

residential buildings) to its nearest site (e.g. McDonald’s stores), if a new site is built at this location. Furthermore, an object is associated with a weight (e.g. the number of customers living at the same apartment building), and we aim at minimizing the weighted average distance (e.g. the average distance from each customer, instead of apartment building, to the nearest McDonald’s). The problem is formally defined as follows.

DEFINITION 1. *Given a set S of sites, a set O of objects with positive-integer weights, and a rectangular query region Q , the Min-Dist Optimal Location (MDOL) is a location l in Q that minimizes:*

$$AD(l) = \frac{\sum_{o \in O} dNN(o, S \cup \{l\}) * o.w}{\sum_{o \in O} o.w} \quad (1)$$

Here $AD(l)$ shows the weighted average distance from each object in O to its nearest site in S , including a hypothetical new site at l . And $dNN(o, S \cup \{l\})$ is the distance from o to that nearest site. Furthermore, $o.w$ is the weight of object o . Once again, throughout this paper whenever we say *distance* we mean L_1 *distance*.

To find a min-dist optimal location is challenging, even though in Section 3 we have a way of computing $AD(l)$ which measures how good a candidate location l is. The reason is, theoretically there are infinite number of locations in the query range Q . It may seem that all these locations are candidates.

In Section 4 we propose two theorems which show that we only need to examine a *finite set of candidate locations*, which are the intersections of some selected vertical lines and horizontal lines. In other words, if we pick the intersection point with the smallest $AD(\cdot)$, it is guaranteed to be an optimal location.

However, the straightforward solution, which computes $AD(\cdot)$ for all such intersection points, is not efficient. The reason is that there may be too many (although finite) such intersection points. Section 5 proposes a progressive algorithm **MDOL_prog**. The algorithm partitions the query range Q into a few cells (by using some of the vertical and horizontal lines), and calculates $AD(\cdot)$ for the corners of these cells. It is guaranteed that any candidate location, whose $AD(\cdot)$ is not computed yet, is in some unpruned cell and in turn can be found if the cell is partitioned. Among the cell corners, the one with the smallest $AD(\cdot)$ can be reported as a temporary optimal location, and the result can be continuously refined.

One key contribution of this paper is that it can calculate a lower bound of $AD(\cdot)$ of all locations in a given cell. This ability brings two benefits. First, along with the temporary optimal location, we can also report the maximum error. This allows the user to abort the calculation if the error is considered to be small enough. More importantly, it is possible to prune a whole cell (i.e., including the non-examined candidate locations in the cell). Two types of lower bounds are proposed, namely, the *data-independent lower bound* (Section 5.2) and the *data-dependent lower bound* (Section 5.3).

Empowered by the ability to calculate lower bounds (for cells), the algorithm **MDOL_prog** chooses the cell(s) with the smallest lower bounds and partitions them. Along with the partitioning, more candidate locations are revealed and

therefore a better temporary optimal location may be found, with a smaller $AD(\cdot)$. On the other hand, the minimum lower bound of unprocessed cells keeps increasing. When the two ends meet, the exact solution of the optimal location will be found. Section 5.5 studies how to select cells to partition and how to partition each cell.

The key contributions of the paper are summarized below.

- We propose the concept of *min-dist optimal-location query*, which is more practical than the existing *max-inf optimal-location query* [2].
- We present a way of computing $AD(l)$ which measures how good a candidate location l is (Section 3).
- We prove that the number of candidate locations is finite, and we have a way of computing them (Section 4). Here, by introducing the concept $VCU(R)$, we can further limit the number of candidate locations.
- We provide a progressive algorithm that computes the min-dist optimal location (Section 5). For pruning purposes, it utilizes novel lower-bound estimators for $AD(\cdot)$ of all locations in a rectangular cell. A batch cell-partitioning method is presented which partitions multiple cells together in a systematic and reasonable way.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 discusses how to compute $AD(l)$ for one candidate location l . Section 4 limits the number of candidate locations. The most crucial part is Section 5, which proposes the algorithm **MDOL_prog**. Performance results appear in Section 6. Finally, Section 7 concludes the paper.

2. RELATED WORK

The max-inf optimal location query was proposed in [2], where the problem was simply called *optimal location query*. As we said, it aimed at finding a location with maximum influence, where the influence of a location is the total weight of objects that are nearer to l than to all sites in S . These objects are called the RNNs of l , where *RNN* stands for *reverse nearest neighbor*. In [2] we first reduced the problem to the problem of finding a location which is contained by the most number of squares, then proposed three algorithms to solve it. We found that the VOL-tree method is the most efficient one. Unfortunately, those solutions do not apply to the newly proposed and more practical problem in this paper.

Since we also utilize the RNN concept when computing $AD(l)$, let’s briefly review it. The RNN problem was introduced by [4]. There are two variations of the RNN query: the monochromatic case [8, 10], and the bichromatic case [9]. We use the bichromatic version of the RNN concept, where there are two datasets (O and S) involved.

To compute the RNNs of a location l , Stanoi *et al.* [9] proposed an algorithm that dynamically constructs the *Voronoi cell* enclosing l . The Voronoi cell of l is a polygon, such that an arbitrary point is inside the polygon if and only if it is closer to l than to any other site in S . To compute this Voronoi cell, only sites in S are needed. Then, set O is probed to find the objects inside the Voronoi cell, which are the RNNs of l . Although there exists a large collection of

computational-geometry work on computing Voronoi cells [1], the solution in [9] is novel in the sense that it only examines a small fraction of the sites in S which are close to l . However, [9] applies on the L_2 metric only.

For the L_1 metric that this paper addresses, we could utilize the computational geometry solution [5]. And similar to [9], we develop an algorithm that computes the L_1 -metric Voronoi cell without examining all sites in S .

• Comparing with Facility Location Problem

Readers familiar with “operations research” may wonder whether the proposed MDOL operation is a solved instance of the facility location (FL) problem [4, 1]. The answer is negative. In the context of optimizing the locations of new McDonald’s, for example, the FL-version of the problem is as follows: given a set X of *pre-decided* locations, the goal is to find a subset X' of X , which minimizes

$$\frac{\sum_{o \in O} dNN(o, X') * o.w}{\sum_{o \in O} o.w}.$$

where the semantics of o , O , $dNN(\cdot)$, and $o.w$ are identical to those in MDOL search (Definition 1). The above problem, however, differs from MDOL retrieval in three important ways. First, the FL problem outputs potentially multiple locations (i.e., set X'), as opposed to only *one* location in an MDOL query. Second, FL knows in advance a *finite* set of candidate locations (i.e., set X), whereas in MDOL search we are given a rectangle Q , inside which all points may be candidate answers (i.e., an infinite number of them). Finally, FL disregards the existing McDonald’s restaurants, which actually play an imperative role in the formulation of MDOL.

• Comparing with k -Medoid/Median Problem

Given a set O of object locations (e.g. customers), the k -medoid query [7] finds a set of medoids $R \subseteq O$ with cardinality k that minimizes the average distance from each object $o \in O$ to its closest medoid in R . The k -median query [3, 6] is a variation, where we find k locations called medians, not necessarily in O , which minimize the average distance (from each object $o \in O$ to its closest median).

However our problem is different from both of them. A fundamental difference is that the existing problems do not assume a set S of *existing* sites (e.g. McDonald’s), but we do. If a city already has at least one McDonald’s store, and some locations should be chosen for new McDonald’s stores, existing work does not apply. In our problem, we want to find an optimal location l , and the nearest neighbor of each residential address is then taken from the UNION of S and $\{l\}$, i.e., every existing site will be considered in each nearest neighbor selection.

Furthermore, while the facility location problem and the k -median/ k -medoid problems are all NP-hard, our problem is not. Our work is a database query-processing paper instead of a theory paper. We find *exact* answers (under the constraints given in Definition 1), not approximate answers with proved approximation ratio (commonly used to tackle NP-hard problems). Even though there may seem to have infinite number of locations in the query region, according to our Theorem 2 we only need to check a finite number (quadratic to the number of objects) of candidate locations

in order to get the exact answer. A straightforward solution is to check all these candidates. The solution is clearly polynomial. NP-hardness is irrelevant here.

In the database community, however, such a straightforward algorithm is too expensive because the data set is typically large. We provide a carefully-designed, theorem-supported and experimentally-evaluated progressive algorithm. In particular, the checking of most candidate locations can be pruned.

To deal with large data sets, we rely on spatial index structures, e.g. to efficiently retrieve the set of objects that consider a particular location l as the closest site (compared with all existing sites in S), as discussed in Section 3.2. In fact the concern of I/O efficiency exists throughout the design of the progressive algorithm. Other examples include the methods to reduce the number of candidate locations (Section 4), the lower-bound theorems (Section 5.3), and the batch cell partitioning (Section 5.5). As described in the performance section, we utilize the R^* -tree spatial index and measures disk I/Os.

3. COMPUTING $AD(L)$ FOR A GIVEN LOCATION L

Before solving the min-dist optimal location query, we first address the following problem: Given a candidate location l , how do we compute the average distance $AD(l)$ as defined in Equation 1?

3.1 $AD(l)$ Can be Computed From $RNN(l)$

Let’s first define AD as the average distance between every object in O to its nearest site in S , without considering a new site. That is:

$$AD = \frac{\sum_{o \in O} dNN(o, S) * o.w}{\sum_{o \in O} o.w} \quad (2)$$

Clearly, $\forall l \in Q$, $AD(l) \leq AD$. If no object in O is closer to l than to its nearest site in S , $AD(l) = AD$. Otherwise, $AD(l) < AD$.

To compute $AD(l)$, by definition we could compute the nearest site (including l) for every object, and then take the average distance between an object to its nearest site. But this is costly. Intuitively, if we add a new McDonald’s store, we only want to visit the nearby residential buildings to compute the new average distance. In fact, we only need to visit objects in $RNN(l)$, as shown in Theorem 1.

THEOREM 1. $AD(l) = AD -$

$$\frac{1}{\sum_{o \in O} o.w} \sum_{o \in RNN(l)} (dNN(o, S) - d(o, l)) * o.w$$

PROOF. From Equations 1 and 2, we have: $AD - AD(l) =$

$$\frac{1}{\sum_{o \in O} o.w} \sum_{o \in O} (dNN(o, S) - dNN(o, S \cup \{l\})) * o.w.$$

Therefore, $AD(l) = AD -$

$$\frac{1}{\sum_{o \in O} o.w} \sum_{o \in O} (dNN(o, S) - dNN(o, S \cup \{l\})) * o.w.$$

Notice that if $o \notin RNN(l)$, $dNN(o, S) = dNN(o, S \cup \{l\})$. To prove the theorem, it remains to point out that for an object $o \in RNN(l)$, $dNN(o, S \cup \{l\}) = d(o, l)$. \square

Theorem 1 tells how to compute $AD(l)$ for an arbitrary location l . Compared with l which varies, S and O can be considered as fixed. Therefore we can pre-compute AD , $\sum_{o \in O} o.w$, and $dNN(o, S)$ for every object. In order to compute $AD(l)$, we only need to find the RNNs of l and compute $d(o, l)$ for every $o \in RNN(l)$.

3.2 The Computation of $RNN(l)$

To find the RNNs of l , we can compute the Voronoi cell of l by examining S , and then locate the objects in O that are spatially enclosed in the Voronoi cell. There exists plenty of work on Voronoi cells, including the L_1 distance case [5] which we need.

However, existing work focused on computing the Voronoi cell of l by examining all sites in S . This is expensive if too many sites exist. Intuitively, it should be enough to examine only the sites close to l . For the L_2 distance, [9] proposed a method to utilize a spatial index structure on the set of sites to quickly identify some sites close to l and compute a range in space, with the guarantee that all sites outside the range can be avoided. We extended this idea to deal with L_1 metric. Due to the space limitation, the L_1 distance method is included in the full version of our paper [12].

4. LIMITING THE NUMBER OF CANDIDATES

If we have only a few candidate locations to choose from, we could compute their $AD(\cdot)$ using the method in Section 3, and then pick the location with the smallest $AD(\cdot)$. The challenge of solving the min-dist optimal-location query lies in the fact that there are too many candidate locations in the query region Q . In fact, theoretically a region contains an infinite number of locations. This section limits the number of candidate locations that need to be checked, yet still guarantees that an exact answer is found.

4.1 The Number of Candidate Locations is Finite

Consider Figure 3. The black dots are the objects. Here the thick-bordered rectangle is the query region Q . The shadowed region is composed of a horizontal extension of Q and a vertical extension of Q , which are defined below.

DEFINITION 2. Given an axis-parallel rectangle Q , the **horizontal extension** of Q is the area derived from infinitely extending Q horizontally. The **vertical extension** of Q is the area derived from infinitely extending Q vertically.

Obviously, the intersection between the horizontal extension and the vertical extension is Q itself.

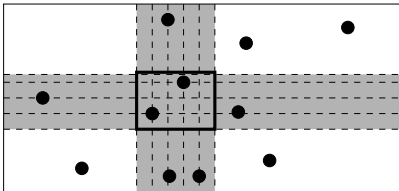
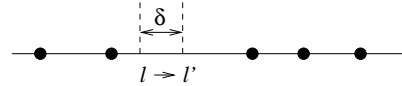


Figure 3: The candidate locations are limited to the intersections of the dashed lines.

Consider each horizontal line that passes through at least one object in the horizontal extension of Q and each vertical line that passes through some object in the vertical extension of Q . Also consider the horizontal and vertical lines that pass through the corners of Q . For instance, in Figure 3 there are six such vertical lines and five such horizontal lines as shown in dashed style. (Note that one line may pass through more than one object.) They make 30 intersection points. According to Theorem 2 below, even though there is an infinite number of locations in Q , we only need to check these intersection points. It is guaranteed that we can find a min-dist optimal location among them.

THEOREM 2. Consider the set of horizontal (and vertical) lines that go through some object in the horizontal (and vertical) extension of Q or go through some end point of Q . There exists a min-dist optimal location at some intersection point of these lines.

PROOF. Consider $RNN(l)$: the set of objects in O that are closer to l than to their nearest sites in S . Let's start with a simple case as illustrated in the figure below, where the objects in $RNN(l)$ as well as l are all located in a horizontal line. Further, assume all objects have weight = 1.



Since all the objects in $RNN(l)$ consider l as the nearest site, we have $dNN(o, S \cup \{l\}) = d(o, l)$. Therefore, the total contribution of such objects to the numerator of $AD(l)$ (in Equation 1) is the sum of distances from each object to l . This total is the summation of two parts: the total distance between l and the objects to its left, and the total distance between l and the objects to its right.

Without loss of generality, assume there are more objects in $RNN(l)$ to the right of l than to its left. Let's move l a little bit to its right side without passing any object, while assuming $RNN(l)$ remains the same during this move. We argue that this can only improve $AD(l)$, i.e. make it smaller. Let there be n_L objects to the left of l , and n_R objects to the right of l . If we move l to l' where the moving distance is δ , the increase of the total distance between l and the objects to its left is $n_L * \delta$, while the decrease of the total distance between l and the objects to its right is $n_R * \delta$. Since we assume $n_L < n_R$, the decrease amount is more than the increase amount. Therefore $AD(l') < AD(l)$.

This tells us that by moving l to the side with more objects, $AD(l)$ always decreases. One may wonder when we can reach the minimum $AD(l)$. The answer is: either when there are equal numbers of objects to the left and right of l , or when we reach a border of the query region Q (remember l should be inside Q). To make sure there are equal numbers of objects to the left and to the right of l , we should make sure that l co-locates with the median object. If the number of objects is even, any location between the two median objects is optimal, including the location of each of these two median objects. In any case, the theorem is correct, i.e. there exists an optimal location either at the same X coordinate as some existing object, or at the smallest (or largest) X coordinate of Q .

The discussion above was based on the following assumptions:

Notation	Meaning
S	the set of sites
O	the set of objects
Q	the query region
l	a candidate location for a new site
$d(p_1, p_2)$	the L_1 distance between two points
$dNN(o, S)$	the L_1 distance between object o to its nearest site in S
$RNN(l)$	the set of objects in O that are closer to l than to their nearest sites in S
$AD(l)$	the average distance between an object in O to its nearest site in $S \cup \{l\}$
AD	the average distance between an object in O to its nearest site in S
$VCU(R)$	the Voronoi cell union of R with regard to S

Table 1: Summary of Notations.

1. When moving l (to l'), the set of RNNs remains unchanged.
2. All objects have weight = 1.
3. The object placement is one-dimensional.

To make the proof complete, let's drop all these assumptions and see why the theorem is still correct.

To drop Assumption 1, we acknowledge that it is possible to have $RNN(l) \neq RNN(l')$. But this does not affect the correctness of the theorem for the following reasons. Let object $o \in RNN(l)$ but $o \notin RNN(l')$. The distance between o to its nearest site is smaller than what we counted under Assumption 1. Therefore the real $AD(l')$ should be even smaller, and thus moving l (to l') remains to improve $AD(l)$. The same discussion applies to the case when $o \notin RNN(l)$ but $o \in RNN(l')$.

To drop Assumption 2, let there be objects whose weight is more than 1. In the problem definition we said the weight of objects should be positive integers like the number of people living in a residential building. The proof of the theorem remains valid if we consider every object o to be multiple objects with weight 1, where the number of copies should be $o.w$.

To drop Assumption 3, let's consider the 2-dimensional case. We can prove the theorem by moving l first horizontally and then vertically. Consider the objects in $RNN(l)$. Let there be n_L objects whose X coordinates are smaller than that of l , and n_R objects whose X coordinates are bigger than that of l . Assume $n_L < n_R$. We claim that by moving l a little bit to the right side will improve $AD(l)$ for the following reason. We know the L_1 distance between two points is their horizontal distance plus their vertical distance. If we move l horizontally, the vertical distance between an object to l remain unchanged. Therefore, according to our discussion in the one-dimensional case, we get a smaller $AD(l)$ since the total horizontal distance is smaller. Next, we move l vertically. And the discussion is the same as in the horizontal case. \square

4.2 Further Limiting the Number of Candidate Locations

According to Theorem 2, to get the set of candidate locations, we considered all objects in either the horizontal extension or the vertical extension of Q . In fact, if we study the theorem proof carefully, we can notice that there is no need to consider all such objects. Instead, we only need to consider the objects which belong to $RNN(l)$ for some location l in Q . Here we introduce a concept called the *Voronoi*

cell union and discuss how it can be used to further limit the number of candidate locations.

DEFINITION 3. Given a set of sites S and a spatial region Q , the **Voronoi cell union** of Q with regard to S is the union of Voronoi cells of every location l in Q .

Let's use $VCU(Q)$ to denote the Voronoi cell union of Q . $VCU(Q)$ is the minimum spatial region that fully contains the Voronoi cell for every location l in Q . An equivalent explanation is that $VCU(Q)$ is a spatial region consisting of the locations which may consider a new site built somewhere in Q as the nearest site.

For ease of reference, Table 1 summarizes the variables that have been introduced.

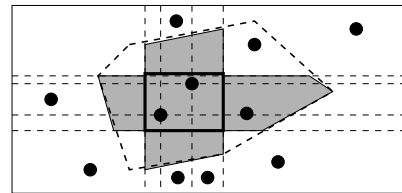


Figure 4: The candidate locations are further reduced using $VCU(Q)$.

The VCU concept can help reduce the number of candidate locations in the following way. When choosing objects which will define the vertical and horizontal lines (whose intersections are the candidates), we only need to choose the objects in $VCU(Q)$. For example, in Figure 4, let the dashed polygon be $VCU(Q)$. We only need to consider the objects in the shadowed region which is the intersection between $VCU(Q)$ and the vertical/horizontal extensions of Q . In this example, there are now four horizontal lines and four vertical lines, which result in 16 candidate locations instead 30. Think the whole space as the United States, and the query region as a city. The range $VCU(Q)$ is typically a small extension of Q , and therefore the savings of this optimization may be big in practice.

We develop a method which efficiently computes $VCU(Q)$. Similar to the case of computing the Voronoi cell of a single location, our method only needs to examine a small subset of sites in S . The sites too far away from Q or located inside Q , do not need to be examined. The algorithm [12] is omitted in this paper due to the space limitation.

5. THE PROGRESSIVE ALGORITHM

We now have an algorithm to accurately compute a min-dist optimal location. That is:

Algorithm MDOL_basic

1. Retrieve the objects in the intersection between $VCU(Q)$ and the horizontal/vertical extensions of Q .
2. Derive the set of candidate locations.
3. Compute the average distance $AD(l)$ for every candidate location l .
4. Return the candidate location with the minimum average distance.

This is a fine algorithm if the number of candidate locations is small. However, if there are too many candidate locations, the basic algorithm is not efficient.

5.1 The Progressive Algorithm Outline

We hereby introduce a progressive algorithm. The idea is illustrated in Figure 5. Here we see five horizontal lines and five vertical lines that partition the query region Q . They make 25 intersection points as candidate locations. It is easily imaginable that in practice the number of candidate locations is much more. Assume we cannot check the average distance for *all* of these candidate locations *together*. What we do is we first partition Q at a coarse granularity, which results in a subset of candidate locations as illustrated in Figure 5(a). The average distance $AD(l)$ is calculated for each of such candidate locations. The location with the smallest $AD(l)$ is returned to the user as a temporary optimal location. Next, we go to a finer partitioning and introduce more candidate locations (Figure 5(b)). A better temporary optimal location may be returned to the user. Eventually, if we go to the finest partitioning, we will check all candidate locations and will find the accurate optimal location.

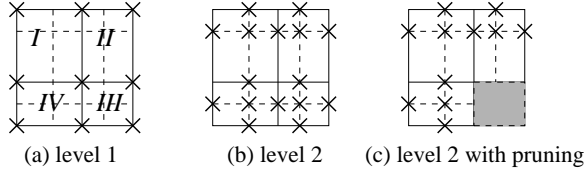


Figure 5: The main idea of the progress algorithm.

One may wonder: “*the description above still checks all candidate locations, so what do you save?*” Well, the description is a skeleton of the algorithm, and we will enhance it by providing a lower-bound computation facility. That is, with a coarse partitioning, the query region Q is partitioned into multiple cells (e.g. I , II , III , and IV in Figure 5(a)). For each cell C , we can compute a lower bound of $AD(l)$ for all locations in C , denoted as $LB(C)$. Empowered with this lower-bound computation facility, our algorithm has the following two abilities:

- We may prune complete cells. Let C be a cell, and l be a candidate location that has been checked. If $LB(C) \geq AD(l)$, we know no location in cell C can reach a lower average distance than that of l . Therefore the computation of all candidate locations in C can be avoided (Figure 5(c)).

- Along with a temporary optimal location that is reported to the user, our algorithm can report a range of average distance values. For instance, our algorithm may not only report a temporary optimal location whose average distance is 3000 (meters), but also claim that the the average distance of the real optimal location should be in the range [2500,3000]. Notice that with new iterations, this range can only be shrunk. The next iteration may report a new location with range [2800, 2900]. This ability gives the user a choice of stopping at an accurate-enough approximate result, in case the accurate answer takes a long time to finish.

5.2 The Data-Independent Lower-Bound

Given a cell C our task is to compute a lower bound of $AD(l)$ for all locations l in C . At this point, we know $AD(c_i)$ for the four corners c_i of C , and we know the perimeter of C . But let’s assume we know no further information. That is, we do not assume any knowledge of sites in S or objects in O . Such a lower bound is called data-independent. With the help of Lemma 1, Corollary 1 shows a first data-independent lower bound, and Theorem 3 gives a better (tighter) one. In the next section we will provide a data-dependent lower bound.

LEMMA 1. For any two locations l and l' , $AD(l') - AD(l) \leq d(l, l')$.

PROOF. We know AD shows the average distance between an object to the nearest site in S . If a new site is built, either at l or l' , this average distance may be improved. For an arbitrary object o , let’s consider the *difference in benefit*: how much more its distance (to the nearest site) can benefit from l than from l' . This difference in benefit is $(dNN(o, S) - dNN(o, S \cup \{l\})) - (dNN(o, S) - dNN(o, S \cup \{l'\})) = dNN(o, S \cup \{l'\}) - dNN(o, S \cup \{l\})$. Notice that $AD(l') - AD(l)$ is the average of such a difference in benefit. We will differentiate two cases to show that this difference in benefit is never more than $d(l, l')$. And that will finish the proof of the lemma.

If $o \notin RNN(l)$, it does not help o by building a new site at l , i.e. $dNN(o, S \cup \{l\}) = dNN(o, S)$. Since $dNN(o, S \cup \{l'\}) \leq dNN(o, S)$, the difference in benefit is less than or equal to 0, which in turn is less than or equal to $d(l, l')$.

If $o \in RNN(l)$, the difference in benefit is $dNN(o, S \cup \{l'\}) - d(o, l)$. Since $dNN(o, S \cup \{l'\}) \leq d(o, l')$, the difference in benefit is no more than $d(o, l') - d(o, l) \leq d(l, l')$. \square

This lemma leads to a straightforward lower bound of a cell given by the following corollary:

COROLLARY 1. Let the corners of a cell C be c_1, c_2, c_3 , and c_4 . Let the perimeter of C be p .

$$\min_{1 \leq i \leq 4} \{AD(c_i)\} - \frac{p}{4}$$

is a lower bound of $AD(l)$ for any location $l \in C$.

PROOF. Consider an arbitrary location $l \in C$. Let c_k be its nearest corner. According to Lemma 1, we have: $AD(l) \geq AD(c_k) - d(l, c_k)$. It is obvious that $d(l, c_k) \leq p/4$ and $AD(c_k) \geq \min_{1 \leq i \leq 4} \{AD(c_i)\}$. So we have

$$AD(l) \geq \min_{1 \leq i \leq 4} \{AD(c_i)\} - \frac{p}{4}.$$

\square

Now we get a lower bound. However, we can do better. Theorem 3 below shows a tighter (larger) lower bound, which may lead to more pruning power.

THEOREM 3. *Let the corners of a cell C be $c_1, c_2, c_3,$ and c_4 , where $\overline{c_1c_4}$ is a diagonal. Let the perimeter of C be p .*

$$\max\left\{\frac{AD(c_1) + AD(c_4)}{2}, \frac{AD(c_2) + AD(c_3)}{2}\right\} - \frac{p}{4} \quad (3)$$

is a lower bound of $AD(l)$ for any location $l \in C$.

PROOF. Due to symmetry, it is sufficient to prove that the following formula holds for every location $l \in C$,

$$AD(l) \geq \frac{AD(c_1) + AD(c_4)}{2} - \frac{p}{4}.$$

According to Lemma 1, we know:

$$\begin{aligned} AD(l) &\geq AD(c_1) - d(l, c_1) \\ AD(l) &\geq AD(c_4) - d(l, c_4) \end{aligned}$$

Therefore, we have:

$$AD(l) \geq \frac{AD(c_1) + AD(c_4)}{2} - \frac{d(l, c_1) + d(l, c_4)}{2}.$$

It remains to point out that $d(l, c_1) + d(l, c_4) = p/2$ holds for any $l \in C$. \square

Figure 6 illustrates the superiority of this new lower bound over the previous one. Based on Corollary 1, we get a lower bound $1000 - p/4$. Theorem 3 shows a better lower bound $3500 - p/4$.

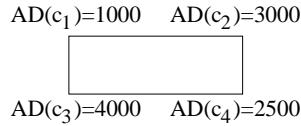


Figure 6: The lower bound in Theorem 3 is $3500 - p/4$, which is better than $1000 - p/4$, the lower bound in Corollary 1.

5.3 The Data-Dependent Lower-Bound

If we know some information about the objects O and the sites S , we may be able to get a tighter lower bound. In the extreme case, if we know all the exact information of O and S , we can compute the exact optimal location. In this section, we will derive a meaningful lower bound (in Theorem 4), assuming no knowledge of O or S except two values: $\sum_{o \in O} o.w$ and $\sum_{o \in VCU(C)} o.w$. The former value is the total weight of objects in the whole space, which can be assumed known. The latter value can be acquired by computing the region $VCU(C)$ and then performing an aggregation query on the (index of the) set of objects.

LEMMA 2. *Given any two locations l and l' on space,*

$$AD(l) - AD(l') \leq \frac{\sum_{o \in RNN(l')} d(l, l') * o.w}{\sum_{o \in O} o.w}.$$

PROOF. For clarity, we can assume every object has weight = 1. This does not lose generality because an object o whose weight $o.w > 1$ can be treated as $o.w$ objects, each of which

having weight = 1. Therefore what we need to prove becomes:

$$AD(l) - AD(l') \leq \frac{|RNN(l')|}{|O|} d(l, l')$$

where $|O|$ and $|RNN(l')|$ are the numbers of objects in O and $RNN(l')$, respectively.

According to the definition of $AD(l)$ and $AD(l')$, we have:

$$AD(l) - AD(l') = \frac{1}{|O|} \sum_{o \in O} (dNN(o, S \cup \{l\}) - dNN(o, S \cup \{l'\})).$$

We divide the summation into two cases, one for those $o \in RNN(l')$ and one for those $o \notin RNN(l')$.

We know

$$dNN(o, S \cup \{l'\}) = \begin{cases} d(o, l'), & \text{if } o \in RNN(l') \\ dNN(o, S), & \text{if } o \notin RNN(l') \end{cases}$$

and we know

$$dNN(o, S \cup \{l\}) \leq dNN(o, S)$$

$$dNN(o, S \cup \{l\}) \leq d(o, l).$$

Therefore, $AD(l) - AD(l')$

$$\begin{aligned} &= \frac{1}{|O|} \sum_{o \in RNN(l')} (dNN(o, S \cup \{l\}) - dNN(o, S \cup \{l'\})) \\ &\quad + \frac{1}{|O|} \sum_{o \notin RNN(l')} (dNN(o, S \cup \{l\}) - dNN(o, S \cup \{l'\})) \\ &= \frac{1}{|O|} \sum_{o \in RNN(l')} (dNN(o, S \cup \{l\}) - d(o, l')) \\ &\quad + \frac{1}{|O|} \sum_{o \notin RNN(l')} (dNN(o, S \cup \{l\}) - dNN(o, S)) \\ &\leq \frac{1}{|O|} \sum_{o \in RNN(l')} (d(o, l) - d(o, l')) \\ &\quad + \frac{1}{|O|} \sum_{o \notin RNN(l')} (dNN(o, S) - dNN(o, S)) \\ &= \frac{1}{|O|} \sum_{o \in RNN(l')} (d(o, l) - d(o, l')) \\ &\leq \frac{1}{|O|} \sum_{o \in RNN(l')} d(l, l') \\ &= \frac{|RNN(l')|}{|O|} d(l, l') \end{aligned}$$

\square

Intuitively, the lemma above can be explained as follows: $(AD(l) - AD(l')) * |O|$ is the total extra benefit of distances to the nearest sites for all objects by building a new site at l' instead of l . The objects which may benefit more from l' than from l are those objects within $RNN(l')$. For each of such object, the maximal extra saving is no more than $d(l, l')$. Therefore, the total extra saving is no more than $d(l, l') * |RNN(l')|$. Based on this lemma, we can prove the following lower bound, which is better than the data-independent lower bound we got before.

THEOREM 4. Let the corners of a cell C be $c_1, c_2, c_3,$ and c_4 , where $\bar{c}_1\bar{c}_4$ is a diagonal. Let the perimeter of C be p .

$$\max\left\{\frac{AD(c_1) + AD(c_4)}{2}, \frac{AD(c_2) + AD(c_3)}{2}\right\} - \frac{p * \sum_{o \in VCU(C)} o.w}{4 * \sum_{o \in O} o.w}$$

is a lower bound of $AD(l)$ for any location $l \in C$.

PROOF. Once again, we can assume every object has weight = 1 without loss of generality. It is sufficient to prove that

$$\frac{AD(c_1) + AD(c_4)}{2} - \frac{p * |VCU(C)|}{4 * |O|}$$

is a lower bound.

According to Lemma 2, we have

$$AD(c_1) - AD(l) \leq \frac{|RNN(l)|}{|O|} d(l, c_1)$$

or

$$AD(l) \geq AD(c_1) - \frac{|RNN(l)|}{|O|} d(l, c_1).$$

Similarly,

$$AD(l) \geq AD(c_4) - \frac{|RNN(l)|}{|O|} d(l, c_4).$$

Therefore, we have: $AD(l)$

$$\begin{aligned} &\geq \frac{AD(c_1) + AD(c_4)}{2} - \frac{|RNN(l)|}{|O|} \frac{d(l, c_1) + d(l, c_4)}{2} \\ &= \frac{AD(c_1) + AD(c_4)}{2} - \frac{|RNN(l)| p}{|O| 4} \end{aligned}$$

It remains to point out that since l is in cell C , $|RNN(l)| \leq |VCU(C)|$. \square

Typically, $\sum_{o \in VCU(C)} o.w$ is much smaller than $\sum_{o \in O} o.w$. Therefore the data-dependent lower bound given in Theorem 4 is generally much better (larger) than the data-independent lower bound given in Theorem 3.

5.4 The Algorithm

Now that we have confidence in computing a lower bound on $AD(l)$ for locations l in any cell C , we are ready to provide the progressive algorithm. Let $LB(C)$ represent the data-dependent lower bound for cell C .

Algorithm MDOL_prog

1. Retrieve the objects in the intersection between $VCU(Q)$ and the horizontal/vertical extensions of Q . Derive the set of horizontal and vertical lines, whose intersections are the candidate locations.
2. Maintain a heap of cells ordered by $LB(\cdot)$. Initially, the heap contains one cell: the query region Q .
3. Set l_{opt} be the corner of Q with minimum $AD(\cdot)$.
4. If the heap is empty, or if $AD(l_{opt}) \leq$ the minimum $LB(\cdot)$ of cells in the heap, **return** l_{opt} as the optimal location.
5. Remove the cell C from the heap with minimum $LB(\cdot)$.
6. If C cannot be partitioned, **goto** Step 4.
7. Partition C into a set of k sub-cells.
8. Compute $AD(\cdot)$ for the corners of all sub-cells, if not computed already. If any corner c_i has $AD(c_i) < AD(l_{opt})$, set $l_{opt} = c_i$.

9. Compute the lower bound $LB(\cdot)$ for every sub-cell using Theorem 4.
10. For every sub-cell C_i where $LB(C_i) < AD(l_{opt})$, insert C_i into the heap.
11. **goto** Step 4.

The algorithm contains initialization (Steps 1, 2, 3) and a loop (Steps 4 through 11). It maintains a heap of cells, and a temporary optimal location l_{opt} . Initially, the heap has one cell: Q . And the temporary optimal location is initialized to be the corner of Q with minimum $AD(\cdot)$. Later on, each iteration removes a cell from the heap and partitions it into sub-cells which will be re-inserted into the heap.

To fully understand the algorithm, let's discuss several key points of it below, and discuss the cell-partitioning issue in Section 5.5.

5.4.1 Cell Pruning using Lower Bound

Along with each cell C , we keep its lower bound $LB(C)$. It is a lower bound of $AD(l)$ for every location l in C . Therefore, if $LB(C) \geq AD(l_{opt})$, we know no location in C can be a better candidate than the current temporary optimal location l_{opt} . So as Step 10 of the algorithm shows, we can prune the examination of a cell C_i (and all candidate locations in it) if $LB(C_i) \geq AD(l_{opt})$.

5.4.2 Continuous Refinement of the Query Result

One feature of the algorithm is that it can quickly report a temporary optimal location, along with its confidence interval, and it can keep refining the result. When the algorithm terminates, the real optimal location can be found. But the user reserves the flexibility of aborting the execution at any time when the quality of the temporal optimal location is high enough (as indicated by the confidence interval). This feature can be helpful especially when the data volume is large and the algorithm takes a long time to terminate.

The confidence interval is $[AD_{low}, AD_{high}]$. Here AD_{low} is the minimum $LB(C_i)$ for all cells in the heap, and $AD_{high} = AD(l_{opt})$. Let the real optimal location be $l_{realopt}$. It is guaranteed that $AD(l_{realopt}) \in [AD_{low}, AD_{high}]$ is true at all times. Therefore, the shorter this interval is, the more confident we are on the temporary optimal location. When the algorithm terminates, the confidence interval shrinks to a single point, and we are certain that we have found the real optimal location.

Once the algorithm starts to run, as time passes by the confidence interval shrinks. On the left side, AD_{low} keeps increasing. This is because when we remove a cell C from the heap (Step 5) and partition it to sub-cells (Step 7) to be inserted to the heap (Step 10), we know $LB(C_i)$ of any sub-cell C_i is at least as large as $LB(C)$. On the right side, AD_{high} keeps decreasing. This is because we only replace l_{opt} with a better candidate location. In summary, the algorithm reports better and better candidate locations whose confidence intervals keeps improving.

5.4.3 The Stopping Condition

Step 4 shows two stopping conditions. If the heap is empty, there is no unprocessed cell and therefore there is no more candidate location to examine. So the temporal optimal location is really optimal. Another condition is when all cells in the heap have $LB(\cdot)$ at least as large as $AD(l_{opt})$. In

this case, no candidate location in any cell in the heap can be better than l_{opt} , and therefore the algorithm can terminate.

One may wonder: “Step 10 has ensured that we only insert a cell C_i into the heap when $LB(C_i) < AD(l_{opt})$. So how can it happen that the heap contains some cell whose $LB(\cdot) \geq AD(l_{opt})$?” The answer is that $AD(l_{opt})$ shrinks as better and better temporary optimal location is found. Therefore it is quite possible that some prunable cell was validly inserted to the heap based on an old $AD(l_{opt})$.

This observation tells us an additional thing we could do (not specified in the algorithm). That is, whenever l_{opt} changes, we remove from the heap every cell C_i where $AD(C_i) \geq AD(l_{opt})$. This additional cost of eager removal can help us keep the heap compact at all times. However, we tend not to implement this eager removal for the following reason. This operation will remove some cells with the largest $LB(\cdot)$. Because the heap is optimized to extract a cell with minimum $LB(\cdot)$, it is costly to remove from the other end.

5.5 Batch Cell Partitioning

When a cell C with the smallest $LB(\cdot)$ is removed from the heap (Step 5), Step 7 of the algorithm MDOL_prog simply said “Partition C into a set of sub-cells”. This section addresses the issue how to partition. In fact, we directly address the extended problem, when multiple cells may be chosen to partition together.

The motivation is that to insert a sub-cell C_i into the heap, we need to access the indices storing O and S (in order to compute $VCU(C_i)$, $LB(C_i)$, and non-computed $AD(\cdot)$ for the corners of C_i), which can be expensive. Therefore we want to batch the access. In other words, we want to compute the associated information for multiple sub-cells together, for each access to the indices of O and S .

The number of new cells to process depends on the available memory. In the extreme case when we had unlimited memory, we could partition Q to the finest level and compute everything by visiting the indices of O and S once. In practice, the allowed number of new cells to process together is not infinity. We therefore face the following **design problems**, assuming we can process k (denoted as partitioning capacity) new cells together:

1. Which cells shall we partition? How many sub-cells shall we partition each cell into?
2. Given a particular cell and the number of sub-cells we aim at partitioning it into, how to partition it?

5.5.1 Solution for Design Problem One

In order to increase the minimum $LB(\cdot)$, we should definitely partition the cells whose $LB(\cdot)$'s are the smallest among all cells in the heap. Moreover, it may not be good enough to only partition the cells whose $LB(\cdot)$'s are equal to the minimum $LB(\cdot)$ – what about a cell whose $LB(\cdot)$ is slightly larger? In general, we want to distribute k to multiple cells, and the smaller $LB(C)$ is, the more sub-cells we partition C into.

We hereby propose a scheme, where the number of sub-cells of C is proportional to $1/LB(C)$. Let C_1, \dots, C_t be the set of t cells in the heap with the smallest $LB(\cdot)$, where t is a pre-defined constant. For every cell C_i , we partition it into $NSC(C_i)$ sub-cells.

$$NSC(C_i) = \frac{k}{LB(C_i) \sum_{j=1}^t \frac{1}{LB(C_j)}} \quad (4)$$

It can be verified that

$$\sum_{i=1}^t NSC(C_i) = k$$

$$\frac{NSC(C_i)}{NSC(C'_i)} = \frac{1/LB(C_i)}{1/LB(C'_i)}$$

Example: Let $t = 4$, $LB(C_1) = 10$, $LB(C_2) = 10$, $LB(C_3) = 100$, and $LB(C_4) = 100$. Let $k = 44$. We have:

$$\sum_{j=1}^t \frac{1}{LB(C_j)} = \frac{1}{10} + \frac{1}{10} + \frac{1}{100} + \frac{1}{100} = 0.22$$

and then:

$$NSC(C_1) = NSC(C_2) = \frac{44}{10 * 0.22} = 20$$

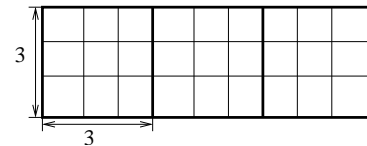
$$NSC(C_3) = NSC(C_4) = \frac{44}{100 * 0.22} = 2.$$

5.5.2 Solution for Design Problem Two

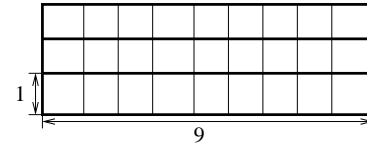
Suppose we aim at partitioning a cell C into k' sub-cells. We need to know how to partition. There are **two sub-problems** here. First, we need to decide how many partitions we should make for the X axis, and how many for the Y axis. Second, we need to choose the vertical and horizontal lines. What guides our design choices in both sub-problems is that we should try to:

- Make sub-cells have similar sizes (or perimeters).
- Make sub-cells be square-shaped.

The reason why we try to make sub-cells have equal sizes is: if some sub-cell has very small sizes and some others have very large sizes, the ones with large sizes, or large perimeter p , may have very small $LB(\cdot)$ (according to Theorem 4), contradictory to our goal of increasing $LB(\cdot)$ of cells in the heap as much as possible.



(a) partitioning into square-like cells



(b) partitioning into thin and long cells

Figure 7: Partitioning into square-like cells results in sub-cells with perimeter 12, while partitioning into thin and long cells results in sub-cells with perimeter 20.

The reason why we try to have square-shaped sub-cells is illustrated in Figure 7. Here cell C has $hu = 9$ horizontal units and $vu = 3$ vertical units. That is, if we partition C into the finest level, we will get $9*3=27$ sub-cells. Our goal

is to partition C into $k' = 3$ sub-cells. Figure 7(a) partitions C into 3 square-like cells, where each cell has perimeter $p = (3 + 3) * 2 = 12$. On the other hand, Figure 7(b) partitions C into 3 thin-and-long cells, where each cell has perimeter $p = (1 + 9) * 2 = 20$. According to Theorem 4, even though both approaches partition C into equal number of sub-cells, the former approach tends to produce a partitioning with larger $LB(\cdot)$ for every sub-cell.

Therefore, we should aim at having

$$\frac{n_x}{n_y} = \frac{w}{h}$$

where n_x is the number of resulted partitions on the X axis (of C), n_y is the number of resulted partitions on the Y axis, w is the width of C , and h is the height of C .

On the other hand, since we aim at partitioning C into k' sub-cells, we have

$$n_x * n_y = k'$$

As a result:

$$n_x = \sqrt{\frac{w \cdot k'}{h}}, n_y = \sqrt{\frac{h \cdot k'}{w}} \quad (5)$$

The above equation has solved the first sub-problem in determining the number of partitions in X and Y . Let's now tackle the second sub-problem on how to partition. Due to symmetry we focus on the X dimension. The task is to choose $n_x - 1$ vertical lines (so as to create n_x partitions for the X range of C). As Figure 8 shows, we should partition in such a way that each partition has roughly the same width.

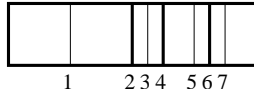


Figure 8: To create n_x partitions (in the X axis), the thick vertical lines show a method of leaving each partition with equal number of unprocessed lines. A better choice is to choose vertical lines 1, 2 and 5, to make each partition have the same width.

Clearly, the $n_x - 1$ hypothetical lines that form an equi-width partitioning (of the X range of C) may not co-locate with existing vertical lines. A straightforward approach is to first compute the equi-width lines, and then for every equi-width line choose the closest existing vertical line. However, as Figure 9 shows, the approach may not work, since multiple equi-width lines may correspond to the same existing vertical line. In the example, both the second and the third equal-width lines consider line 5 as the closest existing line. To solve this problem, we process one equi-width line at a time, from left to right. For each such line, we assign it to the closest vertical line not assigned to any previous one yet. After processing each equi-width line, we check to make sure that the number of remaining vertical lines \geq the number of remaining equi-width lines. If the condition is not met, the remaining equi-width lines are matched with the right-most existing vertical lines. As an example, in Figure 9, after associating vertical line 4 with the first equi-width line, the above condition is not met. Therefore the last two equi-width lines are associated with the last two existing vertical lines (4 and 5). And in turn line 3 is chosen to be associated with the first equi-width line.

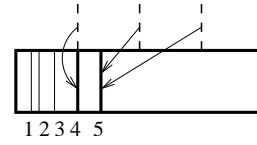


Figure 9: Matching every equi-width line (shown as a dashed line) to the closest existing vertical line may not work.

6. PERFORMANCE

In this section we experimentally evaluate our algorithm and optimizations. We use a real dataset: the 123,593 postal addresses in northeastern part of the United States (New York, Philadelphia and Boston). The dataset is available at the R-tree Portal [11]. For each experiment, given the number of sites, we randomly select some data points as the sites and use the rest as the objects. The objects are stored in an R*-tree, augmented by the L_1 distance from each object to its nearest site. The pagesize of the R*-tree is 4KB. In real applications, the number of sites is typically very small. So, in our experiments, we keep all sites in memory. However, the sites can be organized as an R*-tree and our algorithm still applies. In each experiment, we issue 100 random queries with fixed size, and take their average running time. We use a buffer of size 128 pages and measure the total disk I/Os to the object R*-tree. All experiments are performed on a Dell Pentium IV 3.2GHz PC with 1GB memory. Unless otherwise stated, the experiments use the default parameters as given in Table 2.

Parameter	Default value
Number of sites	100
Query size	1% in each dimension
Partitioning capacity (k)	40

Table 2: The default parameters.

6.1 The Effect of VCU Computation

We first verify that computing the VCU of the query range significantly reduces the number of candidate locations. As Figure 10 shows, computing VCU reduces the number of candidates by about two orders of magnitude. The number of candidates are roughly proportional to the area of the query range for both cases. Thus, in Figure 10, two lines increase almost at the same rate when the query size increases. In the rest of experiments, we always compute VCU of the query range.

6.2 Comparison of the Three Lower Bounds

We have proposed three versions of lower bounds of $AD(l)$ for all locations l in a cell C , as shown in Table 3.

To compare their pruning power, we implement three versions of the algorithm by using these three lower-bounds, respectively. The query size is 0.25% in each dimension. Figure 11 shows the total disk I/Os and running time of these three algorithms. When the number of sites increases, the disk I/Os and running time of all three algorithms decrease, and the gap between DDL and the other two methods also decreases. This is because the VCU of a cell shrinks with

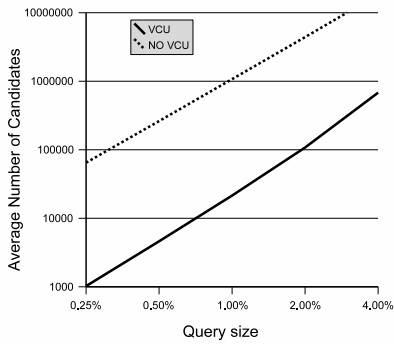


Figure 10: The effect of VCU computation.

Notation	Lower bound	Where
SL	straightforward	Corollary 1
DIL	data-independent	Theorem 3
DDL	data-dependent	Theorem 4

Table 3: The three lower bounds.

more sites and we have fewer candidates. Thus, the number of candidates that need to be pruned also becomes smaller. So DIL has a little better pruning power than SL, while DDL is clearly superior than both of them. In other words, the data-dependent lower bound has the strongest pruning power.

6.3 Impact of Lower-Bound Pruning

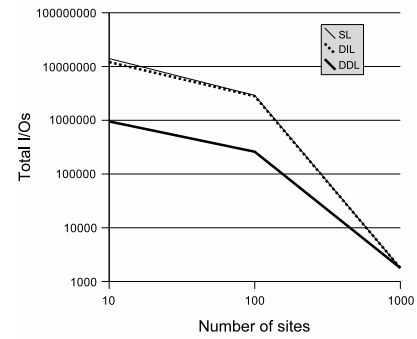
To see how much improvement we can get by using lower-bound pruning, we compare the query I/O performance between the naive algorithm (denoted as *naive*) which checks all candidates, and the algorithm which prunes candidates utilizing the data-dependent lower bound. As shown in Figure 12, using pruning can bring multiple orders of magnitude performance improvement. This is because, with the increase of the query size and the number of candidates, the difference between Naive and DDL in the number of pruned candidates becomes larger.

6.4 The Effect of Batch Partitioning

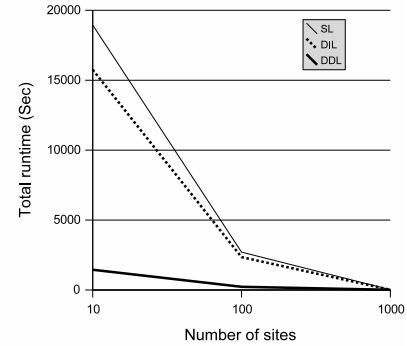
In this section, we examine how the batch partitioning affects the performance. Figure 13 shows the total disk I/Os with respect to the batch-partitioning capacity, i.e. the number of new sub-cells we introduce in a single run. When the batch capacity increases, the total disk I/Os first decrease. That is because, with a larger batch capacity, we can compute more $AD(\cdot)$ and lower bounds at one time. However, when the batch capacity is too large, the performance becomes worse. The reason is that the cells are divided into too fine a granularity. Some disk I/Os are wasted on computing the $AD(\cdot)$ and VCUs for those sub-cells, which could be pruned by using a coarser granularity.

6.5 The Progressiveness

Our progressive algorithm quickly reports a temporary optimal location, and keeps refining the result. In this section we examine how fast the quality of the query result can improve. Since queries are randomly generated, the optimal



(a) The total disk I/Os



(b) The total runtime

Figure 11: Comparison of the three lower bounds.

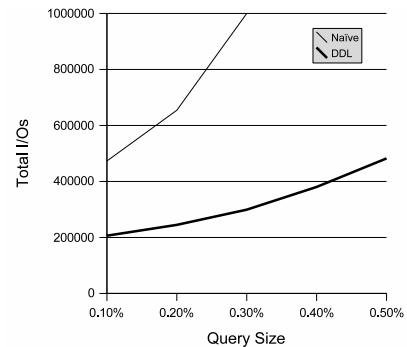


Figure 12: The impact of lower-bound pruning.

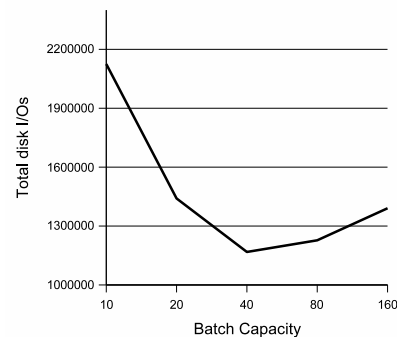


Figure 13: The effect of batch partitioning.

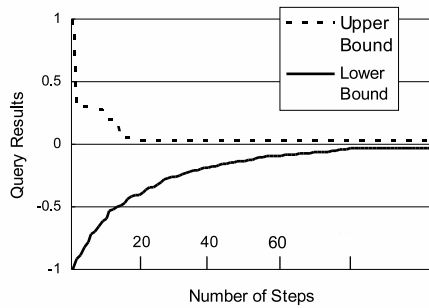


Figure 14: The progressiveness of our algorithm.

locations of different queries may have different $AD(\cdot)$ values. To measure the progressiveness in a unified way, we normalize the result of each query as follows: the final result (i.e. $AD(\cdot)$ of the actual optimal location) is scaled to be 0, the minimal $AD(\cdot)$ of the four corners of the query range is scaled to be 1, and the initial lower bound of the query range is scaled to be -1. After each batch process, we get a new upper bound and lower bound as the intermediate result. They are scaled to real numbers between $[-1,1]$. We run 100 random queries with the default parameters, take the first 100 steps for each run, scale them to the $[-1,1]$ range and report the average. The result is plotted in Figure 14. The (scaled) upper bound approaches the actual result very fast. It will become less than 1% within 20 steps. The (scaled) lower bound, however, approaches the actual result slower. It will become less than 1% in about 80 steps. (Here on average it takes an algorithm 200 steps to find the exact answer.) It tells us that, in many cases, we find a good approximation very fast, but it takes some time to verify whether it is the actual real optimal location.

7. CONCLUSIONS AND FUTURE WORK

This paper proposed and solved the min-dist optimal-location query. Even though there is an infinite number of locations in a query range, we proved that under the L_1 metric we only need to check a finite number of candidates to get an exact answer. We then proposed a progressive algorithm, **MDOL_prog**, that first partitions the query range into some cells, and then recursively partitions each cell into smaller cells. Since any candidate location is the corner of some cell, by partitioning to the finest granularity it is guaranteed that we can find the optimal location. We introduced three lower-bound estimators which enable the pruning of complete cells (and all candidate locations in these cells). The lower-bound estimators provide the progressive nature of the algorithm, as results from earlier runs can be used to prune the search space of later runs. Finally, we proposed the batch-partitioning method. Experimental results revealed that one of the three lower-bound estimators, namely the *data-dependent lower bound*, is clearly better than the other two, and that our progressive algorithm **MDOL_prog** is much more efficient than the naive algorithm.

While this paper solved a real problem and may have strong impact on corporate decision-support systems, it triggers some other interesting problems. What if a franchise

plans to choose $k > 1$ locations together, whose overall impact is optimal? What if one wishes to replace the L_1 metric with L_2 metric or road-network distance? We plan to investigate these problems.

8. ACKNOWLEDGEMENT

Donghui Zhang was partially supported by NSF CAREER Award IIS-0347600. Yufei Tao was partially supported by Grant CityU 1163/04E from the Research Grant Council of the HKSAR government.

9. REFERENCES

- [1] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.
- [2] Y. Du, D. Zhang, and T. Xia. The Optimal-Location Query. In *SSTD*, pages 163–180, 2005.
- [3] L. E. Jackson. The Directional p-Median Problem with Applications to Traffic Quantization and Multiprocessor Scheduling. *Ph.D. thesis, North Carolina State University*. <http://www.csc.ncsu.edu/faculty/rouskas/Ar0ra/Thesis/PHD-Jackson-2003.pdf>, 2003.
- [4] F. Korn and S. Muthukrishnan. Influence Sets Based on Reverse Nearest Neighbor Queries. In *SIGMOD*, pages 201–212, 2000.
- [5] 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.
- [6] N. Megiddo and K. J. Supowit. On the Complexity of Some Common Geometric Location Problems. *SIAM Journal on Computing*, 13(1):182–196, February 1984.
- [7] K. Mouratidis, D. Papadias, and S. Papadimitriou. Medoid Queries in Large Spatial Databases. In *SSTD*, pages 55–72, 2005.
- [8] 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.
- [9] I. Stanoi, M. Riedewald, D. Agrawal, and A. El Abbadi. Discovery of Influence Sets in Frequently Updated Databases. In *VLDB*, pages 99–108, 2001.
- [10] Y. Tao, D. Papadias, and X. Lian. Reverse kNN Search in Arbitrary Dimensionality. In *VLDB*, pages 744–755, 2004.
- [11] Yannis Theodoridis. The R-tree-portal. <http://www.rtreportal.org>, 2003.
- [12] D. Zhang, Y. Du, T. Xia, and Y. Tao. Progressive Computation of The Min-Dist Optimal-Location Query (full version). <http://www.ccs.neu.edu/home/donghui/publications/vldb06full.pdf>, 2006.