Location-based Spatial Queries

Jun Zhang[§] Manli Zhu[§] Dimitris Papadias[§] Yufei Tao[†] Dik Lun Lee[§]

[§]Department of Computer Science
Hong Kong University of Science and Technology
Clear Water Bay, Hong Kong
{ zhangjun, cszhuml, dimitris, dlee}@cs.ust.hk

†Department of Computer Science
Carnegie Mellon University
Pittsburgh, USA
taoyf@cs.cmu.edu

Abstract

In this paper we propose an approach that enables mobile clients to determine the validity of previous queries based on their current locations. In order to make this possible, the server returns in addition to the query result, a *validity region* around the client's location within which the result remains the same. We focus on two of the most common spatial query types, namely nearest neighbor and window queries, define the validity region in each case and propose the corresponding query processing algorithms. In addition, we provide analytical models for estimating the expected size of the validity region. Our techniques can significantly reduce the number of queries issued to the server, while introducing minimal computational and network overhead compared to traditional spatial queries.

1. INTRODUCTION

Spatial databases have been extensively studied during the last two decades and several spatial access methods (SAMs) have been proposed [GG98]. Among the most popular ones is the R-tree and its variations, notably the R*-tree [BKSS90]. R-trees can be viewed as multi-dimensional extensions of B-trees. Figure 1 shows an exemplary R-tree for a set of points {*a, b, c,…*} assuming a capacity of three entries per node. Points that are close in space (e.g., *a, b, c*) are clustered in the same leaf node (E_4) represented as a minimum bounding rectangle (MBR). Nodes are then recursively grouped together following the same principle until the top level, which consists of a single root.

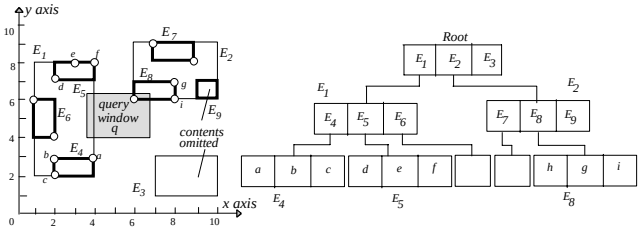


Fig. 1: An R-tree example

R-trees (like most SAMs) were motivated by the need to efficiently process *window queries*, which retrieve all the objects that intersect a window (shaded area in Figure 1). The R-tree answers a window query q as follows. The root is first retrieved and the entries inside it are compared with q. The ones (e.g., E_1, E_2) that intersect q may contain qualifying points and are recursively searched until the leaf (data point) level. Entries not intersecting the query window (e.g., E_3) are not visited.

Another important type of spatial information processing is the nearest neighbor query, which retrieves the data point that is closest to a query point. Roussopoulos et al., [RKV95] propose a branch-and-bound algorithm that searches the R-tree in a depth-first manner. Specifically, starting from the root, all entries are sorted according to their minimum distance (*mindist*) from the query point, and the entry with the smallest value is visited first. The process is repeated recursively until the leaf level where the first potential nearest neighbor is found. During backtracking to the upper levels, the algorithm only visits entries whose *mindist* is smaller than the distance of the nearest neighbor already found. In the example of Figure 2, the algorithm first visits the root entry E_1 (since it has the minimum *mindist*), and then E_4, where the first candidate object (*a*) is retrieved. When backtracking to the previous level, entries E_5 and E_6 are excluded, since their *mindist* is greater than (or equal to) the distance of *a*. Then E_2 and E_8 are accessed, where the actual nearest neighbor (point *h*) is found. Samet and Hjaltason [HS99] develop an improved algorithm, which only visits nodes that may contain the actual nearest neighbor (i.e., in the previous example the algorithm would find point *h*, without first retrieving *a*). Both algorithms can be easily applied for the retrieval of $k > 1$ nearest neighbors.

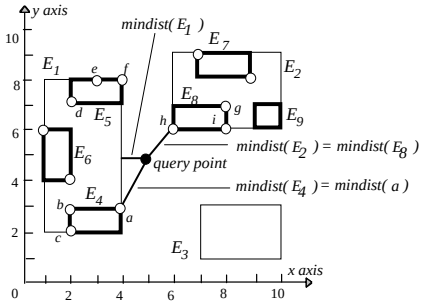


Fig. 2: Nearest neighbor example

The traditional scenario in spatial databases assumes that (i) queries are static and (ii) each query returns a single output and terminates. Consider now an alternative situation where a user with a location-aware mobile device poses a continuous query with respect to his/her current position (e.g., he wants to know the closest restaurant as he moves along). The query is sent to the server, where it is processed and the result is returned to the client via the underlying wireless networks. Due to the mobility of the user, the result may be invalidated immediately as the user's position changes. The conventional approach to attain up-to-date information is to pose a new query to the server after a position update, which could lead to high network overhead and extra processing effort.

ACM SIGMOD'2003, June 9-12, San Diego, California, USA.

Assume in Figure 3 that the user (at position q) issues a nearest neighbor query returning point o. The motivation of this work is that while the user remains in a certain area around the initial position, called the *validity region* (shaded area), the result remains the same. In addition to the query result, the server has to return the validity region of the query, according to which the mobile client is able to determine whether a new query should be issued by verifying whether it is still inside the validity region. Given that the mobile clients have limited storage and processing capabilities compared to the server, it pays off to perform additional processing on the server side (for the initial query) in order to reduce the number of subsequent queries. Furthermore, the representation of the validity region should be compact in order to reduce the network cost, the storage requirements and the computation that must be performed at the client side.

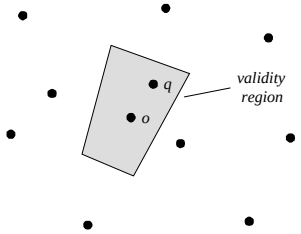


Fig. 3: Validity region

In this paper, we study the problem of determining the validity regions for moving nearest neighbor and window queries on static point datasets, and propose appropriate representations for the corresponding validity regions and query processing techniques. The rest of the paper is organized as follows: Section 2 surveys the related work, while Section 3 defines the validity regions and describes algorithms for nearest neighbors queries. Section 4 discusses window queries, and Section 5 provides an analysis of both query types and formulae for the expected size of the validity regions. Section 6 experimentally evaluates the proposed techniques and Section 7 concludes the paper with a discussion for future work.

2. RELATED WORK

The first spatial query processing techniques for mobile computing were proposed in [ZL01] and [SR01], both dealing with moving nearest neighbor queries on static data. Zheng and Lee [ZL01] pre-compute and store in an R-tree the Voronoi diagram of the dataset. When a nearest neighbor query arrives at the server, the Voronoi diagram is used to efficiently compute the nearest neighbor (e.g., point o in Figure 4).

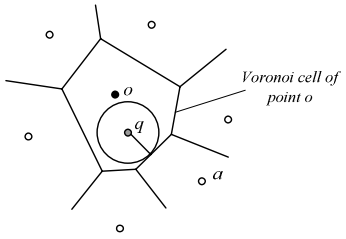


Fig. 4: Example of the [ZL01] technique

In addition to the result, the server sends back to the client the *validity time* T of the result, which is a conservative approximation assuming that the query's speed is below a maximum value. In particular, T is the time that the query point will cross the closest boundary of the Voronoi cell of object o (in which case point a will become the nearest neighbor). A proper value of the maximum query speed, however, is difficult to estimate. A high value will lead to a very short T (reducing the significance of the result), while a low value may cause false misses. Furthermore, the method only deals with single nearest neighbors, as the retrieval of k neighbors would require order-k Voronoi diagrams (for all possible values of k), which are complicated and incur large space overhead.

The technique of Song and Roussopoulos [SR01] does not assume Voronoi diagrams and can be used for any number of neighbors. When a k nearest neighbor query q arrives, the server computes and returns to the client a number $m > k$ of neighbors (using existing algorithms such as [RKV95, HS99]). Let $dist(k)$ and $dist(m)$ be the distances of the k^{th} and m^{th} nearest neighbor from the query point q. If the client re-issues the query at a new location q', it can be easily proven that the new k nearest neighbors will be among the m objects of the first query, provided that $2 \cdot dist(q,q') \leq dist(m)-dist(k)$. Figure 5 shows an example for a 2-nearest neighbor query at location q, where the server returns four results o, a, b and c (the nearest neighbors are o and a). When the client moves to a nearby location q', the two nearest neighbors are o and b. If $2 \cdot dist(q,q') \leq dist(4)-dist(2)$, the client can determine this by computing the new distances (with respect to q') of the four objects, without having to issue a new query to the server. An obvious problem of this approach (not discussed in [SR01]) lies in obtaining a proper value of m. A high value will increase the network overhead and the storage requirements at the client, while a low value may be useless (if it does not reduce the number of queries). In general, m depends on factors, such as data distribution and query frequency, which are difficult to estimate.

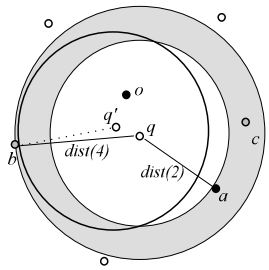
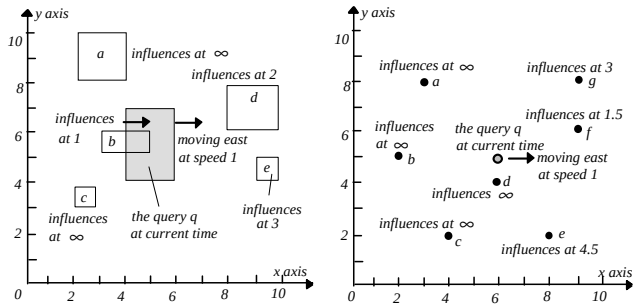


Fig. 5: Example of the [SR01] technique

Given a query moving with steady velocity, [BJKS02, TPS02] return all nearest neighbor results (up to a future timestamp), i.e., the output is a set of tuples $<R_i,T_i>$, where R_i is the set of nearest neighbors during future interval T_i. For this situation (i.e., steady client velocity), the concept of time-parameterized (TP) queries [TP02] can be applied for both window and nearest neighbor queries. In particular when the server receives a request from a client, it executes a TP query and returns $<R,T,C>$, where R is the set of objects satisfying the corresponding spatial query (i.e., current result), T is the validity time of R, and C is the result change at T. From the set of objects in R, and the set of objects C that will cause changes, the client can incrementally compute the next result. Consider, for instance, that a client moving east with speed 1 issues a window query (shaded window of Figure 6a). The server returns $<\{b\},1,\{-b\}>$ meaning that object b currently intersects the query window, but after 1 time unit it will stop doing so (therefore, b should be removed from the result, which will become empty).



(a) TP window query (b) TP nearest neighbor

Fig. 6: Examples of the TP technique

The result of a spatial query changes in the future because some objects "influence" its correctness. For instance, if an object (e.g., *b*) satisfies the query at the current time, it may influence the result when it no longer satisfies it in the future (at time 1). On the other hand, an object not currently in the result (e.g., *d*) may influence the query when it becomes a part of the result (at time 2). Figure 6a shows the influence time of all objects. Some objects, such as *a* and *c*, may never change the result, in which case their influence time is set to ∞. The concept of "influence time" also applies to other types of queries. Figure 6b shows a TP nearest neighbor where the query point *q* is moving east with speed 1. Point *d* is the current nearest neighbor of *q*. In this case, the influence time of an object should be interpreted as the time that it starts to get closer to the query than the current nearest neighbor. For example, the influence time of point *g* is 3, because at this time *g* will come closer to *q* than *d*. Notice that a non-infinite (i.e., different from ∞) influence time does not necessarily mean that the object will change the result; *g* will influence the query at time 3, only if the result does not change before due to another object (actually at time 3 the nearest neighbor is object *f*).

Let $T_{INF}(o, q)$ be the influence time of an object *o* with respect to a query *q*. The expiry time of the current result is the minimum $T_{INF}(o,q)$ of all objects. Therefore, the time-parameterized component (i.e., *C* and *T*) of a TP query can be reduced to a nearest neighbor problem by treating $T_{INF}(o,q)$ as the distance metric: the goal is to find the objects (*C*) with the minimum T_{INF} (*T*). These are the candidates that may generate the change of the result at the expiry time (by adding to or deleting from the previous answer set). T_{INF} for intermediate entries *E* is defined in a way similar to *mindist* in nearest neighbor search: $T_{INF}(E,q)$ is the minimum influence time $T_{INF}(o,q)$ of any object *o* that may lie in the subtree of *E*. Thus, traditional nearest neighbor algorithms (e.g., [RKV95, HS99]) can be applied with appropriate transformations (for details see [TP02]).

These techniques ([TP02, BJKS02, TPS02]), however, presuppose that the future locations of the clients can be calculated using their current movements (i.e., the velocity of the client is known and constant during the lifespan of the query). This assumption may not hold for many applications, where the query velocities are continuously updated as the users change their speed or direction of movement. Motivated by this, in the sequel we describe location-based spatial queries where, instead of time, the validity of the result is determined by the users' location in space. Our methods can capture both nearest neighbor and window queries without any knowledge of the client's velocity.

3. LOCATION-BASED NEAREST NEIGHBORS

In this section, we study the validity regions for nearest neighbor queries and discuss processing algorithms. We start with single nearest neighbor queries, and later we extend our approach to *k*-nearest neighbors. For the following discussion we assume that there exists a spatial index (e.g., R-trees) for the data objects, but no specialized structures (e.g., Voronoi diagrams) for nearest neighbor search. The reasons for this assumption are: i) spatial indices are useful for all query types and not only nearest neighbors; ii) Voronoi diagrams cannot deal efficiently with updates (i.e., a large part of the diagram has to be re-computed for each object update); iii) Voronoi diagrams are inapplicable if the number *k* of neighbors to be retrieved is not known in advance and iv) even if *k* is known, order-*k* Voronoi diagrams are very expensive to compute and incur high space requirements (the server should keep a Voronoi diagram for each possible *k*).

3.1 Validity Region and Influence Sets

Figure 7 shows a single nearest neighbor query *q* whose result (the nearest object of *q*) is point *o*. Let *a* be an arbitrary data point; the perpendicular bisector l_a of *a* and object *o* splits the plane into two half-planes. If the query point *q* moves inside the half-plane which contains point *o*, the distance from *q* to *o* is always smaller than the distance to *a*. The intersection of the half-planes, created by object *o* and all other objects in the dataset, constitutes the validity region of the nearest neighbor query. Formally, we observe that for nearest neighbor queries:

Observation: The validity region *V*(*q*) of a nearest neighbor query *q* is the intersection of the half-planes formed by the perpendicular bisectors of the nearest neighbor of the query point and all the other objects, or equivalently, *V*(*q*) is the Voronoi cell [BKOS97] *VC(o)* of the nearest neighbor point *o* of *q*.

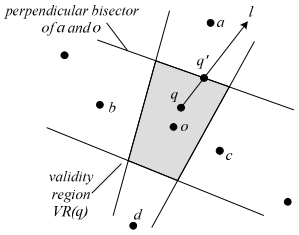


Fig. 7: Single nearest neighbor query example

The representation scheme for validity regions should (i) be as small as possible to reduce the network cost (provided that it captures accurately the shape of the validity region) and (ii) it should facilitate the validity checking which is performed on the client with limited computational capability. We characterize the region by the set of points that determine its edges.

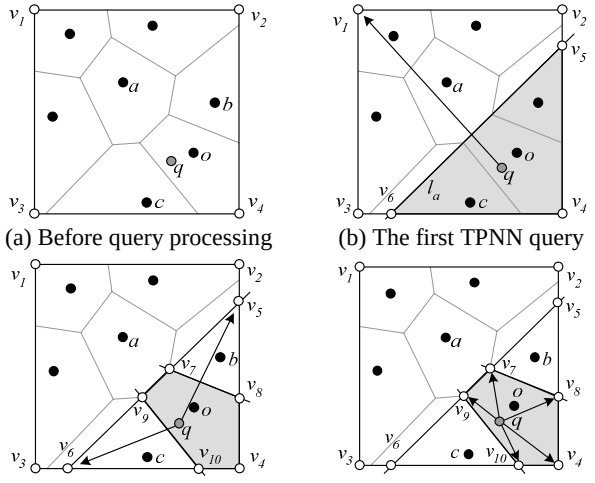
Definition 1: An *influence object* of a query *q* is a data point that *contributes* at least one edge (called the *contribution edge*) to the validity region *V*(*q*). The set of all influence objects is called the influence set $S_{inf}(q)$ of *q*.

In other words, the influence set is the minimal set of objects which determines the validity region. Although adding some other objects to the set does not change the representative region, it is important to eliminate extra objects to reduce the transfer cost. In Figure 7, the influence objects of query *q* are those whose perpendicular bisectors with object *o* constitute the edges of the Voronoi cell *VC(o)*.

Since we do not have the Voronoi diagram for the dataset, we must compute *VC(o)* on the fly. To illustrate this process, consider the query in Figure 7**,** which moves towards direction *l*. When the query reaches position *q'*, its nearest neighbor changes from *o* to *a*. Object *a*, as well as, the distance between *q* and *q'* can be obtained by issuing a time parameterized nearest neighbor (TPNN) query with direction *l* and an arbitrary speed. The TPNN query returns <*R*,*T*,*C*>, where *R*={*o*}, *T* corresponds to the time needed to reach *q'*, and *C*={*-o*, *+a*}. The distance between *q* and *q'* is computed by multiplying the query speed and *T*.

The problem is how to select the directions *l*, in order to obtain the influence points with the minimum number of TPNN queries. The outline of our solution is as follows: (i) the initial validity region is assumed to be the data universe; (ii) a TPNN query with a specific direction *l* (to be discussed shortly) is executed to find an influence object; (iii) the new validity region becomes the intersection of the previous region and the half-plane introduced by the new influence object; (iv) the algorithm terminates when no new influence objects can be found.

Consider a location-based nearest neighbor query *q* (whose nearest object is *o*), as shown in Figure 8a. The initial validity region is the data universe defined by vertices v_1 to v_4. One of the vertices (e.g., v_1) is selected at random, and a TPNN query is issued starting from *q* and pointing towards the chosen vertex. Assume that a new influence object *a* is found by the query, and let l_a be its perpendicular bisector with object *o*. The validity region is then updated to the intersection of the half-plane introduced by l_a and the data universe. The new (triangular) validity region is defined by vertices v_4, v_5 and v_6 in Figure 8b.



(a) Before query processing (b) The first TPNN query

(c) Subsequent TPNN queries (d) Confirmed vertices

Fig. 8: Computation of validity region for nearest neighbor

Next, we choose any one of the three vertices (e.g., v_5) and pose a new TPNN query that retrieves influence object *b* and updates the validity region accordingly. Similarly, we perform the same procedure on v_6, retrieving object *c*. Figure 8c shows the validity region after v_5 and v_6 are processed. Subsequent TPNN queries for vertices v_4, v_7, v_8, v_9 and v_{10} (see Figure 8d) either re-discover existing influence objects (e.g., point *a* for v_7), or fail to discover any point (e.g., v_4). The vertices that do not lead to the discovery of new influence objects are *confirmed* and not considered again. The algorithm terminates when all vertices are confirmed.

Lemma 3.1: The above method: (i) discovers all the influence objects and (ii) does not include any false hits (i.e., the final result does not contain any non-influencing objects) ■

To prove the first part, consider that there is at least an influence object which is not found by the algorithm. Figure 9 shows such an example, where v_1, v_2, v_3, v_4, v_5 and v_6 are the confirmed vertices of the validity region and $a \in S_{inf}$ is not discovered by the algorithm. Since *a* is an influence object, the perpendicular bisector of *a* and *o* cuts the current validity region, meaning that at least one of the existing vertices *v* (e.g., v_1) will be removed by the TPNN query pointing towards *v* (which will discover *a*). This contradicts the fact that *v* is confirmed because the confirmation implies that the TPNN query towards *v* did not discover any new point. Since each TPNN query by definition, either returns an influence object or fails to discover any new object, the proof of the second part of the lemma is trivial.

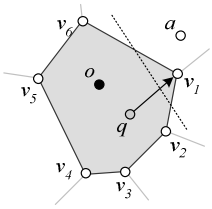


Fig. 9: Illustration for Lemma 3.1

3.2 Query Processing

The server processes a location-based nearest neighbor query in three steps: (i) it performs a nearest neighbor query to obtain the result (i.e., nearest neighbor of the query point); (ii) it iteratively performs TPNN queries to find the influence objects (iii) it returns the query result and the influence set to the client.

Step (i) applies one of the existing nearest neighbor algorithms [RKV95, HS99]. Step (ii) uses the method discussed in Section 3.1 to obtain all the influence objects. A vertex set *V* records all the vertices of the current validity region and each vertex is associated with a flag indicating whether it is confirmed. Initially, *V* contains the four vertices of the data universe. The algorithm then arbitrarily chooses a vertex *v* from *V* and performs the corresponding TPNN query. If the query confirms *v*, its flag is set; otherwise, if the TPNN query returns a new influence object *o*, *o* added to S_{inf}. The new validity region is computed and *V* is updated accordingly. The same process is repeated until all the vertices of *V* are confirmed. Figure 10 shows the pseudo-code for finding the influence set.

```
Algorithm Retrieve_Influence_Set_1NN (q, o)
/* q is the query point, o is the nearest neighbor of q */
1.  S_inf = ∅ // initialize the influence set
2.  V = {universe boundary vertices} // initialize the vertex set
3.  while (V contains non-confirmed vertex)
4.      v = any non-confirmed vertex in V
5.      o_inf = TPNN(q, v, o) // query from q and pointing to v
6.      if (o_inf = ∅ or o_inf ∈ S_inf) confirm v
7.      else  // a new influence object o_inf is discovered
8.          S_inf = S_inf ∪ {o_inf}
9.          update V
10. return S_inf
End Retrieve_Influence_Set_1NN
```

Fig. 10: Algorithm for retrieving the influence set (1NN query)

Lemma 3.2: The algorithm performs $n_{inf} + n_v$ TPNN queries, where n_{inf} is the number of influence objects and n_v is the number of vertices in the final validity region. ■

When the algorithm terminates, there are n_{inf} influence objects and n_v vertices in the validity region. The number of queries required for finding these influence objects is n_{inf}. Furthermore, n_v TPNN queries (that do not discover new influence objects) must be performed to confirm n_v vertices. Since each TPNN query can either find a new influence object or confirm a non-confirmed vertex, the total number of queries is $n_{inf} + n_v$. Notice that if each edge of the validity region is formed by an influence object, $n_{inf} = n_v$; otherwise, $n_{inf} < n_v$, because some edges of the validity region are contributed by the universe boundary (e.g., in Figure 8, $n_{inf} = 3$ and $n_v = 5$).

Finally, the server returns the query result to the client along with the influence objects. The validity checking at the client side is performed by examining whether the new user location is inside the half-plane formed with respect to each influence point, which is equivalent to checking whether the query focus lies in the Voronoi cell of the result.

3.3 Extensions to *k* Nearest Neighbor Queries

The proposed methods can be easily applied to *k* nearest neighbor (*k*NN) queries, where the validity region is the maximal area around the query, where each point has the same set of *k* nearest neighbors (we assume that the order of these *k* neighbors is not important). Figure 11 shows a 2-nearest neighbor query *q* whose result is o_1 and o_2. Let *a* be an arbitrary data point. Consider the intersection of two half-planes, one obtained by l_{a1} (the bisector of *a* and o_1), the other by l_{a2} (the bisector of *a* and o_2) as shown in Figure 11. If the query point *q* moves inside the intersection region, the distances from *q* to o_1 and o_2 are always smaller than the distance to *a*. The intersection of the half-planes, generated by the 2-nearest neighbors of *q* (o_1 and o_2) and all the other objects in the dataset, forms the validity region of the 2-nearest neighbor query.

Observation: The validity region $V(q)$ of a *k*NN query *q* is the intersection of the half-planes formed by the perpendicular bisectors of all the *k* nearest neighbors of the query point and all the other objects, or equivalently, $V(q)$ is the order-*k* Voronoi cell *kVC(q)* [BKOS97] of the query *q*.

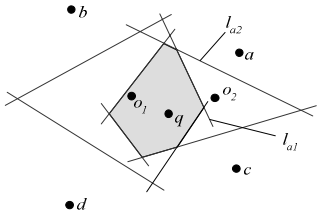


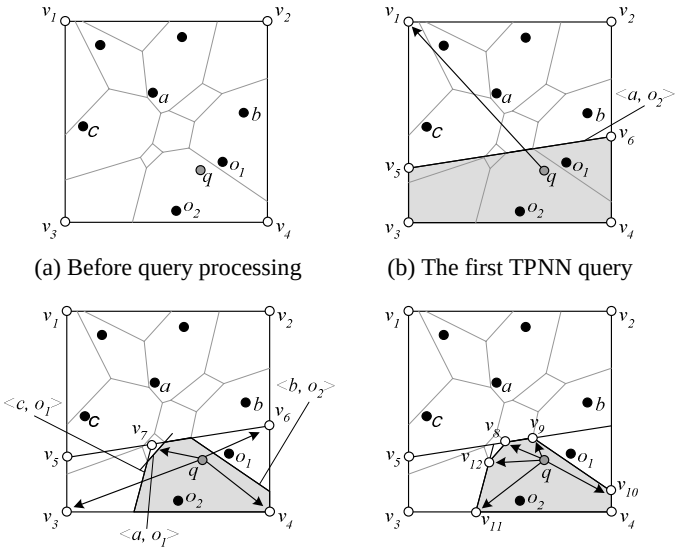
Fig. 11: 2-nearest neighbor query example

The algorithm for computing the influence objects and the validity region, shown in Figure 12, is similar to the one for 1NN queries in Figure 10. The differences are explained as follows: the TPNN query in line 5 of Figure 10 is replaced with a TP*k*NN query. Besides o_{inf}, the TP*k*NN query returns one of the *k* nearest neighbors (denoted as o_r) such that o_{inf} and o_r define the bisector that is nearest to query *q* in the given direction (i.e., from query *q* towards vertex *v*). Consequently, the check on whether o_{inf} is already in S_{inf} (line 6 of Figure 10) is changed to determining whether the pair $<o_{inf}, o_r>$ has been discovered before. An influence pair set S_{inf_p} is used to store the pairs already found.

Algorithm Retrieve_Influence_Set_*k*NN (q, S_o)
/* q is the query point, S_o is the set of nearest neighbors of q */
1. $S_{inf\_p} = \emptyset$ // initialize the influence pair set
2. V = {universe boundary vertices} // initialize the vertex set
3. while (V contains non-confirmed vertex)
4. v = any non-confirmed vertex in V
5. $<o_{inf}, o_r>$ = TP*k*NN(q, v, S_o) // query from q pointing to v
6. if ($o_{inf} = \emptyset$ or $<o_{inf}, o_r> \in S_{inf\_p}$) confirm v
7. else // a new influence object o_{inf} is discovered
8. $S_{inf\_p} = S_{inf\_p} \cup \{<o_{inf}, o_r>\}$
9. update V
10. S_{inf} = all o_{inf} in S_{inf_p}
11. return S_{inf}
End Retrieve_Influence_Set_*k*NN

Fig. 12: Algorithm for retrieving the influence set (*k*NN query)

As an example, consider the 2NN query *q* (whose 2 nearest neighbors are o_1 and o_2) in Figure 13a. Assume that the algorithm first picks v_1 to perform the TP2NN query, which returns influence object *a* together with o_2 (meaning that when the query crosses the bisector of *a* and o_2, object *a* will replace o_2 in the result). The pair $<a, o_2>$ is added to the influence pair set S_{inf_p} and the validity region shrinks as depicted in Figure 13b. The next TP2NN query (towards v_3) discovers pair $<c, o_1>$. No pair is found by the query pointing to v_4, and v_4 is confirmed. The subsequent queries towards v_6 and v_7 return pairs $<b, o_2>$ and $<a, o_1>$ as shown in Figure 13c. Finally, all the vertices are confirmed and the final validity region is obtained as illustrated in Figure 13d.



(a) Before query processing (b) The first TPNN query

(c) Subsequent TPNN queries (d) Confirmed vertices

Fig. 13: Computation of validity region for *k* nearest neighbor

In addition to the influence objects, the algorithm must now send to the client the associated neighbor(s) for each influence object, so that the client can determine if its location is in the appropriate half-planes. It's trivial to adopt Lemma 3.1 to prove the correctness of the algorithm. Similar to Lemma 3.2, Lemma 3.3 gives the number of TP*k*NN queries performed. Notice that the

applicability of the algorithm is beyond the current problem, since it can be used to compute Voronoi cells on-the-fly, using the underlying R-tree.

Lemma 3.3: The algorithm performs $n_{inf\_p} + n_v$ TPkNN queries, where n_{inf_p} is the number of influence pairs and n_v is the number of vertices in the final validity region. ■

Notice that n_{inf_p} equals the number of edges of the final validity region that are not formed by the boundary of the data universe. The number of influence objects is smaller or equal to n_{inf_p}, because an influence object (e.g., a) may contribute multiple pairs, each with respect to a different neighbor.

4. LOCATION-BASED WINDOW QUERIES

The *focus* f of a window query is the centroid of the query window. The validity region $V(q)$ of a query q is the maximal area around the query focus (i.e., $f \in V(q)$) where the query result $R(q)$ does not change. Using the methodology in Section 3, we first define the validity region and influence set, and then propose query processing algorithms.

4.1 Validity Regions and Influence Sets

Let q be a window query with focus f. We call the points that satisfy q *inner objects*, while the points outside the query window *outer objects*. Figure 14 shows an example query where a, b and c constitute the result (i.e., inner objects). The *Minkowski region* of each point (e.g., a) is a rectangle (r_a) identical to the query window whose centroid lies on the corresponding point (a). Observe that when f moves inside r_a, the query result always contains object a. A similar argument also applies to the other inner objects, e.g., when the query focus lies in r_b, b is always in the query result. The intersection of the inner Minkowski regions (shaded area in Figure 14) corresponds to the *inner validity region*, i.e., the maximal area around f where the inner objects will remain in the result. If the query focus exits the inner validity region, the current result is immediately invalidated since at least one of the inner objects will cease to be inside the query window.

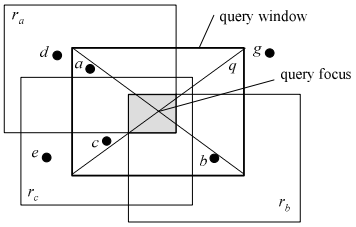


Fig. 14: Inner validity region

Lemma 4.1: The inner validity region of a window query is a rectangle that contains the query focus. ■

Since all inner Minkowski regions contain the query focus, f belongs to their common intersection. Furthermore, the common intersection of a set of rectangles is also a rectangle[1].

The inner validity region is actually a superset of the actual one, since a query window whose focus is in the inner validity region may also contain some additional points (e.g., d and e). In particular, the query starts containing an outer object (e.g., d), if and only if, f enters the corresponding Minkowski region (r_d). Thus, the final validity region (in Figure 15) is the inner validity region after subtracting the outer Minkowski rectangles, i.e., a polygon whose sides are parallel to the x- and y- axes.

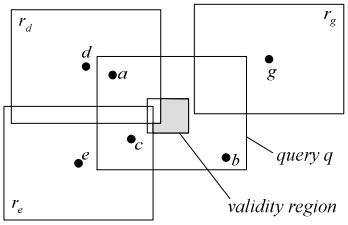


Fig. 15: Validity region

According to Definition 1, the influence set $S_{inf}(q)$ is the minimal set of objects which determines the validity region. $S_{inf}(q)$ for a window query is retrieved in two steps: (i) we first find the set $C_{inf}(q)$ of *candidate* objects that *may* influence the result (i.e., $S_{inf}(q) \subseteq C_{inf}(q)$) and (ii) we compute the final influence set $S_{inf}(q)$ from $C_{inf}(q)$. Lemma 4.2 and Corollary 4.3 identify the inner points that belong to $C_{inf}(q)$.

Lemma 4.2: An inner point is a candidate influence object only if the boundary of its Minkowski region intersects the boundary of the inner validity region. ■

If the Minkowski region r_o (e.g., r_c in Figure 14) of an inner object o (c) does not intersect the boundary of the validity region, then it totally contains it. This, implies, that o does not influence the query, as the result will change before the query focus exits r_o.

Corollary 4.3: The number of inner points in $C_{inf}(q)$ (and $S_{inf}(q)$) is at most 4.

Since the inner validity region is a rectangle, each of its four edges is contributed by one inner point (but an inner point may contribute multiple edges). Actually the inner points in $C_{inf}(q)$ are the ones with the largest/smallest coordinates on the two axes. In the example of Figure 14, a and b contribute two edges each to the inner validity region (because they are the top-left and the bottom-right inner objects). Thus, computing the inner candidate objects reduces to finding the left (right, top, bottom)-most objects. Notice that these points do not necessarily belong to $S_{inf}(q)$, since some edges (e.g., the leftmost edge in Figure 15) of the inner validity region may be eliminated by the insertion of outer points in S_{inf}. In order to compute the candidate outer points, we use Lemma 4.4.

Lemma 4.4: An outer point is a candidate influence object only if its Minkowski region intersects the inner validity region. ■

If the Minkowski region r_o of a candidate outer object o does not intersect the inner validity region, the result will expire before the query focus enters r_o; thus, o cannot influence the result. Points d and e in Figure 15 are candidate outer objects, while $g \notin C_{inf}(q)$.

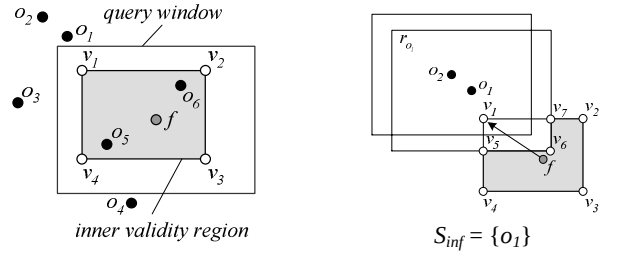
In summary, Lemma 4.2 and 4.4 identify the inner (a,b) and outer (d,e) objects, respectively, that belong to $C_{inf}(q)$. The next step is to find the subset of $C_{inf}(q)$ that comprises $S_{inf}(q)$. Similar to the case of nearest neighbor search, we perform TP window queries on $C_{inf}(q)$ to determine the influence objects. Figure 16a shows an example where $C_{inf}(q)$ contains six candidate influence objects o_1

[1] If all inner points lie on the boundary of the query window, the inner validity region may be a line parallel to the x- or y- axis at f, or a point (i.e., f, if the inner points are at the corner points of the query window).

to o_6. The initial validity region is the inner validity region defined by the vertices v_1 to v_4. One of the vertices (e.g., v_1) is selected, and a TP window query (TPWQ) is issued starting from the query focus f, pointing towards the chosen vertex as depicted in Figure 16b. Object o_1 is returned since its Minkowski region (r_{o1}) is closest to f in the direction from f to v_1. Clearly, o_1 is an influence object because the intersection of r_{o1} and the inner validity region cannot be totally covered by other Minkowski rectangles. After o_1 is added to $S_{inf}(q)$:

(i) The current validity region is updated by subtracting r_{o1}. In the example of Figure 16b, the new validity region is defined by vertices v_2 to v_7.

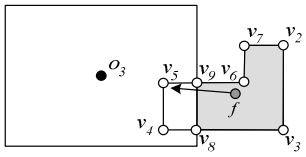
(ii) Some objects in $C_{inf}(q)$ are eliminated because the intersections of their Minkowski regions and the inner validity region are totally covered by the intersection of r_{o1} and the inner validity region (these objects cannot contribute to the final validity region). In the example of Figure 16b, the objects in the upper-left quadrant of o_1 (i.e., o_2), as well as o_1 itself, are removed from $C_{inf}(q)$.



(a) Before query processing

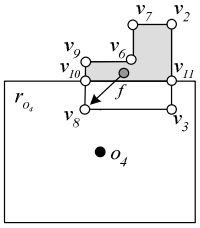
$S_{inf} = \{o_1\}$

(b) TPWQ for v_1



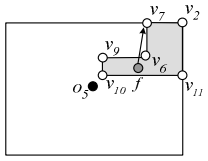
$S_{inf} = \{o_1, o_3\}$

(c) TPWQ query for v_5



$S_{inf} = \{o_1, o_3, o_4\}$

(d) TPWQ for v_8



$S_{inf} = \{o_1, o_3, o_4, o_5\}$

(e) TPWQ for v_7

Fig. 16: Computation of validity region for window query

Next, we perform the same procedure on another vertex over the updated $C_{inf}(q)$. Assume v_5 is selected as illustrated in Figure 16c. The corresponding TP query retrieves o_3 as another influence object. The current validity region is then updated by eliminating vertex v_4 and v_5, and adding v_8 and v_9. We remove o_3 from $C_{inf}(q)$ to avoid considering it again. The subsequent TP window query for vertex v_8 discovers influence object o_4 and shrinks the validity region accordingly, as depicted in Figure 16d. $C_{inf}(q)$ is updated by removing object o_4. The next TP query towards vertex v_{11} does not retrieve any object and v_{11} is *confirmed* as a vertex of the final validity region (the same happens for v_2). Influence object o_5 is found by the TP query for v_7 as illustrated in Figure 16e. Adding o_5 into the influence set does not update the validity region since o_5 is a candidate inner influence object. Thus, v_7 is immediately confirmed. Finally, the algorithm confirms the remaining vertices v_{10}, v_9, v_6 and terminates with $S_{inf} = \{o_1, o_3, o_4, o_5\}$.

Lemma 4.5: The above method: (i) discovers all the influence objects and (ii) does not include any false hits. ■

The proof of Lemma 4.5 is a similar to that of Lemma 3.1 and omitted.

Lemma 4.6: The algorithm performs at most $4+3 \cdot n_{inf}$ TP window queries, where n_{inf} is the number of influence objects. ■

Initially the inner validity region contains 4 vertices. After each influence object is found, we add at most 3 vertices. Thus, the total number of vertices is no more than $4+3 \cdot n_{inf}$. On the other hand, for any vertex, we perform at most one TPWQ towards it. Therefore, the number of TP window queries is no larger than $4+3 \cdot n_{inf}$.

4.2 Query Processing

Once the server receives a window query from a client it performs the following steps: (i) retrieves the query result (i.e., all the inner objects); (ii) computes the candidate inner influence objects; (iii) finds the set of candidate outer influence objects; (iv) among the objects in C_{inf}, selects the ones that belong to S_{inf}, and (v) returns the result and the influence set to the client.

Step (i) is based on traditional window query algorithms (and is, therefore, trivial). Then, the inner candidate objects - step (ii) - are the ones with the minimum/maximum coordinates (Lemma 4.2). In order to find the outer candidate set (step iii) we have to retrieve all objects whose Minkowski regions intersect the inner validity region (Lemma 4.4). This is achieved by applying a new "window" query on the point dataset, where (1) the centroid is the same as that of the inner validity region; (2) the extent on each dimension is the sum of extents of the inner validity region and the initial query; (3) the region of the original window query is subtracted (in order to avoid retrieving the inner objects). Figure 17 illustrates the new query window, continuing the example in Figure 14. It is easy to verify that the Minkowski regions of the points retrieved (e, d) intersect the inner validity region.

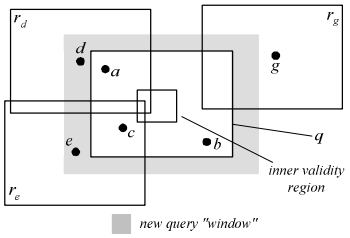


Fig. 17: Obtaining the candidate outer influence points

The elimination of candidate influence objects (step iv) is performed using the algorithm of Figure 18. Main memory TP window queries are repeatedly issued towards the non-confirmed vertices. A vertex is confirmed when the TP query towards it: (i) does not find any influence object; or (ii) discovers an inner influence object.

Algorithm Influence_Checking ($C_{inf}(q)$)
1. $S_{inf} = \emptyset$ // initialize the influence set
2. V = {inner validity region vertices} // initialize the vertex set
3. while (V contains non-confirmed vertex)
4. v = any non-confirmed vertex in V
5. o_{inf} = TPWQ(q, v, $C_{inf}(q)$) // TPWQ from f to v over $C_{inf}(q)$
6. if ($o_{inf} = \emptyset$) confirm v
7. else
8. $S_{inf} = S_{inf} \cup \{o_{inf}\}$
9. if (o_{inf} is an inner object)
10. $C_{inf}(q) = C_{inf}(q) - \{o_{inf}\}$
11. confirm v
12. else
13. $C_{inf}(q) = C_{inf}(q) - \{$objects in pruning region of $o_{inf}\} - \{o_{inf}\}$
14. update V
15. return S_{inf}

End Influence_Checking

Fig. 18: Algorithm for checking the influence objects

Finally the server returns to the client the result of the query, together with S_{inf}. The client can then determine whether the query result is still valid by examining if the focus f' of a future query is inside the validity region. This checking does not require the actual shape of the validity region, i.e., the previous result is still valid if f' is inside the Minkowski rectangles of all influence inner objects and outside the Minkowski rectangles of all outer influence objects. Since, as shown in the experimental evaluation, the number of influence objects for the vast majority of window queries is 4, the computational cost at the client side is minimal.

A last remark concerns the situation that the query result is empty (i.e., there is no inner object), in which case the inner validity region corresponds to the whole data space. The problem now is that there is no "bound" on the how far from the query focus we should search for outer influencing objects. In Figure 19, for instance, depending on the movement of the user, all data objects (o_1 to o_{11}) may potentially influence the query, i.e., the validity region is the whole data space except the Minkowski rectangles (r_{o1} to r_{o11}). Clearly, returning all those points to the client is too expensive and such cases are handled separately. In particular, we perform a nearest neighbor query and find the Minkowski region r that is closest to f. Then, the validity region is approximated by a circle centered at f with radius $mindist(r)$. This process is illustrated in Figure 19 where the shaded region is the conservative approximation of the validity region.

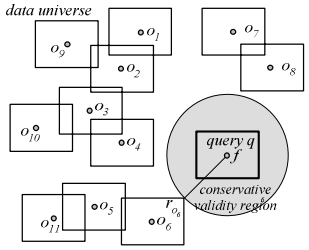


Fig. 19: Conservative validity region

5. ANALYSIS OF LOCATION-BASED QUERIES

The expected size of the validity region is very important for several reasons. First, if the expected size is very small it may not be worth the extra computation performed at the server, or the additional network cost for transferring the validity region. Furthermore, the size of the validity region determines the cost (in terms of the number of disk accesses) at the server; hence, it is needed for query optimization. In this section, we discuss the expected size of the validity region and the query cost of both query types on uniform and non-uniform data. We assume the cardinality of the dataset is N.

We start our discussion with location-based NN queries. The validity region for a NN query is a Voronoi cell (or an order-k Voronoi cell). The expected area $E(A_{VR})$ of an order-k ($k \geq 1$) Voronoi cell for uniform data [OBSC00]:

$$E(A_{VR}) = 1/[(2k-1)N] \qquad (5\text{-}1)$$

The derivation of the cost of location-based nearest neighbors is straightforward given the previous results for kNN and TP kNN queries [BBKK97, WSB98, B00, BBK+01, TP02]. Recall that, as shown in Lemma 3.2 (3.3), a single (k) NN involves one ordinary nearest neighbor query, followed by $n_{inf} + n_v$ ($n_{inf\_p} + n_v$) TP queries. Hence, the total number of node accesses is the sum of every individual (nearest neighbor or TP) query.

We continue with window queries on data uniformly distributed in a square unit universe. Figure 20 illustrates an example query q with focus f over a dataset of points (a, b, …, e, g). Let $dist(\theta)$ be the distance that q must travel towards a direction with angle θ ($0 \leq \theta < 2\pi$) before it reaches the boundary of the validity region $V(q)$ (at which time the result of q is invalidated). If f' is the location where the focus crosses the boundary of $V(q)$ (at angle θ), $dist(\theta)$ equals the distance between f and f'. Figure 20 also demonstrates the corresponding query q' located at f', which covers point e, and changes the initial result {a,b,c} to {a,b,c,e}.

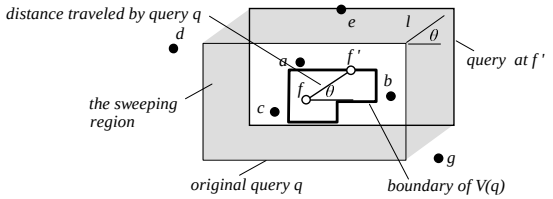


Fig. 20: The sweeping region of a window query q

The area A_{VR} of the validity region can be obtained by integrating $dist(\theta)$ over all θ in [0, 2π) as shown in equation (5-2).

$$A_{VR} = \int_0^{2\pi} \pi \left[dist(\theta) \right]^2 \cdot \frac{d\theta}{2\pi} = \frac{1}{2} \int_0^{2\pi} \left[dist(\theta) \right]^2 \cdot d\theta \qquad (5\text{-}2)$$

Hence, the expected area $E(A_{VR})$ of the validity region can be represented as:

$$E(A_{VR}) = \frac{1}{2} \int_0^{2\pi} E\left[dist(\theta)^2 \right] \cdot d\theta \qquad (5\text{-}3)$$

To derive $E[dist(\theta)^2]$ (along a fixed direction angle θ), we need the probability $P\{dist(\theta) \leq \xi\}$ that the query focus travels no more than distance ξ before crossing the boundary of $V(q)$. It is easier to compute the complement probability $P\{dist(\theta) > \xi\}$ (=1–$P\{dist(\theta) \leq \xi\}$) that the query focus must travel at least ξ (to cross the boundary). We define the *sweeping region* $SR(\xi,\theta)$ as the area swept by the edges of q during traveling distance ξ at direction θ (shaded area in Figure 20).

A crucial observation is that $dist(\theta) > \xi$ if and only if there cannot be any point lying inside $SR(\xi,\theta)$ (otherwise the result of the

query would have changed earlier). Consequently, $P\{dist(\theta)>\xi\} = (1-P_{single})^N$, where P_{single} is the probability that a single data point falls in $SR(\xi,\theta)$, and equals the area of the sweeping region (since we assume uniformity), or formally:

$$P_{single} = 2\xi\left(q_y \cos\theta + q_x \sin\theta\right) - \xi^2\left(\cos\theta \sin\theta\right) \quad (5\text{-}4)$$

where q_x and q_y denote the extents of q along the x- and y-axes, respectively. From equation (5-4), $P\{l(\theta)>\xi\}$ can be obtained as:

$$P\{dist(\theta) \le \xi\} = 1-\left[1-\left[2\xi\left(q_y \cos\theta + q_x \sin\theta\right) - \xi^2\left(\cos\theta \sin\theta\right)\right]\right]^N$$

Taking the derivative of $P\{dist(\theta)\le\xi\}$ with respect to ξ, we obtain its probability density function $p(dist(\theta)=\xi)$, after which we are ready to derive $E[dist(\theta)^2]$:

$$E\left(dist(\theta)^2\right) = \int_0^\infty \xi^2 \cdot p\left(dist(\theta)=\xi\right) d\xi \quad (5\text{-}5)$$

Equation (5-5) holds for all direction angles θ. Hence, by applying it to equation (5-3) we obtain the expected size of the validity region $E(A_{VR})$.

The above analysis can be extended to predict the number of node accesses in answering a location-based window query using R-trees. Recall that our algorithm consists of two steps: the first step retrieves the inner points covered by the query and calculates the inner validity region; the second step searches for the candidate outer points in a "marginal" rectangle computed based on the inner validity (see Figure 17). The number of node accesses in the first step is merely the cost of a traditional window query, which has been studied extensively in the literature [TSS00]. To estimate the cost of the second step, we need to obtain the expected extent of the inner validity region (which as mentioned in Section 4 is a rectangle). As illustrated in Figure 21 (using the dataset and query of Figure 20), the distance between the left boundary of the inner validity region and the original query window equals the distance $dist_{x+}$ that q travels (towards the positive direction of the x-axis) until its left edge hits any inner point (i.e., point c in Figure 21).

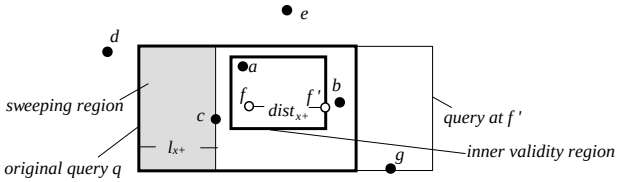


Fig. 21: The extents of the inner validity region

The expected distance of $dist_{x+}$ should be such that, the number of points in the sweeping region (which in this case is the area swept by the left edge of q) contains exactly one data point. Thus, for uniform data, $dist_{x+}$ can be obtained by solving the equation $q_y \cdot dist_{x+} = 1/N$, where q_y is the query extent on the y-axis. Similarly, we can obtain the expected distances $dist_{x-}$, $dist_{y+}$, $dist_{y-}$ that the query needs to travel towards the other directions before its result is invalidated for the first time. Let q_x be the query extent on the x-axis. Then, we have:

$$dist_{x-} = dist_{x+} = 1/\left(N \cdot q_y\right),\ dist_{y-} = dist_{y+} = 1/\left(N \cdot q_x\right) \quad (5\text{-}6)$$

The search region (i.e., the marginal rectangle) in the second step of query processing corresponds to an extended rectangle q' with extents $(q_x+dist_{x+}+dist_{x-})$ and $(q_y+dist_{y+}+dist_{y-})$ on the x- and y-axes respectively, minus the original query q. Hence the number of node accesses for the second query equals the number of R-tree nodes $NA_{intrsct}(q')$ that intersect q', minus the number of nodes $NA_{cont}(q)$ that are fully contained in q. Given the extents of q' and q, $NA_{intrsct}(q')$ and $NA_{cont}(q)$ can be computed using existing models.

The above analysis for nearest neighbor and window query can be adopted for non-uniform data with the aid of histograms. For example, *Minskew* [APR99] partitions the space into a set of disjoint buckets, such that each bucket corresponds to a rectangular region in which the data distribution is almost uniform. Every bucket stores the number of points that fall inside its extent. Our formulae can be applied in conjunction with Minskew (or other histograms) by replacing the data cardinality N in the models with N' obtained from the buckets which contain the data required for the query. Let $b.N$ and $b.Area$ be the number of points in bucket b and the area of its region, respectively. If a query visits buckets $b_1, b_2, \ldots, b_n$, N' is given by the following formula:

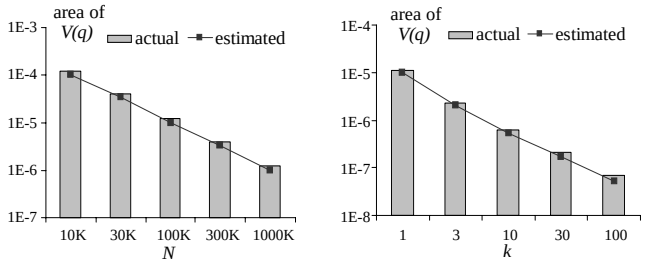
$$N' = \sum_{i=1}^{n} b_i.N \Big/ \sum_{i=1}^{n} b_i.Area \quad (5\text{-}7)$$

For nearest neighbors, we start with the bucket containing the query point and we gradually include its neighboring buckets, until they contain enough points (with respect to the number of neighbors to be retrieved). For window queries, $b_1, b_2, \ldots, b_n$ are the buckets that intersect the boundary of the query window.

6. EXPERIMENTS

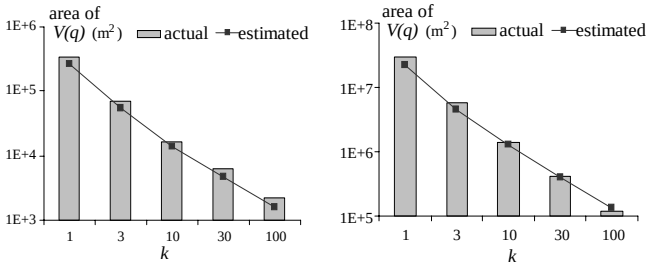
In this section, we evaluate the effectiveness of location-based queries and the efficiency of the proposed algorithms. We use uniformly distributed points in a square unit universe, and two real datasets [Web]: (i) *GR* that contains 23,268 points obtained by taking the centroids of street segments in Greece in a data space 800km×800km; (ii) *NA* containing 569,120 populated places of North America in a data universe of approximately 7000km×7000km. The performance of the algorithms is measured by executing workloads of 500 queries, whose distribution conforms to the distribution of the data objects; window queries have square shape. For real datasets, we employ the analytical models of Section 5 in conjunction with the *Minskew* histogram. The number of buckets for each histogram is set to 500 (constructed from 10,000 initial cells). The R*-tree implementation is based on [BKSS90] with a page size of 4k bytes resulting in a node capacity of 204 entries.

The first set of experiments demonstrates the experimental and estimated sizes of the validity regions for location-based nearest neighbor queries. We first measure the (estimated and experimental) area of the validity region for uniform datasets. In Figure 22a we fix the number k of nearest neighbors to 1 and vary the data cardinality from 10k to 1000k. As expected, the area of the validity region drops linearly with cardinality since the number of Voronoi cells increases (while the area of the data space remains constant). In Figure 22b, queries with different k values are performed on the uniform dataset with N=100k. The validity region shrinks almost linearly with k, which agrees with the observation [OBSC00] that the expected area of an order-k Voronoi cell for uniform data is inversely proportional to $2k$-1. Similar results, regarding the size of the validity region versus k, are obtained from the real datasets as shown in Figure 23. The estimations are accurate in all cases.



(a) Area of $V(q)$ vs. N (b) Area of $V(q)$ vs. k

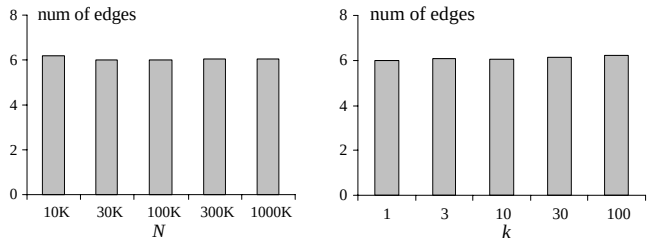
Fig. 22: Area of the $V(q)$ of nearest neighbors (uniform data)



(a) *GR* dataset (b) *NA* dataset

Fig. 23: Area of $V(q)$ vs. k (real data)

Figure 24a (b) illustrates the number of edges in the validity region as a function of N (k) for uniform datasets with k=1 (N =100k). Since validity checking at the client involves determining whether the current position of the client is still inside all the half-planes, the number of edges is a measure for the computation cost at the client side. This number is around 6 under all settings, which is consistent with previous findings [A91, OBSC00] that the average number of edges in a Voronoi cell (or an order-k Voronoi cell) is 6 for uniform datasets.

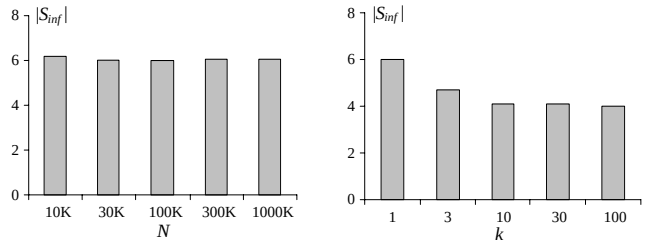


(a) number of edges vs. N (b) number of edges vs. k

Fig. 24: Number of edges of $V(q)$ of nearest neighbors (uniform)

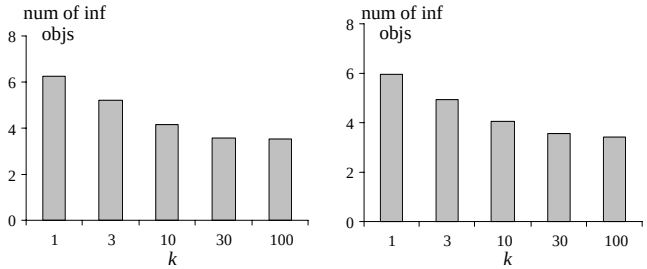
Figure 25 studies the number of influence objects $|S_{inf}|$ for uniform datasets. As shown in Figure 25a, $|S_{inf}|$ for single nearest neighbor queries is approximately 6 for all cardinalities, which is expected since the number of influence objects for a single nearest neighbor query equals the number of edges of the corresponding Voronoi cell. According to Figure 25b, $|S_{inf}|$ decreases to 4 for $k \geq 10$ (N=100k). This is because for $k > 1$, an influence object may contribute more than one edge (since it can form a perpendicular bisector with any of the k nearest objects of the query), while the total number of edges remains around 6. The same results are confirmed by the real datasets in Figure 26.

Next we evaluate the cost of nearest neighbor search on the server side using uniform datasets. Figure 27a (b) shows the number of node (page) accesses as a function of cardinality for k=1 (using an LRU buffer equal to 10% of the R-tree size). The number of node accesses for TPNN queries is about 12 times that of the regular nearest neighbor query because, on the average we need 6 TPNN queries to retrieve the influence objects and another 6 queries to confirm the vertices of the validity region. The buffer reduces the actual cost of the TP queries significantly, since all the queries access similar parts of the data space. Thus, given a relatively small buffer, the overhead imposed by location-based nearest neighbor queries is not significant.



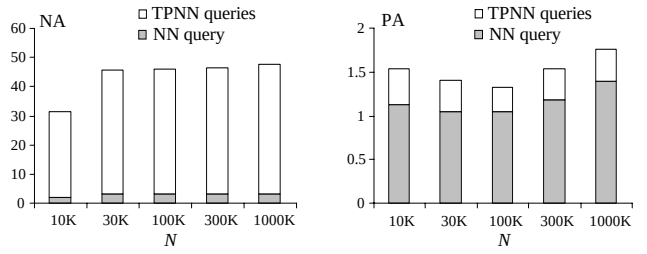
(a) $|S_{inf}|$ vs. N (b) $|S_{inf}|$ vs. k

Fig. 25: $|S_{inf}|$ of nearest neighbor queries (uniform data)



(a) *GR* dataset (b) *NA* dataset

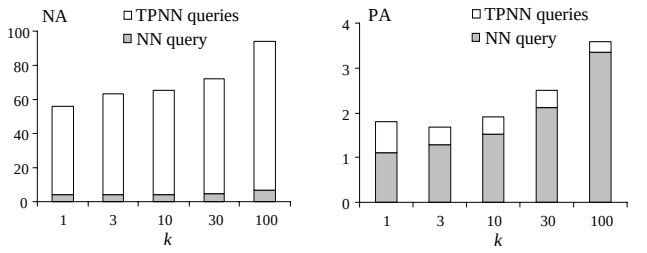
Fig. 26: $|S_{inf}|$ vs. k (uniform data)



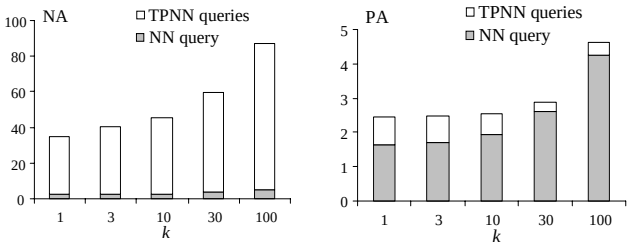
(a) NA vs. N (b) PA vs. N

Fig. 27: Cost of location-based nearest neighbors (uniform data)

In Figure 28 we evaluate the cost of query processing with different values of k using the real datasets (node and page accesses using 10% LRU buffer). Although the number of TPNN queries performed is almost constant (about 12 independently of k) the number of node accesses increases with k, because the cost of each TPNN (and regular nearest neighbor) query increases. The buffer, however, absorbs most of the cost of TPNN queries.



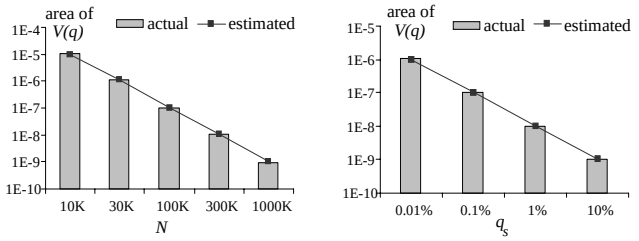
(a) NA vs. k (*GR* dataset) (b) PA vs. k (*GR* dataset)



(c) NA vs. k (*NA* dataset) (d) PA vs. k (*NA* dataset)

Fig. 28: Cost of location-based nearest neighbors (real data)

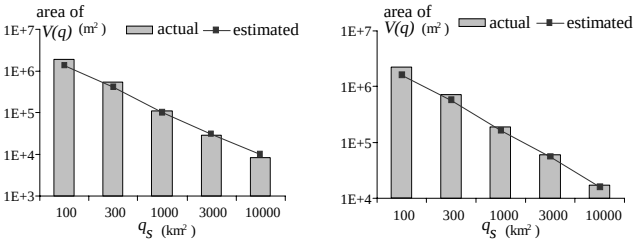
The remaining experiments refer to location-based window queries. Figure 29a fixes the size q_s of the query window to 0.1% of the data space area, and varies the dataset cardinality N for uniform data. Figure 29b shows the size of the validity region as a function of q_s for the 100K dataset. The size of the validity region decreases both with N and q_s. Large N implies high data density, meaning that the results change frequently as the query focus moves. Similarly, a large query is likely to contain numerous objects near the boundary of the window, which will lead to its invalidation. The estimated values are very accurate in all cases.



(a) Area of $V(q)$ vs. N (b) Area of $V(q)$ vs. q_s

Fig. 29: Area of $V(q)$ for window queries (uniform data)

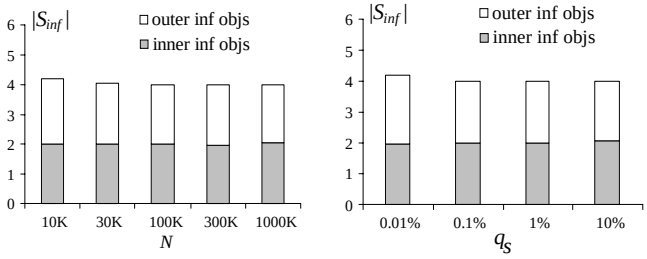
Figure 30 shows the size of the validity region for the two real datasets as a function of q_s, which ranges in 100km^2~10000km^2. The trends are similar to Figure 29b, and the estimation is again accurate despite the large skewness of the datasets. Notice that the sizes of the validity regions are rather large (9,100m^2~1.7×10^{6}m^2 for *GR* and 17,000m^2~2.1×10^{6}m^2 for *NA*), indicating the applicability of the proposed techniques in realistic settings.



(a) *GR* dataset (b) *NA* dataset

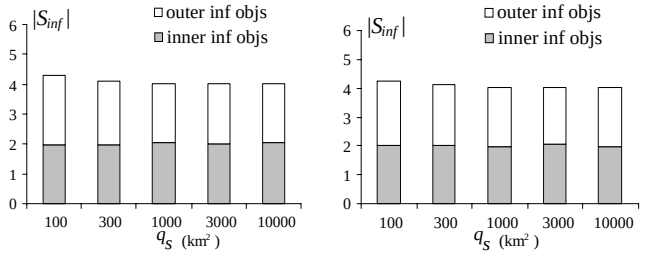
Fig. 30: Area of $V(q)$ vs. q_s for window queries (real data)

Next, we evaluate the number of influence objects $|S_{inf}|$ for window queries. Figure 31a shows $|S_{inf}|$ as a function of dataset cardinality with q_s = 0.1% of the data universe. Figure 31b illustrates $|S_{inf}|$ as a function of q_s for a uniform dataset of 100k points. Figure 32 shows $|S_{inf}|$ for datasets *GR* and *NA* by varying q_s from 100km^2 to 10000km^2. Under all settings, there exist on the average two outer and two inner influence objects, implying that the extra network cost for transmitting the validity region (influence objects) is negligible and that the validity checking at the client side incurs little computational overhead.



(a) $|S_{inf}|$ vs. N (b) $|S_{inf}|$ vs. q_s

Fig. 31: $|S_{inf}|$ for window queries (uniform data)



(a) *GR* dataset (b) *NA* dataset

Fig. 32: $|S_{inf}|$ vs. q_s for window queries (real data)

In order to explain this, consider the example of Figure 33, where an outer object (e.g., a) lies in one of the shaded regions of the extended query window (used to retrieve the candidate outer objects as shown in Figure 17). The Minkowski region of the object eliminates an entire edge of the inner validity region, i.e., the outer object replaces an inner candidate for the influence set, meaning that the total number of influencing objects remains 4. Only objects at the corners of the extended query window can result in non-rectangular validity regions. In general such points are few and for most queries they do not exist.

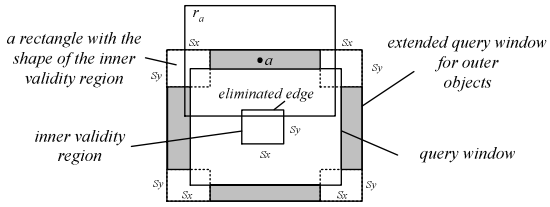
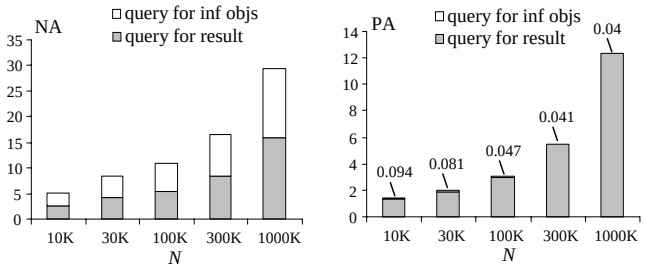


Fig. 33: An example of a window query

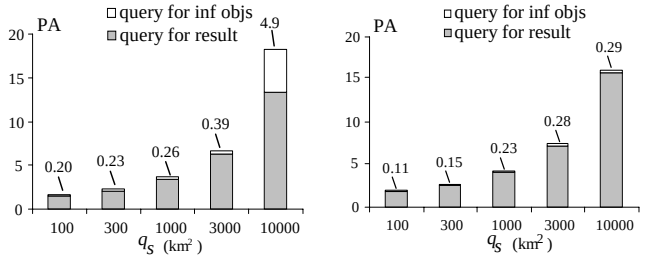
Finally, we study the I/O cost (which is the dominant factor) of location-based window queries at the server side. Figure 34a shows the number of node accesses (NA) as a function of N for uniform datasets. The processing involves two window queries, one for retrieving the result and one for the (candidate) outer influence objects. In Figure 34b we repeat the same experiment using an LRU buffer equal to 10% of the R-tree size (the numbers on the columns are the page faults caused by the queries for the candidate outer objects). Clearly, the cost of the second query drops significantly because most of the accessed nodes are already in the buffer after the first query has been performed. The same can be observed from Figure 35, which shows the number of page accesses for the real datasets as a function of q_s. The only case that the second query has non-trivial cost is for the GR dataset and q_s = 10,000, because, due to the large query size, the buffer is not

enough for all nodes around the query region. In summary, the experiments confirm that our approach incurs minimal overhead to the server if a buffer is used.



(a) NA vs. N (b) PA vs. N

Fig. 34: Cost of window queries vs. N (uniform data)



(a) *GR* Dataset (b) *NA* Dataset

Fig. 35: Page accesses of window queries vs. q_s (real data)

7. CONCLUSION

This paper proposes the concept of location-based spatial queries for mobile computing environments. When a client issues such a query, the server returns, in addition to the result, a validity region for which this result is valid. Thus, before the client issues a new query at another location, it checks whether it is still in the validity region of a previous query; if yes, it can re-use the result. The experimental evaluation confirms the applicability of the proposed approach and shows that the computational and network overhead with respect to traditional queries is small.

We believe that this work is a first but important step towards an important research area. Although spatial queries have been extensively studied, to the best of our knowledge, there exists no previous work that studies validity regions. This concept can be extended to other types queries; for instance, region queries (e.g., find all restaurants within a 5km radius). In this case, the problem is more complex, conceptually and computationally, since the validity region is defined by arcs resulting from cycle intersections.

The incremental computation of the query result based on validity regions is another interesting topic for future work. Consider that a mobile client sends a query to the server immediately after it exits the validity region. It is likely that the new result has significant overlap with the previous one. The incremental computation of the query results and the transfer of the delta (i.e., the new objects added into the result and the objects removed from it) can dramatically reduce the transmission overhead. In summary, location-based queries will play a central role in numerous mobile computing applications. We expect that research interest in such queries will grow as the number of mobile devices and related services continue to increase.

ACKNOWLEDGEMENTS

This work was supported by grants HKUST 6081/01E, HKUST 6079/01E, HKUST 6197/02E and HKUST 6255/02E from Hong Kong RGC.

REFERENCES

[A91] Aurenhammer, F. Voronoi Diagrams: A Survey of a Fundamental Geometric Data Structure. *ACM Computing Surveys*, Vol 23(3), 345-405, 1991.

[APR99] Acharya, S., Poosala, V., Ramaswamy, S. Selectivity Estimation in Spatial Databases. *SIGMOD*, 1999

[B00] Bohm, C. A Cost Model for Query Processing in High Dimensional Data Spaces. *ACM TODS*, Vol. 25(2), pp. 129-178, 2000.

[BKSS90] Beckmann, N., Kriegel, H. P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[BBKK97] Berchtold, S., Bohm, C., Keim, D.A., Kriegel, H. A Cost Model for Nearest Neighbor Search in High-Dimensional Data Space. *PODS*, 1997.

[BBK+01] Berchtold, S., Bohm, C., Keim, D., Krebs, F., Kriegel, H.P. On Optimizing Nearest Neighbor Queries in High-Dimensional Data Spaces. *ICDT*, 2001.

[BJKS02] Benetis, R., Jensen, C., Karciauskas, G., Saltenis, S. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. *IDEAS*, 2002.

[BKSS90] Beckmann, N., Kriegel, H.P., Schneider, R., Seeger, B. The R*-tree: An Efficient and Robust Access Method for Points and Rectangles. *SIGMOD*, 1990.

[BKOS97] de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. Computational Geometry. pp. 145-161. *Springer*, 1997.

[GG98] Gaede, V., Günther, O. Multidimensional Access Methods. *ACM Computing Surveys*, 30(2), 1998.

[HS99] Samet, H., Hjaltason, G. Distance Browsing in Spatial Databases. *ACM TODS*, 1999.

[OBSC00] Okabe, A., Boots, B., Sugihara, K., Chiu, S. Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. pp. 291-410. *John Wiley*, 2000.

[RKV95] Roussopoulos, N., Kelly, S., Vincent, F. Nearest Neighbor Queries. *SIGMOD*, 1995.

[SR01] Song, Z., Roussopoulos, N. K-Nearest Neighbor Search for Moving Query Point. *SSTD*, 2001.

[TP02] Tao, Y., Papadias, D. Time Parameterized Queries in Spatio-Temporal Databases. *SIGMOD*, 2002.

[TPS02] Tao, Y., Papadias, D., Shen, Q. Continuous Nearest Neighbor Search. *VLDB*, 2002.

[TSS00] Theodoridis, Y., Stefanakis, E., Sellis, T. Efficient Cost Models for Spatial Queries Using R-trees. *TKDE*, 12(1):19-32, 2000.

[Web] dias.cti.gr/~ytheod/research/datasets/ spatial.html

[WSB98] Weber, R., Schek, H.J., Blott, S. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. *VLDB*, 1998.

[ZL01] Zheng, B., Lee, D. Semantic Caching in Location-Dependent Query Processing. *SSTD*, 2001.