A Safe Zone Based Approach for Monitoring Moving Skyline Queries

Muhammad Aamir Cheema[†], Xuemin Lin[‡†*], Wenjie Zhang[†], Ying Zhang[†]
[†]*The University of New South Wales, Australia* [‡]*East China Normal University, China*
{macheema,lxue,zhangw,yingz}@cse.unsw.edu.au

ABSTRACT

Given a set of criterions, an object o dominates another object o' if o is more preferable than o' according to *every* criterion. A skyline query returns every object that is not dominated by any other object. In this paper, we study the problem of continuously monitoring a moving skyline query where one of the criterions is the distance between the objects and the moving query. We propose a *safe zone* based approach to address the challenge of efficiently updating the results as the query moves. A safe zone is the area such that the results of a query remain unchanged as long as the query lies inside this area. Hence, the results are required to be updated only when the query leaves its safe zone. Although the main focus of this paper is to present the techniques for Euclidean distance metric, the proposed techniques are applicable to any metric distance (e.g., Manhattan distance, road network distance). We present several non-trivial optimizations and propose an efficient algorithm for safe zone construction. Our experiments demonstrate that the cost of our safe zone based approach is reasonably close to a lower bound cost and is three orders of magnitude lower than the cost of a naïve algorithm.

Categories and Subject Descriptors

H.2.8 [**Database Applications**]: Spatial databases and GIS

General Terms

Algorithms

1. INTRODUCTION

Due to the exponential increase in the usage of smart phones, availability of inexpensive position locators and cheap network bandwidth, location-based services are becoming increasingly popular. Skyhook reported that the number of location-based applications being developed each month is increasing exponentially. Consequently, in the past few years, significant research attention has been given to continuously monitor various spatial queries such as k nearest neighbors (kNN) queries [27, 18], range queries [27, 4] and reverse k nearest neighbors queries [13, 26].

Each of the above mentioned spatial queries retrieves the objects based on their distances from the query location. However, in many real world applications, distance is not usually the only criterion desired by the users. In this paper, we focus on continuously monitoring the spatial queries that involve multiple criterions (distance being one of the criterions). Since skyline is a natural and popular choice for the applications involving multi-criteria decision making [2, 24, 14, 20, 21, 22, 8], in this paper, we study the problem of continuously monitoring skyline for a moving query where distance between the objects and the query is one of the criterions.

Consider the example of a car driver who is looking for restaurants. He may be interested in the restaurants that are close to his location, have good reputations and are cheap, i.e., distance, rank and price are the three criterions. A traditional continuous kNN query monitors the k closest restaurants regardless of their reputations and food prices. On the other hand, a skyline query returns every restaurant that is not *dominated* by any other restaurant (i.e., for every returned restaurant o, there does not exist any other restaurant that is closer to the query, has a better reputation and is cheaper). Since the distances between the car and the restaurants change as the car moves, the skyline is needed to be updated continuously. In this paper, we present efficient techniques to continuously monitor the moving skyline queries.

Note that some users may want to define additional constraints on skyline objects so that the result size is reduced and it is easier for them to make the final choice. For instance, a user may want to continuously monitor k-closest skyline objects or only the skyline objects that lie within a certain distance range. In Section 5.2, we show that our proposed techniques can be easily adopted to continuously monitor such filtered skyline objects.

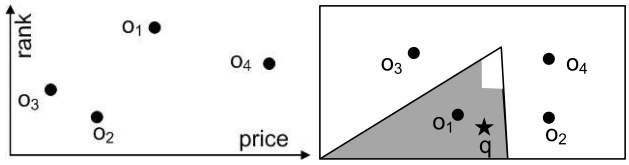
1.1 Solution Highlights

In the past few years, several *safe zone* based approaches [27, 4, 18] have been proposed to monitor various continuous spatial queries. Such algorithms do not only return the current results but also a safe zone which is an area such that the results remain unchanged as long as the query remains inside the safe zone. Hence, the results of the query are not required to be updated unless the query leaves its safe zone. Due to the effectiveness and popularity of the safe zone based approaches, we also propose an efficient safe

*Corresponding author

zone based solution.

Fig. 1 shows an example of four restaurants (o_1 to o_4) and a query q. Fig. 1(a) depicts the rank and average food price of each restaurant and Fig. 1(b) shows the locations of the restaurants and the query q. Assume that a lower rank indicates a better reputation. A restaurant o is a skyline object if there does not exist a restaurant o' that is more preferable than o according to *every* criterion (i.e., price, rank and distance). Note that o_2 and o_3 are the skyline objects because there does not exist a restaurant that has a lower price *and* a lower rank. o_1 is also a skyline object. This is because although o_2 and o_3 have lower ranks and lower prices than o_1, the restaurant o_1 is closer to q than o_2 and o_3 (see Fig. 1(b)). o_4 is not a skyline object because o_2 has a lower rank, a lower price and is closer to q than o_4. Safe zone is the area shown shaded in Fig. 1(b). The results do not change as long as q lies in the safe zone.



(a) Price and rank (b) Location coordinates

Figure 1: Skyline objects are o_1, o_2 and o_3 and safe zone is the shaded area

We formalize the safe zone by introducing a novel concept of *impact region*. Impact region of an object o is the area such that o is one of the skyline objects of a query q if and only if q lies in this area. In Fig. 1(b), the impact region of o_1 is the triangle. The impact region does not only help in computing the safe zone but it can also be used for market analysis and targeted marketing. For instance, a person that is present in the impact region of a restaurant is likely to be influenced by an advertisement because the restaurant is one of its skyline objects. Similarly, demographics of the impact region of a restaurant can be used by the market researchers to analyse its business.

We show that the safe zone can be constructed if the impact region of every object is known. More specifically, we show that the safe zone consists of every point that lies inside the impact region of *every* skyline object and lies outside the impact region of *every* non-skyline object. In Fig. 1, safe zone is the shaded area. After formalizing the safe zone, we present a basic algorithm to construct the safe zone.

We note that the basic algorithm is quite expensive mainly because the computation of the impact region of an object is not cheap as it requires considering all other objects. Based on several non-trivial observations, we show that the safe zone can be constructed accurately even if partially computed impact regions are used. More specifically, we show that the safe zone can be constructed accurately even when only the skyline objects are considered for the impact region construction. This significantly improves the performance of the basic algorithm because the number of skyline objects is usually much smaller than the total number of objects. Furthermore, we present effective pruning rules to prune the objects that do not affect the safe zone construction. These optimizations improve the performance up to three orders of magnitude and enable us to compute the safe zone with a cost quite close to the lower bound cost.

1.2 Contributions

Our proposed approach has the following features.

Efficiency. A safe zone computation algorithm returns the current results of a query as well as a safe zone and guarantees that the results remain valid unless the query leaves the safe zone. Hence, the lower bound cost of computing the safe zone is the cost of computing the skyline objects. IO and CPU cost of our safe zone construction algorithm is close to the IO and CPU cost of BBS [20] which is an IO optimal skyline algorithm, i.e., IO cost of our safe zone construction algorithm is close to the lower bound IO cost. This also implies that while the overhead of computing the safe zone is small as compared to the cost of computing the skyline objects, the benefit is large because the results are not required to be updated as long as the query remains inside the safe zone. This enables our algorithm to monitor the skyline quite efficiently.

Applicability to arbitrary distance metric. Although the focus of this paper is to present the techniques for the case where the distances between objects and query are computed using Euclidean distance metric in 2d space, the core ideas (Section 3 and 4) can be used to efficiently construct the safe zone for arbitrary distance metric (e.g., Manhattan distance in 3d space, network distance in road networks).

Empirically evaluated. For a strict evaluation, we compare our algorithm with a specially designed algorithm called *supreme* algorithm. The supreme algorithm assumes the existence of an oracle and meets the lower bound IO cost for safe zone construction. More specifically, the supreme algorithm computes the skyline objects using BBS [20] which is an IO optimal skyline algorithm. We assume that the oracle computes the safe zone without incurring any IO or CPU cost and returns it to the supreme algorithm. Our extensive experimental study demonstrates that the cost of our algorithm is reasonably close to the cost of the supreme algorithm. Moreover, our algorithm performs three orders of magnitude better than a naïve algorithm.

The rest of the paper is organized as follows. We discuss the related work in section 2. In Section 3, we formally define the problem, formalize the safe zone and present a basic algorithm. Section 4 presents several novel optimizations that significantly improve the basic algorithm and are applicable to arbitrary metric spaces. Section 5 presents our branch and bound algorithm specifically designed for Euclidean space. An extensive experimental study is presented in Section 6. Section 7 concludes the paper.

2. RELATED WORK

2.1 Skyline Queries

Snapshot Queries. A snapshot skyline query retrieves the set of skyline objects only once and the results are not required to be updated. Some of the notable snapshot skyline algorithms include *block-nested loop* (BNL) [2], *divide and conquer* (D&C) [2], *bitmap* [24], *index* [24], *nearest neighbor* (NN) [14] and *branch and bound search* (BBS) [20]. BBS is superior to the other algorithms and is IO optimal, i.e., it does not access any node of R-tree that cannot contain a skyline object.

Continuous Queries. Continuous skyline queries have been studied under various settings, e.g., updating skyline in data streams [16, 17], skyline maintenance due to deletions [25] and skyline monitoring for dynamically changing points [12, 15, 11]. Below, we describe the most closely related work and discuss why these techniques are not suitable to solve the problem studied in this paper.

The work that is most closely related to our work is done by Huang *et. al* [12] who propose a kinetic-based data struc-

ture to update the skyline results. Lee *et. al* [15] also study a similar problem. However, both of these works rely on the assumption that the velocities of the moving points are known. Unfortunately, this assumption does not hold in many real world applications where the points (e.g., cars) frequently change their motion pattern (e.g., speed and direction). Furthermore, the extension of their techniques is non-trivial for the scenarios where velocities are unknown.

Hsueh *et. al* [11] present an algorithm to update the skyline when the data objects change their attribute values. They use a pre-computed *second skyline* to efficiently update the skyline. The proposed technique is specifically designed for the scenario where the update ratio is small and performs well under such scenarios. However, it is not suitable when a large number of objects continuously issue updates. Note that in our problem setting, due to the change in query location, the distance attributes of *all* the objects change. This is equivalent to the scenario where all of the objects issue updates and this makes the pre-computed second skyline useless.

2.2 Voronoi Cell Computation

Our proposed approach requires computing *Voronoi cells* of the objects. Voronoi cell of an object o_1 consists of every point x for which $dist(x, o_1)$ is smaller than the distance of x from any other object in the data set. In other words, o_1 is the nearest neighbor of any point x if and only if x lies inside the Voronoi cell of o_1. In Fig. 2(b), the shaded triangle is the Voronoi cell of o_1. In the past few years, several approaches [6, 27, 23] have been proposed to compute the Voronoi cell of an object. Below, we discuss the techniques proposed in [6, 7] because we use it in our algorithm.

A perpendicular bisector between two points o_1 and o_2 divides the space into two halves. Let the half-space that contains o_2 be denoted as $H_{o_2:o_1}$ (see the white area in Fig. 2(a)). The half-space $H_{o_2:o_1}$ has the property that every point x that lies in $H_{o_2:o_1}$ is closer to o_2 than o_1, i.e., $dist(x, o_2) < dist(x, o_1)$. Note that $H_{o_2:o_1}$ cannot be a part of the Voronoi cell of o_1 because o_1 cannot be the nearest neighbor of any point x in $H_{o_2:o_1}$.

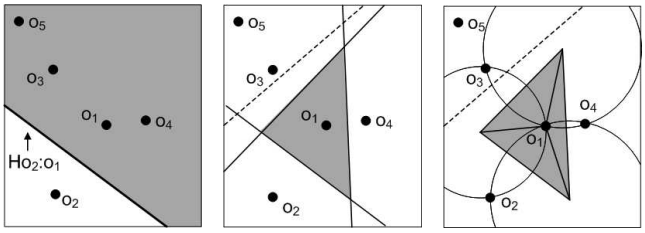


Figure 2: Computing Voronoi cell of o_1

Based on the property of half-space, the Voronoi cell of o_1 can be constructed as follows. Initially, the Voronoi cell is set to the whole data universe (e.g., the rectangle in Fig. 2). Then, each object o_i is considered iteratively, and the Voronoi cell is updated by removing $H_{o_i:o_1}$. For example, in Fig. 2(a), the Voronoi cell is updated to the shaded area when o_2 is considered. Fig. 2(b) shows the Voronoi cell of o_1 (the shaded area) after all the objects have been considered.

Cheema *et. al* [6] present an efficient branch and bound approach to compute the Voronoi cell. They observe that an object o' cannot affect the shape of the Voronoi cell of o if, for *every* vertex v of the current Voronoi cell V, $dist(o, v) < dist(o', v)$. In Fig. 2(c), suppose that the current Voronoi cell is the shaded area. The half-space $H_{o_5:o_1}$ (denoted by broken line) between o_1 and o_5 does not remove any part of the shaded area because, for every vertex v of the Voronoi cell, $dist(o_1, v) < dist(o_5, v)$ (i.e., o_5 does not lie in any of the circles shown in Fig. 2(c)). Hence, Voronoi cell of o_1 can be computed correctly even if o_5 is pruned.

Algorithm 1 Compute Voronoi Cell

Input: a set of objects O, an object $o \in O$
Output: Voronoi cell of o
1: initialize V as the whole data universe
2: insert root of R-tree in a min-heap h
3: **while** h is not empty **do**
4: deheap an entry e
5: **for** each vertex v of V **do**
6: **if** $mindist(v, e) < dist(v, o)$ **then**
7: mark e as valid; break
8: **if** e is valid **then**
9: **if** e is an intermediate node or leaf **then**
10: insert every child c in h with key $mindist(o, c)$
11: **else if** e is an object **then**
12: update V by removing every point x from V for which $dist(e, x) < dist(o, x)$

The authors propose a branch and bound algorithm where the data objects are assumed to be indexed by an R-tree. Algorithm 1 shows the details. A de-heaped entry e can be ignored if, for every vertex v of V, $mindist(v, e) \geq dist(v, o)$. Otherwise, it is considered valid (lines 5 to 7). If the entry is valid and is an intermediate node or a leaf node, its children are inserted in the heap (lines 8 to 10). Otherwise, if the entry e is valid and is a data object, the Voronoi cell V is updated by removing $H_{e:o}$ (line 12). The algorithm stops when the heap becomes empty.

We remark that Algorithm 1 can be modified to compute Voronoi cell for general metric spaces. The lines 5 to 7 are not applicable because, in general metric spaces, Voronoi cell may not be a polygon. However, the entries can be pruned using Lemma 10 presented in [6] which uses the triangle inequality for pruning.

3. GETTING STARTED

3.1 Problem Definition

Let O be a set of objects. In addition to location coordinates, each object has d attributes (dimensions). The i-th attribute value of an object o is denoted as $o[i]$. The distance between a query q and an object o is denoted as $dist(q, o)$ and is considered the (d+1)-th dimension of the object, i.e., $o[d+1] = dist(q, o)$. Hence, each object is considered to have $(d+1)$ dimensions. Since $dist(q, o)$ changes with the change in query location, the distance is called the *dynamic* dimension of o. Other attributes of the objects are not affected by the query movement and are called *static* dimensions of the objects.

Complete Dominance. An object o is completely dominated by another object o' if for *every* dimension $1 \leq i \leq (d+1)$, $o'[i] \leq o[i]$ and for at least one dimension $1 \leq j \leq (d+1)$, $o'[j] < o[j]$. This dominance relationship is called complete dominance because it involves all dimensions (static and dynamic) in contrast to the static dominance relationship (defined in Section 3.2) that considers only the static dimensions of the objects.

Skyline Query. A skyline query returns every object o that is not completely dominated by any other object. Since the value of $(d+1)$-th dimension (i.e., $dist(q, o)$) of every object o changes as the query changes its location, the skyline is needed to be continuously updated. In this paper, we

study the problem of continuously monitoring the skyline of a moving query.

Table 1: Notations

Notation	Definition
$o =_s o'$	o statically equals o' (Section 3.2)
$o \prec_s o'$	o statically dominates o' (Section 3.2)
$o \preceq_s o'$	$o \prec_s o'$ or $o =_s o'$
$A(o)$	Affecting set of o (Definition 1)
$IR(o)$	Impact region of o (Definition 2)
$PA(o)$	Pseudo-affecting set of o (Definition 4)
$PIR(o)$	Pseudo-impact region of o (Definition 5)
Z	Safe zone
$NN(x, S)$	Nearest neighbor of x among a set of objects S

3.2 Formalizing Safe Zone

Throughout this section, we use Fig. 3 to explain the concepts. Fig. 3(a) shows 5 objects according to their static dimensions (price and rank) and Fig. 3(b) shows the same objects according to their location coordinates. We remark that although Fig. 3(b) shows the objects in two dimensional Euclidean space, the proposed ideas are immediately applicable to general metric spaces (e.g., road network distance, Manhattan distance in 3d space). For the ease of presentation, we first introduce some terms and notations.

Static equality. An object o is statically equal to o' if, for every *static* dimension i (i.e., $1 \leq i \leq d$), $o[i] = o'[i]$. We denote the static equality as $o =_s o'$.

Static dominance. An object o is statically dominated by another object o' if o is not statically equal to o' and for every *static* dimension i, $o'[i] \leq o[i]$. We use $o' \prec_s o$ to denote that o' statically dominates o. We use $o' \preceq_s o$ to denote that o' either statically dominates o or is statically equal to o. In Fig. 3(a), $o_2 \preceq_s o_4$ and $o_1 \not\preceq_s o_4$.

For the ease of presentation, we assume that for any two objects o and o', $dist(q, o) \neq dist(q, o')$. We remark that this assumption is made only for the ease of presentation (by avoiding the boundary conditions) and our techniques can be applied even when this assumption does not hold.

Complete dominance revisited. To assist us in explaining our techniques, we define complete dominance using the notations defined above. Specifically, an object o is completely dominated by another object o' if $o' \preceq_s o$ and $dist(q, o') < dist(q, o)$. In other words, o' completely dominates o if o' is at least as good as o on static dimensions and is closer to the query than o. In Fig. 3, o_2 completely dominates o_4 because $o_2 \preceq_s o_4$ (see Fig. 3(a)) and $dist(q, o_2) < dist(q, o_4)$ (see Fig. 3(b)). On the other hand, o_2 does not completely dominate o_1. This is because, although $o_2 \preceq_s o_1$, $dist(q, o_2) \not< dist(q, o_1)$.

Condition for skyline membership. Note that an object o' cannot completely dominate o if $o' \not\preceq_s o$ no matter whether $dist(q, o') < dist(q, o)$ or not. This implies that only the objects that statically dominate or equal o can completely dominate o. Based on this, Lemma 1 defines the condition that an object o must satisfy in order to be a skyline object.

LEMMA 1 : An object o is a skyline object if and only if for every other object o' for which $o' \preceq_s o$, $dist(q, o') > dist(q, o)$.

Proof is straightforward and is omitted. Intuitively, Lemma 1 states that o is a skyline object if o is closer to q than every object o' that is at least as good as o on static dimensions (i.e., $o' \preceq_s o$). Otherwise, o' completely dominates o and o is not a skyline object.

In Fig. 3, o_4 is not a skyline object because there exists an object o_2 such that $o_2 \preceq_s o_4$ and $dist(q, o_2) < dist(q, o_4)$. On the other hand, o_1 is a skyline object because o_1 is closer to q than o_2, o_3 and o_5.

According to the condition specified by Lemma 1, the locations of only the objects that statically dominate or are equal to o are important in deciding whether o is a skyline object or not. The set consisting of such objects is called affecting set of o. Below, we give a formal definition.

DEFINITION 1 : **Affecting set.** Affecting set $A(o)$ of an object o consists of every object $o' \in O$ for which $o' \preceq_s o$.

By definition, the affecting set $A(o)$ of o always includes o. In the example of Fig. 3(a), the affecting set of o_1 is $A(o_1) = \{o_1, o_2, o_3, o_5\}$. Similarly, $A(o_2) = \{o_2\}$, $A(o_3) = \{o_3\}$, $A(o_4) = \{o_2, o_3, o_4\}$ and $A(o_5) = \{o_5\}$.

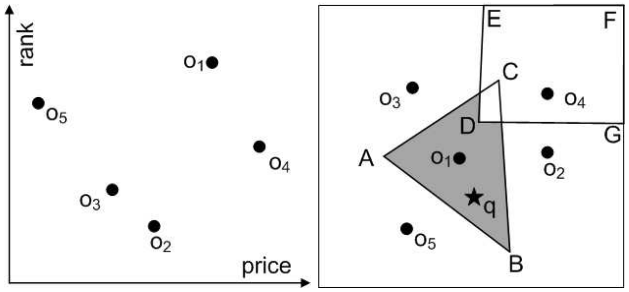
Let x be a point and S be a set of objects. $NN(x, S)$ denotes the nearest neighbor (closest object) of x among the objects in S.

LEMMA 2 : An object o is a skyline object if and only if $NN(x, A(o)) = o$.

The proof is straight forward because, according to Lemma 1, an object o is a skyline object if and only if o is closer to q than every object o' for which $o' \preceq_s o$. In Fig. 3, $NN(q, A(o_1)) = o_1$ and $NN(q, A(o_4)) = o_2$. Hence, o_1 is a skyline object whereas o_4 is not.

Now, we introduce the concept of *impact region*. Impact region $IR(o)$ of an object o is the area such that o is a skyline object of q if and only if q lies inside $IR(o)$. Below, we give a formal definition.

DEFINITION 2 : **Impact region.** The impact region $IR(o)$ of an object o consists of every point x in the space for which $NN(x, A(o)) = o$, i.e., every point x for which o is the closest object in $A(o)$.



(a) Static dimensions (b) Location coordinates

Figure 3: Illustration of safe zone

Assume that we draw a Voronoi diagram on the locations of objects in $A(o)$. Let $Vor(o, A(o))$ be the Voronoi cell in this Voronoi diagram related to the object o. As stated in Section 2.2, a Voronoi cell $Vor(o, A(o))$ has the property that o is the nearest object (among the objects in $A(o)$) of a point x if and only if x lies inside $Vor(o, A(o))$. This implies that the impact region of an object o is its Voronoi cell constructed using the set of objects $A(o)$. We remark that the concept of Voronoi diagram and Voronoi cell is applicable to arbitrary metric spaces (e.g., network Voronoi diagram and weighted distance Voronoi diagram [19]).

EXAMPLE 1 : Recall that Fig. 3(a) shows that $A(o_1) = \{o_1, o_2, o_3, o_5\}$. Voronoi cell of o_1 constructed using these objects is the triangle $\triangle ABC$ (see Fig. 3(b)). Note that o_1 remains the closest object of q among the objects in $A(o_1)$

as long as q remains in $\triangle ABC$. Hence, the impact region of o_1 is $Vor(o_1, A(o_1)) = \triangle ABC$. The impact region of o_4 is the Voronoi cell $Vor(o_4, A(o_4))$ (the polygon $DEFG$ in Fig. 3(b)) where $A(o_4) = \{o_2, o_3, o_4\}$. Note that o_4 becomes the skyline object only when q enters in $DEFG$. ■

Note that any object o for which $A(o) = o$ (i.e., no other object o' exists s.t. $o' \preceq_s o$) is always a skyline object regardless of the location of the query or other objects. In other words, the impact region of such an object is the whole data space. For instance, in Fig. 3, o_2, o_3 and o_5 are always the skyline objects and their impact regions correspond to the whole data space.

Safe Zone. Now, we formalize the safe zone. By the definition of impact region, an object o remains a skyline object as long as q remains inside its impact region. Similarly, an object o' remains a non-skyline object as long as q remains outside its impact region. For example, in Fig. 3(b), o_1 remains a skyline object as long as q remains inside the triangle ABC and o_4 remains a non-skyline object as long as q remains outside the polygon $DEFG$. This implies that the results of the query q remain unchanged as long as q remains inside the impact region of every skyline object and remains outside the impact region of every non-skyline object. Hence, the safe zone can be defined using the impact regions of the objects.

DEFINITION 3 : **Safe Zone.** Let $IR^c(o)$ denote the complement of the impact region of an object o, i.e., the area outside the impact region of o. Let S denote the set of skyline objects of q. The safe zone of the query q is $Z = \cap_{o_i \in S} IR(o_i) \bigcap \cap_{o_j \in O-S} IR^c(o_j)$.

In plain words, the safe zone consists of every point that lies inside the impact region of every skyline object and lies outside the impact region of every non-skyline object.

EXAMPLE 2 : Consider the example of Fig. 3. Note that o_1, o_2, o_3 and o_5 are the skyline objects because their impact regions contain q. The object o_4 is a non-skyline object because its impact region does not contain q. The safe zone is the area shown shaded in Fig. 3(b) and is defined by $IR(o_1) \cap IR(o_2) \cap IR(o_3) \cap IR(o_5) \cap IR^c(o_4)$. As mentioned earlier, the impact regions $IR(o_2)$, $IR(o_3)$ and $IR(o_5)$ correspond to the whole data space. Hence, the safe zone can also be obtained by $Z = IR(o_1) \cap IR^c(o_4) = IR(o_1) - IR(o_4)$. ■

3.3 A Basic Algorithm

A straightforward approach to compute the safe zone is shown in Algorithm 2. The safe zone is initialized as the whole data universe, e.g., in Euclidean space, we initialize the safe zone as a rectangle that covers the whole data space. Since the safe zone is being constructed and is not the final safe zone, we call it *evolving safe zone* and denote it as Z_e. For each object o, we compute its impact region (lines 3 and 4). The Voronoi cell is constructed using Algorithm 1 as we described in Section 2.2. If the object o is a skyline object (i.e., q lies in $IR(o)$) then the evolving safe zone is updated by taking its intersection with the impact region of o (line 6). Otherwise if o is a non-skyline object, the evolving safe zone must be updated by taking its intersection with $IR^c(o)$ (the complement of the impact region of o). Note that $Z_e \cap IR^c(o) = Z_e - IR(o)$ which implies that we can update the safe zone by subtracting $IR(o)$ from it (line 9).

Possibility of a materialized approach. A possible approach to continuously monitor the skyline queries is to materialize the impact regions or to materialize all possible safe zones using the impact regions of the objects. However, these materialized techniques have the following limitations: i) the materialized approach cannot efficiently deal with the data updates, e.g., a deletion or insertion may change the affecting sets of a large number of objects which may invalidate a large number of materialized impact regions; ii) the materialized approach does not work if a user intends to monitor skyline queries on a subset of data (e.g., on restaurants that lie in a constrained region, or on the restaurants that sell Chinese food); iii) spatial indexes such as R-trees are useful for several spatial queries in contrast to the materialized approach that is useful only for the skyline queries. We also remark that a pre-built Voronoi diagram (constructed using all data objects) is not useful in computing the impact regions. This is because the impact region of each object corresponds to a Voronoi cell constructed using a different set of objects, i.e., its affecting set.

Algorithm 2 A Basic Algorithm

Input: q: the query point, O: the set of objects
Output: S: the set of skyline objects, Z: the safe zone
Description:
1: initialize safe zone Z_e as the whole data universe
2: **for** each object $o \in O$ **do**
3: $A(o) \leftarrow$ the set consisting of every $o' \in O$ s.t. $o' \preceq_s o$
4: $IR(o) = Vor(o, A(o))$ /* `Algorithm 1` */
5: **if** $q \in IR(o)$ **then** /* o `is a skyline object` */
6: $Z_e \leftarrow Z_e \cap IR(o)$
7: add o in S
8: **else** /* o `is a non-skyline object` */
9: $Z_e \leftarrow Z_e - IR(o)$
10: **return** set of skyline objects S and safe zone $Z = Z_e$

4. OPTIMIZATIONS

Algorithm 2 has the following two major limitations: **i)** at line 2, the algorithm considers every object regardless of whether its impact region affects the shape of the evolving safe zone or not; **ii)** at line 3, the algorithm computes the affecting set of an object o by considering all the objects in O which requires traversing the whole data set O for each object. In this section, we present optimizations that address several limitations of the basic algorithm including the two major limitations mentioned above.

4.1 Using Pseudo-Impact Regions

First, we address the second limitation discussed above. Let S be the set of skyline objects of the query q. We prove that the safe zone can be correctly computed even if, at line 3 of Algorithm 2, the affecting set $A(o)$ is created using only the skyline objects, i.e., the set consisting of every object $o' \in S$ for which $o' \preceq_s o$. This optimization significantly improves the performance mainly because the size of S is significantly smaller than the size of O. More analytical details of its advantages are presented later in Section 4.1.3.

Before we present the details and proof of correctness of this novel optimization, we define few terms and notations.

DEFINITION 4 : **Pseudo-affecting set.** Let S be the set consisting of all skyline objects. Pseudo-affecting set $PA(o)$ of an object o is a set consisting of o and every object $o' \in S$ for which $o' \preceq_s o$. Note that $PA(o)$ always includes o regardless of whether o is a skyline object or not.

EXAMPLE 3 : Consider the example of Fig. 4(a) where four objects are shown according to their static dimensions (price and rank). Assume that we know that the set of skyline objects is $S = \{o_1, o_2\}$ (the skyline objects are shown as filled circles and non-skyline objects are shown as hollow circles).

While the affecting set of o_4 is $A(o_4) = \{o_1, o_2, o_3, o_4\}$, its pseudo-affecting set is $PA(o_4) = \{o_1, o_2, o_4\}$. For other objects, $A(o_1) = PA(o_1) = \{o_1, o_2\}$, $A(o_2) = PA(o_2) = \{o_2\}$ and $A(o_3) = PA(o_3) = \{o_2, o_3\}$. ■

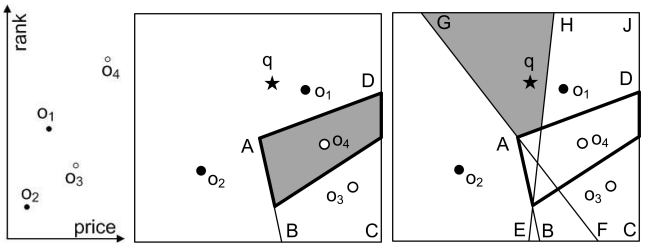


Figure 4: Pseudo Impact Region

DEFINITION 5 : **Pseudo-impact region.** Pseudo-impact region $PIR(o)$ of an object o consists of every point x for which $NN(x, PA(o)) = o$. In other words, $PIR(o)$ corresponds to the Voronoi cell $Vor(o, PA(o))$ of o constructed using the objects in pseudo-affecting set $PA(o)$.

EXAMPLE 4 : Consider the example of Fig. 4(b). The impact region of o_4 is the shaded area and it corresponds to the Voronoi cell of o_4 constructed using $A(o_4) = \{o_1, o_2, o_3, o_4\}$. On the other hand, the pseudo-impact region of o_4 is the polygon $ABCD$ that corresponds to the Voronoi cell of o_4 constructed using $PA(o_4) = \{o_1, o_2, o_4\}$. Note that the pseudo-impact region of an object always contains the impact region of the object, i.e., $IR(o) \subseteq PIR(o)$. As shown in Example 3, the pseudo-affecting sets of other objects are the same as their corresponding affecting sets. Hence, for those objects their impact regions are the same as their pseudo-impact regions. Fig. 4(c) shows the impact regions of all the objects. More specifically, the impact region of o_2 corresponds to the whole data space and $IR(o_2) = PIR(o_2)$. Moreover, $IR(o_3) = PIR(o_3) = HECJ$ and $IR(o_1) = PIR(o_1) = GFCJ$. ■

DEFINITION 6 : **Pseudo-safe zone.** The pseudo-safe zone Z_P is the area that consists of every point x that lies inside the pseudo-impact region of every skyline object and lies outside the pseudo-impact region of every non-skyline object. Formally, $Z_P = \cap_{o_i \in S} PIR(o_i) \bigcap \cap_{o_j \in O-S} PIR^c(o_j)$ where $PIR^c(o_j)$ denotes the complement of $PIR(o_j)$.

Hereafter, the safe zone Z that we defined in previous section is called *original safe zone* if not clear by context.

EXAMPLE 5 : Consider the example of Fig. 4(c). In Example 4, we listed the impact region and pseudo-impact region of every object. The pseudo-safe zone Z_P is the area shown shaded in Fig. 4(c). This is obtained by $PIR(o_1) \cap PIR(o_2) \cap PIR^c(o_3) \cap PIR^c(o_4)$. Note that the original safe zone Z can be obtained as $Z = IR(o_1) \cap IR(o_2) \cap IR^c(o_3) \cap IR^c(o_4)$ and it also corresponds to the shaded area of Fig. 4(c). ■

Note that the original safe zone Z is constructed using the impact regions whereas the pseudo-safe zone Z_P is computed using the pseudo-impact regions. Although the impact region of an object is always smaller than or equal to its pseudo-impact region (i.e., $IR(o) \subseteq PIR(o)$), we prove that the pseudo-safe zone is always equal to the original safe zone, i.e., $Z = Z_P$.

4.1.1 Proof of correctness ($Z = Z_P$)

For the ease of presentation, we break the proof into several lemmas and the main theorem (Theorem 1) is built on these lemmas.

LEMMA 3 : Let o and o' be two objects. If $o' \in A(o)$ (i.e., $o' \preceq_s o$) then $A(o') \subseteq A(o)$.

Proof is straightforward and is omitted.

LEMMA 4 : Let x be a point that lies in Z or Z_P. For every object o, $NN(x, A(o)) = NN(x, PA(o))$.

PROOF. We prove this by contradiction. Assume that $NN(x, A(o)) \neq NN(x, PA(o))$. According to the definition of pseudo-affecting set, $PA(o) \subseteq A(o)$. Hence, the inequality implies that there exists an object $o' \in A(o)$ such that $NN(x, A(o)) = o'$ and $o' \notin PA(o)$. This implies that o' is a non-skyline object (otherwise $o' \in PA(o)$). Below, we prove that $NN(x, A(o))$ cannot be a non-skyline object o' for every x that lies in Z or Z_P.

By definition of original safe zone Z, if x lies in Z then x lies outside the impact region of every non-skyline object o', i.e., $NN(x, A(o')) \neq o'$. According to Lemma 3, $A(o') \subseteq A(o)$ because $o' \in A(o)$. Since $NN(x, A(o')) \neq o'$, $NN(x, A(o)) \neq o'$.

By definition of pseudo-safe zone Z_P, if x lies in Z_P then x lies outside pseudo-impact region of every non-skyline object o', i.e., $NN(x, PA(o')) \neq o'$. Note that $PA(o') \subseteq A(o')$ and $A(o') \subseteq A(o)$. This implies that $NN(x, A(o)) \neq o'$. □

LEMMA 5 : Let x be a point that lies in Z or Z_P. For every object o, x lies in $PIR(o)$ if and only if x lies in $IR(o)$.

PROOF. As per the definition of impact region $IR(o)$, x is a point in $IR(o)$ if and only if $NN(x, A(o)) = o$. According to the definition of pseudo-impact region $PIR(o)$, x is a point in $PIR(o)$ if and only if $NN(x, PA(o)) = o$. Since x is a point in Z or Z_P, Lemma 4 implies that $NN(x, A(o)) = NN(x, PA(o))$. Hence, it immediately follows that x lies in $PIR(o)$ if and only if it lies in $IR(o)$. □

THEOREM 1 : $Z = Z_P$.

PROOF. We prove that every point $x \in Z$ satisfies $x \in Z_P$ and every point $y \in Z_P$ satisfies $y \in Z$. By definition of the original safe zone Z, x lies inside $IR(o_i)$ of every skyline object o_i and lies outside $IR(o_j)$ of every non-skyline object o_j. Since x is a point in Z, according to Lemma 5, x lies inside $PIR(o)$ of any object o if and only if x lies inside $IR(o)$. This implies that x lies inside every $PIR(o_i)$ and lies outside every $PIR(o_j)$. Hence, x lies in the pseudo-safe zone Z_P. Following the similar arguments, it can also be proved that every point $y \in Z_P$ satisfies $y \in Z$. □

4.1.2 Computing pseudo-affecting set

Recall that we compute the pseudo-affecting set $PA(o)$ by selecting every object $o' \in S$ for which $o' \preceq_s o$ where S consists of all skyline objects. However, this requires computing S which may not be known. In this section, we solve this issue as follows: We propose an access order that guarantees that, for each object o, all the objects in $PA(o)$ are accessed before o (Lemma 6). Furthermore, for each accessed object, we show that we can determine whether it is a skyline object or not by using its pseudo-impact region (Lemma 7).

Proposed access order. Assume that $\sum_{i=1}^{d} o[i]$ is called the static score of an object o and is denoted as $o.score$. We access the objects in ascending order of $o.score$. If two objects have the same static score then we prefer the object that is closer to the query q.

LEMMA 6 : The above access order guarantees that, for every accessed object o, there does not exist any skyline object o' that satisfies $o' \preceq_s o$ but has not been accessed before o.

PROOF. For every object o' accessed after o, $o.score \leq o'.score$. It immediately follows that o' cannot statically dominate o (i.e., $o' \not\prec_s o$). Hence, o' may only satisfy $o' =_s o$. However, if $o' =_s o$ then o' cannot be a skyline object because o satisfies $o \preceq_s o'$ and $dist(q, o) < dist(q, o')$. This is because, according to the proposed access order, $dist(q, o) < dist(q, o')$ if $o.score = o'.score$. □

The next issue is how to determine whether an object o is a skyline object or not. In the basic algorithm, o is guaranteed to be a skyline object if and only if q lies in $IR(o)$ (line 5 of Algorithm 2). Although $PIR(o)$ is always at least as large as $IR(o)$, Lemma 7 shows that such guarantee can be provided even if $PIR(o)$ is used instead of $IR(o)$.

LEMMA 7 : An object o is a skyline object if and only if q lies in the pseudo-impact region $PIR(o)$ of o.

PROOF. By definition of the safe zone Z, q lies in the safe zone Z. Since q is a point in the safe zone Z, q lies in $PIR(o)$ if and only if $q \in IR(o)$ (see Lemma 5). Since o is a skyline object if and only if $q \in IR(o)$, it immediately follows that o is a skyline object if and only if $q \in PIR(o)$. □

Remark: We remark that although our proposed access order is similar to the order used in BBS [20], it is not the same. Let $o.score + dist(q, o)$ be the *overall* score of an object. BBS accesses the objects in ascending order of their overall scores. The example below shows that this access order is not useful for our problem, i.e., Lemma 6 does not hold if this access order is used. Consider the example of Fig. 4 and assume that $dist(q, o_1) = 1$ and $dist(q, o_2) = 10$. Assume that the domain range for both price and rank is from 0 to 1. Clearly, o_1 will be accessed before o_2 because the overall score of o_1 is smaller. However, note that o_2 is a skyline object and statically dominates o_1 (see Fig. 4(a)).

Algorithm 3 An Improved Algorithm

```
1: initialize safe zone Ze as the whole data universe
2: S = φ
3: for each object o ∈ O in ascending order of o.score (break
   ties on dist(q, o)) do
4:    PA(o) ← set containing o and every o' ∈ S s.t. o' ⪯s o
5:    PIR(o) = Vor(o, PA(o))
6:    if q ∈ PIR(o) then /* o is a skyline object */
7:       Ze ← Ze ∩ PIR(o)
8:       add o in S
9:    else /* o is a non-skyline object */
10:      Ze ← Ze − PIR(o)
11: return set of skyline objects S and safe zone Z = Ze
```

An improved algorithm. To summarize the ideas presented so far, Algorithm 3 presents an improved approach to construct the safe zone. The set of skyline object S is initially empty (line 2). At line 3, we access the objects in the proposed order. An object o is added to S if q lies inside its pseudo-impact region (line 8). The pseudo-affecting set of each object is constructed using S (line 4). For each object o, the evolving safe zone Z_e is updated by an intersection or difference operation depending on whether o is a skyline object or a non-skyline object (line 7 and 10).

4.1.3 Discussion

Now, we analyse the impact of the optimization used in Algorithm 3. Assuming that the values of objects in one dimension are independent to their values in the other dimensions, it is well known (e.g., see [20]) that the expected number of skyline objects is $O(\log^d N)$ when the total number of objects is N and the total number of criterions is $d + 1$ (dynamic and static dimensions). In other words, the expected size of S is $O(\log^d N)$. This reduces the cost of line 4 from $O(N)$ to $O(\log^d N)$. Furthermore, since the expected size of S is significantly smaller, we may store S in a main memory data structure (e.g., a main-memory R-tree) to speed up the computation of pseudo-affecting set $PA(o)$.

The optimization presented in this section also significantly improves the cost of computing Voronoi cell at line 5. This is because the expected size of pseudo-affecting set is significantly smaller than the size of affecting set as stated in the lemma below.

LEMMA 8 : For any object o, let n be the number of objects in its affecting set $A(o)$. The expected number of objects in its pseudo-affecting set $PA(o)$ is $O(\log^d n)$.

PROOF. Assume that a skyline query is issued on only the objects in $A(o)$ and the returned skyline objects are called sub-skyline objects. Clearly, the expected number of sub-skyline objects is $O(\log^d n)$ assuming that the values are independent to dimensions. Next, we show that an object $o' \in A(o)$ is a skyline object (i.e., $o' \in S$) if and only if o' is a sub-skyline object. This implies that $o' \in PA(o)$ if and only if o' is a sub-skyline object and completes the proof.

Assume that o_x is an object that completely dominates o', i.e., $o_x \preceq_s o'$ and $dist(q, o_x) < dist(q, o')$. Since $o_x \preceq_s o'$, $o_x \in A(o')$ which implies that $o_x \in A(o)$. Hence, each object $o' \in A(o)$ can be completely dominated by only the objects in $A(o)$. Hence, $o' \in A(o)$ is a skyline object if and only if o' is a sub-skyline object. □

We remark that the intersection and difference operations between Z_e and $PIR(o)$ can be conducted cheaply. This is because $PIR(o)$ is a Voronoi cell and the average number of edges of Voronoi cell is at most 6 [19]. Furthermore, our experiments demonstrate that the average number of edges of the safe zone is around 5 for all data sets.

4.2 Pruning Irrelevant Objects

In this section, we present techniques to prune the objects that do not affect the shape of the evolving safe zone. Furthermore, we present techniques to efficiently update the safe zone using the pseudo-impact regions.

LEMMA 9 : An object o does not affect the shape of evolving safe zone Z_e if its pseudo-impact region $PIR(o)$ does not intersect Z_e.

PROOF. By definition of safe zone Z, q lies in Z. Since $Z \subseteq Z_e$, q lies in Z_e. Since $PIR(o)$ does not intersect Z_e, it implies that $PIR(o)$ does not contain q. Hence o is a non-skyline object (see Lemma 7). Since o is a non-skyline object, the updated safe zone $Z_e - PIR(o)$ is the same as Z_e. Hence, o does not change the shape of Z_e. □

Let $mindist(x, Z_e)$ and $maxdist(x, Z_e)$ denote minimum and maximum distance of a point x from Z_e, respectively. The following two lemmas identify the objects that can be pruned.

LEMMA 10 : Let o' be a skyline object such that $o' \preceq_s o$ (i.e., $o' \in PA(o)$). If $maxdist(o', Z_e) < mindist(o, Z_e)$ then o does not affect shape of the safe zone and can be pruned.

PROOF. We prove that $PIR(o)$ does not intersect with Z_e, i.e., $PIR(o)$ does not contain any point of Z_e. Let x be a point in Z_e. Since $maxdist(o', Z_e) < mindist(o, Z_e)$, $dist(o', x) < dist(o, x)$. This implies that for every point $x \in Z_e$, $NN(x, PA(o)) \neq o$. Hence, by the definition of pseudo-impact region, x cannot be a point in $PIR(o)$. □

At line 4 of Algorithm 3, an object o can be pruned if there exist an object $o' \in S$ that satisfies the above condition.

LEMMA 11 : An object o can be pruned if, for every point $x \in Z_e$, there exists a skyline object o' such that $o' \preceq_s o$ and $dist(x, o') < dist(x, o)$.

PROOF. We prove that $PIR(o)$ does not contain any point of Z_e. It can be immediately verified that for every point $x \in Z_e$, $NN(x, PA(o)) \neq o$ because there exists an object $o' \in PA(o)$ that is closer to x. Hence, x cannot be a point in $PIR(o)$ (by definition of $PIR(o)$). □

In Section 4.2.1 (see Advantage 2), we show that this pruning rule can be applied during the computation of pseudo-impact region of an object o (at line 5 of Algorithm 3).

4.2.1 Updating safe zone using $PIR(o)$

In this section, we show that the safe zone can be efficiently updated even if exact pseudo-impact region $PIR(o)$ is not computed. Recall that the evolving safe zone is updated either by $Z_e \leftarrow Z_e \cap PIR(o)$ or $Z_e \leftarrow Z_e - PIR(o)$ depending on whether o is a skyline object or a non-skyline object. Let A, B and C be three sets such that $C = A \cap B$. It can be easily verified that $A \cap B = C$ and $A - B = A - C$. This essentially means that, to update the evolving safe zone Z_e, we are not required to compute the exact $PIR(o)$. Instead, if we correctly compute $C = Z_e \cap PIR(o)$ (i.e., C is the part of $PIR(o)$ that overlaps with Z_e) then we can correctly update Z_e either by $Z_e \leftarrow Z_e \cap PIR(o) \leftarrow C$ or $Z_e \leftarrow Z_e - PIR(o) \leftarrow Z_e - C$.

EXAMPLE 6 : Consider the example of Fig. 5(a) where the evolving safe zone Z_e is the tilted rectangle and $PIR(o_1)$ corresponds to the triangle. Let $C = Z_e \cap PIR(o_1)$ (the shaded area of Fig. 5(a)). Regardless of whether o_1 is a skyline object or a non-skyline object, Z_e can be updated using C. More specifically, if o_1 is a skyline object $Z_e = C$ (the shaded area of Z_e). Otherwise, $Z_e \leftarrow Z_e - C$ (the white area of the rectangle). ■

We utilize the above observation to improve the performance. To construct $C = Z_e \cap PIR(o)$, we call Algorithm 1 with a minor modification. More specifically, at line 1 of Algorithm 1, the Voronoi cell is initialized to the evolving safe zone Z_e instead of initializing it to the whole data universe. Note that Algorithm 1 iteratively removes the part that cannot be a part of the Voronoi cell of o (see line 12). Hence, with this modification, Algorithm 1 returns only the part of Voronoi cell that overlaps with the evolving safe zone Z_e, i.e., it returns $C = Z_e \cap PIR(o)$. This modification has the following advantages.

Advantage 1. Voronoi cell V is initialized to Z_e (a smaller area as compared to the whole data universe). Moreover, at any stage during the execution of Algorithm 1, V is at most as large as it would be without the above modification. Hence, pruning of entries (lines 5 to 7 of Algorithm 1) is significantly more effective.

EXAMPLE 7 : Consider Fig. 5(a) and assume that we call Algorithm 1 and initialize V to the evolving safe zone Z_e

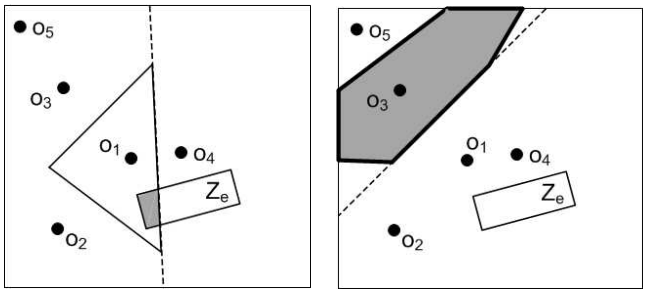


Figure 5: Updating safe zone using $PIR(o)$

(the tilted rectangle). Assume that the algorithm considers the object o_4 and updates V using the half-space between o_1 and o_4 (the broken line). In effect, V is updated to the shaded area of Fig. 5(a). The algorithm can now ignore all remaining objects by applying the pruning condition mentioned in Section 2.2. Hence, we only need to access o_4 to update Z_e. In contrast, computing exact $PIR(o_1)$ (the triangle) requires accessing o_2, o_3 and o_4 (see Section 2.2). ■

Advantage 2. At any stage during the execution of Algorithm 1, if the updated Voronoi cell V (at line 12) becomes empty, we can terminate the algorithm because this essentially means that $Z_e \cap PIR(o) = \phi$. Note that this is an indirect application of the pruning rule implied by Lemma 11.

EXAMPLE 8 : Consider the example of Fig. 5(b) and assume that we have to update the evolving safe zone Z_e using the pseudo-impact region of a non-skyline object o_3. If Algorithm 1 initializes V to the whole data universe then the algorithm accesses all the objects and returns $PIR(o_3)$ (the shaded area of Fig. 5(b)). Note that $PIR(o_3)$ does not affect Z_e. Now assume that Algorithm 1 is called and V is initialized to Z_e. Assume that the algorithm first considers the object o_1 and updates the current Voronoi cell using the half-space between o_1 and o_3 (the broken line). The updated Voronoi cell in this case becomes empty and the algorithm can terminate returning ϕ. Hence, Z_e remains unaffected. Note that the modified version accesses only one object. ■

Advantage 3. Let $C = Z_e \cap PIR(o)$. If o is a skyline object, the evolving safe zone is updated as $Z_e \leftarrow Z_e \cap PIR(o) = C$. Since, the modified version of Algorithm 1 returns C, we can simply update Z_e by C, i.e., the intersection between the polygons is not required.

A final issue requiring attention is that, at line 6 of Algorithm 3, we decide whether an object o is a skyline object or not based on whether q is inside $PIR(o)$ or not. However, the optimization presented in this section returns $C = Z_e \cap PIR(o)$ instead of $PIR(o)$. Lemma 12 shows that we can check whether o is a skyline object or not using C.

LEMMA 12 : An object o is a skyline object if and only if q lies in C where $C = Z_e \cap PIR(o)$.

PROOF. According to Lemma 7, o is a skyline object if and only if q lies in $PIR(o)$. By definition of safe zone Z, q lies in Z. Since $Z \subseteq Z_e$, q lies in Z_e. Hence, o is a skyline if and only if q lies in $Z_e \cap PIR(o)$. □

5. BRANCH AND BOUND ALGORITHM

Based on the ideas presented in the previous section, we present our branch and bound algorithm specially designed for the case where distance between an object and query is

computed using Euclidean distance metric. Although our techniques can be applied on any data structure, due to the simplicity and popularity of R-trees, we assume that the data objects are indexed by a disk-resident R-tree.

Algorithm 4 shows the details of our branch and bound algorithm. Safe zone Z_e is initialized to the whole data universe. A min-heap is initialized with root entry of the R-tree. To access the objects in the order proposed in Section 4.1.2, the heap stores the entries according to the static scores of their lower corners (see line 7). In other words, key of an entry e is $\sum_{i=1}^{d} e^L[i]$ where e^L denotes the lower corner of an entry e (i.e., e^L is a point of e consisting of lowest value on each dimension). Ties are broken by preferring the entry that has smaller minimum distance from q. If ties are still unbreakable, preference is given to the entry that is at a deeper level in the tree (e.g., a data object is preferred to a leaf node). If still not broken, we break the tie arbitrarily.

Algorithm 4 Branch and Bound Algorithm

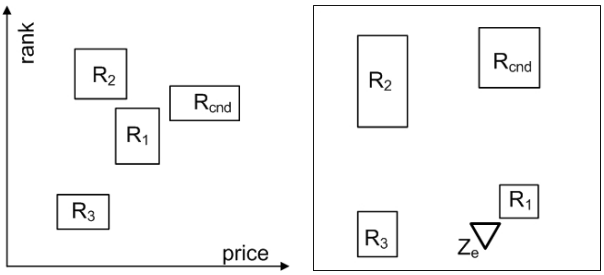
```
1: Initialize Ze as the whole data universe
2: insert root of R-tree in a min-heap H
3: while H is not empty do
4:    deheap an entry e
5:    if e is not pruned then /* using Algorithm 5 */
6:       if e is a leaf or intermediate node then
7:          insert every child c in heap with key set to the static
            score of lower corner of c
8:       else
9:          C ← Ze ∩ PIR(e)
10:         if q ∈ C then /* e is a skyline object */
11:            Ze ← C
12:            add e in S and insert e in main-memory R-tree
13:         else
14:            Ze ← Ze − C
15: return set of skyline objects S and safe zone Z = Ze
```

Entries are de-heaped iteratively. For each de-heaped entry e, we check if e can be pruned or not (line 5) by using our pruning algorithm that will be introduced later in Section 5.1. If e cannot be pruned and is a leaf or intermediate node then its children are inserted in the heap (line 7). Otherwise, if e is a data object then we need to update the evolving safe zone Z_e using the pseudo-impact region of e. We utilize the idea presented in Section 4.2.1 to avoid exact computation of $PIR(e)$ and compute $C = Z_e \cap PIR(e)$ (see line 9). According to Lemma 12, e is a skyline object if q lies in C (line 10). If e is a skyline object then the safe zone Z_e is updated and e is inserted in a main-memory R-tree (line 12). This main-memory R-tree is used in our pruning algorithm. If e is not a skyline object, the safe zone Z_e is updated appropriately (line 14). The algorithm returns skyline objects and the safe zone when the heap becomes empty.

5.1 Pruning Algorithm

Recall that Algorithm 4 indexes the accessed skyline objects in a main-memory R-tree. Hereafter, this R-tree is called a local R-tree. The disk-resident R-tree that indexes all of the data objects is called global R-tree. Let R_{cnd} be an entry of the global R-tree and R_{fil} be an entry of the local R-tree. In this section, we present an approach to prune R_{cnd} (the candidate node) using R_{fil} (the filtering node).

Fig. 6 shows an example where R_{cnd} and three filtering entries R_1, R_2 and R_3 are shown. Each entry R is a four-dimensional rectangle (two static dimensions and two location coordinates). Fig. 6(a) shows the rectangles according to their static dimensions and Fig. 6(b) shows the rectangles according to their location coordinates. For a rectangle R, we use R^L and R^H to denote the lower and upper corners of R, respectively. $R_a^H \preceq_s R_b^L$ denotes that the upper corner of R_a statically dominates or equals the lower corner of R_b. In Fig. 6(a), $R_3^H \preceq_s R_{cnd}^L$ (upper corner of R_3 statically dominates lower corner of R_{cnd}) whereas $R_2^L \not\preceq_s R_{cnd}^L$ (lower corner of R_2 does not statically dominate lower corner of R_{cnd}). $mindist(R, A)$ denotes the minimum distance between a rectangle R and a shape A (a point or polygon) computed using their location coordinates. Maximum distance is defined in a similar way.



(a) Static dimensions (b) Location coordinates

Figure 6: Illustration of pruning

PRUNING RULE 1 : R_{cnd} can be pruned if $R_{fil}^H \preceq_s R_{cnd}^L$ and $maxdist(R_{fil}, Z_e) < mindist(R_{cnd}, Z_e)$.

PROOF. Let o be an object in R_{cnd} and o' be an object in R_{fil}. Since $R_{fil}^H \preceq_s R_{cnd}^L$, it implies that $o' \in PA(o)$ because $o' \preceq_s o$ and o' is skyline object ($o' \in R_{fil}$ which is an entry of local R-tree that indexes only the skyline objects). Moreover $maxdist(o', Z_e) < mindist(o, Z_e)$ because $maxdist(R_{fil}, Z_e) < mindist(R_{cnd}, Z_e)$. Hence, according to Lemma 10, o does not affect the shape of the evolving safe zone Z_e. This holds for every $o \in R_{cnd}$. □

Next, we present a tighter pruning rule based on Lemma 11.

PRUNING RULE 2 : R_{cnd} can be pruned if both of the following conditions hold: 1) $R_{fil}^H \preceq_s R_{cnd}^L$; and 2) for every point $x \in Z_e$, every point $y \in R_{fil}$ and every point $z \in R_{cnd}$, $dist(y, x) < dist(z, x)$.

PROOF. Let o be an object in R_{cnd}. As shown in the proof of Pruning Rule 1, if $R_{fil}^H \preceq_s R_{cnd}^L$ then every object $o' \in R_{fil}$ is a skyline object and satisfies $o' \preceq_s o$. If the second condition holds then, for every $x \in Z_e$, there always exists an object $o' \in R_{fil}$ for which $dist(o', x) < dist(o, x)$. Hence, according to Lemma 11, o can be pruned. □

Given three rectangles A, B and C, Emrich *et. al* [9] present an efficient approach that returns true if and only if, for every point $x \in A$, $y \in B$ and $z \in C$, $dist(y, x) < dist(z, x)$. Since their proposed approach applies only on the rectangles, we approximate the safe zone Z_e by a minimum bounding rectangle to apply Pruning Rule 2.

EXAMPLE 9 : In the example of Fig. 6, R_{cnd} cannot be pruned by R_1 because $R_1^H \not\preceq_s R_{cnd}^L$ (see Fig. 6(a)). Similarly, R_2 cannot prune R_{cnd}. However, R_3 satisfies $R_3^H \preceq_s R_{cnd}^L$. Note that R_3 also satisfies the second condition (please see Fig. 6(b)). Hence, R_{cnd} can be pruned by R_3. ■

LEMMA 13 : None of the objects in R_{fil} can prune R_{cnd} if $R_{fil}^L \not\preceq_s R_{cnd}^L$ or $maxdist(R_{cnd}, Z_e) < mindist(R_{fil}, Z_e)$.

The proof is straight forward because none of the children of R_{fil} satisfies the pruning condition in Pruning Rule 1

and 2. Hence, in this case, the children of R_{fil} can be ignored because they cannot prune R_{cnd}. In the example of Fig. 6(a), $R_2^L \not\preceq_s R_{cnd}^L$ which implies that no child of R_2 can be used for pruning R_{cnd}. Similarly, in Fig. 6(b), assume that R_1 is to be pruned. Note that although $R_3^L \preceq_s R_1^L$ (see Fig. 6(a)), no child of R_3 can prune R_1 because $maxdist(R_1, Z_e) < mindist(R_3, Z_e)$ (see Fig. 6(b)).

Algorithm 5 Pruning Algorithm

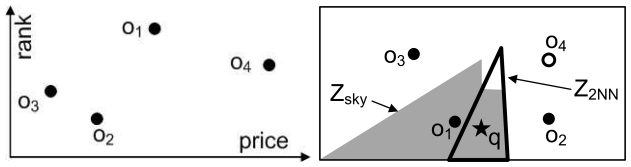
Input: R_{cnd}: the candidate node, A main memory R-tree that indexes only the skyline objects
Output: Return true if R_{cnd} can be pruned.
Description:
1: initialize H with root of the main-memory R-tree
2: **while** H is not empty **do**
3: deheap an entry R_{fil}
4: **if** $maxdist(R_{cnd}, Z_e) < mindist(R_{fil}, Z_e)$ **then**
5: **return false**
6: **if** R_{cnd} can be pruned using Pruning Rule 1 or 2 **then**
7: **return true**
8: **if** $R_{fil}^L \preceq_s R_{cnd}^L$ **then**
9: **if** R_{fil} is a leaf or intermediate node **then**
10: insert its every child c in H with key $mindist(c, Z_e)$
11: **return false**

Algorithm 5 shows the details of our pruning algorithm. A heap H is initialized with root of the main-memory R-tree that indexes only the skyline objects. Each entry e is inserted in heap H with key set to $mindist(e, Z_e)$ (line 10). The entries from heap are de-heaped iteratively. The algorithm returns false if $maxdist(R_{cnd}, Z_e) < mindist(R_{fil}, Z_e)$ for a de-heaped entry R_{fil} (see line 5). This is because, according to Lemma 13, none of R_{fil} and its children can prune R_{cnd}. Moreover, every entry e in the heap H has $mindist(e, Z_e)$ at least equal to $mindist(R_{fil}, Z_e)$ (according to order of entries in heap). Hence, no such entry e or its children can prune R_{cnd}.

If R_{cnd} can be pruned by applying Pruning Rule 1 or 2, then the algorithm returns true (line 7). Otherwise, the children of R_{fil} are inserted in heap H only if $R_{fil}^L \preceq_s R_{cnd}^L$ (see Lemma 13). The algorithm returns false if the heap becomes empty.

5.2 Monitoring Filtered Skyline Objects

In this section, we show that our techniques can be easily adopted to continuously monitor the skyline objects that satisfy certain conditions. Note that the conditions defined on static dimensions are trivial to handle. Once the skyline S and the safe zone is computed, the objects can be filtered by applying the conditions. Since the set of skyline objects does not change as long as the query is inside the safe zone, the filtering remains valid unless the query leaves the safe zone. Next, we present the techniques to handle the conditions defined on the dynamic dimension, i.e., distance.



(a) Price and rank (b) Location coordinates

Figure 7: 2-closest skyline objects are o_1 and o_2

Assume that a user wants to continuously monitor k-closest skyline objects. First, the skyline S and the safe zone are computed. To avoid ambiguity, we call this safe zone Z_{sky} because the set of *skyline* objects remain valid as long as q remains inside Z_{sky}. Fig. 7 reproduces the example we presented in Section 1. Z_{sky} is the shaded area and the set of skyline objects is $S = \{o_1, o_2, o_3\}$.

We consider the set of skyline objects S and compute the k-nearest neighbors (kNN) of q and the order-k Voronoi cell of S that contains q. This order-k Voronoi cell is denoted as Z_{kNN} because the k-nearest neighbors (kNN) of q among S remain the same as long as q remains inside Z_{kNN} [27, 10]. In Fig. 7(b), order-2 Voronoi cell (computed using S) that contains q is the triangle shown using thick lines. The 2-NNs of q among S are o_1 and o_2 and the 2-NNs remain the same as long as q is inside Z_{2NN}. Note that the results of k-closest skyline remain valid as long as q is inside both Z_{sky} and Z_{kNN}. If q leaves Z_{sky} then the set of skyline objects S changes. Hence, Z_{sky} and Z_{kNN} are recomputed. Otherwise, if q is inside Z_{sky} but leaves Z_{kNN} then only a new Z_{kNN} is to be computed.

Note that the above framework can also be used to monitor the skyline objects that lie within a given distance range r from the user. The only difference is that we need to compute Z_{range} instead of Z_{kNN} where Z_{range} is the safe zone for a range query issued on the set of objects S. The techniques presented in [10] and [5] can be immediately applied to compute Z_{kNN} and Z_{range}, respectively.

6. EXPERIMENTS

6.1 Experimental Setup

A naïve approach to monitor moving skyline queries is to compute the safe zone as we described in the basic algorithm (Algorithm 2). Another naïve approach is to call existing algorithms such as BBS [20] to re-compute the skyline whenever the results are to be updated (e.g., after every t time units). However, our experiments demonstrated that both of these naïve approaches perform quite poorly (e.g., both algorithms are almost three orders of magnitude worse than our algorithm).

For a strict evaluation of our algorithm, we specially design the *supreme* algorithm that assumes the existence of an oracle and meets the lower bound IO cost.

Supreme Algorithm. We assume that there exists an oracle that computes the safe zone without incurring any IO or CPU cost. The supreme algorithm uses BBS [20] to compute the skyline objects and assumes that the oracle returns it the safe zone. The results are updated by calling BBS again only when the query leaves the safe zone. Note that the cost of computing the skyline objects is the lower bound cost for every algorithm that computes the safe zone. Since BBS has been shown to be IO optimal [20] for skyline queries, the supreme algorithm meets the lower bound IO cost for the safe zone computation. Our experimental results demonstrate that the performance of our algorithm is reasonably close to the supreme algorithm.

Table 2: System Parameters

Parameter	Range
Number of objects (×1000)	50, 75, **100**, 125, 150
Dimensionality of R-tree	3, **4**, 5, 6
Speed of queries in km/hr	40, 60, **80**, 100, 120
Distribution on static dimensions	unif, **norm**, corr, anti

Our evaluation framework is similar to the framework used in existing safe zone based techniques [4, 27, 18] for continuous spatial queries. Table 2 shows the parameters we use in our experiments and the default values are shown in bold. The objects are indexed by a disk-resident R-tree with node size set to 4096 bytes. The dimensionality of

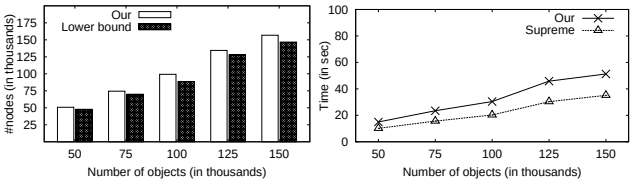
the objects vary from 3 to 6 (including two location coordinates). We generate different data sets each following a different distribution on static dimensions, i.e., Uniform (unif for short), Normal (norm), Correlated (corr) and Anti-correlated (anti) [2]. The location coordinates of the objects are extracted from a real dataset [1] that contains 175,813 points of interest (POIs) in North America and corresponds to a data universe of 5000Km×5000Km. To run the experiments for varying number of objects, we randomly choose the required number of POIs from the real data set. One hundred queries are generated using the popular Brinkhoff data generator [3] that simulates cars (queries) moving on the road network of North America. *The results of each query are monitored for* 5 *minutes and the experiments report the total cost of monitoring all queries for* 5 *minutes.*

6.2 Performance Evaluation

As we mentioned earlier, the naïve algorithms perform quite poorly. Therefore, for a better illustration of the comparison, we only show the results for our algorithm and the supreme algorithm. Nevertheless, towards the end of this section, we present Fig. 13 that not only compares our algorithm with a naïve algorithm but also demonstrates the effectiveness of our proposed optimizations.

6.2.1 Effect of data cardinality

Fig. 8 studies the effect of data cardinality on both of the algorithms. Our algorithm incurs more IOs (the number of accessed R-tree nodes) as compared to the supreme algorithm because our algorithm also needs to consider the non-skyline objects to construct the safe zone in contrast to the supreme algorithm that only computes the skyline objects. The CPU cost of our algorithm is higher mainly because it not only processes non-skyline objects but also computes the pseudo-impact regions to construct the safe zone. Nevertheless, the cost of our algorithm is reasonably close to the cost of supreme algorithm which shows the effectiveness of our proposed optimizations.



(a) Node accesses (b) CPU time

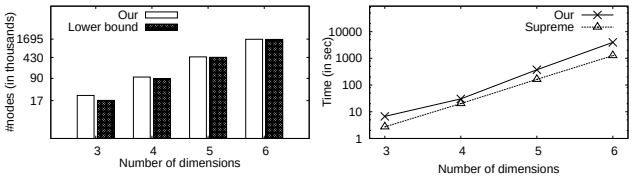
Figure 8: Effect of data cardinality

6.2.2 Effect of dimensionality

In Fig. 9, we change the number of static dimensions from 1 to 4 (the location coordinates are two dimensional). It is well known [20, 2] that the cost of skyline computation algorithms significantly increases with the increase in dimensionality. The same can be observed in Fig. 9. However, the cost of our algorithm is close to the cost of the supreme algorithm which demonstrates that the cost of our algorithm increases mainly because the cost of skyline computation increases (i.e., the safe zone construction does not add a large overhead).

6.2.3 Effect of data distribution

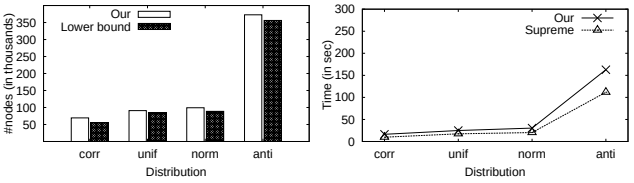
In Fig. 10 we study the effect of data distribution. The distribution of location coordinates does not significantly affect the cost of the algorithms. Therefore, we only present



(a) Node accesses (b) CPU time

Figure 9: Effect of dimensionality

the results for the distributions of values on static dimensions. More specifically, Fig. 10 shows the effect of correlated (shown as corr), uniform (unif), normal (norm) and anti-correlated (anti) distributions. In accordance with the results reported in existing work [20, 2], the skyline algorithms perform best for correlated distribution and the worst for anti-correlated distribution. Note that the cost of our algorithm remains reasonably close to the cost of the supreme algorithm for all data distributions.

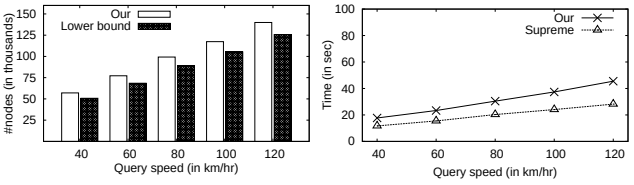


(a) Node accesses (b) CPU time

Figure 10: Effect of distribution

6.2.4 Effect of query speed

In Fig. 11, we run the experiments where the average speed of queries varies from 40 km/hr to 120 km/hr. The cost increases with the increase in query speed because the queries leave their respective safe zones more often and the safe zones are required to be recomputed more frequently. Note that IO and CPU cost of our algorithm is close to the cost of supreme algorithm.



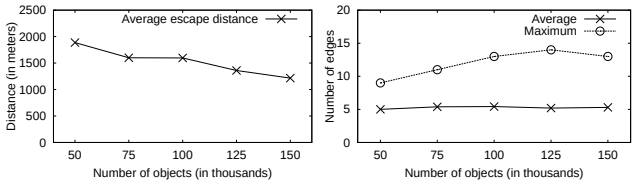
(a) Node accesses (b) CPU time

Figure 11: Effect of query speed

6.2.5 Effectiveness of the safe zone

Note that a safe zone based approach is not effective if a query leaves its safe zone too frequently. Hence, the average distance a query travels before it leaves the safe zone is an important measure to verify the effectiveness of the safe zone. In Fig. 12(a), we show the average escapee distance which is the average distance the queries travel before leaving their respective safe zones. Fig. 12(a) shows that the average distance varies from 1300 meters to 1900 meters. The average escape distance decreases with the increase in data cardinality because the safe zone shrinks.

If the safe zone is a very complex shape then the clients, which usually have low computational power, may not be able to efficiently check whether they lie inside the safe zone or not. Hence, the shape of the safe zone is also an important measure to evaluate its effectiveness. The safe zone in our case is always a polygon and the cost of checking whether a



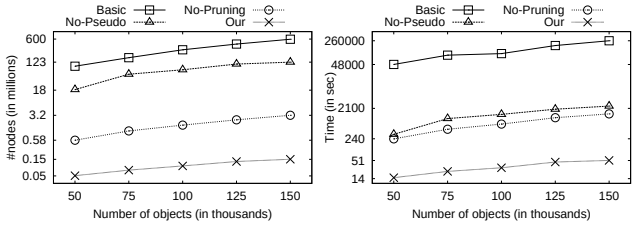
(a) Escape distance (b) Edges of safe zone

Figure 12: Effectiveness of safe zone

client lies inside the safe zone or not is linear to the number of edges of the polygon. Fig. 12(b) shows that the average number of edges of the safe zone is around 5 whereas the maximum number of edges for any safe zone is 14. We conducted the same experiments for other settings (e.g., varying distribution) and observed that the average number of edges is always between 5 to 6.

6.2.6 *Effectiveness of proposed optimizations*

In Fig. 13, we evaluate the effectiveness of the optimizations we presented in Section 4. More specifically, *Basic* is the basic algorithm (Algorithm 2). *No-Pseudo* is the same as Algorithm 4 except that it computes the safe zone by using exact impact regions instead of using pseudo-impact regions. However, *No-Pseudo* algorithm uses our proposed pruning rules. In contrast, *No-Pruning* algorithm uses the concept of pseudo-safe zone but does not employ the pruning rules used in Algorithm 4.



(a) Node accesses (b) CPU time

Figure 13: Effectiveness of proposed optimizations

Fig. 13 shows that our algorithm is several orders of magnitude better than the basic algorithm (note the log scale). The IO cost of *No-Pseudo* is quite high because it does not use the main-memory R-tree (i.e., for each accessed object, all other objects have to be considered). Although the IO cost of *No-Pruning* is lower than the cost of *No-Pseudo*, it is 20 to 30 times higher than the IO cost of our algorithm. This shows the effectiveness of our pruning rules.

7. CONCLUSIONS

We are the first to present a safe zone based approach to continuously monitor skyline for queries moving in arbitrary fashion. We propose efficient safe zone construction techniques that can be applied on arbitrary metric spaces. Our experiments demonstrate that the cost of our proposed algorithm is close to the lower bound cost and is more than three orders of magnitudes lower than a naïve algorithm.

Acknowledgments. Muhammad Aamir Cheema is supported by ARC DE130101002 and ARC DP130103405. The research of Xuemin Lin is supported by NSFC61232006, NSFC61021004, ARC DP110102937 and ARC DP120104168. Wenjie Zhang is supported by ARC DP120104168 and ARC DE120102144. The research of Ying Zhang is supported by ARC DP130103245 and ARC DP110104880.

8. REFERENCES

[1] `http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm`.
[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, 2001.
[3] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
[4] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In *ICDE*, 2010.
[5] M. A. Cheema, L. Brankovic, X. Lin, W. Zhang, and W. Wang. Continuous monitoring of distance-based range queries. *IEEE Trans. Knowl. Data Eng.*, 23(8):1182–1199, 2011.
[6] M. A. Cheema, X. Lin, W. Zhang, and Y. Zhang. Influence zone: Efficiently processing reverse k nearest neighbors queries. In *ICDE*, pages 577–588, 2011.
[7] M. A. Cheema, W. Zhang, X. Lin, and Y. Zhang. Efficiently processing snapshot and continuous reverse k nearest neighbors queries. *VLDB J.*, 21(5):703–728, 2012.
[8] K. Deng, X. Zhou, and H. T. Shen. Multi-source skyline query processing in road networks. In *ICDE*, 2007.
[9] T. Emrich, H.-P. Kriegel, P. Kröger, M. Renz, and A. Züfle. Boosting spatial pruning: on optimal pruning of mbrs. In *SIGMOD*, 2010.
[10] M. Hasan, M. A. Cheema, X. Lin, and Y. Zhang. Efficient construction of safe regions for moving knn queries over dynamic datasets. In *SSTD*, 2009.
[11] Y. Hsueh, R. Zimmermann, and W. Ku. Efficient updates for continuous skyline computations. In *DEXA*, 2008.
[12] Z. Huang, H. Lu, B. C. Ooi, and A. K. H. Tung. Continuous skyline queries for moving objects. *IEEE TKDE*, 18(12):1645–1658, 2006.
[13] J. M. Kang, M. F. Mokbel, S. Shekhar, T. Xia, and D. Zhang. Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In *ICDE*, 2007.
[14] D. Kossmann, F. Ramsak, and S. Rost. Shooting stars in the sky: An online algorithm for skyline queries. In *VLDB*, pages 275–286, 2002.
[15] M.-W. Lee and S. won Hwang. Continuous skylining on volatile moving data. In *ICDE*, pages 1568–1575, 2009.
[16] X. Lin, Y. Yuan, W. Wang, and H. Lu. Stabbing the sky: Efficient skyline computation over sliding windows. In *ICDE*, pages 502–513, 2005.
[17] H. Lu, Y. Zhou, and J. Haustad. Continuous skyline monitoring over distributed data streams. In *SSDBM*, 2010.
[18] S. Nutanong, R. Zhang, E. Tanin, and L. Kulik. The v*-diagram: a query-dependent approach to moving knn queries. *PVLDB*, 2008.
[19] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu. *Spatial Tessellations: Concepts and Applications of Voronoi Diagrams*. Wiley, 1999.
[20] D. Papadias, Y. Tao, G. Fu, and B. Seeger. An optimal and progressive algorithm for skyline queries. In *SIGMOD*, 2003.
[21] D. Sacharidis, P. Bouros, and T. K. Sellis. Caching dynamic skyline queries. In *SSDBM*, pages 455–472, 2008.
[22] M. Sharifzadeh and C. Shahabi. The spatial skyline queries. In *VLDB*, pages 751–762, 2006.
[23] I. Stanoi, M. Riedewald, D. Agrawal, and A. E. Abbadi. Discovery of influence sets in frequently updated databases. In *VLDB*, 2001.
[24] K.-L. Tan, P.-K. Eng, and B. C. Ooi. Efficient progressive skyline computation. In *VLDB*, pages 301–310, 2001.
[25] P. Wu, D. Agrawal, Ö. Egecioglu, and A. E. Abbadi. Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In *ICDE*, 2007.
[26] T. Xia and D. Zhang. Continuous reverse nearest neighbor monitoring. In *ICDE*, page 77, 2006.
[27] J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based spatial queries. In *SIGMOD*, 2003.