

# On Monitoring the top- $k$ Unsafe Places

Donghui Zhang Yang Du Ling Hu  
College of Computer and Information Science  
Northeastern University  
{donghui,duy,audnyhu}@ccs.neu.edu

**Abstract**—In a city, protecting units like police cars move around and protect places such as banks and residential buildings. Different places may have different requirements in how many protecting units should be nearby. If any place has less protecting units around than it requires, it is an unsafe place. This paper studies the *Continuous Top- $k$  Unsafe Places (CTUP)* query, which continuously monitors the  $k$  least safe places while the protecting units keep sending their location updates to the server. The CTUP query is a novel addition to the family of continuous location-based queries, an emerging area due to the recent advances in dynamic location-aware environments. Solutions to existing continuous location-based queries and to the traditional top- $k$  queries do not apply. This paper proposes two solutions to this new query, the *BasicCTUP* scheme and the *OptCTUP* scheme. Experiments are conducted to evaluate the proposed solutions.

## I. INTRODUCTION

The main duty of the police is to protect public safety. Police presence is a deterrent to crime. Crime is not randomly distributed in space but instead concentrates in space and time in a predictable fashion [2]. Theoretically, therefore, locating officers where and when crime is concentrated can prevent crime. Response time for officers on patrol to calls is often critical to the apprehension of a criminal. In the vast majority of citizen initiated calls to police, response times are simply too long for a criminal to be apprehended. Sometimes this is due to the nature of the crime (e.g. a burglary that took place hours before a resident returns home to discover what has happened), but sometimes this is due to the location of a patrol car via the location of a crime. In an emergency situation (e.g. a shooting, murder, terrorist activity) police response time can be a critical component to the successful resolution (e.g. apprehension of suspect or ideally the prevention of completion of a crime) of a case.

Because of the predictable nature of crime (in space and time) some places naturally need more protection than others. Some of these places will always have higher than expected rates of crime (e.g. unsupervised parking lots, shopping malls, areas around transit stations etc.) while some may require special action due to an unpredictable occurrence (e.g. a terrorist threat on a specific place like an office building, embassy, airport, or other public place). Some places may require less police attention to prevent crime (e.g. a wealthy suburban neighborhood) while other places may place heavier demands on police resources (e.g. an inner city neighborhood). In some unusual cases, for example, the “Midwest Bank Robbers” [1], criminals may create a distraction to lure the

police away from a location before committing the planned crime.

In light of the above discussion, to enable police agencies to better protect public safety, we propose to study the problem of continuously monitoring the top- $k$  unsafe places. In more detail, in a dynamic environment when police cars move around and send location updates to a server:

**The Continuous Top- $k$  Unsafe Place (CTUP) query** is to continuously monitor the  $k$  places with the smallest *safeties*.

Here the *safety* of a place is defined as the difference between the number of protecting units (i.e. police cars) currently protecting the place and the number of protecting units required to protect it. A protecting unit  $u$  is considered to be protecting a place  $p$ , if the distance between  $u$  and  $p$  is within some given threshold (e.g. 1 mile). Alternatively, let the *protecting region* of  $u$  be a circular region centered at  $u$  with radius being 1 mile,  $u$  protects  $p$  if  $p$  is in the protecting region of  $u$ .

The CTUP query is a novel addition to the family of continuous location-based queries, which has emerged to be an interesting research area due to the recent advances in dynamic location-aware environments [11], [16]. The research area integrates spatial database research with data stream research. Existing continuous location-based queries include the continuous range query [6], [9], [14], [15], [19], the continuous nearest neighbor query [10], [18], [21], [24], [26], the continuous reverse nearest neighbor query [13], [22], and the continuous aggregation query [12], [7]. The model of these queries are: the objects of interest have locations which may change over time, and one or multiple location-based queries are registered whose results may change over time. The server is supposed to continuously monitor the query results. Compared with traditional spatial database queries, the continuous location-based queries deal with moving objects and/or moving queries and the query results need to be continuously monitored. Compared with continuous queries in data stream studies, the continuous location-based queries have a spatial component.

In all continuous location-based queries, there exists a naïve solution: to re-compute the query result (using existing spatial database technologies) upon each location update. But the assumption here is that location updates are received at the server so frequently that this naïve approach is prohibitively expensive.

Compared with existing continuous location-based queries, the new CTUP query is especially challenging because there is not a good monitoring region. Consider the continuous range query. After computing the initial range query result, the server only needs to modify the range query result when some object moves into the query region or when some object moves outside the query region. The size of the monitoring region is typically very small compared with the size of the whole space. Therefore most location updates (which happen completely outside the monitoring region) do not affect the query result. It is very crucial to have a small monitoring region, because a very large monitoring region will degrade the monitoring algorithms to the naïve solution of recomputing everything upon each location update. Unfortunately, in the CTUP query, there does not exist a small monitoring region. One possible monitoring region is the union of the protecting regions of all protecting units. But this monitoring region can be too large, because the police departments aim to protect the whole city.

Top- $k$  query processing is a well studied topic [4], [5], [8], [17], [23], [25], [20]. Two existing top- $k$  processing schemes are particularly related to the CTUP query: Mouratidis *et al.* [17] which proposed an efficient way to continuously monitor the top- $k$  query result over a sliding window, and Yi *et al.* [25] which studied how to efficiently maintain a materialize top- $k$  view given changes in a large base table. However, none of these traditional top- $k$  solutions can be used to efficiently solve the CTUP query, as explained in Section V.

This paper proposes two schemes to solve the CTUP query, a basic version called **BasicCTUP** and an optimized version called **OptCTUP**. The idea is to partition the space into disjoint cells and maintain a lower bound safety for the places in each cell. Let the safety of the  $k^{th}$  unsafe place be denoted as  $SK$ . Upon a location update, if a cell  $C$  has a lower bound more than  $SK$ , there is no need to retrieve the places in  $C$  and compute their safeties.

Implementing the above idea efficiently is not trivial. For instance, BasicCTUP, which implements the idea, has three drawbacks: (a) The lower bounds may decrease unnecessarily; (b) Too many places are maintained in memory; and (c) There exists the “flashing” phenomenon, i.e. a cell may need to be “illuminated” to calculate the safeties of all places in it upon every location update. The optimized scheme OptCTUP, which consists of the *Decrease Once Optimization (DOO)*, can successfully address all the three drawbacks.

The key contributions of this paper can be summarized as follows:

- It proposes the CTUP query, a novel addition to the family of continuous location-based queries. Solving this query may have a strong impact on the society because it may help the police to more efficiently combat crime and to better protect public safety.
- It proposes two solutions to the CTUP query. The OptCTUP scheme can successfully address all the three identified drawbacks of the BasicCTUP scheme. In par-

ticular, to address the problem that cell lower bounds decrease too fast, the DOO optimization is proposed.

- It experimentally evaluates the proposed schemes and reveals some insights on how to choose certain system parameters.

The rest of the paper is organized as follows. Section II formally defines the CTUP problem. When the places are modeled as points, Section III describes the BasicCTUP scheme, and Section IV presents the OptCTUP scheme. Section V discusses related work on continuous location-based queries and top- $k$  queries and why existing solutions do not apply to our problem. Performance studies appear in Section VI. Finally, Section VII concludes the paper and summarize our future work.

## II. MODEL AND PROBLEM DEFINITION

This section first introduces the data model assumed in this paper, then formally defines the CTUP problem.

### A. Model

Assume a centralized server exists. The server stores the locations and shapes of all places, and keeps track of the most-recently-reported locations of protecting units. The protecting units move around and send location updates through some wireless channel to the server whenever necessary (e.g. one meter away from the location reported previously). Following the storage model in [18], [26], the 2D space, formed by longitude and latitude, is partitioned into disjoint cells, e.g. by a grid consisting of  $X \times Y$  cells. The protection area of a protecting unit is a circular area which is centered at the unit and has a radius of  $R$ . The places inside the circle are protected by the unit, while the places outside the circle are not.

Our storage model uses a two-level structure, which separates the storage of all places from other data. The lower level stores all places along with their geographic and descriptive information (e.g. map of the city and various annotations). The data in the lower level are infrequently updated. The higher level maintains all units, the information of cells, and a very small fraction of places (with their safeties). The data in the higher level change continuously.

In real applications, the number of places is typically very large. Therefore, the set of places may not fit in memory. That is why in Fig. 1 and Fig. 2 the two levels are denoted as “memory” and “disk”. However, our model also applies to the case when the set of places fit in memory. In both cases, our two-level model achieves efficiency (I/O efficiency and computational efficiency, respectively). If there is a large amount of places and they can not fit in memory, the lower level corresponds to the disk and the higher level corresponds to memory. To access some place in a cell, unless the place is already maintained in memory, all places in the cell are loaded from disk. If the memory is large enough to hold all places, this model divides the memory into two pieces and utilizes one piece to simulate disk. It still improves the computational

efficiency. The reason is that, upon each location update of a unit, it is too expensive to compare (the old and new locations of) the unit with *all* places. So we compare it with the small fraction of places maintained in the higher level, but only compare it with places stored in the lower level when absolutely necessary.

### B. Problem Definition

Let  $U$  be a set of protecting units (or units in short) such as police cars. Each unit has a location and a circular protection region (e.g. the area within 1 mile to the unit). Let  $P$  be a set of places that need to be protected. The definitions below assume the places are modeled as points. The definition and solution to the problem when the places have extent are omitted due to space limitations.

*Definition 1:* A unit  $u \in U$  **protects** a place  $p \in P$  if  $p$  is spatially contained in the protecting region of  $u$ .

Independent to where the units are placed, each place has a required protection. Different places may have different required protections. For instance, a bank may require six police cars nearby, while a residential building may require one.

*Definition 2:* The **required protection**  $RP(p)$  of a place  $p \in P$  is the required number of units in  $U$  that protect  $p$ . The **actual protection**  $AP(p)$  of a place  $p \in P$  is the total number of units in  $U$  that are currently protecting  $p$ .

*Definition 3:* The **safety** of place  $p \in P$  is the difference between its actual protection and its required protection, *i.e.*  $safety(p) = AP(p) - RP(p)$ .

A negative safety indicates that the actual protection a place receives does not meet its requirement. The smaller  $safety(p)$  is, the less safe  $p$  is. This paper addresses the continuous top- $k$  unsafe place query.

*Definition 4:* The **Continuous Top- $k$  Unsafe Place (CTUP) query** continuously monitors the  $k$  places in  $P$  with the smallest safeties.

For ease of presentation we define a special variable  $SK$ . The safety of the  $k^{th}$  unsafe place is denoted as  $SK$ . Any place with safety less than  $SK$  is a top- $k$  unsafe place. Any place with safety more than  $SK$  is not a top- $k$  unsafe place.

## III. THE BASIC SOLUTION

This section proposes the **BasicCTUP** scheme, which is the basic version of our proposed solution for the CTUP query. The scheme is composed of maintained information, initialization algorithm and update algorithm.

### A. Maintained Information

Fig. 1 illustrates the information that is maintained in memory by the BasicCTUP scheme. The space (e.g. map of a city) is spatially partitioned into a set of non-intersecting cells. A cell is either *dark* or *illuminated*. Most cells are *dark*,

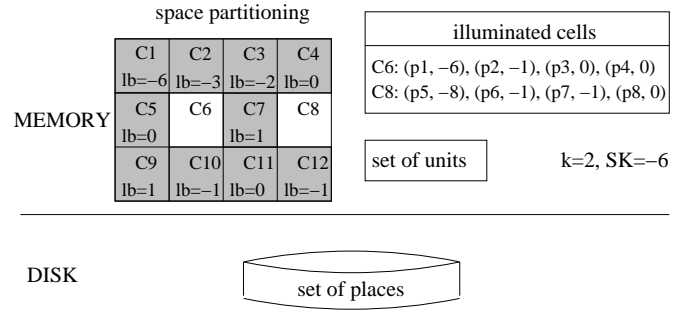


Fig. 1. Illustration of BasicCTUP.

which means that the places in them and the safeties for these places are not maintained. However, for each dark cell, a lower bound of the safeties among all places inside it is maintained. For an *illuminated* cell, we maintain the locations and safeties for all places in it. Also maintained in memory is the set of all units.

In the example of Fig. 1, the space is partitioned into 12 cells  $C_1, \dots, C_{12}$ . A dark cell (e.g.  $C_1$ ) maintains a lower bound (e.g.  $C_1.lb = -6$ ). It is guaranteed that in cell  $C_1$ , no place has safety less than  $-6$ . For the two illuminated cells  $C_6$  and  $C_8$ , the places in them, along with their safeties, are maintained in memory.

For efficient query support, BasicCTUP ensures that every cell that contains some top- $k$  unsafe places is illuminated. This way, the top- $k$  unsafe places (and their safeties) are maintained at all times, therefore are continuously monitored. In Fig. 1,  $k = 2$ , and the top- $k$  unsafe places are  $p_5$  with safety =  $-8$  and  $p_1$  with safety =  $-6$ . The safety of the  $k^{th}$  unsafe place is  $SK = -6$ .

### B. Initialization

The initialization algorithm described as follows is executed only once.

- 1) For every cell, derive the protecting units whose protecting regions intersect the cell. Load the places in the cell to memory and compute their safeties. Take the smallest safety as the lower bound of the cell. Keep this lower bound but discard the places from memory. Mark all cells dark.
- 2) Illuminate the cells in increasing order of lower bound, adjusting the value of  $SK$  as the safety of each place is calculated.
- 3) Terminate when  $SK \leq$  the lower bounds of all dark cells.

### C. Update

After initialization, the following update algorithm is executed whenever the server receives a location update from some unit.

- 1) Update the safeties of maintained places, *i.e.* the places in the illuminated cells.

- 2) Update the lower bound of every dark cell. This step is non-trivial and will be discussed in detail after the update algorithm.
- 3) For every dark cell whose lower bound  $< SK$ , illuminate it.
- 4) For every illuminated cell which does not contain any top- $k$  unsafe place, *darken* it, i.e. discard all places in it from memory.

We need to illuminate a dark cell whose lower bound  $< SK$ , because the cell may contain some top- $k$  unsafe places. We need to darken an illuminated cell which does not contain any top- $k$  unsafe place, because otherwise the algorithm may end up maintaining too much information.

The remaining of this section focuses on discussing how to maintain the lower bound of a cell upon one location update of some unit.

old \ new	N/P	F
N	0	+
P	-	0
F	-	0

TABLE I  
LOWER-BOUND MAINTENANCE IN BASICCTUP.

The protecting region of a unit has three possible relationships with a cell: not intersect (N), partially intersect (P), and fully contain (F). Table I describes how the lower bound of a cell is affected in different cases.

- **N  $\rightarrow$  N/P:** The old protecting region does not intersect a cell, and the new protecting region either does not intersect the cell or only partially intersects it. The cell lower bound should remain the same (denoted as 0).
- **N  $\rightarrow$  F:** The old protecting region does not intersect a cell, and the new protecting region fully contains the cell. Every place in the cell will be safer. So the cell lower bound is increased by 1 (denoted as +).
- **P  $\rightarrow$  N/P:** It is possible that a place is contained in the old protecting region but not the new one. So the cell lower bound should decrease by 1 (denoted as -).
- **P  $\rightarrow$  F:** Some place may have the same safety before and after the location update. So the cell lower bound should remain the same.
- **F  $\rightarrow$  N/P:** The cell lower bound should decrease by 1.
- **F  $\rightarrow$  F:** The cell lower bound should remain the same.

#### IV. THE OPTIMIZED SOLUTION

The basic scheme is more efficient than the naïve algorithm which maintains the safeties of all places, because a location update typically does not need to check individual places in the dark cells. However, it still has some efficiency problems.

This section first identifies three drawbacks of BasicCTUP, then proposes the **OptCTUP** scheme which successfully addresses all the three identified drawbacks.

#### A. The Drawbacks of BasicCTUP

- **Drawback one:** *The lower bounds of dark cells may decrease unnecessarily.* Suppose a unit  $u$  updates its location 20 times while remaining to partially intersect with a cell  $C$ . According to the P $\rightarrow$ N/P case in Table I, the lower bound of  $C$  should be decreased by 20. But intuitively, the correct way is to decrease by one. Another scenario of unnecessary decrease of cell lower bound is when a unit alternates between partially intersecting a cell and fully containing a cell. In Table I, although P $\rightarrow$ F corresponds to 0, F $\rightarrow$ N/P corresponds to -. So if the unit alternates 20 times, the cell's lower bound will be decreased by 10. To unnecessarily decrease cell lower bounds is not efficient, because it unnecessarily increases the number of times to illuminate cells.
- **Drawback two:** *Too many places are maintained in memory.* BasicCTUP maintains in memory all places in all illuminated cells. Many of such maintained places will never be a top- $k$  unsafe place and therefore maintaining them is wasteful. Consider the example of Fig. 1. While  $p_5$  (e.g. a bank whose required protection is high) is very unsafe, all the other places in the same cell (e.g. residential buildings) are quite safe and are unnecessarily maintained.
- **Drawback three:** *There exists the "flashing" phenomenon.* That is, a cell may need to be illuminated and darkened too frequently. Consider a dark cell  $C$  which contains a place whose safety is equal to (or slightly larger than)  $SK$ . One (or a few) update may decrease  $C$ 's lower bound to below  $SK$ . So  $C$  needs to be illuminated. But after illumination, BasicCTUP may find that no place in  $C$  is top- $k$  unsafe, and so it darkens  $C$ . It is obviously very inefficient if every update illuminates and then darkens a cell. For example, in Fig. 1, since  $C_1.lb = SK$ , a single update which decreases  $C_1.lb$  will need to illuminate  $C_1$ . If after illumination, no place in  $C_1$  has safety  $< -6$ , BasicCTUP will darken  $C_1$ . Then another update will illuminate  $C_1$  again, and so on.

The OptCTUP scheme addresses drawback one, by integrating an optimization technique called DOO, which ensures that a cell's lower bound is decreased at most once due to the movement of one unit. To address drawback two, OptCTUP keeps all cells dark, while selectively maintaining some unsafe places. The lower bound of a dark cell is calculated without considering the maintained places. To address drawback three, OptCTUP requires that immediately after illuminating a cell, the cell's lower bound should be  $\geq SK + \Delta$  for some system parameter  $\Delta$ .

#### B. The Decrease Once Optimization (DOO)

The **Decrease Once Optimization (DOO)** ensures that the lower bound of a cell  $C$  is decreased at most once due to multiple location updates for a single unit  $u$ . The idea is to

insert  $(u, C)$  into a hash table denoted as *DecHash* upon the first decrease of  $C$ 's lower bound according to the location update of  $u$ . When  $u$  updates its location again, because  $(u, C)$  is already in *DecHash*, the lower bound of  $C$  does not decrease. To implement this idea, we also need to take care of the cases when  $(u, C)$  needs to be removed from *DecHash*.

old \ new	N/P	F
N	0	$+, h^-$
P	0 (if in hash) $-, h^+$ (otherwise)	$+, h^-$ (if in hash) 0 (otherwise)
F	$-, h^+$	0

TABLE II  
LOWER-BOUND MAINTENANCE IN OPTCTUP.

The scheme for maintaining cell lower bounds after applying DOO is illustrated in Table II. Here  $h^+$  stands for inserting  $(u, C)$  into *DecHash*, and  $h^-$  stands for deleting  $(u, C)$  from *DecHash*. In more details:

- **N**  $\rightarrow$  **N/P**: 0 (as in BasicCTUP).
- **N**  $\rightarrow$  **F**:  $+$  (as in BasicCTUP), but should be followed by an attempt to remove  $(u, C)$  from *DecHash*. The reason is that, after increasing the lower bound of  $C$ , it should appear as if  $u$  has never been used to decrease the lower bound of  $C$ , and therefore  $(u, C)$  should be removed from the hash table to enable decreasing  $C$ 's lower bound using  $u$  in the future.
- **P**  $\rightarrow$  **N/P**: This case plays the biggest role in illustrating that using DOO is better. In BasicCTUP, this case always decreases  $C$ 's lower bound. In OptCTUP which uses DOO, the lower bound is decreased only if  $(u, C)$  is not in *DecHash*. The decreasing is accompanied by inserting  $(u, C)$  into *DecHash*, so that future location updates of  $u$  do not decrease  $C$ 's lower bound again.
- **P**  $\rightarrow$  **F**: While in BasicCTUP, this case does not change  $C$ 's lower bound, in OptCTUP the lower bound may increase, in case  $(u, C) \in \text{DecHash}$ .
- **F**  $\rightarrow$  **N/P**:  $-$  (as in BasicCTUP), but should be followed by inserting  $(u, C)$  into *DecHash* to record the fact that  $u$  has already been used to decrease  $C$ 's lower bound.
- **F**  $\rightarrow$  **F**: 0 (as in BasicCTUP).

### C. Maintained Information

Fig. 2 illustrates the information which is maintained in memory by the OptCTUP scheme. Compared with the BasicCTUP scheme (Fig. 1), the differences are:

- To solve Drawback 1, a new data structure called *DecHash* is maintained.
- To solve Drawback 2, all cells are dark. Instead of maintaining "illuminated cells", OptCTUP maintains some places. For instance, in Fig. 1 to maintain the illuminated cell  $C_8$ , the places  $(p_6, -1)$ ,  $(p_7, -1)$ , and  $(p_8, 0)$  are maintained unnecessarily. And in Fig. 2, for  $C_8$  only  $p_5$  is maintained.

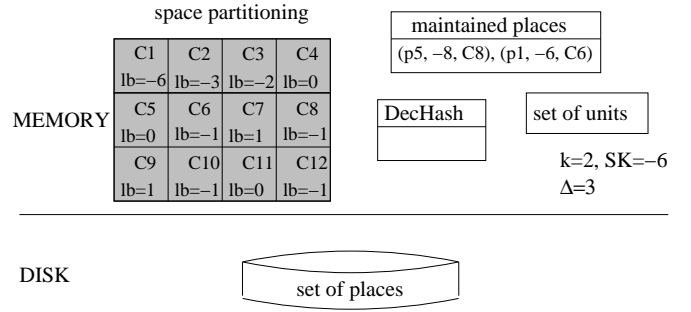


Fig. 2. Illustration of OptCTUP.

- The lower bound of a cell is computed without considering the maintained places. For instance, the lower bound of cell  $C_8$ , which is shown as -1, is computed without considering  $p_5$ .
- To solve Drawback 3, a new variable  $\Delta$  is introduced. If an update decreases the lower bound of  $C_1$ , the new lower bound will be less than  $SK$  and therefore OptCTUP will access  $C_1$ . To avoid the "flashing" phenomenon, every place in  $C_1$  whose safety  $< SK + \Delta$  ( $=-3$  in Fig. 2) will be maintained in memory. Now that the new lower bound of  $C_1 \geq SK + \Delta$ , the lower bound can decrease  $\Delta$  times before  $C_1$  needs to be accessed again.

### D. Initialization

- 1) For every cell, derive the protecting units whose protecting regions intersect the cell. Load the places in the cell to memory and compute their safeties. Take the smallest safety as the lower bound of the cell. Keep this lower bound but discard the places from memory.
- 2) Access the cells in increasing order of lower bound. To access a cell, load all places in it to memory and compute their safeties. Adjust the value of  $SK$  as the safety of each place is calculated. All places in these accessed cells will be kept in memory temporarily. Stop accessing new cells when  $SK \leq$  the lower bounds of all non-accessed cells.
- 3) For each accessed cell, remove from memory the places in it whose safeties  $\geq SK + \Delta$ . Change the lower bound of this cell to be the minimum safety of the removed places.
- 4) Set *DecHash* =  $\emptyset$ .

### E. Update

- 1) Update the safeties of maintained places.
- 2) Update the lower bound of the cells intersecting the old or new protecting regions of the unit whose position changed. The update strategy should follow Table II.
- 3) For every cell whose lower bound  $< SK$ , load the places in it to memory and calculate their safeties, adjust  $SK$ , remove from memory the places in this cell whose safeties  $\geq SK + \Delta$ , and set the cell's lower bound to be the minimum safety of thus removed places.

## V. RELATED WORK

### A. Continuous Location-based Queries

Due to the recent advances in dynamic location-aware environments [11], [16], various types of continuous location-based queries have been studied. The models of these queries are: the objects of interest have locations which may change over time, and one or multiple location-based queries are registered whose results may change over time. The server is supposed to continuously monitor the query results. An assumption is that the location updates come so frequently that re-computing the results from scratch upon each update is not acceptable. Existing continuous location-based query processing algorithms utilize the fact that a *small monitoring region* exists.

The continuous range query [6], [9], [14], [15], [19] monitors the objects that are spatially contained in each query region. The idea for efficient updates is to pay attention only to location updates that cross the boundary of the query region. That is, either when an object moves into the region, or when an object moves out of the region. Here the monitoring region is the query region. Most updates happen outside the region and do not affect the query result that is being continuously monitored.

The continuous nearest neighbor query [10], [18], [21], [24], [26] monitors the nearest neighbor(s) of each query location  $q$ . Here the monitoring region is a circle whose center is  $q$  and whose radius is the distance from  $q$  to its nearest neighbor. The nearest neighbor query result needs to be updated only if an object moves inside the monitoring region (and becomes the new nearest neighbor), or when the previous nearest neighbor moves away from  $q$ . The scheme can be extended to monitor  $k$  nearest neighbors instead of one. The idea is to enlarge the monitoring region such that it contains  $k$  objects.

The continuous reverse nearest neighbor query [13], [22] monitors the reverse nearest neighbors of each query location  $q$ , i.e. the set of objects who consider  $q$  as their nearest neighbor. There are two versions of the problem: the monochromatic case and the bichromatic case. The monochromatic case [22] considers a single data set. An object  $o$  is a reverse nearest neighbor of  $q$ , if  $o$  is closer to  $q$  than to any other object. The bichromatic case [13] involves two data sets, e.g. a set of mobile users and a set of wireless stations. A mobile user is a reverse nearest neighbor of  $q$ , if the user is closer to  $q$  than to any wireless station. In both cases, the monitoring region is more complicated than the monitoring regions in the continuous range query and nearest neighbor query. For instance, the monitoring region for the monochromatic case [22] for one query location is composed of six pie-shaped regions and six circular regions. Nevertheless, the size of such monitoring regions is still small compared with the size of the whole space.

The continuous aggregation query [12] is similar to the continuous range query in that a query region is given. The difference is that, instead of monitoring the actual objects in the query region, it monitors the number of objects in the

region. A reverse version is to monitor the dense areas [7], i.e. the regions which has sizes smaller than some given threshold and which contain at least a given number of objects.

The CTUP query studied in this paper is a new continuous location-based query. It is different from all the existing continuous location-based queries in that there is not a small monitoring region. The existing queries typically have small monitoring regions, therefore can utilize the following algorithm: “only process the small fraction of location updates that happen inside the monitoring region.” But in the new CTUP query, there does not exist a small monitoring region. A natural monitoring region is the union of the protecting regions of all protecting units. But this monitoring region is large. If the police department distributes the protecting units in such a way that they cover most area of the whole city, the monitoring region is almost as large as the whole space. With a large monitoring region, the existing algorithm degenerates to the naïve approach of re-computing the query result upon (almost) every location update, which is prohibitively expensive.

### B. Top- $k$ Queries

Given a dataset where every record has  $d$  attributes, the **top- $k$  query** specifies a preference function that maps each record to a numeric score, and returns the  $k$  records which the largest preference scores. In the literature the top- $k$  query was also called the *top- $k$  model-based query* [4] and the *ranked query* [8].

Very related to our work is Mouratidis *et al.* [17], which proposed an efficient way to continuously monitor the top- $k$  query result over a sliding window. That is, it monitors, among all recent records (in a given sliding window), the  $k$  records with the largest preference scores. Here the sliding window can be either time-based or count-based. In the time-based sliding window, a record expires if it entered the sliding window at least  $T$  time units ago. In this case, the number of records in the sliding window can vary, depending on the speed of the stream. As a comparison, the size-based sliding window keeps a fixed number of records. The proposed approach partitions the attribute space into axis-parallel cells. For instance, if every record has two attributes, using these two attributes as dimensions a 2D space is formed and then partitioned. Assume the preference function is monotone, e.g. to increase the value of any attribute of a record always leads to a larger (or equal) preference score. Then in every cell, a lower bound of the preference score occurs at its lower-left corner, and an upper bound of the preference score occurs at its upper-right corner. Furthermore, if cell  $L$  is located to the lower left of another cell  $U$ , any object in  $L$  will have a smaller (or equal) value than an arbitrary object in  $U$ . Using these properties an efficient algorithm was proposed, which first examines the cell located to the upper-right of the whole space, then examines the cells next to that upper-right cell, and so on.

At first glance, our proposed CTUP query is similar to [17] in that they both continuously monitor top- $k$  query result and they both utilize space partitioning. However, there are several differences. First, CTUP does not have the notion of sliding

window, and therefore never expires objects. Secondly, CTUP deals with two data sets (the set of places and the set of protecting units), while [17] involves a single data set. Thirdly, CTUP considers spatial objects, not records in a traditional database. For instance, CTUP needs to calculate the distance between a place  $p$  and a protecting unit  $u$  to determine whether  $u$  protects  $p$ . Finally, the spaces being partitioned in CTUP and [17] are different. In [17], the space is formed by the attributes of the records. If each record has five attributes, [17] partitions a 5D space. Our work partitions the 2D map, where one dimension is longitude and the other dimension is latitude. If we want to utilize the solution of [17] to solve our problem, we would need to partition a different space. In particular, since the preference score for each place is the safety of the place, which is computed by its actual protection subtracted by its required protection, we would partition the 2D space with dimensions actual protection and required partition. This approach is problematic, because the actual protection of an object is not fixed. But in [17], the value for each attribute of a record remains constant.

Also very related to this paper is Yi *et al.* [25]. It studied how to efficiently materialize the top- $k$  view for a large base table. The top- $k$  view enables the system to answer the top- $k$  query without examining the base table. However, the top- $k$  view is not self-maintainable. Deletion or update may cause a tuple to leave the top- $k$  list. In this case, the view needs to be ‘refilled’ by executing the top- $k$  query, which incurs expensive I/Os to the base table. To avoid frequent ‘refill’ query execution, the proposed solution is to maintain top- $k'$  objects in the view instead of top- $k$  objects, where  $k'$  changes at runtime between  $k$  and some pre-determined  $K > k$ . Initially  $k' = K$  and the top- $K$  tuples are maintained in the view. If a deletion or update causes a tuple to leave the current top  $k'$ , decrease  $k'$  by 1. The view is refilled to  $K$  whenever  $k'$  drops below  $k$ . Based on this solution, the authors proposed the cost model to measure the amortized cost of this simple algorithm. Their analysis indicates the optimal value of  $K$  under various circumstances and shows how to choose  $K$  dynamically according to the workload.

In the CTUP query, conceptually each place can be regarded as a tuple in some base table with an attribute *safety*, and then the CTUP query is logically reduced to maintaining top- $k$  views solvable by traditional top- $k$  algorithms such as [25]. However, in the CTUP query there does not actually exist such a base table telling us the safety of each place. The naïve approach of maintaining the base table, i.e. the safeties for all places, is prohibitively costly.

There are some other good solutions to the traditional top- $k$  query in the literature. The *Onion technique* [4] focuses on linear preference functions and preprocesses the objects into a set of layers based on the concept of *convex hull*. To retrieve top- $k$  objects, the Onion technique retrieves objects in increasing layer numbers while maintaining the maximum preference score of objects in the current layer. The *PREFER* system [8] stores the materialized views of multiple sorted lists of objects, each based on some preference function. At

query time, the preference function most similar to the query preference function is selected and the corresponding view is examined. Typically more than  $k$  records in the materialized view are needed to find the query results. But intuitively the more similar the two preference function are, the fewer records need to be examined. Fagin *et al.* [5] proposed the *TA algorithm* and the *NRA algorithm*. Both algorithms merge  $d$  sorted lists, one per attribute of the object. While scanning down the  $d$  lists in parallel, the upper bound for the preference score of all unseen objects is maintained, which enables the algorithms to report top- $k$  objects progressively, without finishing examining all objects. The difference between TA and NRA lies in whether random access is allowed to retrieve an object’s scores for other attributes if the object is seen in one sorted list.

The top- $k$  query has also been studied in some other environments. Xia *et al.* [23] proposed a method for computing the top- $k$  most influential spatial sites. The problem considers two data sets, a set of objects and a set of sites. The *influence* of a site is the size of its reverse nearest neighbor set, i.e. the number of objects closer to the site than to any other site. Given a spatial region, the goal is to find the  $k$  sites inside the query region with the largest influences. An efficient algorithm was proposed based on the R-tree index. Soliman *et al.* [20] studied the top- $k$  query in uncertain databases.

## VI. PERFORMANCE

This section experimentally compares three algorithms:

- **Naïve**: The straightforward method of recomputing the *safety* of all places upon each update.
- **BasicCTUP**: proposed in Section III.
- **OptCTUP**: proposed in Section IV.

<i>Parameter</i>	<i>Default Value</i>
Number of units ( $ U $ )	<b>150</b>
Number of places ( $ P $ )	<b>15K</b>
Number of TUPs ( $k$ )	<b>15</b>
Adjustable Parameter ( $\Delta$ )	<b>6</b>
Unit Protection Range	<b>0.1</b>
Partition Granularity	<b>10</b>

TABLE III  
DEFAULT PARAMETER VALUES.

The algorithms are implemented in Java, running on a Dell PC with a Pentium IV 3.2GHz processor and 4GB memory. The OS is Windows XP. The experimental data are generated using the *Network-based Generator of Moving Objects* [3] in the following way. The protecting units are generated as objects moving along the roadnet of Oldenburg, Germany. The places are randomly generated. The default parameter values are shown in Table III. The parameter settings of the following experiments will not be listed if the default values are used. We first compare the initialization time and average update time of the naïve algorithm against BasicCTUP and OptCTUP algorithms. Then we evaluate the update cost of

BasicCTUP and OptCTUP by varying different parameters. The DOO optimization algorithm is examined in Section VI-B. The effectiveness of  $\Delta$  is measured in OptCTUP algorithm with DOO optimization.

### A. Initialization and Update Cost

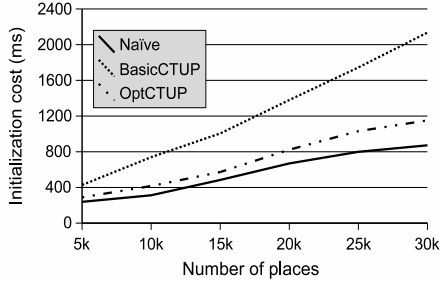


Fig. 3. Comparison of initialization time.

Fig. 3 compares the initialization time cost of the three algorithms. The Naïve algorithm has the shortest initialization time since it does not need to compute and maintain any information other than the result top- $k$  unsafe places. OptCTUP is close and BasicCTUP is the worst. Both of them need to also maintain cell lower bound information.

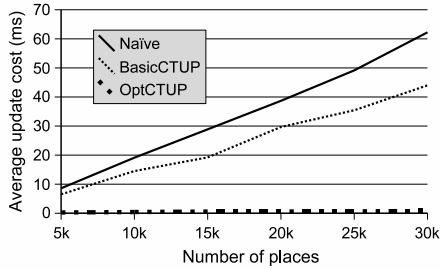


Fig. 4. Comparison of update cost.

Fig. 4 compares the update cost. OptCTUP clearly outperforms the other two approaches by a large margin. BasicCTUP is better than Naïve, but is still much worse than OptCTUP.

Fig. 5, Fig. 6 and Fig. 7 compare the update cost of BasicCTUP and OptCTUP while varying  $k$ , the partitioning granularity, and the protection range. The cost of the algorithms are shown in logarithmic scale. Again OptCTUP is clearly superior because it prevents the lower bound of cells from dropping too fast and reduces the number of maintained places.

### B. The Effect of DOO

Fig. 8 measures the effect of the Decrease Once Optimization (DOO) which is used to update the *lower-bound* of cells. The figure shows the update cost of the OptCTUP algorithm with and without applying DOO. Using DOO is clearly better than its counterpart, especially as the number of places increases. DOO is an effective way to prevent the *lower-bound* of cells from decreasing too fast.

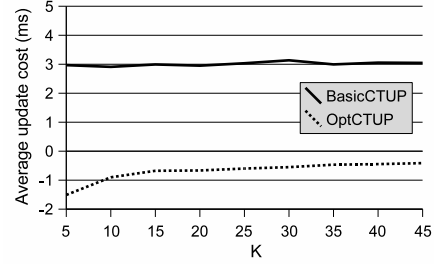


Fig. 5. Update cost varying  $k$ .

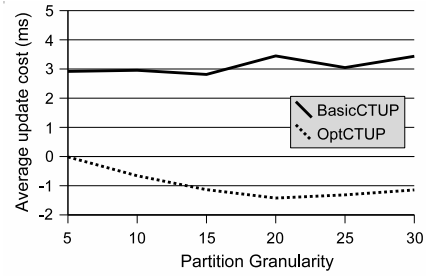


Fig. 6. Update cost varying partitioning granularity.

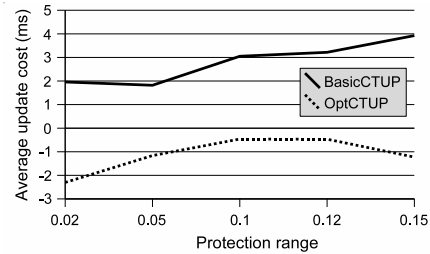


Fig. 7. Update cost varying protection range.

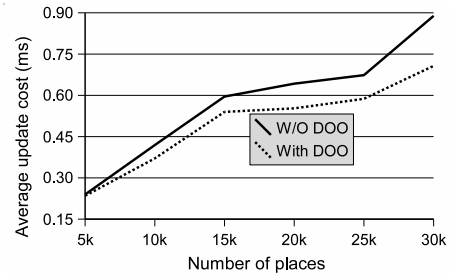


Fig. 8. Update cost varying the number of places.

### C. The Effect of $\Delta$

Fig. 9 measures the update cost when varying the value of  $\Delta$ . Further, the update cost is broken down into two parts: the cost to modify the maintained information, and the cost to access cells. As  $\Delta$  increases, more information needs to be maintained, and therefore the cost to modify the maintained information increases, while the cost to access cells decreases. In the experiments the data in the cells are also kept in memory and therefore the cell access cost is relatively small. It is expected that if the data in the cells are actually stored on disk, the cell access time will increase significantly, but the observed trends should be similar.

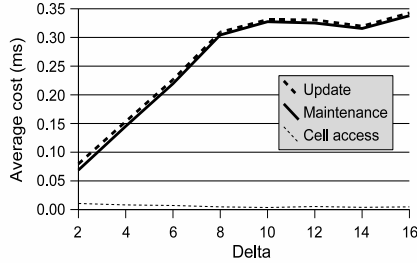


Fig. 9. Update cost and its two parts, the cost to update the maintained information and the cost to access cells, when varying  $\Delta$ .

## VII. CONCLUSIONS AND FUTURE WORK

This paper proposed a novel continuous location-based query: the CTUP query which continuously monitors the top- $k$  unsafe places, as the protecting units like police cars move around and send location updates. Two spatial partitioning based solutions are provided, namely BasicCTUP and OptCTUP. The BasicCTUP scheme, although simple, has some efficiency drawbacks. The OptCTUP scheme resolves these drawbacks and therefore is more robust. Experimental results have reconfirmed that while both proposed solutions are much better than naïve solutions, OptCTUP is clearly superior than BasicCTUP. The CTUP query and the OptCTUP solution have a potential to enable the police to better protect public safety.

There are several future research directions to the proposed work. First, the places may have extent, either because some place may have non-negligible extent or because some nearby places should be combined. Second, the protection of unit to a place can be modeled as a decaying function, i.e. the farther away, the less protected. Third, instead of monitoring top- $k$  unsafe places, the users may ask to monitor all places with safety lower than a threshold. Fourth, instead of monitoring, the user may want the system to continuously predicting the unsafe places in the near future.

**Acknowledgement:** Donghui Zhang's research has been partially supported by NSF CAREER Award IIS-0347600.

## REFERENCES

- [1] Midwest Bank Robbers. [http://en.wikipedia.org/wiki/Midwest\\_Bank\\_Robbers](http://en.wikipedia.org/wiki/Midwest_Bank_Robbers).
- [2] P. L. Brantingham and P. J. Brantingham. *Environmental Criminology*. Waveland Press, Illinois, 1991.
- [3] Thomas Brinkhoff. A Framework for Generating Network-Based Moving Objects. *Geoinformatica*, 6(2):153–180, 2002.
- [4] Y.-C. Chang, L. D. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The Onion Technique: Indexing for Linear Optimization Queries. In *SIGMOD*, pages 391–402, 2000.
- [5] R. Fagin, A. Lotem, and M. Naor. Optimal Aggregation Algorithms for Middleware. In *PODS*, pages 102–113, 2001.
- [6] B. Gedik and L. Liu. MobiEyes: Distributed Processing of Continuously Moving Queries on Moving Objects in a Mobile System. In *EDBT*, 2004.
- [7] M. Hadjieleftheriou, G. Kollios, D. Gunopulos, and V. J. Tsotras. On-Line Discovery of Dense Areas in Spatio-temporal Databases. In *SSTD*, pages 306–324, 2003.
- [8] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A System for the Efficient Execution of Multi-parametric Ranked Queries. In *SIGMOD*, pages 259–270, 2001.
- [9] H. Hu, J. Xu, and D. L. Lee. A Generic Framework for Monitoring Continuous Spatial Queries over Moving Objects. In *SIGMOD*, pages 479–490, 2005.
- [10] G. S. Iwerks, H. Samet, and K. P. Smith. Continuous K-Nearest Neighbor Queries for Continuously Moving Points with Updates. In *VLDB*, pages 512–523, 2003.
- [11] C. S. Jensen. Location-Based Services. In *Database Aspects of Location-Based Services*, pages 115–148. Morgan Kaufmann, 2004.
- [12] C. S. Jensen, D. Lin, B. C. Ooi, and R. Zhang. Effective Density Queries on Continuously Moving Objects. In *ICDE*, 2006.
- [13] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous Evaluation of Monochromatic and Bichromatic Reverse Nearest Neighbors. In *ICDE*, 2007.
- [14] I. Lazaridis, K. Porakaew, and S. Mehrotra. Dynamic Queries over Mobile Objects. In *EDBT*, pages 269–286, 2002.
- [15] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD*, pages 623–634, 2004.
- [16] Mohamed F. Mokbel and Walid G. Aref. PLACE: A Scalable Location-aware Database Server for Spatio-temporal Data Streams. *IEEE Data Engineering Bulletin*, 28(3):3–10, 2005.
- [17] K. Mouratidis, S. Bakiras, and D. Papadias. Continuous Monitoring of Top-k Queries over Sliding Windows. In *SIGMOD*, pages 635–646, 2006.
- [18] K. Mouratidis, M. Hadjieleftheriou, and D. Papadias. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD*, pages 634–645, 2005.
- [19] S. Prabhakar, Y. Xia, D. V. Kalashnikov, W. G. Aref, and S. E. Hambrusch. Query Indexing and Velocity Constrained Indexing: Scalable Techniques for Continuous Queries on Moving Objects. *IEEE Transactions on Computers*, 51(10):1124–1140, 2002.
- [20] M. A. Soliman, I. F. Ilyas, and K. C. Chang. Top-k Query Processing in Uncertain Databases. In *SIGMOD*, pages 635–646, 2007.
- [21] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *VLDB*, pages 287–298, 2002.
- [22] T. Xia and D. Zhang. Continuous Reverse Nearest Neighbor Monitoring. In *ICDE*, 2006.
- [23] T. Xia, D. Zhang, E. Kanoulas, and Y. Du. On Computing Top-t Most Influential Spatial Sites. In *VLDB*, pages 946–957, 2005.
- [24] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE*, pages 643–654, 2005.
- [25] K. Yi, H. Yu, J. Yang, G. Xia, and Y. Chen. Efficient Maintenance of Materialized Top-k Views. In *ICDE*, pages 189–200, 2003.
- [26] X. Yu, K. Q. Pu, and N. Koudas. Monitoring K-Nearest Neighbor Queries Over Moving Objects. In *ICDE*, pages 631–642, 2005.