The Optimal-Location Query

Yang Du Donghui Zhang* Tian Xia

College of Computer & Information Science
Northeastern University, Boston, MA 02115
{duy,donghui,tianxia}@ccs.neu.edu

Abstract. We propose and solve the optimal-location query in spatial databases. Given a set S of sites, a set O of weighted objects, and a spatial region Q, the *optimal-location query* returns a location in Q with maximum influence. Here the *influence* of a location l is the total weight of its RNNs, i.e. the total weight of objects in O that are closer to l than to any site in S. This new query has practical applications, but is very challenging to solve. Existing work on computing RNNs assumes a single query location, and thus cannot be used to compute optimal locations. The reason is that there are infinite candidate locations in Q. If we check a finite set of candidate locations, the result can be inaccurate, i.e. the revealed location may not have maximum influence. This paper proposes three methods that *accurately* compute optimal locations. The first method uses a standard R*-tree. To compute an optimal location, the method retrieves certain objects from the R*-tree and sends them as a stream to a plane-sweep algorithm, which uses a new data structure called the aSB-tree to ensure query efficiency. The second method is based on a new index structure called the *OL-tree*, which novelly extends the k-d-B-tree to store segmented rectangular records. The OL-tree is only of theoretical usage for it is not space efficient. The most practical approach is based on a new index structure called the *Virtual OL-tree*. These methods are theoretically and experimentally evaluated.

1 Introduction

Spatial databases play more and more important roles in applications such as corporation decision-support systems. For instance, an interesting query that the McDonald's Corporation may ask again and again is: "what is the optimal location in a given region to put a new McDonald's store?" Here an optimal location can be defined as a location which geographically benefits the most number of customers. This example motivates the optimal-location query. In general, let S be the set of sites (e.g. existing McDonald's stores) and let O be the set of weighted objects (e.g. residential buildings, where the weight for a building is the number of residents in it). Given a spatial region Q, the *optimal-location query* computes a location l in Q which maximizes the total weight of objects that are closer to l than to any site.

* Supported by NSF CAREER Award IIS-0347600.

We focus our discussions on the L_1 distance (also known as Manhattan distance) for it more accurately models the driving distance in a city road network [SKC93]. Given two locations (x_1, y_1) and (x_2, y_2), their L_1 distance is $|x_1 - x_2| + |y_1 - y_2|$. If a road network consists of a set of north-south roads and a set of east-west roads (e.g. in Manhattan), the L_1 distance is the shortest driving distance. When we say "the closest site of o", we mean the site whose L_1 distance to o is the smallest.

A closely related problem is the *bichromatic Reverse Nearest Neighbor (RNN) query* [KM00,YL01,SRAE01]. There, a query location is given, and the RNN query computes the set of objects in O that are closer to l than to any site in S. There are three differences between the bichromatic RNN query and our newly proposed optimal-location query. A small difference is that the RNN query considers L_2 distance (also known as Euclidean distance). The most important difference is that the optimal-location query involves a query region Q, which consists of infinite number of candidate locations. One can approximate Q as a grid, and limit the candidates to the finite set of grid intersections. But this approach cannot accurately compute optimal locations, for the optimal location may be off the grid. The third difference is that the optimal-location query is interested in the influence of a candidate location, or the total weight of objects in the RNN set, instead of the RNN objects themselves.

This paper proposes three methods that accurately compute optimal locations. The first solution (Section 4) assumes we have an R*-tree indexing the set O of objects. Similar to how the Rdnn-tree [YL01] extends the R-tree, we assume the R-tree stores some extra information. Every object stores the L_1 distance to its closest site in S, and every index entry stores the maximum L_1 distance of objects in the sub-tree. In particular, we propose a concept called the *nn_buffer*, for an object o. It is a spatial contour such that a location l is inside *o.nn_buffer*, if and only if o is closer to l than to any site. As we will see later, each such contour, based on L_1 distance, has the shape of a diamond which has four right angles. If we rotate the coordinate by 45° counter-clockwise, every *nn_buffer* is an axis-parallel square in the rotated coordinate. The solution follows two steps. The first step is to retrieve from the R*-tree those objects which may affect the influence of some locations in Q. The objects are identified in certain order which enables a plane-sweep algorithm (as the second step) to go through the stream of objects once and identify an optimal location. The only objects that may affect the influence of locations in the query region Q are the ones whose *nn_buffers* intersect with Q. Our approach retrieves such objects in increasing order of *nn_buffer.x_low* in the rotated coordinate, even though the R*-tree was built in the original coordinate. This enables the run-time plane sweep. A naive plane-sweep solution has $O(n^2)$ cost, where n is the number of objects in the stream. We propose the *aggregation SB-tree (aSB-tree)*, extended from the SB-tree [YW01], to reduce the query cost to $O(n \log n)$.

Our second solution (Section 5.1) to the optimal-location query is based on a new specialized aggregation index called the *OL-tree*. It is disk-based, balanced and dynamically updateable. The index is built in the rotated coordinate. It

is a novel extension to the k-d-B-tree. While the k-d-B-tree maintains point objects, the OL-tree keeps axis-parallel squares. In the OL-tree, each index entry maintains a value called *fullcover* to count how many squares fully contain the range of it. Besides, Each index entry stores *maxoverlap*: the maximum local influence in the sub-tree, and *maxrange*, a rectangular region where any location in it has maximum local influence.

The two solutions have interesting tradeoffs. The R*-tree based solution has efficient (linear) space cost. However, as objects are not pre-aggregated, a query needs to examine all objects whose *nn_buffers* intersect with Q. If Q has large size, the query performance is poor. On the other hand, the OL-tree is a specialized aggregation index, whose space overhead is higher since an object may have many copies. But it may have faster query support. For instance, if Q intersects with the *maxrange* stored at the root, the algorithm instantly returns.

The third solution (Section 5) combines the benefits of the previous two approaches. As in solution 1, we use an R*-tree to store the objects. But to guide the search, we use a small, in-memory OL-tree-like structure. This index is named the *Virtual OL-tree (VOL-tree)*. It looks like an OL-tree, but it does not store any *nn_buffer*. A leaf entry has the same meaning of an index entry. It corresponds to a spatial range, and it logically references a node that stores (pieces of) *nn_buffers* in that range. These *nn_buffers* can be retrieved from the R*-tree dynamically. Because the VOL-tree is small, each leaf entry may correspond to many *nn_buffers* (as a comparison, each OL-tree leaf node has at most B *nn_buffers*). Thus it is more costly to maintain *maxoverlap* in a VOL-tree, which may require to retrieve all *nn_buffers* corresponding to some leaf entry. For this reason, instead of maintaining *maxoverlap*, the VOL-tree maintains *lower_max* and *upper_max*, which are a lower bound and an upper bound of *maxoverlap*. In particular, *maxrange* is associated with *lower_max*.

This paper contributes in several ways to the understanding of the emerging class of located-based applications.

1. We propose the optimal-location query. It has practical applications such as corporate decision-support systems.
2. We present an R*-tree-based solution. In particular, our solution retrieves objects of interest in some given order and then uses a plane-sweep algorithm to identify an optimal location. The plane-sweep algorithm uses a new data structure called the aSB-tree to improve the query performance from $O(n^2)$ to $O(n \log n)$.
3. We introduce a theoretical solution based on a new index structure called the OL-tree. It is a specialized index with higher space cost but possibly more efficient query performance.
4. We provide a practical solution based on the Virtual OL-tree, which is both space efficient and query efficient.
5. We show experimental results on real datasets which reveal the tradeoffs of the proposed methods.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 provides problem transformation and introduces the rotated space.

Section 4 presents the R*-tree-based solution. Section 5 presents the OL-tree-based and the Virtual-OL-tree-based solutions. Section 7 shows performance results. And Section 8 concludes the paper.

2 Related Work

The nearest neighbor (NN) query, since its introduction in [RKV95], has received vast attention in spatial database research community. One recent variation introduced by [KM00] was the RNN query. That is, given a query location l, find the objects in a given set O that consider l as their nearest neighbor. Note that existing work assumes the Euclidean distance while we focus on the Manhattan distance.

There are two variations of the RNN query: the monochromatic case and the bichromatic case. In the monochromatic case [SAE00,TPL04], the distance between an object $o \in O$ and the query location l is compared with the distances between o and other objects in O. In the bichromatic case [SRAE01], there is another dataset: a set S of sites. And the distance between o and l is compared with the distances between o and sites in S. Many real-life applications correspond to the bichromatic case. For instance, given a new location, compute the set of residence buildings that are closer to this location than to any existing McDonald's store. In [Smi97], it was proved that for the monochromatic case, the number of RNNs is bounded. For instance, there are at most 6 RNNs in the 2D case and at most 12 RNNs in the 3D case. But for the bichromatic case, the number of RNNs is unbounded even for the 2D case.

One solution to the bichromatic RNN query is based on precomputation [KM00,YL01]. (It also works for the monochromatic case.) The idea of [KM00] is to build an R-tree that stores circles instead of points. Every circle is centered at some object o, with radius being the distance from o to its nearest site. Precomputation is required to get these distances. Given a query location l, its RNNs are retrieved by locating the circles that enclose l.

Yang and Lin [YL01] proposed the Rdnn-tree which combines the R-tree of circles with the R-tree of objects. It is an R-tree of objects, where every object stores the distance to its closest site, while every index entry stores the maximum distance of all objects in the sub-tree. The structure logically maintains the R-tree of circles. It remains to determine, given a location l and an index entry e, whether the sub-tree referenced by e may contain some object whose "circle" (not stored) encloses l. The solution is to expand the index entry's MBR outward by the associated maximum distance. If the expanded region does not enclose l, there is no need to check the sub-tree.

The R-tree that we use, in the first solution to the optimal-location query, is the Rdnn-tree [YL01] which stores L_1 distance instead of L_2 distance. Our concept of *nn_buffer* corresponds to their concept of *circle* for each object. However, in this paper both the addressed problem and the R-tree-based solution are different from [YL01]. Our problem takes as input a spatial region and aims at identifying an optimal location (with maximum influence), while [YL01] takes

as input a single location and finds its RNNs. As for the solution, while [YL01] finds the objects whose circles enclose the given location and terminates, we find the objects whose *nn_buffers* intersects with the given region as the first stage of a pipeline process. The second pipeline stage takes this stream of objects and identifies an optimal location via plane sweep.

Another bichromatic RNN query solution was proposed by [SRAE01]. The idea is to dynamically construct the *influence region* of the query location l. Here, the influence region is defined as a polygon in space which encloses and only encloses all possible RNNs of l. This is equivalent to the *Voronoi cell* enclosing l [BKOS97]. Conceptually, if we draw a bisector line between l and a site s, any object located on the l side of the bisector will have smaller Euclidean distance to l than to s. The l side of the bisector is a half plane. If we compare l against all sites and take the intersection of these l-side half planes, we get the Voronoi cell containing l.

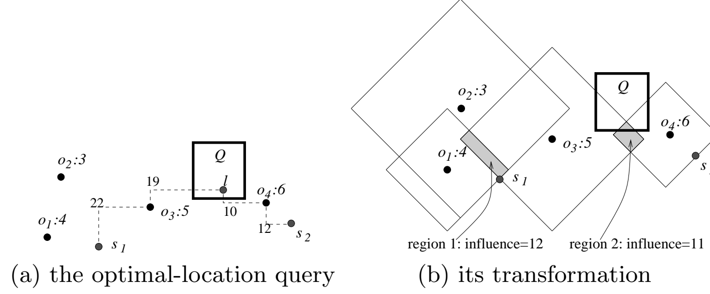
Of course, to compare with all sites is expensive. [SRAE01] provides a clever way to compute the rectangle that is guaranteed to contain all sites needed for computing the exact Voronoi cell of l. Then the Voronoi cell of l can be computed by only examining these sites with in the rectangle. So to find the RNNs of l, a range query using the Voronoi cell is performed on the R-tree of objects.

If we had limited candidate locations, the approach could be extended to solve the optimal location problem, with two modifications. First, we now need to construct a Voronoi cell with regards to the L_1 distance [LW80]. Second, for each location, we need to know its influence rather than the actual RNN objects. So we can index the set of objects using the aggregation R-tree (aR-tree) [PKZT01] where each index entry stores the total weight of objects in the sub-tree. If the MBR of an index entry is contained in a Voronoi cell, the stored total weight contributes to the computation of influence, without browsing the sub-tree.

Unfortunately, in the optimal-location query, there are infinite number of candidate locations. So the approach of [SRAE01] (with the above modifications) does not work. Before presenting our solutions, let's first study a problem reduction.

3 Problem Transformation

An illustration of the optimal-location query, defined in Section 1, appears in Figure 1(a). There are four objects and two sites. In particular, the object o_3 with weight 5 has s_1 as the closest site, where $d(o_3, s_1)$=22. And the object o_4 with weight 6 has s_2 as the closest site, where $d(o_4, s_2)$=12. The influence of a location is the total weight of objects that are closer to this location than to their closest sites. For instance, the influence of l is the total weight of o_3 and o_4, which is 5+6=11. Given a query region Q, the optimal-location query finds an location inside Q with maximum influence. In this example, l is an optimal location. There may be more than one optimal location. The query asks for one of them.



(a) the optimal-location query (b) its transformation

Fig. 1. In (a), l is an optimal location, with influence 11. The transformation in (b) shows that any location in the intersection between Q and region 2 is an optimal location.

To efficiently compute an optimal location, we first define the concept of *nn_buffer*, and then transform the optimal-location query into the problem of finding a location with maximum overlap among objects' *nn_buffers*.

definition 1 *Let s be the closest site of an object o. The nn_buffer of o is a contour such that $\forall$ l on the contour, $d(l, o) = d(o, s)$. Here $d()$ is L_1 distance. Given a MBR of objects, let t be the maximum distance between any such object to its closest site, the nn_buffer of the MBR is a outside contour such that $\forall$ l on the contour, the minimum distance between l and the MBR is t.*

(a) (b)

Fig. 2. The *nn_buffer* of an object and the *nn_buffer* of an MBR.

As shown in Figure 2(a), the *nn_buffer* of an object o is a diamond with four right angles. It is easy to check that the L_1 distance between o and any location on the boundary of the diamond is fixed.

The weight of object o contributes to the influence of a location l, if and only if l is inside the *nn_buffer* of o. So as shown in Figure 1(b), an optimal location is a location l inside Q which maximizes the total weight of objects whose *nn_buffers* contain l. The concept of *nn_buffer* can also be defined for an *minimum bounding rectangle (MBR)* of a set of objects. The *nn_buffer* of an MBR is the tightest contour which is guaranteed to contain the *nn_buffers* of all objects in the MBR, without knowing the locations of the objects. The *nn_buffer* of an MBR is a polygon with eight edges, as illustrated in Figure 2(b).

Consider the coordinate which has the same origin as in the original coordinate, but whose X and Y axes are rotated 45° counter-clockwise. We call it the $\circlearrowleft 45^{\circ}$ (reads rotate-45-degree) coordinate (Figure 3). In this paper, the R*-tree indexes in the original coordinate (to satisfy the possible need for other applications), while the aSB-tree, the OL-tree and the VOL-tree indexes in the $\circlearrowleft 45^{\circ}$ coordinate.

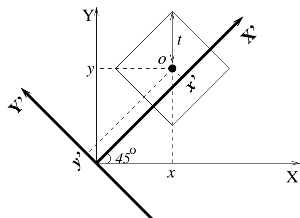


Fig. 3. Illustration of the rotated coordinate.

Our analysis shows that an object o located at (x, y) in the original coordinate is mapped to $(\frac{x+y}{\sqrt{2}}, \frac{-x+y}{\sqrt{2}})$ in the $\circlearrowleft 45^{\circ}$ coordinate. Furthermore, let t be the L_1 distance from o to its closest site. The *nn_buffer* of o is an axis-parallel square in the $\circlearrowleft 45^{\circ}$ coordinate, whose lower-left corner and upper-right corner are: $(\frac{x+y-t}{\sqrt{2}}, \frac{-x+y-t}{\sqrt{2}})$ and $(\frac{x+y+t}{\sqrt{2}}, \frac{-x+y+t}{\sqrt{2}})$.

4 The R*-tree-based Solution

Our first solution to the optimal-location query assumes the objects are indexed by an R*-tree. Similar to how the Rdnn-tree [YL01] extends the R*-tree, we assume every object stores the L_1 distance to its closest site, and every index entry stores the maximum L_1 distance of objects in the sub-tree.

The R*-tree indexes objects in the original coordinate (not the $\circlearrowleft 45^{\circ}$ coordinate), since there may be other applications that need to access the data in the original coordinate. However the plane-sweep algorithm works in the $\circlearrowleft 45^{\circ}$ coordinate. In order to do the plain sweep, we have to retrieve the objects in increasing order of their *nn_buffer*'s *x_low* in the $\circlearrowleft 45^{\circ}$ coordinate. Section 4.1 shows how to retrieve objects, Section 4.2 describe a naive plane sweep algorithm with $O(n^2)$ cost, Section 4.3 propose the aSB-tree structure which can reduce the worst-case query cost to $O(n \log n)$, and Section 4.4 extends the algorithms to incorporate a rotated query region.

4.1 Retrieving Objects from The R*-tree

To retrieve the objects whose *nn_buffers* intersects with Q, we can browse the R*-tree in a top-down fashion, similar to the range query. The difference is that to determine whether to expand an sub-tree, instead of checking whether its MBR intersects with Q, we check whether the MBR's *nn_buffer* intersects with Q.

The remaining issue of object retrieval is how to return objects in increasing order of their *nn_buffer*'s *x_low* in the $\circlearrowleft 45^{\circ}$ coordinate. This is achieved by using a best-first search. That is, we keep a heap of the R*-tree's index entries as

well as objects. The entries are ordered in increasing *nn_buffer.x_low* in the 45° coordinate. Initially, the heap contains the index entry referencing the whole tree. In each iteration, the entry e with minimum *nn_buffer.x_low* is extracted. If e is a object, output it (to be sent, as the next element of an input stream, to the plane-sweep algorithm discussed in the next section). Otherwise, e is an index entry. We examine every entry se in the node referenced by e, and push se into the heap if its *nn_buffer* intersects with Q.

4.2 The Naive Plane Sweep

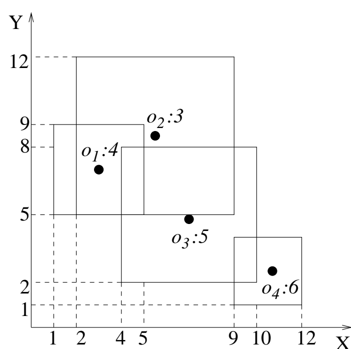


Fig. 4. *nn_buffers* in the rotated coordinate.

In the rotated coordinate, the *nn_buffers* are axis-parallel squares. To find the optimal location, the basic idea is to perform a plane sweep in increasing order of X. For each particular X, the Y axis is partitioned into a set of intervals, each associated with an influence value. For instance, at X=4, the Y axis is partitioned into six intervals: (-∞,2):0, (2,5):5, (5,8):12, (8,9):7, (9,12):3, and (12,∞):0. Whenever a change happens, update the set of intervals. During the process, always maintain a location with maximum influence. In fact we can maintain a rectangular region with maximum influence, instead of a single location.

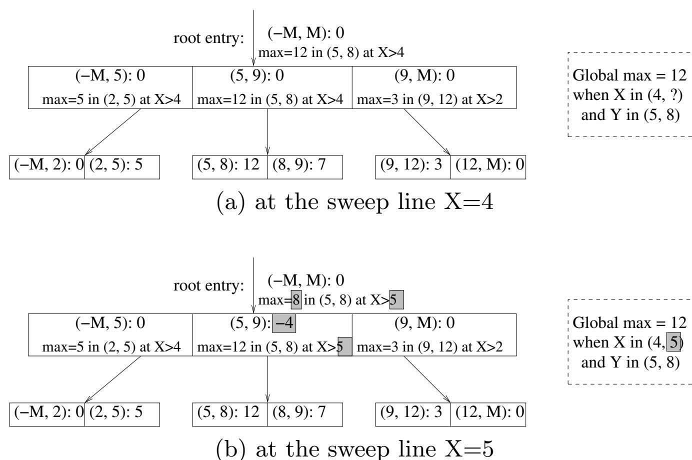
At the end, any location in the maintained region is an optimal location. As an example, in Figure 4, any location in the X range of (4,5) and Y range of (5,8) is an optimal location, with influence 12.

4.3 The aSB-tree

The naive plane sweep has $O(n^2)$ worst-case performance. The reason is that there are $O(n)$ events to handle, and each event needs to scan through $O(n)$ intervals that partition the Y axis. We hereby propose a data structure called the *Aggregation SB-tree (aSB-tree)*, derived from the SB-tree [YW01]. The new structure enables any event to be processed in $O(\log n)$ time, and therefore reduces the overall cost to $O(n \log n)$.

The idea is to organize the intervals (that partition the Y axis) into a balanced B-tree-like structure. To insert a new Y range which may affect many intervals in the naive approach, with the aSB-tree we only need to update two paths from root to leaf. The key idea that enables this is: if the Y range to be inserted fully contains the interval of an index entry, we do not insert into the sub-tree. Instead, we update a value *fullcover* maintained along with the index

entry. The aSB-tree extends the SB-tree by storing the max influence and the corresponding spatial region in the sub-tree. Figure 5 shows a two-level aSB-tree, which corresponds to Figure 4 right after processing the event at $X = 4$ and the event at $X = 5$. Let's examine it in more details.



(a) at the sweep line X=4

(b) at the sweep line X=5

Fig. 5. An example of aSB-tree.

Properties inherited from the SB-tree:

- The aSB-tree is a balanced tree structure. The maximum number of entries in a (leaf or index) node is fixed. Except the root, every node must be at least half full.
- Every entry corresponds to an interval (a Y range). For any index entry e, all intervals of entries in $Node(e)$ form an exact partition of $e.interval$. E.g. in Figure 5(a), the root entry has an interval as the whole Y space.
- Every leaf entry has a value *influence*. In Figure 5(a), the leaf entry (5,8):12 means that right after the current $X = 4$, any location with $Y \in (5, 8)$ has influence 12.
- Every index entry has a value *fullcover*, which corresponds to the total weight of inserted Y ranges which fully cover the entry's interval. E.g. the second index entry in the root of Figure 5(a), which has (5,9):0, means its fullcover is 0, while the interval is (5,9).
- To insert a range I with weight w, we update (at most) two paths from the root to the leaf level. These are the nodes whose referencing entry's interval partially intersects with I. E.g. Figure 5(b) shows the result after processing the event at $X = 5$, i.e. inserting I=(5,9) and w=-4. In particular, the insertion stopped at the root node, since no entry in the root node has an interval partially intersecting with I. For any entry (e.g. the second index entry in root) whose interval is contained in I, w is added to its $fullcover$. An overflow/underflow, if happens, is treated like in the B-tree.

Properties extended from the SB-tree:

- There is a gap between what the SB-tree provides and what we need. The ultimate goal we need is: after all the *nn_buffers* are seen, report an optimal location with its influence. To do so, separate from the aSB-tree we maintain a globally maximum influence and its spatial range.
- This global max is maintained after processing each insertion. Here every index entry in the aSB-tree stores the local maximum influence of some location in the sub-tree. It is local because the actual influence should consists of the $fullcover$ of index entries for all ancestor nodes. For instance, in the second index entry in Figure 5(b), the local maximum influence is 12. But the actual maximum influence is 12+(-4)=8. Since the old global max is no smaller than the new one, it is not changed.
- Along with each local maximum influence stored at an index entry, or with the global max, we also store the corresponding spatial region. That is, any location in this region has this maximum influence. For a local max, its corresponding region only needs to store the left X border for the right border is not known yet. With a new insertion, it is possible to close the right border of the previous max region and start a new one. E.g. in Figure 5(b), two index entries' local max region change their left border from $X = 4$ to $X = 5$. The right border of the global max region may need to be closed correspondingly. All the additional information can be maintained along with the insertion process, by following the insertion paths backwards. So the update cost remains $O(\log n)$ as in the SB-tree. Therefore, the plane-sweep algorithm integrated with the aSB-tree has $O(n \log n)$ query cost.

4.4 Extension to Involve A Query Region

In the original coordinate, the query region Q is an axis-parallel rectangle. Thus, in the 45° coordinate, the query region Q becomes a rectangle rotated 45° clockwise (as shown in Figure 6). To perform the query correctly, our aSB-tree based plane-sweep algorithm needs to be extended as follows:

- The Y space of the aSB-tree is not the whole space (-M,M), but the Y projection of Q. This is because we only care about the locations in Q. In Figure 6, the Y space of the aSB-tree should be (y_l, y_h).
- For each *nn_buffer*, we calculate the smallest X (called *start*) when it 'enters' Q and the largest X (called *end*) when it 'leaves' Q. The insertion/removal events occur at these calculated X values, instead of the *x_low* and *x_high* of the *nn_buffers*.

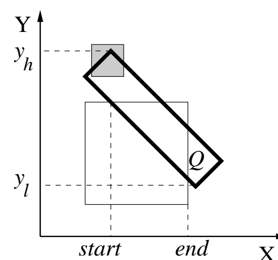


Fig. 6. Illustration of a rotated query region.

- Finally, it is no longer true that whenever the aSB-tree is updated due to an event, the current maximum influence is known only by checking the root entry. The reason is that the maintained maximum influence may be in a region outside Q. To address this issue, we perform a *range-max query* on the aSB-tree. That is to find the maximum influence within the actual Y range. The range-max query can be performed in $O(\log n)$ time, since it only needs to examine two paths of the aSB-tree. The reason is that, similar to the insertion algorithm, a sub-tree whose interval is contained in the query interval does not need to be expanded.

A side note is: even though we use the aSB-tree as an in-memory structure to improve the plain sweep, if needed the structure can be implemented as a disk-based index like the SB-tree.

5 The Virtual OL-tree

The R*-tree based solution examines all objects whose *nn_buffers* intersect with the query region Q, and thus is not efficient when a large Q results in the examination of many objects. This section first proposes an theoretical solution to the optimal-location query based on a new index structure called the *Optimal-Location Tree (OL-tree)*. Then we extend it to a more practical and efficient solution based on the *Virtual Optimal-Location Tree (VOL-tree)*.

5.1 The OL-tree-based Solution

The OL-tree is a k-d-B-tree-like structure which is balanced, disk-based and dynamically-updateable. Roughly speaking, it stores the *nn_buffers* in the 45° coordinate. Like the k-d-B-tree, the OL-tree is a space-partitioning method (versus a data-partitioning method like the R*-tree). Unlike the k-d-B-tree, the OL-tree stores rectangular records in its leaf nodes. If a square partially intersects with the ranges of multiple index entries, it is split and multiple copies are inserted. However, if the square fully contains the range of some index entry, we only update a value called *fullcover* stored along with the index entry, without further inserting into the sub-tree. Each index entry stores *maxoverlap*: the maximum local influence in the sub-tree. That is, the maximum influence in the sub-tree, subtracted by the $fullcover$ values of all ancestor index entries. A rectangular region *maxrange* is also stored, where any location in it has maximum local influence. Due to the space limitation, we skip the details of the update and query processing.

The OL-tree may cause cascading split of child nodes if splitting an index node. One may wonder how bad the space complexity can be. We argue that the space complexity of the OL-tree (with an additional requirement) is $O(n^2/B)$ for the following reasons. First, the total number of leaf entries is $O(n^2)$. With n axis-parallel squares, there are $O(n)$ different X positions and $O(n)$ different Y positions, which form $O(n^2)$ cells. In the worse case each cell is stored in the tree

separately. Thus there are at most $O(n^2)$ leaf entries. Second, the total number of nodes in $O(n^2/B)$. The linear storage of the k-d-B-tree can be guaranteed by re-organization of sub-trees which contain too few leaf entries. Similarly, the OL-tree with $O(n^2)$ leaf entries needs $O(n^2/B)$ nodes.

This bound reveals that the OL-tree is not a practical spatial index structure. In the next subsection we introduce a practical structure, named the VOL-tree, to solve the optimal-location query.

5.2 The VOL-tree Structure

The OL-tree has more than linear space because if an *nn_buffer* is split into multiple pieces, each of them is physically stored in some leaf node(s). What if we do not physically store any leaf node of the OL-tree? We can use an R*-tree to store the original objects, and whenever the content of a leaf node is needed, we perform a range query on the R*-tree. This is the key idea to the *Virtual OL-tree (VOL-tree)*.

It is challenging to implement this idea. As we already spend the space to store the R*-tree, it is ideal to have a small VOL-tree that fits in memory. On the other hand, as there are $O(n^2/B)$ leaf nodes in an OL-tree, there are $O(n^2/B^2)$ index nodes, which would be the size of the VOL-tree if we treat it as an OL-tree without leaf nodes. There is a big gap. Thus we claim that the VOL-tree is NOT merely an OL-tree without the leaf level. It has to be much much smaller, possibly only consisting of one index level besides the root. A consequence is that each leaf entry of the VOL-tree corresponds to a virtual node (content stored in the R*-tree) with much more than B *nn_buffers*. So a crucial issue jumps out: it is expensive to maintain *maxrange* and *maxoverlap* because an update requires us to perform plane sweeps on the virtual nodes.

To address this issue, we propose another change from the OL-tree: along with each index entry, instead of keeping the accurate *maxoverlap*, keep two values *lowermax* and *uppermax*, which are a lower bound and an upper bound of *maxoverlap*.

In more detail, the entries in the tree are as follows:

- An index entry e has the following format: (*range*, *nodeID*, *fullcover*, *lowermax*, *maxrange*, *uppermax*). Here *range* is the spatial range of the corresponding sub-tree, and *nodeID* points to the referenced node. The value *fullcover* is the total weight of *nn_buffers* whose insertion stopped at e (such a *nn_buffer* contains $e.range$, but not the *range* of e's parent).
- The values *lowermax* and *uppermax* are some lower and upper bounds of the maximum local influence in $e.range$. And *maxrange* is a rectangle fully contained in $e.range$ where every location in *maxrange* has local influence $= lowermax$.
- A leaf entry and an index entry have the same content, with a minor difference that a leaf entry's *nodeID* is empty.

Algorithm *VOLTreeQuery*
Input: Query region Q, VOL-tree *root*.
Return: An optimal location in Q.

1. **if** $root.maxrange \cap Q \neq \emptyset$ and ($root.lowermax = root.uppermax$ or $Q \subseteq root.maxrange$), **return** any location in $root.maxrange \cap Q$.
2. $heap$.Insert($root$, 0, $root.uppermax$)
3. Set *opt_loc* as an arbitrary location in Q, and $opt_inf = 0$,
4. **while** *heap* is not empty
   (a) (e, min, $upper$) = $heap$.ExtractMaxUpper().
   (b) **if** $upper \leq opt_inf$, **return** *opt_loc*.
   (c) **if** e references an index node
      for every entry se in $Node(e.nodeID)$ s.t. $se.range \cap Q \neq \emptyset$
         A. Set $m = min + se.fullcover$, and $u = min + se.fullcover + se.uppermax$.
         B. **if** $u \leq opt_inf$, **goto** next entry.
         C. **if** $opt_inf < m$, set $opt_inf = m$ and *opt_loc* be any location in $se.range \cap Q$.
         D. **if** $se.maxrange \cap Q \neq \emptyset$,
            (i) $l = min + se.fullcover + se.lowermax$
            (ii) **if** $opt_inf < l$, set $opt_inf = l$ and *opt_loc* be any location in $se.maxrange \cap Q$.
            (iii) **if** $u \neq l$ and $(Q \cap se.range) \not\subseteq se.maxrange$, $heap$.Insert(se, m, u)
         E. **else**
            $heap$.Insert(se, m, u)
         F. **end if**
      end for
   (d) **else**
      A. Using $e.range \cap Q$ as a new query region, retrieve *nn_buffers* from the R*-tree of objects. Use plane sweep to find an optimal location (inf, loc) within the new query region.
      B. **if** $opt_inf < min + inf$, set $opt_loc = loc$, and $opt_inf = min + inf$.
   (e) **end if**
5. **end while**
6. **return** *opt_loc*.

Fig. 7. Finding an optimal location using the VOL-tree.

5.3 The VOL-tree Query Algorithm

Figure 7 shows the optimal-location-query algorithm in the VOL-tree. We start from the root node. In the VOL-tree, even if $root.maxrange$ intersects with Q, it is possible that some location in $Q - root.maxrange$ has an influence larger than $root.lowermax$ (when $root.lowermax < root.uppermax$). So as Step 1 shows, we can safely return a location in $root.maxrange \cap Q$ only if $root.lowermax = root.uppermax$ or Q is completely inside $root.maxrange$.

Step 2 inserts the root entry into a heap. Every entry in the heap has, besides an index entry, two values *min* and *upper*. These are the actual (not local) lower bound and upper bound of influence for locations in the sub-tree. Meanwhile, we maintain the currently seen optimal location *opt_loc* along with its influence opt_inf, initialized to be an arbitrary location with influence 0 (Step 3).

While the heap is not empty, we process each element at a time. In each iteration, the heap entry with maximum *upper* is extracted. As Step 4(b) of the algorithm shows, if this extracted *upper* is no larger than opt_inf, we can

determine that *opt_loc* is an optimal location and thus the algorithm returns. The crucial steps are Step 4(c) which expands an index node and Step 4(d) which expands a leaf node.

To expand an index node, we examine every child entry *se* whose *range* intersects with Q, and try to push *se.nodeID* into the heap. Here the new lower bound is $m = min + se.fullcover$, and the new upper bound is $u = min + se.fullcover + se.lowermax$. There are two pruning opportunities. First, if the new upper bound u is no larger than opt_inf, there is no need to expand the sub-tree (Step 4(c)B). Second, if *se.maxrange* intersects with Q, we already know the influence of the locations within the intersection, and thus we may have the chance to update the maintained optimal location before expanding the sub-tree (Step 4(c)D). It likely causes other entries to be pruned earlier.

To expand a leaf node (Step 4(d)), we go to the R*-tree to retrieve the *nn_buffers* that intersect with $e.range \cap Q$ and then perform a plane sweep technique of Section 4 to compute a location inside Q with maximum global influence inf. If this influence is bigger than opt_inf, we update the maintained opt_inf and *opt_loc*.

5.4 The Update Algorithm

The VOL-tree can be bulk-loaded. Due to space limitations, the algorithm is omitted. We only point out that immediately after bulk-loading, every entry in the VOL-tree has accurate local maximum information, i.e. $lowermax = uppermax$. With dynamic update, this may not be true. Let us examine the update algorithm below.

The insertion algorithm is shown in Figure 8, while a deletion is treated as an insertion with negative weight. To insert into an index node (Step 1), we consider every child entry *se* whose *range* intersects with the parameter R. If *se.range* is contained in R, we simply add w to $se.fullcover$. If *se.range* partially intersects with R, we recursively insert into the sub-tree referenced by *se*. After insertion, we need to re-aggregate the *lowermax*, *uppermax* and *maxrange* if necessary.

When e refers to a virtual leaf node which is not stored, the actual object is maintained in a separate R*-tree. So we only need to modify *e.lowermax*, *e.uppermax* and *e.maxrange*. The update of *e.uppermax* is simple. As Step 2(a) shows, for a positive weight, *e.uppermax* is increased by w. For a negative weight, *e.uppermax* remains unchanged. We may modify *e.lowermax* and/or *e.maxrange* only if *e.maxrange* intersects with R. For a positive weight (Step 2(c)), the intersection part of *e.maxrange* and R is the new *e.maxrange*, with weight increased by w. For a negative weight (Step 2(d)), there are two cases. If *e.maxrange* is fully covered by R, we decrease *e.lowermax*. Otherwise, we shrink *e.maxrange* to $e.maxrange - R$ but keep *e.lowermax* unchanged.

6 Performance

In this section, we report experimental results on the R*-tree approach and the VOL-tree approach. In our experiments we used real datasets: the Digital Chart

Algorithm *VOLTreeInsert*

Input: Range R, Weight w, VOL-tree index entry e.

Pre-condition: R intersects with, but does not fully contain, $e.range$.

Action: Insert range R with weight w to the sub-tree referenced by e.

1. **if** e refers to a node in the VOL-tree
   (a) **for** every se in $Node(e)$ s.t. R contains $se.range$, $se.fullcover$ += w.
   (b) **for** every se in $Node(e)$ s.t. R partially intersects with $se.range$, VOLTreeInsert(R, w, se).
   (c) Let se_0 be the entry in $Node(e)$ with maximum $se.fullcover+se.lowermax$.
   (d) Set $e.maxrange = se_0.maxrange$ and $e.lowermax = se_0.fullcover + se_0.lowermax$.
   (e) $e.uppermax$=max$\{se.fullcover+se.uppermax\}$ for all entry se in $Node(e)$.
2. **else** /* e refers to a virtual leaf node */
   (a) **if** $w > 0$, $e.uppermax$ += w.
   (b) **if** $e.maxrange \cap R = \emptyset$, **return**.
   (c) **if** $w > 0$
      i. $e.maxrange = e.maxrange \cap R$
      ii. $e.lowermax+ = w$
   (d) **else**
      i. **if** $e.maxrange \subseteq R$, $e.lowermax+ = w$.
      ii. **else** $e.maxrange = e.maxrange - R$.
   (e) **end if**
3. **end if**

Fig. 8. The Insertion algorithm of the VOL-tree.

of the World from the R-tree Portal [The03]. It contains two type of point data: the populated places and cultural landmarks in North America, a total of 24,493 and 9,203 points respectively. We use the populated places as the objects and cultural landmarks as the sites. From the dataset, we generated an object R*-tree for all populated place, which is augmented by the L_1 distance from each object to its nearest site. We set the page size to 1k and the default buffer size to 256 pages. All the programs were written in Java and run on a Pentium IV Dell PC equipped with 3.2GHz CPU.

From our preliminary experimental results, we found that it does not help to make the VOL-tree disk based. So the VOL-tree is in memory, and only the I/O of R*-tree will be measured. However, it consumes part of the buffer. For example, if the size of the VOL-tree is 50 pages, the buffer available to the R*-tree retrieval in the VOL-tree based method should be $256 - 50 = 206$. For each experiment we start with a clean buffer, run 100 random queries, and measure the total I/Os. Buffer will not be flushed during the execution of 100 queries. Our preliminary experimental results show that, with fixed area of the query range, the shape of the query range has little impact on the query performance of the VOL-tree based methods. Thus we always use square query ranges.

In many applications, the datasets are known in advance. For instance, the set of McDonald's stores and the set of residential buildings can be given in advance when building the index, although changes may happen later on. Therefore, we use the VOL-tree based method with high bulk-loading percentage (80% and

100%). In the experiments, we compare the performance of three methods listed in Table 1.

Name of method	Explanation
R*	R*-tree based method (without using VOL-tree)
VOL80	VOL-tree based method with 80% of the objects being bulk loaded
VOL100	VOL-tree based method with 100% of the objects being bulk loaded

Table 1. There different settings for experiments.

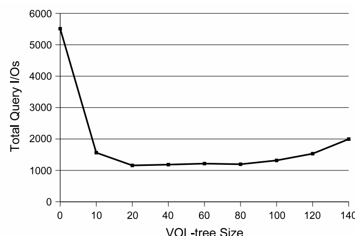


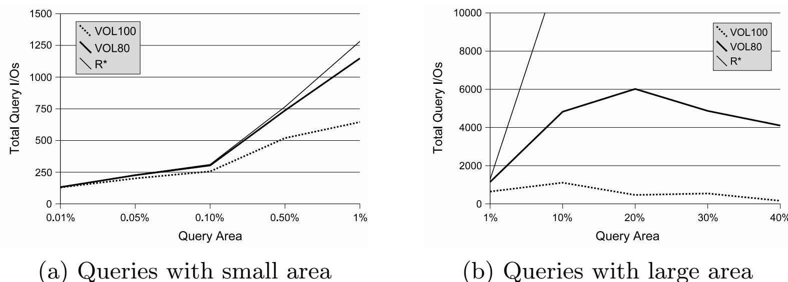
Fig. 9. The I/O performance of the VOL-tree for various size.

To utilize the VOL-tree, the first question to answer is how large the VOL-tree should be. Figure 9 shows the I/O of the R*-tree of various sizes (in the unit of the page size). When the size of the VOL-tree is small (< 20 pages), the I/Os become close to the R*-tree based method (which corresponds to VOL-tree size = 0). When the size of the VOL-tree is large (> 80 pages), the I/Os also increase. That is because the larger VOL-tree does not help much to prune the search space, but it uses a large proportion of the buffer, which results in the worse I/O of the R*-tree.

From the results, we draw the conclusion that a small VOL-tree is sufficient. Thus, in the later experiments, we set the size of VOL-tree to 20 pages.

To study the effect of the size of query range on the I/Os, we change the *area* of the query range. Figure 10 (a) and (b) shows the results when the query range is small and is large respectively. When the query area is smaller than 1% of the whole space, their performances are very close although VOL-tree methods outperform. When the query area is larger than 1%, the R*-tree based method has I/Os of more than 10,000 so we do not even show it in figure. An expected fact is that when the query range becomes very large, the performance of both VOL80 and VOL100 improve. That is because, the large query ranges is more likely to intersect with *maxrange*s stored in the VOL-tree, and thus is more likely to prune some subtrees.

The updates increase the difference between *lowermax* (*uppermax*) and the local maximal influence, thus decreasing the pruning capability. Figure 11 shows how updates affect the I/O performance. We bulk load some objects and insert the others. The X-axis presents the percentage of the number of inserted objects to the number of bulk loaded objects. For example, $X = 50\%$ corresponds to the case when we bulk load 2/3 of the objects and insert the remaining 1/3. With



(a) Queries with small area (b) Queries with large area

Fig. 10. The I/O performance of of the VOL-tree for various query area.

the increase of the percentage, the I/O performance decreases. After about 50% insertions, the performance becomes comparable to the R*-tree based method. There are two reasons for that. First, the VOL-tree uses some buffer of the R*-tree. Second, the VOL-tree may cause multiple scans of same page. We need to point out that even if an application is update intensive, the VOL-tree based method is still a good choice since the tree can be rebuilt in part or in full. And the rebuilding cost is amortized. Furthermore, the rebuilding can be integrated with the query processing.

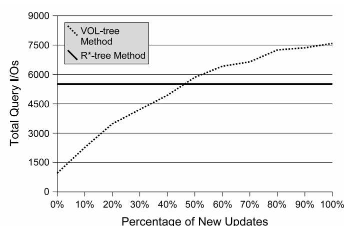


Fig. 11. The Effect of the Updates.

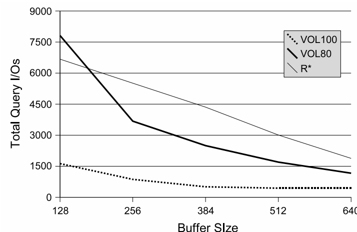


Fig. 12. The Effect of the Buffer Size.

Figure 12 shows how the buffer size affects the I/O performance of the VOL-tree. When buffer size is 128, the R* outperforms VOL80. That is because the VOL-tree has size of 20 and occupies about 20% the buffer. After the buffer size is doubled to 256, the I/O of VOL80 dramatically drops to below R*. With the increase of the buffer size, the performance of all the three methods get improved, while the VOL-tree based method is again better.

7 Conclusions

In this paper we proposed and solved the optimal-location query. The query has real applications, e.g. in corporate decision-support systems. We presented three solutions to accurately answer such a query. In particular, the VOL-tree approach is the most efficient. The approach uses an R*-tree to index the objects,

while a small, in-memory VOL-tree is used to prune the search space. The query performance is much better than the plain R*-tree approach, especially when the query size is large. (Notice that the R*-tree approach is already optimized via a new index called the aSB-tree.) For instance, if the query area is 5% of the space, the VOL-tree approach computes an optimal location 6 times faster than the R*-tree approach. If the query size increases, the improvement increases as well, which can be multiple orders of magnitude better. Also, the size of the VOL-tree is small. In our experiments, while the R*-tree of objects is over 700 disk pages, the VOL-tree is only 20 pages. The VOL-tree has very efficient updates, as the index is small and updating it does not need to touch the R*-tree (except for ordinary object insertion/removal). One set of experiments showed that within 50% new updates, the VOL-tree approach remained to have better query performance. Of course, if there are too many updates, the VOL-tree can be re-built and the cost is amortized across all the new updates. In summary, the VOL-tree approach is the most efficient solution to the optimal-location query.

References

[BKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Sprinter Verlag, 1997.

[KM00] F. Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *SIGMOD*, pages 201–212, 2000.

[LW80] D. T. Lee and C. K. Wong. Voronoi Diagram in L_1 (L_∞) Metrics with 2-Dimensional Storage Applications. *SIAM Journal on Computing*, 9:200–211, 1980.

[PKZT01] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP Operations in Spatial Data Warehouses. In *SSTD*, pages 443–459, 2001.

[RKV95] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest Neighbor Queries. In *SIGMOD*, pages 71–79, 1995.

[SAE00] I. Stanoi, D. Agrawal, and A. El Abbadi. Reverse Nearest Neighbor Queries for Dynamic Databases. In *ACM/SIGMOD Int. Workshop on Research Issues on Data Mining and Knowledge Discovery (DMKD)*, pages 44–53, 2000.

[SKC93] S. Shekhar, A. Kohli, and M. Coyle. Path Computation Algorithms for Advanced Traveller Information System (ATIS). In *ICDE*, pages 31–39, 1993.

[Smi97] M. Smid. Closest Point Problems in Computational Geometry. In J.-R. Sack and J. Urrutia, editors, *Handbook on Computational Geometry*. Elsevier Science Publishing, 1997.

[SRAE01] I. Stanoi, M. Riedewald, D. Agrawal, and A. El Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *VLDB*, pages 99–108, 2001.

[The03] Yannis Theodoridis. The R-tree-portal. *http://www.rtreeportal.org*, 2003.

[TPL04] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, pages 744–755, 2004.

[YL01] C. Yang and K.-I. Lin. An Index Structure for Efficient Reverse Nearest Neighbor Queries. In *ICDE*, pages 485–492, 2001.

[YW01] J. Yang and J. Widom. Incremental Computation and Maintenance of Temporal Aggregates. In *ICDE*, pages 51–60, 2001.