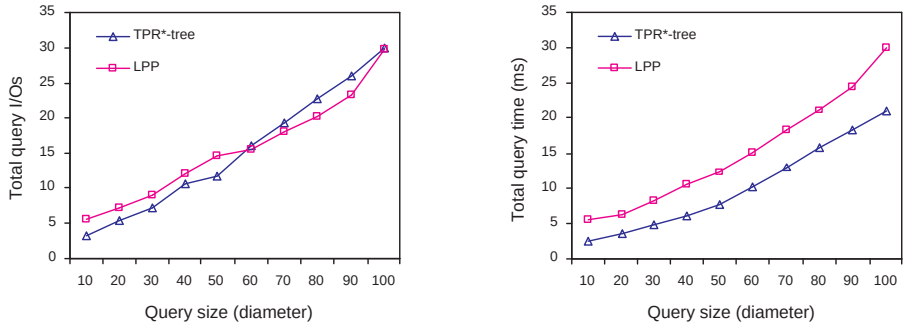


(a) Total I/Os (b) Total CPU time

Figure 16: Impact of Number of Agents and Data Sizes on Range Query Performance

6.2.4 Impact of Query Size

In this section, we analyze the effect of the query size, varying the query diameter from 10 to 100 for a dataset of size 100K. Figure 17 shows that the query costs of both TPR*-tree and the LPP system increase with the query size. The reason is straightforward. Larger query ranges contain



(a) Total I/Os (b) Total CPU time

Figure 17: Impact of Query Size on Range Query Performance

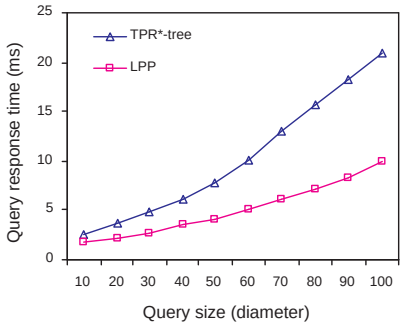


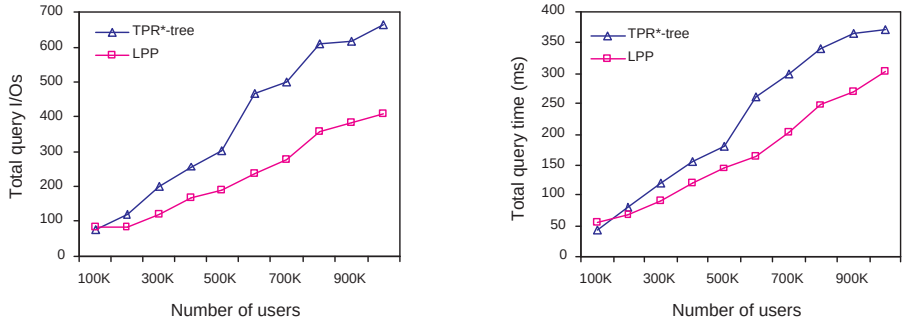
Figure 18: Query Response Time with Varying the Query Size

more objects and therefore lead to more tree node accesses. We can also observe that the total query I/Os of the LPP system is quite close and sometimes less than that of the TPR*-tree, while the total query time of the LPP system is slightly longer.

Figure 18 plots the response time of the TPR*-tree and the LPP system, which shows the similar performance patterns as that of previous experiments.

6.2.5 Impact of Skewed Data

To analyze the query performance on skewed data, we use datasets of exponential distribution with the same skewed parameters from 100K to 1M. Figure 19 shows the experiment results. It is inter-



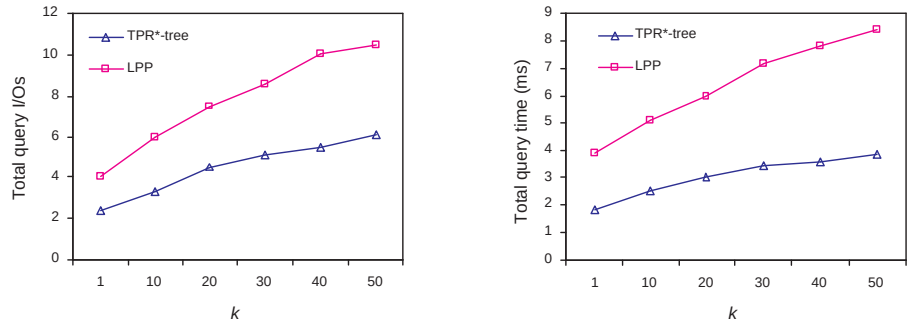
(a) Total I/Os (b) Total CPU time

Figure 19: Impact of Skewed Data on Range Query Performance

esting to see that the LPP system performs much better than the TPR*-tree for skewed data. Both the total I/O cost and CPU time of the LPP system are less than that of the TPR*-tree, and the differences between them increases as the data size increases. A reason for such behavior is as follows. Overlaps among MBRs in the TPR*-tree become more severe when the dataset become skewed and large. The LPP system partitions the dataset into subdatasets with respect to agents, and hence reduces the chance of overlaps which leads to the enhancement of the query performance.

6.3 K Nearest Neighbor Query Performance

We proceed to evaluate the efficiency of kNN queries. Because the kNN query is treated as an incrementally expanded range queries, the performance difference between the TPR*-tree and the



(a) Total I/Os (b) Total CPU time

Figure 20: Impact of k on kNN Query Performance

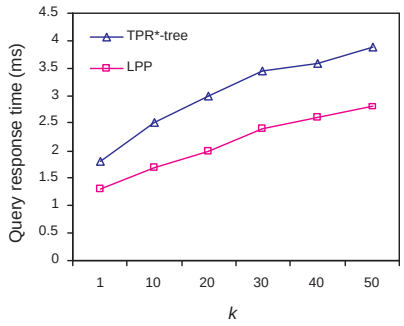


Figure 21: Query Response Time when Varying k

LPP system exhibits a behavior similar to that of range queries when considering the effect of super query, data size, number of agents and so forth. Here, we present a representative result which is the impact of the value of k, that is, the number of required nearest neighbors.

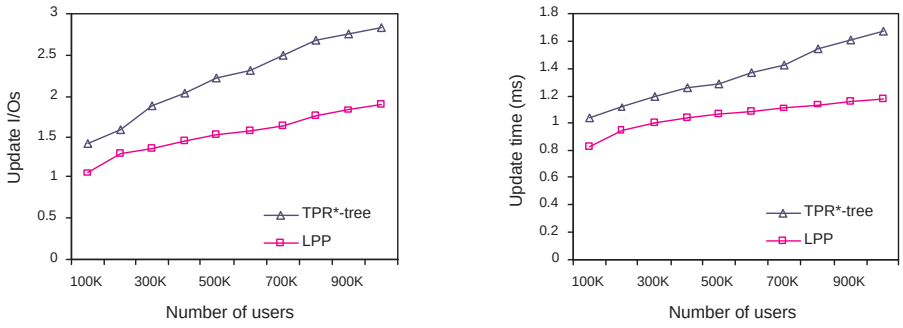
As shown in Figure 20, the total query cost increases for both the TPR*-tree and the LPP system as k increases. The LPP system has higher query cost because the server must execute a kNN query in each sub-dataset corresponding to each agent, and the search range would be bigger for the same k in a smaller dataset. However, the response time of the LPP system could be still better than that of the TPR*-tree as we can observe from the Figure 21.

6.4 Update Performance

We now compare the average update cost (amortized over insertion and deletion) of our model against the TPR*-tree.

6.4.1 Impact of Data Size

First we examine the update performance with respect to the dataset size. We compute the average update cost after the maximum update interval of 120 time units. From Figure 22, we can see that our model achieves better performance than the TPR*-tree. Both the I/O and CPU costs incurred by



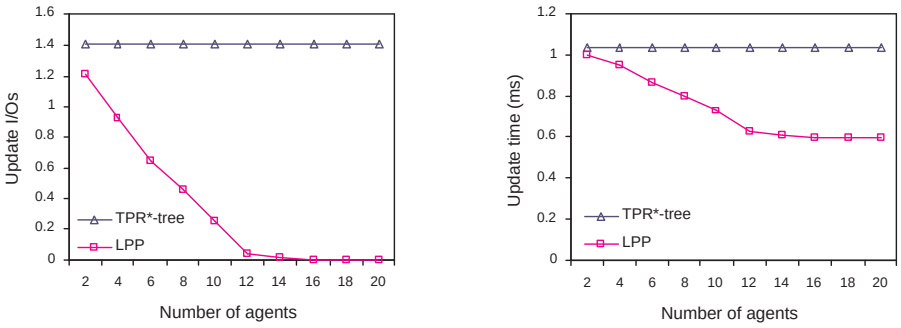
(a) I/Os (b) CPU time

Figure 22: Impact of Data Sizes on Update Performance

our approach are less than that of the TPR*-tree. Moreover, the update cost of our model increases slower than that of the TPR*-tree. A reason is that an update is sent to only one agent. The whole dataset has been partitioned by agents, and then the server handles each update only in a small partition corresponding to the agent, which leads to reduced update costs. This result is very important because the update performance is crucial when dealing with moving object databases where the update frequency is much higher than that of the queries.

6.4.2 Impact of the Number of Agents

In this section, we investigate the update performance of our model when using varying values for the number of agents in the system. Figure 23 shows the I/O and CPU costs of the update. Observe that the update cost of our model is smaller than that of the TPR*-tree and keeps decreasing when the number of agents increases. This is because the user data is distributed among agents. The more agents, the fewer number of data that this agent is responsible for, and hence the dataset of this agent maintained by the server is smaller. It is obvious that an update executed in a smaller dataset would be more efficient.

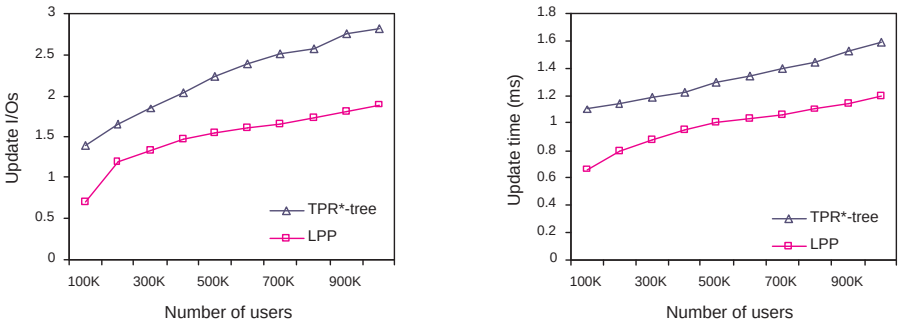


(a) I/Os (b) CPU time

Figure 23: Effect of Number of Agents on Update Performance

6.4.3 Impact of Skewed Data

Finally, we evaluate the update performance in the skewed datasets that we used in the experiments on queries. As shown in Figure 24, both the TPR*-tree and the LPP system have a performance



(a) Total I/Os (b) Total CPU time

Figure 24: Effect of Data Distribution on Update Performance

similar to that of the uniform datasets. A reason for this behavior is that the update cost is already very small and hence less affected by the skewed data.

7 Conclusions and Future Work

In this paper, we propose a novel system framework to address the problems of location privacy in moving-object environments. Our framework achieves both high assurance privacy and good performance. Specifically, our framework uses a number of agents in-between users and servers. Agents are lightweight systems which do not store any user information, but only perform data transformation. In this way, our system can prevent servers from knowing exact locations of users, and even map topology. We have also developed a privacy model to analyze the degree of privacy protection.

We have carried out extensive performance studies to assess the impact of various parameters. We have tested our technique on both uniform and skewed data, and have analyzed the impact of various parameters, such as data size, number of agents, query size. We have also compared the performance of our technique with a traditional approach – the TPR*-tree – which does not consider privacy. The results show that our approach outperforms the TPR*-tree with regards to update operations.

Several promising directions for future work exist. An important extension is the support for continuous queries. Another relevant direction is how to set up system configurations so that the privacy level of the system satisfies a given threshold. Further, we can consider how to satisfy individual privacy requirement in the system. Yet another direction is to refine the proposed privacy protection metrics by taking into account priori knowledge that the adversary may possess, in order to have a better assessment on privacy risks.

Acknowledgement

The work reported in this paper has been supported by UM Research Board under the project “Preserving Location Privacy in Pervasive Environments”, the Research Grants Council of the Hong Kong SAR, China (Project No. HKU 513806E) and the Research Center for Ubiquitous Computing, Central Allocation Group Research Projects (HKBU 1/05C).

References

[1] A. R. Beresford and F. Stajano. Location privacy in pervasive computing. *IEEE Pervasive Computing*, 2(1):46–55, 2003.

[2] D. L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Comm. of ACM*, 24(2):84–88, 1981.

[3] K. Chen and L. Liu. A random rotation perturbation approach to privacy preserving data classification. In *Proc. ICDM’05*, 2005.

[4] R. Cheng, Y. Zhang, E. Bertino, and S. Prabhakar. Preserving user location privacy in mobile data management infrastructures. In *Proc. Workshop on Privacy Enhancing Technologies*, 2006.

[5] D. L. Donoho and X. Huo. Beamlet pyramids: a new form of multiresolution analysis suited for extracting lines, curves, and objects from very noisy image data. In *Proc. SPIE*, pages 434–444, 2000.

[6] B. Gedik and L. Liu. A customizable k-anonymity model for protecting location privacy. In *Proc. IEEE ICDCS*, pages 620–629, 2005.

[7] G. Ghinita, P. Kalnis, A. Khoshgozaran, C. Shahabi, and K.-L.Tan. Private queries in location based services: Anonymizers are not necessary. In *Proc. SIGMOD*, 2008.

[8] M. Gruteser and D. Grunwald. Anonymous usage of location-based services through spatial and temporal cloaking. In *Proc. MobiSys*, pages 31–42, 2003.

[9] U. Hengartner and P. Steenkiste. Protecting access to people location information. In *Proc. SPC*, pages 25–38, 2003.

[10] B. Hore, S. Mehrotra, and G. Tsudik. A privacy-preserving index for range queries. In *Proc. VLDB*, pages 720–731, 2004.

[11] X. Huo and D. L. Donoho. Recovering filamentary objects in severely degraded binary images using beamlet-decorated partitioning. In *Proc. ICASSP*, 2002.

[12] 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.

[13] A. Khoshgozaran and C. Shahabi. Blind evaluation of nearest neighbor queries using space transformation to preserve location privacy. In *Proc. SSTD*, pages 239–257, 2007.

[14] D. Lin, E. Bertino, R. Cheng, and S. Prabhakar. Position transformation: A location privacy protection method for moving objects. In *Proc. ACM GIS Workshop on Security and Privacy*, 2008.

[15] R. P. Minch. Privacy issues in location-aware mobile devices. In *Proc. HICSS*, 2004.

[16] M. F. Mokbel, C. Y. Chow, and W. G. Aref. The new casper: Query processing for location services without compromising privacy. In *Proc. VLDB*, pages 763–774, 2006.

[17] J. M. Patel, Y. Chen, and V. P. Chakka. Stripes: An efficient index for predicted trajectories. In *Proc. ACM SIGMOD*, pages 637–646, 2004.

[18] M. Reiter and A. Rubin. Crowds:anonymity for web transactions. *ACM Trans. On Inform. and Sys. Security*, 1(1):66–92, 1998.

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

[20] E. Snekkenes. Concepts for personal location privacy policies. In *Proc. ACM EC*, pages 48–57, 2001.

[21] Y. Tao, D. Papadias, and Jimeng Sun. The tpr*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*, pages 790–801, 2003.

[22] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *TKDE*, pages 16(10): 1169–1184, 2004.

[23] M. Yiu, Y. Tao, and N. Mamoulis. The b^{dual}-tree: Indexing moving objects by space-filling curves in the dual space. *VLDB Journal*, 2006.

[24] M. L. Yiu, C. S. Jensen, X. Huang, and H. Lu. A random rotation perturbation approach to privacy preserving data classification. In *Proc. ICDE*, 2008.