Efficient Valid Scope for Location-Dependent Spatial Queries in Mobile Environments

Ken C. K. Lee[†‡] Wang-Chien Lee[†] Hong Va Leong[‡] Brandon Unger[†] Baihua Zheng[§]

[†]Department of Computer Science and Engineering, Pennsylvania State University, USA
[‡]Department of Computing, The Hong Kong Polytechnic University, Hong Kong
[§]School of Information Systems, Singapore Management University, Singapore
[†]{cklee,wlee,bunger}@cse.psu.edu
[‡]{cscklee,cshleong}@comp.polyu.edu.hk
[§]bhzheng@smu.edu.sg

***Abstract*— In mobile environments, mobile clients can access information with respect to their locations by submitting Location-Dependent Spatial Queries (LDSQs) to Location-Based Service (LBS) servers. Owing to scarce wireless channel bandwidth and limited client battery life, frequent LDSQ submission from clients must be avoided. Observing that LDSQs issued from a client located at nearby positions would likely return the same query results, we explore the idea of *valid scope*, which represents a spatial area in which a set of LDSQs will retrieve exactly the same set of query results. With a valid scope derived and an LDSQ result cached, a client can assert whether the new LDSQs can be answered with the maintained LDSQ result, thus eliminating the need of sending LDSQs to the server. Contention on the wireless channel and client energy consumed for data transmission can be considerably reduced. In this paper, we design efficient algorithms to compute the valid scope for common types of LDSQs, including nearest neighbor queries, range queries and window queries. Through an extensive set of experiments, our proposed valid scope computation algorithms are shown to outperform existing approaches.**

I. Introduction

With rapid technological advancement of portable devices, positioning equipment and wireless communication, the vision of mobile computing has moved closer to reality. Among many applications developed for mobile environments, **L**ocation-**B**ased **S**ervices (**LBS**) belong to a major category of those killer applications. LBS provides useful information to the users at the right time in the right place. Typically, location-related information and requests for this information are expressed as spatial objects, or simply objects, and **L**ocation-**D**ependent **S**patial **Q**ueries (**LDSQ**s), respectively. Example LDSQs issued by mobile clients include "where are those ATMs within 1 mile from my current position?" and "where is the nearest gas station with respect to my current position?". Figure 1 depicts a client-server system model in which LBS is commonly deployed. Here, mobile clients send LDSQs via a base station to the LBS server querying for spatial objects. Typical types of LDSQs include nearest neighbor (NN) queries, range queries and window queries. An NN query finds an object within a set of spatial objects which is closest to the current user position; a kNN ($k > 1$) query extends NN query to search for k nearest objects. A range query retrieves objects within a specified distance from the current user position. A window query searches for objects within a spatial window centered at the current user position.

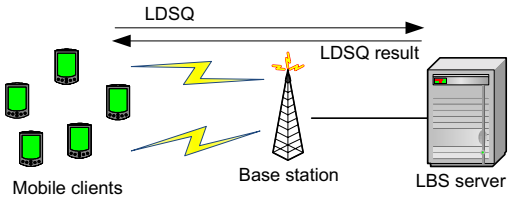


Figure 1. Client-server based LBS system model

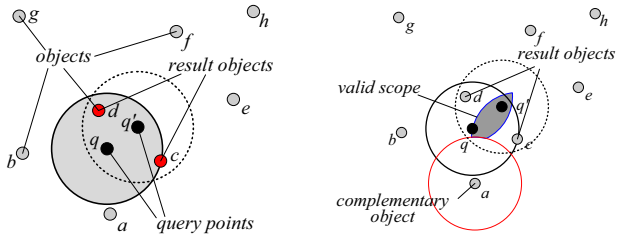
Unlike conventional spatial queries, LDSQs are parameterized with a query point that represents the current user location. As such, the result of an LDSQ is dependent on the client location where the query is issued. A client who is interested in keeping track of updated LDSQ results while moving will need to repeatedly evaluate the query to ensure the correctness of LDSQ results. However, staying proactive to reevaluate LDSQs certainly consumes a lot of precious wireless bandwidth and client energy and imposes extra load on the server. In fact, the results of LDSQs with respect to similar positions are most likely to remain the same. For instance, as shown in Figure 2(a), a range query issued at two query points q and q' cover exactly the same result set (i.e., objects c and d). Suppose that the result of the LDSQ issued at q is maintained by a client, reevaluating the query issued at q' can be avoided if the client can assert that the query results are the same. As

This paper is based on "Efficient Valid Scope Computation for Location-Dependent Spatial Queries in Mobile and Wireless Environments," by K.C.K. Lee, W.C. Lee, H.V. Leong, B. Unger, and B. Zheng, which appeared in the Proceedings of Third International Conference on Ubiquitous Information Management and Communication, Seoul, Korea, January 2009.

Wang-Chien Lee and Ken C. K. Lee are supported in part by the National Science Foundation under Grant numbers IIS-0534343 and CNS-0626709. Hong Va Leong and Ken C. K. Lee are supported in part by the Hong Kong Research Grant Council under Grant number HKBU 1/05C.

doi:10.4304/jsw.5.2.133-145

a result, expensive query execution costs could be saved. We call this the self answerability property of the revised query at the client [6].



(a) Range queries q and q' (b) Valid scope

Figure 2. Illustration of valid scope

To avoid unnecessary LDSQ reevaluation, a valid scope that represents a spatial area can be derived and associated with a query result. Inside the valid scope, the associated result is guaranteed to be identical for the corresponding query. Thus, a client inside a valid scope can simply reuse the result for the query, and perform query reevaluation only when it moves out of the valid scope. To represent a valid scope, it is straightforward to retrieve and label all objects as result objects and non-result objects. However, retrieval of all objects from the LBS server is very time and energy consuming. Instead, we determine a small and representative subset of non-result objects, called "*complementary objects*", together with result objects to represent a valid scope. Referring to our example, the valid scope of an LDSQ result is formed as shown in Figure 2(b) by two result objects, i.e., c and d and one complementary objects, a. Now, the client with the LDSQ result and a few complementary objects can determine if the result can be reused. By reducing the number of LDSQ reevaluations, contention on wireless bandwidth and client energy consumption can be alleviated. The server load can be relieved as well.

Although algorithms to determine valid scope have been recently proposed, as to be discussed shortly, they are inefficient and cannot support various types of LDSQs. Motivated by the importance of valid scopes to LDSQs and the need of highly efficient valid scope algorithms, we develop in this paper a suite of efficient online valid scope computation algorithms for various LDSQs. As opposed to existing works, our algorithms can determine an LDSQ result and its valid scope together with only a single index lookup, thus shortening the valid scope computation and I/O access time. Besides, our algorithms are generic to support more types of LDSQs while existing ones cannot. We conduct an extensive set of experiments to validate the effectiveness of valid scopes in reducing the evaluation cost of LDSQs and to measure the efficiency of our algorithms in comparison with existing works. In summary, we make the following important contributions.

1) We investigate the valid scope determination problem and transform it into an issue of identifying complementary objects with respect to LDSQ result objects, based on which efficient online valid scope computation algorithms can be developed.
2) We devise valid scope determination algorithms for common types of LDSQs, that include nearest neighbor, k nearest neighbor, range and window queries and explore some corresponding optimization techniques.
3) We implement our proposed algorithms and conduct experiments with both synthetic and real data sets to test their scalability and practicality. The results well demonstrate the effectiveness of valid scope and the efficiency of our online valid scope computation algorithms.

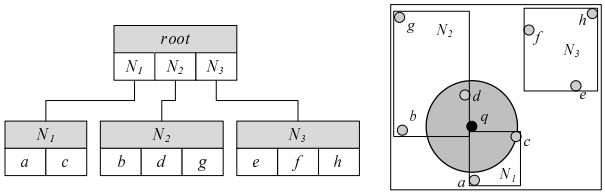
The remainder of the paper is organized as follows. Section II provides some background of this research and reviews important related works. Section III, Section IV, and Section V present the valid scope determination algorithms for NN (including kNN) queries, range queries and window queries, respectively. Section VI describes the experiment settings and studies the simulation results. Section VII concludes this paper, with an outline of our future research directions.

II. PRELIMINARY

In this section, we first review the R-tree index and the best-first search algorithm on the R-tree that are useful to both LDSQ processing and valid scope computation. Then, we review closely related works.

A. R-tree and Best-First Search Algorithm

In this paper, we assume that all the objects maintained by an LBS server are indexed by an R-tree [3] on their spatial coordinates. R-tree clusters spatially close objects, represents them using minimum bounding boxes (MBBs) and recursively groups MBBs until the root of the index is formed. Figure 3(a) depicts an R-tree with a maximum fanout of three. At the bottom, 8 objects labeled 'a' through 'h' are grouped into 3 MBBs, i.e., N_1, N_2 and N_3. Then, the three MBBs are grouped to form the root of the index. The positions of objects and MBBs are shown in Figure 3(b).



(a) R-tree (b) Object MBB

Figure 3. R-tree and search algorithm

To efficiently retrieve objects required by a spatial query, many efficient search algorithms are developed based on the notion of best-first traversal [4] upon R-tree. Best-first search algorithm organizes unexplored index nodes and objects to be accessed in a priority queue according to their minimum distances with respect to a query point (i.e., mindists [8]). The search initializes the priority queue with the root node. It repeatedly dequeues the head entry, which can be an index node or an object, of the queue for evaluation until the queue becomes empty

or all the remaining objects in the queue do not satisfy the query. The use of the priority guarantees that the head entry is at a minimum distance to the query point among all unexplored entries in the R-tree. The best-first search algorithm is shown in Figure 4. It explores the head entry ϵ in a priority queue (line 2-9). If ϵ is a node, it is expanded into its child nodes (line 5-6). Otherwise, it is checked against the query and collected as a result object if it satisfies the query (line 8-9). We simply use a termination condition to indicate when the search completes, so that it could be easily generalized for various queries. For kNN query, the search terminates when the result set contains k result objects. For range and window query, the termination condition is satisfied when all the remaining objects in the queue are outside the search area.

```
Algorithm BestFirstSearch(T,q)
Input.   an R-tree (T), a query point (q)
Local.   a priority queue (P)
Output.  a result set of objects (R)
Begin
 1.  P.enqueue(T.root, mindist(T.root, q));
 2.  while (P is not empty and the termination
         condition is not satisfied) do
 3.    (ϵ, d) ← P.dequeue();
 4.    if (ϵ is a node) then
 5.      foreach child c of ϵ do
 6.        P.enqueue(c, mindist(c, q));
 7.    else
 8.      if (ϵ satisfies the query) then
 9.        R ← R ∪ {ϵ};
10.  output R;
End
```

Figure 4. Algorithm BestFirstSearch

Entry (ϵ)	Priority queue (in ascending mindist order)
$root$	N_2, N_1, N_3
N_2	N_1, d, N_3, b, g
N_1	d, c, a, N_3, b, g
d	c, a, N_3, b, g
c	a, N_3, b, g

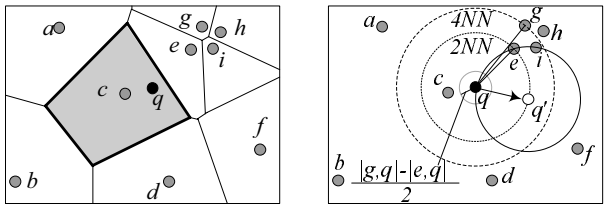
Figure 5. Trace based on a 2NN query with best-first search

To facilitate our discussion, we illustrate best-first search algorithm using a running example. Suppose a 2NN query is issued at a point q, based on an R-tree shown in Figure 3(a). The trace of the algorithm for each iteration is outlined in Figure 5. First, the root node, the initial entry in the priority queue, is fetched and examined. All its child nodes N_1, N_2, and N_3 are placed back to the queue, sorted according to their mindists, in the order N_2, N_1, N_3. As N_2 is the head entry, it is dequeued and replaced by its children b, d and g in the queue, sorted in mindists order. Next, N_1 is the closest one and is explored and expanded into a and c. At this moment, d at the head of the queue is dequeued and collected as part of the result. After that, c is also dequeued and collected. Now, c and d form the 2NN query result and the search completes. All the remaining entries in the queue, which represent the non-result objects, are guaranteed to be located farther away, in non-decreasing distance order, from the query point.

As to be discussed, our approaches are based on best-first search algorithm because of three reasons. First, it can support various types of LDSQs that usually access proximate objects around query points. Second, upon termination, the remaining priority queue content represents *all* the non-result objects, based on which a valid scope for the query result can be derived. Third, as those non-result objects that affect the determination of a valid scope are expected to be close to the result objects and the query point, the remaining priority queue already has them sorted based on their distances to the query point. By tracking both result and non-result objects in a single priority queue, our valid scope computation incurs only one index lookup. Since valid scope formulation is dependent on the types of LDSQs, we shall discuss in detail the algorithms for nearest neighbor, range and window queries in subsequent sections.

B. Related Work

If mobile clients can be assured that the result of a previous LDSQ remains valid for the new LDSQs, unnecessary query reevaluation can be avoided. To equip mobile clients with such a capability, a number of related research works in the literatures are studied and reported. In [2], window query results are maintained as semantic regions. Any window query fully covered by existing semantic regions is guaranteed to be answerable by the client locally. Specific for handling NN query, the work [13] associates each NN result with a precalculated Voronoi cell [1] as in Figure 6(a). The result is asserted to be valid if the client (i.e., the query point) is inside the corresponding cell. For kNN query, the search for m ($m > k$) NN objects can be extended with respect to the same query point q [9]. Due to triangular inequality, a client can be assured that kNN is contained by the mNN result if the distance it moved from q is less than half of the distance difference between k-th NN and m-th NN with respect to q. However, this is ineffective to reduce query reevaluation. For example in Figure 6(b), based on q, a 2NN is executed as a 4NN and δ is the distance between the second NN and the fourth NN object. A new query point q' located more than a distance of $\delta/2$ away from q will thus induce reevaluation. In fact, the 4NN result still covers 2NN objects to q', which implies that this over-conservative estimation cannot effectively eliminate unnecessary queries.

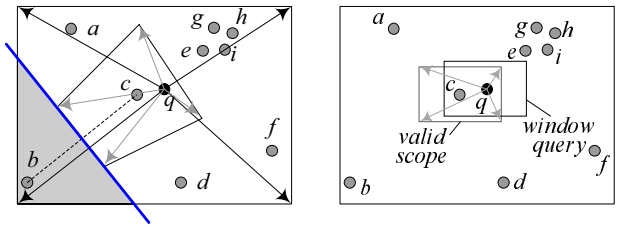


(a) Voronoi cell (b) mNN query

Figure 6. Voronoi cell and mNN query

The algorithms to determine the valid scopes in [12] are similar to what we are investigating in this paper. There are two steps in those algorithms. The first step

determines the LDSQ result. The second step determines the valid scope that is initialized as the entire object space and keeps being refined until client movement inside the region is certainly not violating the result validity. More specifically, the second step executes a number of time parameterized (TP) queries [10] to simulate all possible client movement paths. Figure 7(a) shows the valid scope refinement for an NN query result. TPNN queries are issued towards all the vertices of the region to probe non-result objects that influence the query result (i.e., complementary objects, in our terminology). If non-result objects are found before the TPNN reaches the target vertices, the valid scope is trimmed along a bisector formed between the NN result object and the closest non-result object. Otherwise, the query result is considered to be valid at the edge of the scope. Figure 7(b) illustrates the valid scope determination for a window query result. TPWINDOW queries are initiated from a query point q towards all vertices of the region. Then, the valid scope is refined by removing the portion that touches any non-result object. Again, the issuance of TPWINDOW queries is repeated until no non-result object can be probed. The number of TP queries required is highly related to the complexity of the valid scope. As reported in [12], this valid scope algorithm has to repeatedly access an R-tree index, resulting in a large number of disk accesses. These algorithms are clearly inefficient, since they involve complicated polygon manipulation. If the search range is very complex or not of the shape of a polygon (e.g., circle as in range query), the valid scope computation will be very computationally and I/O expensive. To address this, we introduce more efficient online valid scope computation algorithms in this paper.



(a) NN query (b) Window query

Figure 7. Valid scope refinement based on TP-based approach

Recently we have also studied the problem of valid scope computation in mobile broadcast environments [7]. Although the basic idea of the server-based valid scope computation algorithms presented in this paper bear some similarity to the approach for broadcast environments, the challenges faced by algorithm designs for the broadcast and point-to-point communication environments are very different. Thus, the focus are clearly not the same. In broadcast environments, where the delivery order of objects are fixed in a broadcast channel, the emphasis of valid scope computation algorithm is on how to determine the search space for complementary objects, in order to optimize the tuning and access time. In the point-to-point communication environment in this paper, we focus on how to integrate existing query processing algorithm with valid scope computation so as to reduce bandwidth consumption and the overall server processing cost.

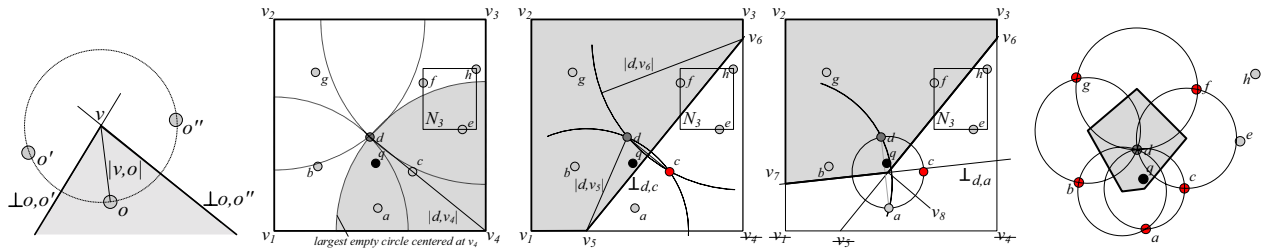
III. Nearest Neighbor Query

In this section, we present online valid scope computation algorithms for NN and kNN query and discuss the result validity check for new queries from the client.

A. Valid Scope for NN Query

Let the set of objects in our environment be O. For an NN or kNN query p, let the set of result objects be R_p and the remaining non-result objects be N_p. Then $O = R_p \cup N_p$ and $R_p \cap N_p = \emptyset$. Assume that for an NN query p, it has the nearest neighbor object o, then the result set for the NN query is $R_p = \{o\}$. Let $V(p)$ be the valid scope for the NN query result R_p. It is obvious that the Voronoi cell of o, denoted by $\diamondsuit(o)$, is the valid scope of the corresponding query result with respect to p. Thus, $V(p) = \diamondsuit(o)$. Object o is guaranteed to be the nearest neighbor to any point inside $\diamondsuit(o)$. The formation of a Voronoi cell is based on half-planes. Given two objects o and o', two half-planes, $HP_{o,o'}(o)$ and $HP_{o,o'}(o')$ are formed sharing a bisector $\perp_{o,o'}$ between o and o'. With a set of non-result objects denoted by N_p, $\diamondsuit(o) = \cap_{o' \in N_p} HP_{o,o'}(o)$, i.e., an intersection of all the half-planes that cover o formed against all the non-result objects. Obviously, examining all non-result objects to determine their half-planes and to derive the Voronoi cell of an NN object is totally impractical. In fact, only those non-result objects that contribute the bisector as the Voronoi cell perimeter are needed. We refer to those objects as a set of *complementary objects*, denoted by C_p ($C_p \subseteq N_p$). Thus, the valid scope for an NN query result $R_p = \{o\}$ is formed as $\cap_{o' \in C_p} HP_{o,o'}(o)$ instead. The search of C_p, which is very dependent on the object distribution, becomes the main challenge of determining the valid scope for NN query result efficiently.

To tackle this challenge, we exploit the largest empty circle property, which is one of the most important Voronoi cell properties, to identify C_p. The largest empty circle for a set of objects O is a circle with the largest radius, such that there is no object in O staying inside the circle. Furthermore, the center of the circle is inside the convex hull of the set of objects O and at least two objects lie on the boundary of the circle. In $\diamondsuit(o)$, each vertex, v, is formed by an intersection of two bisectors, $\perp_{o,o'}$ and $\perp_{o,o''}$, of a Voronoi cell and it should be equidistant to all o, o' and o''. Then v should have its largest empty circle that covers the object, o, inside the cell and at least two objects outside the cell. Formally, let us denote $|v_i, v_j|$ for the Euclidean distance between two points v_i and v_j and $\overline{i,j}$ for the line joining them. We also denote the largest empty circle centering at v by $\odot(v, |v, o|)$, where the radius is $|v, o|$. This is illustrated in Figure 8(a). The shaded area represents a part of the Voronoi cell of an object o, and point v is one of the vertices of $\diamondsuit(o)$. On the other hand, each object o' that contributes one edge of $\diamondsuit(o)$ must be touched by two



(a) Largest empty circle (b) Initial search space (c) Examining c (d) Examining a (e) Valid scope

Figure 8. Determination of valid scope for NN query result

largest empty circles, centered at the two endpoints of the edge contributed by o'. If a vertex v is valid (i.e., belonging to $\diamondsuit(o)$), its corresponding circle $\odot(v, |v, o|)$ must be empty. Otherwise, there should exist another object that stays inside the corresponding circle and is close to o. As a whole, the union of all the largest empty circles $\odot(v, |v, o|)$ of existing vertices v forms the search space of possible non-result objects needed in C_p. Since only objects inside the circle $\odot(v, |v, o|)$ will impact the validity of v, objects outside all those largest empty circles can be safely skipped.

In our approach, the determination of NN query result and the valid scope for the result takes two steps and shares one index lookup based on best-first search. In the first step, it retrieves the nearest object. Based on it, a tentative valid scope is formed as the entire service area. Then the largest empty circles of all the corners of the area form the search area for C_p. In the second step, non-result objects are examined in the remaining priority queue to refine the valid scope. Every time when an object o' is found inside the circle $\odot(v, |v, o|)$ associated with a vertex v, v becomes invalid. A bisector between the newly detected object o' and NN object o is formed, which will intersect the original edges (that connect to vertex v) at two new vertices. Then, both the valid scope and the set of complementary objects are revised. Meanwhile, when an index node is covered by any existing largest empty circle, it is expanded and all its children are queued for later examination. Those index nodes or objects that are out of all the existing largest empty circles are skipped from detail examination. The second step terminates when the priority queue becomes empty, and the valid scope is finalized.

For simplicity, we only show the second step, i.e., valid scope refinement and depict it in Figure 9. We assume that the result set R is already retrieved and the priority queue, P, is retained. Based on the service area, a tentative valid scope is formed. Then the search algorithm repeatedly dequeues the head entry ϵ from P and examines it. We check ϵ against all existing largest empty circles. If ϵ is an index node, it is expanded and all its children are queued for later examination. Otherwise (i.e., ϵ is an object), for each of those empty circles where ϵ is covered, the corresponding vertex is removed; the new vertices are then introduced and the set of complementary objects C is updated. Once the queue becomes empty, the search terminates and C becomes the output.

We illustrate the valid scope computation in Figure 8(b) through Figure 8(e). Suppose object d is the NN answer object and the remaining objects, c, a, N_3, b, g in the priority queue are sorted according to their distance to q as a result of the best-first-search. Initially the service area is taken as the tentative valid scope (see Figure 8(b)). The corners of the area, namely, v_1, v_2, v_3 and v_4, form four largest empty circles. Then c, the second nearest object is examined. The bisector $\perp_{c,d}$ intersects the existing edges $\overline{v_1, v_4}$ and $\overline{v_3, v_4}$ at v_5 and v_6 respectively. Then v_4 is replaced with the new vertices v_5 and v_6. The corresponding largest empty circle $\odot(v_4, |v_4, d|)$ is removed and two new circles, $\odot(v_5, |v_5, d|)$ and $\odot(v_6, |v_6, d|)$ are formed (see Figure 8(c)). Similarly after examining a, the search space is refined based on bisector $\perp_{a,d}$, as shown in Figure 8(d). The next step is the refinement based on the next nearest object b, with the bisector $\perp_{b,d}$. Now node N_3 is expanded and f is examined with the bisector $\perp_{f,d}$. Finally, bisector $\perp_{g,d}$ is used and the valid scope is formed with all largest empty circles not covering other objects as depicted in Figure 8(e). The objects a, b, c, f, and g that help to define the largest empty circles are collected as complementary objects.

Algorithm ValidScopeForNN(R,P,p)
Input. An NN result set (R), a priority queue (P), a query point (p)
Local. A set of largest empty circles (L)
Output. Complementary set (C)
Begin
1. initialize L with largest empty circles centered at the corners of service area;
2. while (P is not empty) do
3. $(\epsilon, d) \leftarrow P$.dequene();
4. if ($\exists l \in L, \epsilon \in l$) then
5. if (ϵ is an index node) then
6. forall children c of ϵ do
7. P.enqueue($(c, |p, c|)$);
8. else /* ϵ is an object */
9. update L and C;
10. **output** C;

End.

Figure 9. Algorithm ValidScopeForNN

Compared with the TP-based approach, our approach to compute a valid scope is more efficient. First, it does not need to access the R-tree multiple times. Second, it does not need to validate all vertices. Instead it examines the objects in distance order until no largest empty circle is violated. All those operations are performed within one single index lookup. Finally, we devise Algorithm ValidityTestForNN, depicted in Figure 10, for clients to detect if an existing NN query result remains valid for

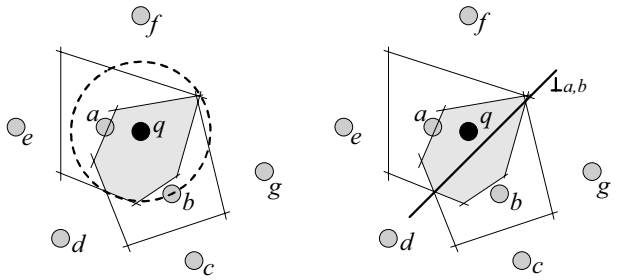
a new query point. The basic idea is to check if any complementary object is found to be closer than the NN object to q. Whenever the algorithm is invoked, it sorts the NN object and all complementary objects. If the first object is no longer the NN result object, it reports invalid for a reevaluation.

Algorithm ValidityTestForNN(R,C,p)
Input. An NN result set (R), a set of complementary objects (C), a query point (p)
Output. valid (if same NN result) or invalid (otherwise)
Begin
1. add R and C to a sorted list;
2. if (the first object of the list is NN result) then
3. **output** valid;
4. else
5. **output** invalid;
End.

Figure 10. Algorithm ValidityTestForNN

B. Valid Scope for kNN Query

kNN query is an extension of NN query. It can be classified into order-sensitive and order-insensitive NN queries. The former is concerned with the distance order of result kNN objects. The objects in the result set are sorted based on their distance to the query point, and the result set $\langle a, b\rangle$ is considered to be different from the set $\langle b, a\rangle$. The latter does not care about the distance order. As long as the same set of objects are included in the result set, the result is considered to be identical. Our discussion starts with valid scope determination for order-insensitive kNN query followed by the discussion of the necessary extension for order-sensitive kNN query.



(a) Order-insensitive kNN (b) Order-sensitive kNN

Figure 11. Valid scope for kNN query

Order-Insensitive kNN Query. The valid scope for an order-insensitive kNN query is an order-k Voronoi cell [1]. Suppose k objects $o_1, \cdots o_k$ form the result set, R_p, of a kNN query evaluated at a query point p and the rest belong to non-result objects, $N_p = O - R_p$. Each NN object o_i has its own valid scope formed as $\diamondsuit(o_i) = \cap_{o \in C_i} HP_{o_i,o'}(o_i)$, where the set of complementary objects $C_i \subseteq N_p$. The order-k Voronoi cell, i.e., $V(p)$, is formed as the intersection of all $\diamondsuit(o_i)$, i.e., $\cap_{o_i \in R_p} \diamondsuit(o_i)$. Figure 11(a) shows a scenario in which a 2NN query is issued at query point, q, and objects a and b form the result set. The shaded area represents the valid scope of the result set, constituted by a's and b's own Voronoi cells. To determine the valid scope for kNN query result, we simply extend our algorithm ValidScopeForNN supporting NN query. Initially, kNN objects need to be identified as the result set. Next, for each individual NN object, o_i, we maintain L_i and C_i ($1 \leq i \leq k$) to keep track of the largest empty circles and set of complementary objects for every o_i. The rest of the logic is similar to that already discussed. The final valid scope is the intersection of individual NN valid scopes. Similar to NN query, the validity check for kNN query is to check if any complementary object appears closer to a query point than any existing result object.

Order-Sensitive kNN Query. To facilitate the checking of order, we partition the valid scope of kNN query result by adding bisectors between the kNN objects. As shown in Figure 11(b), the bisector $\perp_{a,b}$ is introduced and it partitions the valid scope into two smaller portions. Within the valid scope, when the client moves from one divided portion to another, the result is considered to become invalid due to the change of distance order. However, the client needs only to reorder the result objects, since the set of objects still remains the same, as the client is still staying within the combined valid scope of the unordered result set. The query is still self-answerable and no complementary query to the server is needed.

IV. Range Query

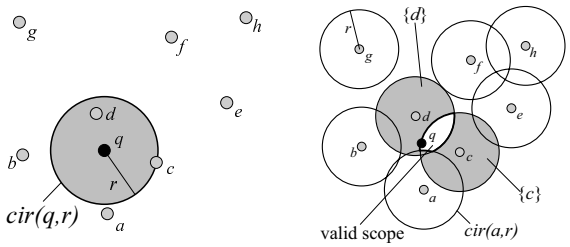
Range queries search for objects within specified query ranges. In the following, we discuss valid scope determination algorithm for answering range queries, with circular query regions.

A. Valid Scope for Range Query

The query region for a range query Q_p with range r is a circle, i.e., $\odot(p, r)$, where p and r represent the query point (i.e., the current user position) and the query range, respectively. To answer a range query, objects located inside $\odot(p, r)$ are collected as the result set R_p and all remaining objects form the non-result set N_p. Every object in R_p is covered by $\odot(p, r)$, whereas all objects in N_p are not.

Given a query range r, let us define the Minkowski region, $\tilde{M}(S)$, for a space S as $\tilde{M}(S) = \{m | \exists s \in S, |m, s| \leq r\}$. For a single object o, it is obvious that $\tilde{M}(o) = \odot(o, r)$ and we call it a Minkowski circle. Corresponding to the query region $\odot(p, r)$, all objects in R_p should have their Minkowski regions $\tilde{M}(o)$ ($o \in R_p$) covering p, i.e., $\forall o \in R_p,\ p \in \tilde{M}(o)$. On the other hand, no non-result object o' in N_p has its Minkowski region, $\tilde{M}(o')$, enclosing p, i.e., $\forall o' \in N_p,\ p \notin \tilde{M}(o')$. As shown in Figure 12(a), a circular search space $\odot(q, r)$ is generated for a range query issued at q with a radius of r. The result objects are c and d. Other objects like a lying outside $\odot(q, r)$ are non-result objects. On the other hand, Figure 12(b) shows the Minkowski circles of all objects. Clearly, only $\odot(c, r)$ and $\odot(d, r)$ cover q while others, e.g., $\odot(a, r)$, do not.

Specifically, the valid scope for p, denoted by $V(p)$, which represents an area in which R_p remains valid, can



(a) $\odot(q, r)$ (or $cir(q, r)$) (b) Minkowski circles

Figure 12. Circular search range and Minkowski circles

be computed as an area equal to the intersection of the Minkowski regions of all the result objects minus those Minkowski regions of the non-result objects, as expressed in Equation (1).

$$V(p) = \bigcap_{o \in R_p} \tilde{M}(o) - \bigcup_{o' \in N_p} \tilde{M}(o') \qquad (1)$$

Continuing with our example, a valid scope for a result set $\{c, d\}$ is shown in Figure 12(b). However, making use of N_p and its Minkowski region to derive a valid scope for the query is totally impractical, due to the need of a large number of object examination operations. To address this, we propose a different approach in this paper. Observing that $A - B \equiv A - A \cap B$, where A and B are two sets, those non-result objects o' with the property that $\tilde{M}(o') \cap \bigcup_{o \in R_p} \tilde{M}(o) = \emptyset$ can be safely discarded as their Minkowski regions do *not* have any impact on the formation of the valid scope. Thus, we introduce a set of complementary objects C_p, i.e., a small and representative subset of non-result objects ($C_p \subseteq N_p$) in place of N_p to derive the valid scope. The formulation of a valid scope can then be revised and stated in Equation (2).

$$V(p) = \bigcap_{o \in R_p} \tilde{M}(o) - \bigcup_{o' \in C_p} \tilde{M}(o') \qquad (2)$$

In Equation (2), the set of complementary objects C_p refer to those objects whose Minkowski regions overlap with those of the result objects, i.e., $\{o' | o' \in N_p \wedge \tilde{M}(o') \cap \bigcup_{o \in R_p} \tilde{M}(o) \neq \emptyset\}$. With R_p and C_p available at the client side, the client can determine whether a new query point p' is within the current valid scope $V(p)$ of query p by checking whether p' stays inside all the Minkowski regions of all the result objects and outside those of the complementary objects. Consequently, it simplifies the validation process and also avoids the need of using complicated polygon representation for the valid scope. In effect, we represent the valid scope on the fly and on demand in the form of an algorithm. Algorithm ValidityTest, as outlined in Figure 13, is devised to perform validity test on the valid scope (i.e., whether $p' \in V(p)$).

Now, the only remaining problem is to compute efficiently for the set of complementary objects, C_p. The derivation of C_p depends on the availability of the result set R_p. Recall that when we compute for R_p, we adopt the best-first search approach. In the process of maintaining the priority queue, all the remaining elements in the queue represent all non-result objects, be it in the form

Algorithm ValidityTest(R,C,p)
Input. NN result set (R), a set of complementary objects (C), a query point (p)
Output. valid (if same NN result) or invalid (otherwise)
Begin
1. if ($\exists o \in R, p \notin \tilde{M}(o)$) then
2. **output** invalid;
3. else if ($\exists o' \in C, p \in \tilde{M}(o')$) then
4. **output** invalid;
5. else
6. **output** valid;
End.

Figure 13. Algorithm ValidityTest

of an actual object, or a minimum bounding box for a collection of objects. As a side effect, N_p is available and ordered in a useful way. Since only those non-result objects whose Minkowski circles overlap with those of all the result objects are taken as complementary objects, the valid scope computation algorithm for range query then becomes quite straightforward. By computing for the two sets, R_p and C_p, in such a way, only one single expensive index lookup is required. Note that we have assumed that the valid scope for the range query is formed when the result set is non-empty. If the result set is empty, a nearest surrounder query [5] can be evaluated to identify all the nearest neighbor objects in all directions with respect to the query point p. Then the area bounded by all those nearest surrounders can be adopted as the valid scope for the empty result set.

Algorithm ValidScopeForRange(R,P,p,r)
Input. NN result set (R), a priority queue (P), a query point (p), a query range (r)
Output. Complementary set (C)
Begin
1. while (P is not empty) do
2. $(\epsilon, d) \leftarrow P$.dequene();
3. if ($\forall o \in R, |o, \epsilon| \leq 2r$) then /* Lemma 1 */
4. if (ϵ is an index node) then
5. forall children c of ϵ do
6. P.enqueue($(c, |p, c|)$);
7. else
8. $C \leftarrow C \cup \{\epsilon\}$; /* ϵ is an object */
9. **output** C;
End.

Figure 14. Algorithm ValidScopeForRange

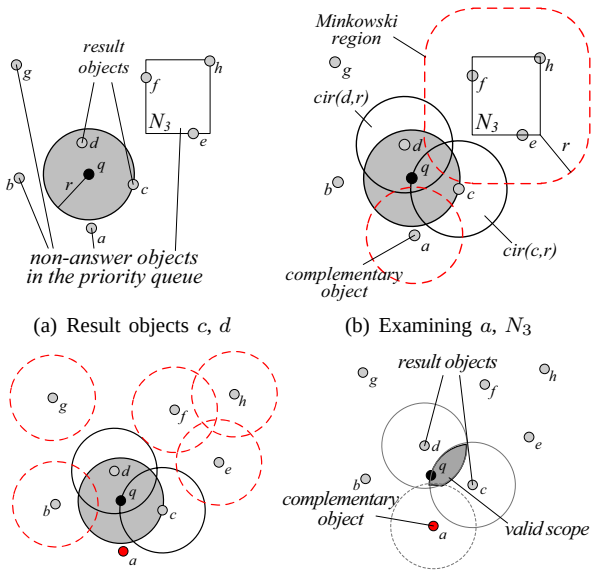
Figure 14 outlines the algorithm that determines the valid scope for range query. The result set is assumed to be determined in prior by best-first search algorithm and the priority queue is retained in order to identify the complementary objects. The algorithm iteratively examines the head entry ϵ from the priority queue P. For each examination, if the Minkowski circle for ϵ overlaps all Minkowski circles of the result objects (line 3), we examine it in greater details (line 4-8); we will ignore it otherwise. Since ϵ can be an MBB, for computational efficiency, we measure the distance bound between ϵ and the result objects, o, instead based on Lemma 1. Then, if ϵ is an index node, it is expanded and all its children are enqueued for future examination (line 4-6). Otherwise, it is incorporated to the current set of complementary

objects, C (line 7). The algorithm terminates when P becomes empty.

Lemma 1: Given a result object, o, the maximum distance of complementary objects from o is bounded by $2r$ for a query with radius r.

Proof. Those complementary objects o' that can affect the valid scope of the result object o should have their Minkowski circles intersecting that of o, i.e., $\odot(o,r) \cap \odot(o',r) \neq \emptyset$. In other words, o' should be at a distance of no more than $2r$ away from o. □

To illustrate how the algorithm works, let us consider a range query at q with radius r, based on a collection of objects in a running example as shown in Figure 3(a). Right after best-first search is completed, the result set R_q includes objects c and d and the priority queue P contains a, N_3, b and g in that order. The object positions are shown in Figure 15(a). First, a, the head entry in P is examined and its Minkowski circle is found to be covered by those of b and c, as in Figure 15(b). Thus, a is collected into the set of complementary objects, C. Next, N_3 whose distances to both c and d are less than $2r$ is expanded into e, f and h. After that, all objects are examined but none of them has their Minkowski circles covered by those of the result objects as shown in Figure 15(c). The search finally terminates. The valid scope is formed by the intersection of the Minkowski circles of two result objects, c and d, and one complementary object, a, as depicted in Figure 15(d).



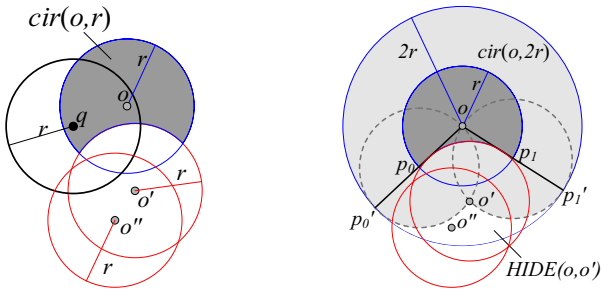
(a) Result objects c, d (b) Examining a, N_3

(c) Remaining objects (d) Valid scope

Figure 15. Determination of valid scope for range query

B. Optimizing the Search Space

We have discussed the process in searching for complementary objects that are used to represent a valid scope purely based on result objects. In fact, we can further prune the search space if the complementary objects are also taken into consideration. Let us consider an example as shown in Figure 16 where the range query is $\odot(q,r)$ and o is the only result object. Based on o, o' and o'' are considered to be complementary objects since their Minkowski circles overlap with $\odot(o,r)$ (also denoted as $cir(o,r)$). However, the area represented by $\bigcap_{o\in R_q} \tilde{M}(o) - \tilde{M}(o')$ is exactly the same as $[\bigcap_{o\in R_q} \tilde{M}(o) - \tilde{M}(o')] - \tilde{M}(o'')$, implying that o'' has no impact on the valid scope. Including o'' to represent a valid scope and for result validity check is therefore redundant. We call objects like o'' in this case *false complementary objects*. In what follows, we discuss an additional filter to identify false complementary objects for removal.



(a) False complementary object o'' (b) Hiding area

Figure 16. Hiding area for range query

We first outline how a false complementary object can be identified for a range query, based on the example shown in Figure 16(a). Based on observation, we can conclude that any object falling inside the white area in Figure 16(b) is a false complementary object as its impact on the valid scope is preempted by that of o'. In our approach, we call this white area the *hiding area* and formalize it as follows. Given a result object, o, and a complementary object, o', a hiding area $HIDE(o,o')$ can be formulated as in Equation (3).

$$HIDE(o,o') = \begin{array}{l} \lhd(\odot(o,2r), \angle_{p_0'op_1'}) - \\ [\lhd(\odot(p_0,r), \angle_{op_0p_0'}) \bigcup \\ \lhd(\odot(p_1,r), \angle_{op_1p_1'})] \end{array} \quad (3)$$

The first term represents a sector ($\lhd$) of a circle $\odot(o,2r)$ with angle at o between p_0' and p_1'; the second and third terms stand for sectors (actually semicircles) centering at p_0 and p_1 with radius r. Finally, all non-result objects that fall inside the hiding areas belonging to the same complementary object for all result objects can be safely ignored.

It is noteworthy that identifying and ignoring false complementary objects can reduce the number of complementary objects to be delivered to the client, thus reducing bandwidth consumption, client storage for representing a valid scope and the computational overhead for result validity. However, the complicated logic of formulating hiding area, especially for range query, would incur significant server processing cost. In the evaluation section, we will study the performance due to optimization.

V. Window Query

Window queries search for objects within a rectangular window centered at the current query point. Riding on

their similarity to range queries, we will discuss the valid scope algorithm applicable to window queries.

A. Valid Scope for Window Query

The query region for a window query Q_p is a rectangular window specified as $\square(p, 2l, 2h)$, where l and h are extents from the query point, p, in x- and y-dimensions, respectively. As a result, the size of the querying region for a window query is $4lh$, as compared to the size of πr^2 in a range query. Since the same concepts for range queries are applicable to window queries, we discuss the algorithm to compute the valid scope for window queries, where the associated Minkowski region is defined in the form of rectangles, which we call Minkowski rectangles.

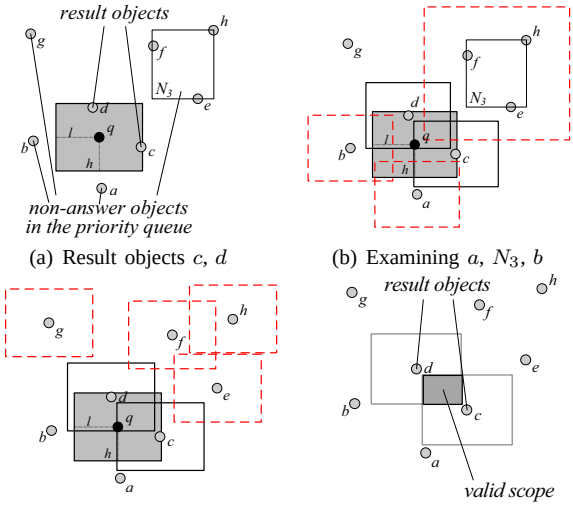
Algorithm ValidScopeForWindow to determine the valid scope for range queries is outlined in Figure 17. It repeatedly examines entries from the priority queue, P and checks them against result objects in R (line 1-8). Different from the processing logic for range queries, the search area is expressed as a rectangle. Here, we identify a set of false victim objects that are discarded from the search for the initial result set as the set of objects that are not located inside the search window, even though they are at a smaller Euclidean distance to the query point p. Before the algorithm is invoked, those false victim objects are reinserted to the priority queue, since they could still contribute to the result, owing to the rectangular shape of the search region. Whenever the Minkowski rectangles of an entry ϵ touch those of the result objects, detailed examination takes place (line 3-8). In place of complicated intersection determination between rectangles, Lemma 2 suggests an efficient way of determination by comparing the distance between two objects along x- and y-dimensions. If ϵ is an index node, all its children are enqueued for later investigation (line 4-6). Otherwise, it is collected into a set of complementary objects, C (line 8). Once P becomes empty, the computation of valid scope is completed, and the set of complementary objects is returned (line 9).

Algorithm ValidScopeForWindow(R,P,p,W,H)
Input. NN result set (R), priority queue (P), a query point (p), window width ($W = 2l$), window height ($H = 2h$)
Output. Complementary set (C)
Begin
1. while (P is not empty) do
2. $(\epsilon, d) \leftarrow P$.dequeue();
3. if ($\forall o \in R, |o, \epsilon|_x \leq 2l \wedge |o, \epsilon|_y \leq 2h$) then /* Lemma 2 */
4. if (ϵ is an index node) then
5. forall children c of ϵ do
6. P.enqueue($(c, |p, c|)$);
7. else /* ϵ is an object */
8. $C \leftarrow C \cup \{\epsilon\}$;
9. **output** C;
End.

Figure 17. Algorithm ValidScopeForWindow

Lemma 2: Given a result object, o, the maximum distances of complementary objects along x- and y-dimensions are bounded by $2l$ and $2h$, respectively.

Proof. Those complementary objects o' that can affect the valid scope should have Minkowski rectangles intersecting that of the result object o, i.e., $\square(o, 2l, 2h) \cap \square(o', 2l, 2h) \neq \emptyset$. In other words, o' should be no more than a distance of $2l$ and $2h$ away from o along the x- and y-dimensions, respectively. □



(a) Result objects c, d (b) Examining a, N_3, b

(c) Remaining objects (d) Valid scope

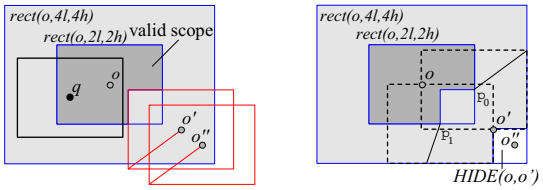
Figure 18. Determination of valid scope for window query

We illustrate the algorithm with an example as shown from Figure 18(a) though Figure 18(d). The window query is issued at q and its width and height are $W = 2l$ and $H = 2h$, respectively. At the beginning, the result set include c and d while the priority queue maintains a, N_3, b and g in sorted distance order from q, as depicted in Figure 18(a). First, as shown in Figure 18(b), a is examined. Its Minkowski rectangle is covered by that of d and it is discarded. Next, N_3 whose Minkowski rectangle is covered by those of c and d is expanded into e, f and h. Similarly, b is examined and it is discarded. Later, as shown in Figure 18(c), all the remaining objects that are not close to the result objects are examined and discarded. Finally, no complementary objects are found for the formation of valid scope as shown in Figure 18(d).

B. Search Space Optimization

As with range queries, we could exploit the concept of the hiding area to prune the search space for false complementary objects to improve the efficiency. We illustrate the formulation of the hiding area for window query in Figure 19. When a complementary object o' is considered, a hiding area is defined as the region behind the complementary object o', i.e., a white area in the figure with respect to a result object. Any object (e.g. o'') staying inside the hiding area that belongs to the complementary object can be safely ignored.

The hiding area for a window query could be formulated in a similar way to that of a range query. Here, $HIDE(o, o')$ can be formulated as in Equation (4), where p_0 and p_1 are the intersection points of the perimeters of $\square(o, 2l, 2h)$ and $\square(o', 2l, 2h)$. Here $\square_{1/4}(o, 2l, 2h, o')$



(a) False complementary object o'' (b) Hiding area

Figure 19. Hiding area for window query

represents a quadrant of rectangle centered at o where o' resides.

$$HIDE(o,o') = \square_{1/4}\,(o,2l,2h,o') - [\square\,(p_0,2l,2h) \cup \square\,(p_1,2l,2h)] \quad (4)$$

Again, identifying and ignoring false complementary objects can reduce complementary object transmission to the client, thus reducing bandwidth consumption, client storage and computational overhead. We study the performance due to optimization in next section.

VI. Performance Evaluation

This section evaluates the performance of our proposed geometric valid scope computation and compare it with state-of-art approach, namely, TP query-based approach as discussed in Section II. Our evaluation mainly focuses on *overall system performance improvement* by adopting valid scope for LDSQs, which is the motivation of our work and *extra overhead* incurred by computing/transmitting the valid scope.

To measure the performance for different aspects, we use five metrics, namely, *query submission rate*, *bandwidth consumption*, *server execution time*, *server I/O cost* and *area of valid scope*. Query submission rate measures the ratio of queries submitted to the server for processing. Bandwidth consumption counts the amount of data in unit of kilobytes downloaded to the clients, including query result and valid scope if needed. Logically, low query submission rate and low bandwidth consumption indicate the effectiveness of approaches in allowing queries to be answered locally. Server execution time measures the time when the query is received by the server to the time when the query is processed and the valid scope is computed, if any. Meanwhile, I/O cost counts the number of pages accessed. Both server execution time and I/O cost measure the overhead incurred by valid scope computation algorithms. Finally, the area of valid scope estimates the coverage in a space that LDSQ results remain identical to the cached LDSQ result. The larger a valid scope is, the more likely new LDSQs are found to be the same as the cached LDSQ result, thus the smaller the query submission rate.

In our evaluation, we implement baseline bare query processing and TP query-based approach in addition to our proposed approach that determines the valid scope based on result objects and complementary objects, according to the geometric relationship. Our implementation is in GNU C++ and the algorithms are labeled as Bare, Valid Scope (TPQ) and Valid Scope (Geo), respectively for presentation convenience. Bare does not maintain any LDSQ result and submits all queries to the server for processing. We use both synthetic and real object sets in our experiments. Two synthetic object sets, namely, Uni and Gau, are generated based on uniform and Gaussian distributions, respectively. Both object sets consist of $10K$ objects. More specifically, we set the mean and standard deviation of Gaussian distribution to 500 and 100, respectively for the synthetic object set. The real object set, obtained from U.S. Census Bureau TIGER/Line [11], contains $11K$ shopping malls across the country. The locations of the objects in these object sets are normalized to a service area of 1000×1000 units. We also fix the size of an object (that includes object location) and object location at 256 bytes and 16 bytes, respectively.

Parameters	Values
Approach	Bare: bare query processing Valid Scope (TPQ): TP query-based approach Valid Scope (Geo): our geometric approach
Object set	Uni : synthetic (uniform, 10000) Gau : synthetic (Gaussian, 10000) Real: real (shopping malls, 11000)
Service area	1000×1000 units
Query	kNN query (k: 1, 4, 16, 64) range query (r: 5, 10, 15, 20) window query ($l = h$: 5, 10, 15, 20)
Server cache	5% of the R-tree size

TABLE I.
Experiment Parameters

Corresponding to the object distributions, we generate client locations where queries are issued and processed. All the four discussed types of queries, namely, NN query, kNN query, range query and window query are evaluated. The value of k in kNN is ranged between 1 (subsuming NN query) and 64 and the radii for range query is varied from 5 up to 20 units. We assume that all window queries are square-shaped and their side lengths are ranged between 5 to 20 units. We run our experiments on Solaris Blade 1000 Workstations equipped with 1GB RAM and SunOS 5.10 operating system. Furthermore, all the experimented object sets are indexed by R-tree with a disk page size of 4KB. In addition, a cache with size equal to 5% of R-tree index size managed by LRU replacement policy is used to alleviate some server I/O cost for query processing and valid scope computation if needed. While we have conducted the experiments on different possible settings, we select a representative set of experiment results to report due to space constraints. Finally, Table I summarizes all the experiment parameters. Unless specified otherwise, the underlined values are used as the default values in our experiments.

In the following, we first examine the overall system performance improvement by adopting the valid scope. Then we study the overhead incurred by the valid scope computation. As to be discussed, valid scope is shown to be useful in avoiding unnecessary LDSQs being submitted to the server. Meanwhile, our approach is shown to out-

perform TP query-based approaches as it can considerably reduce computational and I/O costs.

A. Experiment 1: Effectiveness of Valid Scope

We examine the effectiveness of valid scope to enable a client to answer LDSQs that effectively return the same results, based on the cached LDSQ result. In this experiment, we simulate mobile clients moving in the service area based on a random walk model. Initially 10 mobile clients are randomly allocated in the service area and then they proceed for 100 steps in their movement. Whenever a client completes one step movement, it makes a turn in a random direction and proceeds to the next stop in a distance randomly drawn from 0 to D from the current position, where D is the maximum distance moved and it is a perimeter in this experiment. For smaller values of maximum distance moved, it is more likely that the new LDSQs are issued within the valid scope associated with a cached LDSQ result. In this case, the valid scope should be very effective. Also in each stop, every client issues one LDSQ. As such, each client issues 100 queries during its movement cycle. The value of k in kNN query, the radius of range query, and the side length of window query are set to 4, 15, 15, respectively. The results presented in Figure 20 are obtained by averaging all the experiment results from all 100 queries issued by the 10 clients.

First, Figure 20(a) shows the performance of valid scope for kNN query. In the figure, we can see that the query submission rate of Valid Scope (Geo) is lower than that Bare. Here, Valid Scope (TPQ) is not included for brevity, since it provides the same valid scope as Valid Scope (Geo). Also, we can observe that the query submission rate increases with maximum distance moved for all the evaluated object sets. This is because more new LDSQs are issued out of the valid scope of the current LDSQ result. Next, we study the bandwidth consumption, server execution time and I/O cost averaged by all issued queries. Since some queries can be answered by the clients when they adopt valid scopes, on average, the bandwidth consumed for Valid Scope (Geo) and Valid Scope (TPQ) are the same and lower than that for Bare, as shown in Figure 20(b).

In terms of query submission rate and bandwidth consumption, Valid Scope (Geo) and Valid Scope (TPQ) are equally effective as they generate identical valid scopes. However, as shown in Figure 20(c), Valid Scope (TPQ) incurs a very high execution time. This is attributed to its exhaustive number of TP queries required to refine the valid scope. With the same reasons, a large number of accesses on the R-tree results in high I/O costs, as shown in Figure 20(d). Thus we can observe that Valid Scope (Geo) can save the bandwidth consumption at the expense of server execution time. This is well justified since wireless bandwidth is very scarce and cannot be expanded easily, while server computation power can be increased if more resources are available. Also, we can observe Valid Scope (Geo) demands similar I/O cost as Bare, thereby indicating the I/O efficiency of our approach in processing LDSQ and computing valid scope with a single index lookup.

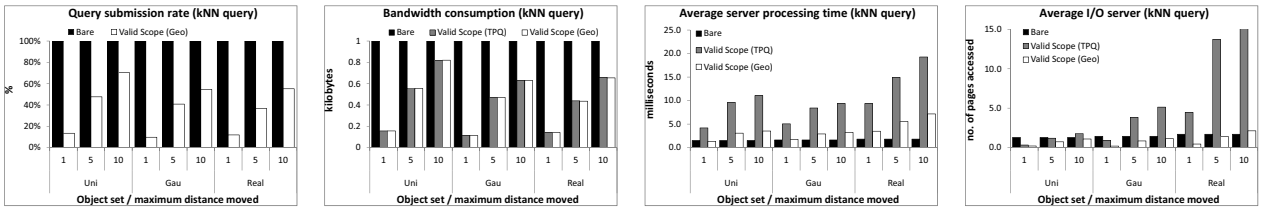
Next, we examine the overall system performance improved for range query. Since Valid Scope (TPQ) does not support range query, it is not included in the evaluation. In general, the similar observation as that for kNN query can be made here, that Valid Scope (Geo) can save LDSQ submission and bandwidth consumption as shown in Figure 21(a) and Figure 21(b). In particular, we examine the improvement induced by search space optimization as discussed in Section IV-B. We include an additional approach, i.e., our valid scope computation approach with **no** search space optimization and label it as Valid Scope (Geo) - NO-OPT. As shown in Figure 21(c), Valid Scope (Geo) - NO-OPT saves some processing overhead. Meanwhile, since no false complementary object is removed, slightly higher bandwidth and I/O costs, as shown in Figure 21(d), are resulted.

Finally, we examine the performance for window query that Valid Scope (TPQ) can support. The results are depicted in Figure 22. Again, valid scope is effective to save LDSQs from being submitted to the server and hence the bandwidth. Valid Scope (Geo) outperforms Valid Scope (TPQ) in terms of execution time and I/O costs. Besides, we can see that Valid Scope (Geo) - NO-OPT reduces execution time by inducing fewer I/O cost and bandwidth.

Based on the evaluations on various queries, valid scope is proven to be effective in mobile environments for reducing the scarce bandwidth consumption. Meanwhile, our proposed valid scope computation that exploits the geometric relationship between result objects and complementary objects and index lookup sharing with query processing clearly outperform more standard TP query-based approach. Besides, our approach following the same principle can also support range queries, but the TP query-based approach cannot.

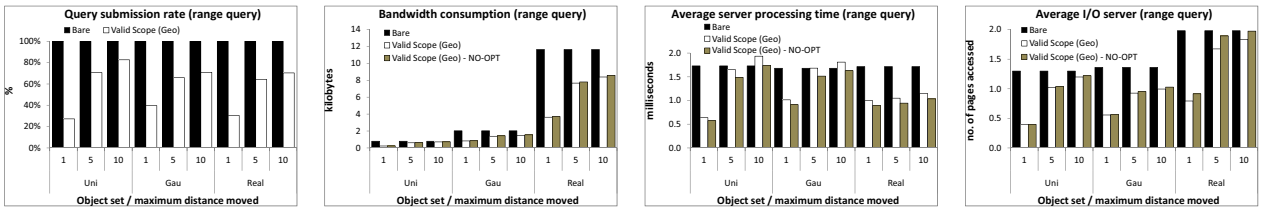
B. Experiment 2: Overhead of Valid Scope Computation

In the second set of experiments, we focus on the overhead incurred by valid scope computation. We issue 100 LDSQs which are independent to one another. The results presented in this section are obtained by averaging those from the 100 queries. First of all, we investigate the area of the valid scope as shown in Figure 23, which explains why the application of valid scope can improve the system performance. In general, for kNN query, the area of valid scope shrinks with increase in the value of k. This is because the valid scope for kNN query is formed by intersecting Voronoi cells of result objects. When more result objects are obtained, the area of intersection among those result objects becomes smaller. However, for range and window query, we made two observations. For Uni, when the search area increases, the area of valid scope shrinks, but it expands for Gau and Real. Recall that a valid scope for range or window query is formed by intersecting Minkowski regions. If the result



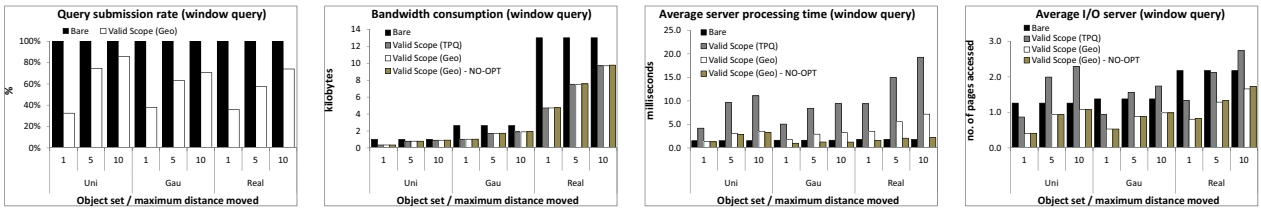
(a) Query submission rate (b) Bandwidth consumption (c) Server execution time (d) Server I/O cost

Figure 20. kNN query



(a) Query submission rate (b) Bandwidth consumption (c) Server execution time (d) Server I/O cost

Figure 21. Range query



(a) Query submission rate (b) Bandwidth consumption (c) Server execution time (d) Server I/O cost

Figure 22. Window query

objects are far apart and are distributed evenly in the space, as is the case of Uni, the overlapping area of their Minkowski regions becomes very small. On the contrary, when the result objects are closely located, especially for very skewed object sets, the overlapping area of their Minkowski regions would be much larger.

Figure 24 and Figure 25 depict the server execution time and I/O cost needed for query processing and valid scope computation. Compared with Bare, which only involves the execution of LDSQs, our approach Valid Scope (Geo) actually incurs a certain degree of extra overhead in computing for the valid scope. Clearly Valid Scope (TPQ) is the worst among all. Furthermore, we can see that Valid Scope (Geo) - NO-OPT leads to an improvement of the server execution time of Valid Scope (Geo), with only a slightly higher I/O cost (about one more page accessed). That means Valid Scope (Geo) - NO-OPT is an attractive alternative for Valid Scope (Geo). Also due to the little extra bandwidth consumed, as shown in Figure 26, as incurred by Valid Scope (Geo) - NO-OPT, we would like to consider the incorporation of false complementary object removal algorithm into the client, as the next step of our work. This can make good use of computation power of mobile clients in relieving the server load.

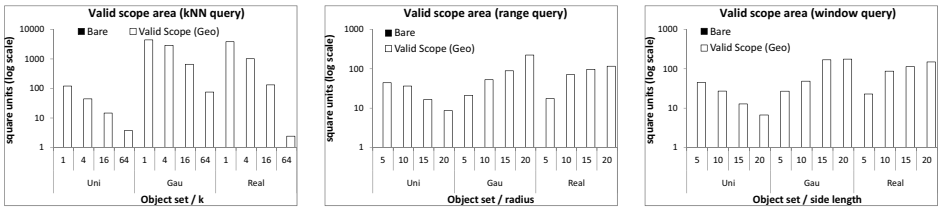
VII. Conclusion

In this paper, we studied an important adaptation of valid scope that contributes in avoiding redundant LDSQs if their results would be identical to the previous LDSQ results returned to mobile clients for usage. We propose new valid scope computation algorithms that exploit the geometric relationship between result objects and some important non-result objects, namely, complementary objects, to derive and represent the valid scope. Algorithms are defined to support NN queries, kNN queries, range queries and window queries. Our approach is very I/O-efficient since it can share the index access with LDSQ processing, by only incurring a single index lookup. For range queries, we considered search space optimization approach in identifying false complementary objects and ignoring them from the construction of a valid scope. The approach is then also applied to window queries. We conducted an extensive set of experiments to evaluate the system performance gained by adopting valid scope and compare our approach with state-of-art approach based on time parameterized queries. As proven in our evaluation, the concept of valid scope can effectively improve the system performance and our proposed algorithms are more efficient than existing ones. As future work of this research, we are investigating into incorporating false complementary object removal algorithm onto the client side to offload the server burden.

References

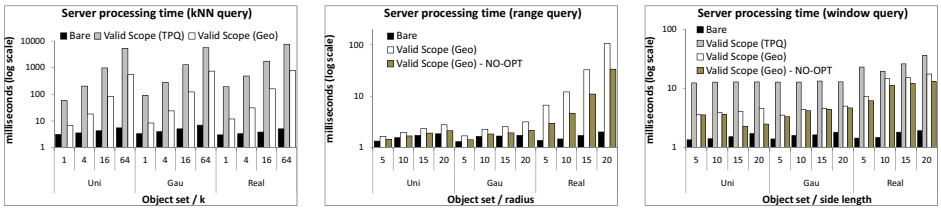
[1] F. Aurenhammer. Voronoi Diagrams - A Survey of a Fundamental Geometric Data Structure. *ACM Computing Survey*, 23(3):345–405, 1991.

[2] S. Dar, M. J. Franklin, B. T. Jónsson, D. Srivastava, and M. Tan. Semantic Data Caching and Replacement. In



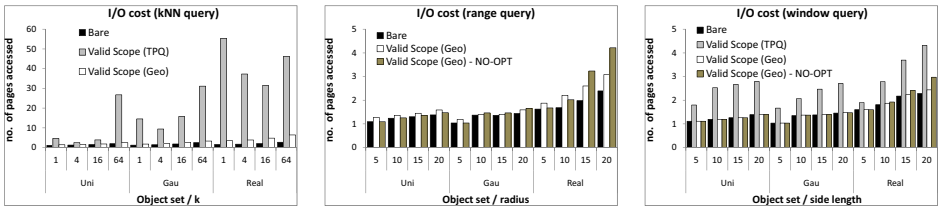
(a) kNN query (b) Range query (c) Window query

Figure 23. Area of valid scope



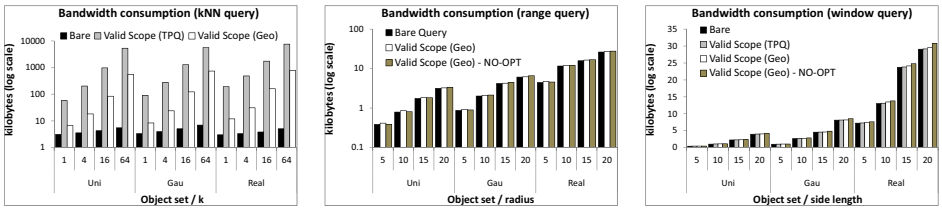
(a) kNN query (b) Range query (c) Window query

Figure 24. Server execution time



(a) kNN query (b) Range query (c) Window query

Figure 25. I/O cost



(a) kNN query (b) Range query (c) Window query

Figure 26. Bandwidth consumption

Proceedings of International Conference on Very Large Data Bases, pages 330–341, 1996.

[3] A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[4] G. R. Hjaltason and H. Samet. Distance Browsing in Spatial Databases. *ACM Transactions on Database Systems*, 24(2):265–318, 1999.

[5] K. C. K. Lee, W.-C. Lee, and H. V. Leong. Nearest Surrounder Queries. In *Proceedings of International Conference on Data Engineering*, pages 85–94, 2006.

[6] K. C. K. Lee, H. V. Leong, and A. Si. Semantic Data Broadcast for a Mobile Environment based on Dynamic and Adaptive Chunking. *IEEE Transactions on Computer*, 51(10):1253–1268, 2002.

[7] K. C. K. Lee, J. Schiffman, B. Zheng, and W.-C. Lee. Valid Scope Computation for Location-Dependent Spatial Query in Mobile Broadcast Environments. In *Proceedings of ACM International Conference on Information and Knowledge Management*, pages 1231–1240, 2008.

[8] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 71–79, 1995.

[9] Z. Song and N. Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *Proceedings of International Symposium on Advances in Spatial and Temporal Databases*, pages 79–96, 2001.

[10] Y. Tao and D. Papadias. Time-Parameterized Queries in Spatio-Temporal Databases. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 334–345, 2002.

[11] U.S. Census Bureau. Topologically Integrated Geographic Encoding and Referencing System. U.S. Census Bureau - TIGER/Line.

[12] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based Spatial Queries. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 443–454, 2003.

[13] B. Zheng and D. L. Lee. Semantic Caching in Location-Dependent Query Processing. In *Proceedings of International Symposium on Advances in Spatial and Temporal Databases*, pages 97–116, 2001.