CONTINUOUS MONITORING OF SPATIAL QUERIES IN REMOTE SYSTEM ENVIRONMENTS

Madduri Sree Radha Manga Mani, Ganta Chamundeswari
RamaChandra College of Engineering, Eluru, A.P.

1 INTRODUCTION

With the availability of inexpensive mobile devices, position locators and cheap wireless networks, location based services are gaining increasing popularity. Some examples of the location based services include fleet management, geo-social networking (also called location-based networking), traffic monitoring, location-based games, location based advertisement and strategic planning etc. Due to the popularity of these services, various applications have been developed that require continuous monitoring of spatial queries. For example, a person driving a car may issue k nearest neighbors (kNN) query to continuously monitor k closest restaurants. Similarly, a taxi driver might issue a query to continuously monitor the passengers that are within 5 Km of his location.

Driven by such applications, continuous monitoring of spatial queries has received significant research attention in the past few years. For example, several algorithms have been proposed to answer kNN queries \[14, 22, 20\], range queries \[8, 12, 18, 3\], constrained kNN queries \[7, 9\] and aggregate kNN queries \[14\]. Although various algorithms have been proposed to solve each of these spatial queries, to the best of our knowledge, there does not exist a unified algorithm that solves all the above mentioned queries. In this paper, we propose a unified algorithm to monitor a broad class of spatial queries including the above mentioned spatial queries.

2 Background

2.1 Preliminaries

In this section, we formally define few inherently different spatial queries.

k nearest neighbors query. A kNN \[10, 16, 11, 14, 22, 20\] query returns the k closest objects from the query point q.

Given a set of objects O and a query point q, the kNN query returns a set N of k objects such that for each \(n_i \in N\) the

\[\text{dist}(n_i, q) = \min_{o \in O} \text{dist}(o, q)\]

Constrained k nearest neighbors query. A constrained kNN query \[7, 9\] returns k objects closest to the query point q among the objects that lie inside a constrained region CR. Given a set of objects O, a query point q and a constrained region CR, the constrained kNN query returns a set N containing k objects such that for each \(n_i \in N\)

\[\text{dist}(n_i, q) \leq \text{dist}(o', q)\]

where \(o' \in O - N\) and both \(o'\) and \(n_i\) lie in the constrained region CR.

Aggregate k nearest neighbors query. Given a set of objects O and a query set Q with m numbers of instances \(q_1, ..., q_m\), the aggregate kNN query \[21, 15, 13\] returns k objects with the minimum aggregate distance from Q. Let \(\text{aggdist}(o, Q)\) be the aggregate distance of an object o from

\[\text{aggdist}(o, Q) = \sum_{q \in Q} \text{dist}(o, q)\]
Q. An aggregate $k\text{NN}$ query returns an answer set $N$ containing $k$ objects such that for each $n \in N$, $aggdist(n, Q) \leq aggdist(o', Q)$ where $o' \in O - N$. Below we define the aggregate distance functions for some common aggregate $k\text{NN}$ queries.

1. Sum-aggregate $\text{KNN}$ query uses $aggdist(o, Q) = \sum_{q \in Q} \text{dist}(o, q)$. 
2. Max-aggregate $\text{KNN}$ query uses $aggdist(o, Q) = \max_{q \in Q} \text{dist}(o, q)$. 
3. Min-aggregate $\text{KNN}$ query uses $aggdist(o, Q) = \min_{q \in Q} \text{dist}(o, q)$.

Consider that a group of friends wants to meet at a restaurant such that the total distance traveled by them to reach the restaurant is minimum. A sum-aggregate $\text{KNN}$ query ($k = 1$) returns the location of such a restaurant.

2.2 Related Work

[22, 20, 14] are some notable techniques that use grid-based index to monitor spatial queries. Conceptual Partitioning Monitoring (CPM) technique [14] organizes the grid cells into conceptual rectangles and assigns each rectangle a direction and a level number. The direction is $R$, $D$, $L$, $U$ (for right, down, left, up) and the level number is the number of cells in between the rectangle and $q$ as shown in Fig. 1.

CPM first initializes an empty min-heap $H$. It inserts the query cell $cq$ with key set to zero and the level zero rectangles $(R_0, D_0, L_0, U_0)$ with the keys set to minimum distances between the query $q$ and the rectangles into $H$. The entries are de-heaped iteratively. If a de-heaped entry $e$ is a cell then it checks all its objects and updates $q,k\text{NN}$ (the set of $k\text{NN}$ for the query $q$) and $q,\text{distk}$ (the distance of current $k$th $\text{NN}$ from $q$). If $e$ is a rectangle, it inserts all the cells inside the rectangle and the next level rectangle in the same direction into the heap $H$. The algorithm stops when the heap becomes empty or when $e$’s distance is greater than $q,\text{distk}$.

The update of each object is handled as follows. (1) Incoming update: CPM removes the $k$th NN and inserts the object in $q,k\text{NN}$. (2) Outgoing update: CPM removes the object from $q,k\text{NN}$ and finds the next NN by visiting the remaining entries in the heap. In case the query moves, CPM starts from the scratch. CPM outperforms both YPK-CNN [22] and SEA-CNN [20]. CPM can also be extended to solve few other spatial queries.

CPM can be used to answer continuous constrained $k\text{NN}$ queries by making a small change. More specifically, the algorithm inserts only the rectangles and the cells that intersect the constrained region into the heap. Figure 1 shows an example where the constrained region is a polygon $R$. The constrained $NN$ is $O_2$ and the rectangles/cells shown shadowed are inserted into the heap by CPM.

CPM can also be extended to answer Aggregate $k\text{NN}$ queries. In case of a conventional $k\text{NN}$ query, the algorithm starts with the query cell $cq$. For aggregate $k\text{NN}$ query, the algorithm computes a rectangle $M$ such that all query instances lie in $M$. At the initial phase, the algorithm inserts the rectangle $M$ and the zero level rectangles in the heap with their aggregated distance $aggdist(e, Q)$ (e.g., $\min_{q \in Q} \text{dist}(e, q_i)$) from the query set $Q$. For instance, in the Fig. 2 the min aggregate $NN$ is $O_3$ and all rectangles shown in solid lines are inserted in the heap to compute $O_3$. Please note that the algorithm inserts all the shadowed cells into the heap. Note that CPM inserts many unnecessary cells into the heap. Fig. 3 shows the minimum number of cells (shown shaded) that are needed to be inserted in heap by an optimal algorithms. We show that our approach significantly reduces the number of cells inserted in the heap.

CircularTrip [5] and iSEE [19] also efficiently monitor $k\text{NN}$ queries. CircularTrip visits the cells around query point round by round until all NNs are retrieved. On the other hand iSEE computes a visit order list around the query point to efficiently answer the $k\text{NN}$ query. However, extension of these algorithms for other spatial queries (e.g., aggregate $k\text{NN}$ query) is non-trivial.

3 Problem definition

Let $p$ be a point and $R$ and $R_c$ be two hyper-rectangles in a $d$-dimensional space $\mathbb{R}^d$. If $R$ contains $R_c$ (i.e., $R_c$ inside the hyper-rectangle $R$) then $R$ is called the child of $R$. Consider a function $f(p)$ that returns the score of a given point $p$. An upper bound score $SU(R)$ of a hyper-rectangle $R$ is defined as,

$$SU(R) = \max_{p \in R}(f(p))$$

where $p \in R$ denotes a $p$ that lies in the hyper-rectangle $R$. Similarly, the lower bound score $SL(R)$ is defined as,

$$SL(R) = \min_{p \in R}(f(p))$$

Versatile Scoring function: A function $f$ is called a versatile scoring function if $SU(R) \geq SU(R_c)$ and $SL(R) \leq SL(R_c)$ for any $R$ and $R_c$ where $R_c$ is a child rectangle of $R$. We denote the versatile scoring function as $vsf()$. The versatile score of a given point $p$ is denoted as $vsf(p)$.

Consider a function $f(p) = \text{dist}(p, q)$ where $\text{dist}(p, q)$ is the Euclidean distance between the point $p$ and a given point $q$. Hence, the upper bound score $SU(R)$ is the maximum Euclidean distance between the rectangle $R$ and the fixed point $q$. Similarly, the lower bound score $SL(R)$ is the minimum Euclidean distance between the rectangle $R$ and the fixed point $q$. Note that $f(p) = \text{dist}(p, q)$ is a versatile scoring function.

3.1 Versatile top-$k$ queries

Consider a set of objects $O = \{o_1, \ldots, o_N\}$ where $o_i$ denotes the spatial location of $i$th object. Also, consider a versatile scoring function $vsf()$ to compute the score of the objects. A top-$k$ query returns a set of $k$ objects $N = \{n_1, \ldots, n_k\}$ such that $vsf(n_i) \leq vsf(o')$ for any $n_i \in N$ and any $o' \in O - N$. Hence, top-$k$ query returns $k$ objects having smallest scores.

In this paper we study the continuous monitoring of top-$k$ query where the top-$k$ results are updated with the changes in the datasets. We follow timestamp model where
the results are required to be updated after every t time units.

3.2 Modeling spatial query to top-k query

We can model various spatial queries to a versatile top-k query by defining a suitable versatile scoring function. The versatile scoring functions for some popular spatial queries are given below.

k nearest neighbors queries:

$$vsf(o) = dist(o, q)$$

Here, the dist(o, q) is the Euclidean distance between an object o and the query point q.

Furthest k neighbors queries:

$$vsf(o) = -dist(o, q)$$

Please note that not the object furthest from the query q has the smallest score. Hence, the further objects are preferred in this case.

Aggregate k nearest neighbors queries:

Below we define the scoring functions for Sum, Max and Min aggregate k nearest neighbors queries, respectively.

i) Sum-Aggregate k nearest neighbors queries: $$vsf(o) = \sum_{i=1}^{k} dist(o, qi)$$

ii) Max-Aggregate k nearest neighbors queries: $$vsf(o) = \max_{i \in Q(dist(o, qi))}$$

iii) Min-Aggregate k nearest neighbors queries: $$vsf(o) = \min_{i \in Q(dist(o, qi))}$$

Note that we can also define the versatile scoring functions for other queries like constrained furthest neighbors query and constrained aggregate kNNs query etc.

Next, we define the versatile scoring function to model another spatial query which is not essentially a top-k query. This demonstrates that our proposed unified algorithm can be applied to answer several other queries that are not top-k queries.

Circular Range queries: Given a set of objects O, a query point q and a positive value r. A circular range query [8, 12, 3] returns every object n ∈ O that lies within distance r of the query location q (i.e., every object such that dist(n, q) ≤ r). We call such query a circular range query because the search space is a circle around the query point q with the radius r. Below we define the versatile scoring function for the circular range query.

$$vsf(o) = \begin{cases} 1, & \text{if dist(o, q) ≤ r;} \\ \infty, & \text{otherwise.} \end{cases}$$

Here, r is the radius of the circular range query. To model the circular range query to a versatile top-k query we need to make the following small changes: i) every object with score equal to kth object’s score must also be reported; i) an object with score ∞ must not be reported. Note that we if the range contains more than k objects then all the objects inside the range are reported. On the other hand, if the range contains less than k objects then objects outside the range are not reported.

4 Technique

4.1 Conceptual Grid Tree

In this section, we briefly describe the conceptual grid tree which we introduced in [4] and later used to answer other spatial queries in [3, 9]. The conceptual grid tree is the backbone of our approach. First, we briefly describe the grid index and then we describe the conceptual grid tree which is a conceptual visualization of the grid index.

In a grid based index, the whole space is divided into a number of cells, where the size of each cell is $\delta \times \delta$. Hence, the extent of each cell in a dimension is $\delta$. A particular cell is denoted as $[i, j]$ where i is the column number and j is the row number. The lower left cell of the grid is $[0, 0]$. An object o with the position $(x, y)$ is located into the cell $[\lceil x/\delta \rceil, \lceil y/\delta \rceil]$. I.e., a cell $[i, j]$ contains all the objects with x-coordinate in the range $i \cdot \delta$ to $(i + 1) \cdot \delta$ and y-coordinate in the range $j \cdot \delta$ to $(j + 1) \cdot \delta$.

In our proposed conceptual grid tree structure we assume a grid that consists of $2^n \times 2^n$ cells. The grid is treated as a conceptual tree where the root contains all $2^n \times 2^n$ grid cells. Each intermediate entry e in a level l (for root $l = 0$) is recursively divided into four children of equal sized rectangles such that each child of an entry e contains $x/4$ cells where x is the number of cells contained by the intermediate entry e. I.e., if an entry e at level l contains $2^{n-l} \times 2^{n-l}$ cells then each child of the entry e will contain $2^{n-l-1} \times 2^{n-l-1}$ cells. Every leaf level entry contains four grid cells.

The root, intermediate entries and the grid cells are shown in Fig. 4. In Fig. 4 the grid size is $4 \times 4$ (i.e., $2^2 \times 2^2$ grid cells). Hence, the root contains all $2^2 \times 2^2$ cells. An intermediate entry with level 1 contains $2^1 \times 2^1$ cells (i.e., four cells).

Please note that the Grid-tree is just a conceptual visualization of the grid and it does not exist physically (i.e., we do not need pointers to store entries and its children). In Fig. 4 the rectangles with dotted lines are considered as conceptual structure and the rest are physical structure. Therefore, root and the intermediate entries are conceptual only and they are not stored in the memory.

To retrieve the children of an entry (or root), we divide its rectangle into four equal sized rectangles such that each child has side length $d/2$ where $d$ is the side length of its parent. A rectangle with side length equal to $\delta$ (the width of a grid cell) refers to a cell $[i, j]$ of the grid.

4.2 Unified Algorithm

Initial computation Most of the spatial queries algorithms that can be applied on other tree structure (e.g., R-tree) can easily be applied on the conceptual grid tree. The advantage of using this grid tree over previously used grid based access methods is that if an intermediate entry of the tree lies in the pruned region then none of its cells are accessed.

Algorithm 1 CGTree-based Unified Initial Computation

Input: q: query point with the versatile scoring function(vsf()); k: an integer
Output: top-k query results
1: q.score$k = \infty$; q.ka = _; H = _
2: Initialize a min-heap H with root entry of the conceptual grid tree
3: while H/= _ do
The initial computation of the unified algorithm using the Conceptual Grid Tree is presented in Algorithm 1. The main idea is similar to that of applying BFS search [11] on R-tree based data structure. Specifically, the algorithm starts de-heaps the entries. If a de-heaped entry e is a grid cell then it visits the cell and updates q.kA and q.scorek where q.kA is the answer set and q.scorek is the kth smallest score of objects in q.kA (line 8). If |q.kA| < k (i.e., the size of the answer set is less then k) then q.scorek is set to infinity. Please recall that the width of a cell is δ. So, the algorithm checks the width of each entry e to identify whether e is a grid cell or not (line 7).

If the de-heaped entry e is not a grid cell, then the algorithm inserts its children into the heap H with their lower bound scores (lines 10 to 12). The algorithm terminates when the heap becomes empty (line 3) or when a de-heaped entry e has its lower bound score SL(e) ≥ q.scorek (line 5). This guarantees the correctness of the algorithm. This is because any cell c for which SL(c) ≥ q.scorek cannot contain an object that has a score smaller than q.scorek (and cannot be the answer for this reason). When the de-heaped entry e has its lower bound score SL(e) ≥ q.scorek, every remaining entry e′ in the heap H has its lower bound score SL(e′) ≥ q.scorek because the entries are accessed in ascending order of their lower bound scores.

**Continuous Monitoring** Before we present the continuous monitoring algorithm, we introduce the data structure that is used for efficient update of the results.

The system stores a query table and an object table to record the information about the queries and the objects. An object table stores the id and location of all objects. The query table stores the query id, query location, the answer set q.kA and the cellList (cells that the query has visited to retrieve all objects in the answer set q.kA). Each cell of the grid stores two lists namely object list and query list. The object list of a cell c contains the object id of every object that lies in c. The query list of a cell c contains the id of every query that has visited c (by visiting c we mean that it has considered the objects that lie inside it (line 8 of Algorithm 1)). The query list is used to quickly identify the queries that might have been affected by the object movement in a cell c.

**Handling a single update:** Assume that an object o reports a location update and old and onew correspond to its old and new locations, respectively. The object update can affect the results of a query q in the following three ways:

1. **Internal update:** vsf(old) ≤ q.scorek and vsf(onew) ≤ q.scorek; clearly, only the order of the answer set may have been affected, so we update the order of q.kA accordingly.
2. **Incoming update:** vsf(old) > q.scorek and vsf(onew) ≤ q.scorek; this means that o is now a part of q.kA. Hence, o is inserted in q.kA.
3. **Outgoing update:** vsf(old) ≤ q.scorek and vsf(onew) > q.scorek; i.e., o is not part of the answer set anymore. Therefore, we delete o from q.kA.

**The complete update handling module:** The update handling module consists of two phases. In the first phase, we receive the object updates. For each object update, we reflect its effect on the results according to the three scenarios described earlier. In the second phase, we compute the final results. Algorithm 2 presents the details.

### Algorithm 2 Continuous Monitoring

**Input:** location updates  
**Output:** q.kA

**Phase 1:** First, we receive the object updates and for each object update, we identify the queries that might have been affected by this update. It can be immediately verified that only the queries in the query lists of cold and cnew may have been affected where cold and cnew denote the old and new cells of the object, respectively. For each affected query q, the update is handled (lines 3 to 6) as mentioned previously (e.g., internal update, incoming update or outgoing update).

**Phase 2:** After all the updates are received, the results of the queries are updated as follows; if q.kA contains more than k objects in it (more incoming updates than the outgoing updates), the results are updated by keeping only the top k objects. Otherwise, if q.kA contains less than k objects, we expand the search region so that q.kA contains k objects. The expansion is similar to the Algorithm 1 except the following changes. Any entry e that has SL(e) ≤ q.scorek are not inserted into the heap. This is because such entries have already been explored.

The stopping criteria is same as the initial computation i.e., we stop when a de-heaped entry e has SL(e) ≥ q.scorek.

If a query changes its location the versatile score become invalid. Hence, the results are computed by calling the Algorithm 1 (i.e., we compute the result for the query from the scratch).

**Proof of optimality and correctness** Before we prove the optimality, we define two terms; accessing and visiting a cell. We say that a cell has been accessed if the algorithm inserts it in the heap (e.g., line 12 of Algorithm 1). If a cell is de-heaped from the heap and the algorithm retrieves the objects in this cell, we say that the cell has been visited by the algorithm (e.g., line 8 of Algorithm 1). Please note that the cost of visiting a cell is usually significantly higher than the cost of accessing a cell.

We prove that our algorithm is optimal in number of visited cells (i.e., it does not visit any unnecessary cell to answer the query). To prove the correctness, we show that our algorithm visits all the cells that must be visited to compute the correct results.
Continuous Monitoring of Spatial Queries In Remote System Environments

Proof. Let $q_{old}.score_k$ and $q_{new}.score_k$ be the scores of $k^{th}$ object before and after the update, respectively. Consider the case when $q_{old}.score_k \geq q_{new}.score_k$ (i.e., the number of incoming updates is at least equal to the number of outgoing updates). This implies $|q_{old} \Delta_k| \geq k$ (line 8 of Algorithm 2) and we do not need to visit any new cell to update the result. Therefore, we only need to consider the case when $q_{old}.score_k < q_{new}.score_k$ (line 9 of Algorithm 2). Below, we prove the optimality and correctness of our algorithm for this case.

Let C be the set of minimum cells that have to be visited in order to guarantee the correct results. First, we identify C and show that our algorithm does not visit any unnecessary cell $c'$ such that $c' \in C$. A cell $c'$ for which $SU(c') \leq q_{old}.score_k$ is not required to be visited. This is because all the objects in this cell have been considered earlier. Similarly, a cell $c'$ for which $SL(c') \leq q_{new}.score_k$ is not required to be visited. This is because every object in such cell has score at least equal to $q_{new}.score_k$. Therefore, the set C of minimum cells consists of every cell c that satisfies the following two inequalities.

$SU(c) > q_{old}.score_k$ (1)

$SL(c) < q_{new}.score_k$ (2)

Please note that in our update handling algorithm, we ignore the cells that have $SU(c) \leq q_{old}.score_k$ and terminate the algorithm when $SL(c) \geq q_{new}.score_k$ (see Section 4.2 Phase 2). Thus, we satisfy both of the above inequalities. Therefore, our algorithm does not visit any unnecessary cell and is optimal in the number of visited cells.

Please note that the initial computation can be considered as a special case of update handling where $q_{old}.score_k$ is set to zero. As a proof of correctness, we show that our algorithm visits all the cells in the set C. Recall that we maintain the cells in a heap based on their lower bound scores. Therefore, the cells are visited in the ascending order of their lower bound scores and it guarantees that every cell c for which $SL(c) < q_{new}.score_k$ is visited.

5 EXPERIMENTS

We choose three inherently different spatial queries and run experiments to evaluate the efficiency of our unified algorithm. More specifically, we run the experiments for the constrained kNN queries, the aggregate kNN queries and the furthest k neighbors queries. Since our algorithm is based on the Conceptual Grid Tree, we refer to it as CGT.

We compare our algorithm with CPM [14]. As mentioned by the authors, it can be modified to answer constrained kNN and aggregate kNN queries. We extend CPM to answer furthest k neighbors query as follows. We compute the furthest conceptual rectangles from the query cell $c_0$ in all four directions (i.e., right, down, left, up). Initially, we insert the furthest rectangles in the heap with their keys set to the maximum distances between the query and the rectangles. After de-heaping a rectangle, the previous level (closer to the query cell) rectangle in the same direction is inserted in the heap. We use a max heap and thus retrieve the rectangles in descending order of their maximum distances from the query.

For a proof of correctness, we show that our algorithm visits all the cells $c$ that satisfies the following two inequalities.

$SU(c) > q_{old}.score_k$ (1)

$SL(c) < q_{new}.score_k$ (2)

Please note that in our update handling algorithm, we ignore the cells that have $SU(c) \leq q_{old}.score_k$ and terminate the algorithm when $SL(c) \geq q_{new}.score_k$ (see Section 4.2 Phase 2). Thus, we satisfy both of the above inequalities. Therefore, our algorithm does not visit any unnecessary cell and is optimal in the number of visited cells.

Please note that the initial computation can be considered as a special case of update handling where $q_{old}.score_k$ is set to zero. As a proof of correctness, we show that our algorithm visits all the cells in the set C. Recall that we maintain the cells in a heap based on their lower bound scores. Therefore, the cells are visited in the ascending order of their lower bound scores and it guarantees that every cell c for which $SL(c) < q_{new}.score_k$ is visited.

CONCLUSION

We are first to present a unified algorithm to answer a broad class of spatial queries. Our proposed algorithm is optimal in the sense that it visits minimum number of cells throughout the life of a continuous query. Our extensive experimental results demonstrate that for each inherently different spatial queries our unified algorithm significantly outperforms existing best known algorithm.

REFERENCES


