Noname manuscript No.
(will be inserted by the editor)

Continuous Monitoring of Moving Skyline and Top-k Queries

Arif Hidayat, Muhammad Aamir Cheema, Xuemin Lin, Wenjie Zhang · Ying Zhang

Received: date / Accepted: date

Abstract Given a set of criteria, 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. A top-k query returns k most preferred objects according to a given scoring function. In this paper, we study the problem of continuously monitoring moving skyline queries and moving top-k queries where one of the criteria is the distance between the objects and the moving query. We propose *safe zone* based techniques 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. We present several non-trivial optimizations and propose an efficient algorithm for safe zone construction for both the skyline queries and top-k queries. Our techniques for the moving top-k queries are generic in the sense that these are immediately applicable to any top-k query as long as its scoring function is monotonic. Furthermore, we show that the proposed techniques can also be extended to monitor various other queries for different distance metrics. Our experiments demonstrate that the cost of our techniques is reasonably close to a lower bound cost and is several orders of magnitude lower than the cost of a naïve algorithm.

Arif Hidayat
Brawijaya University, Indonesia
Monash University, Australia
E-mail: arifhidayat@ub.ac.id, arif.hidayat@monash.edu

Muhammad Aamir Cheema
Monash University, Australia
E-mail: aamir.cheema@monash.edu

Xuemin Lin and Wenjie Zhang
University of New South Wales, Australia
E-mail: {lxue,zhangw}@cse.unsw.edu.au

Ying Zhang
University of Technology Sydney, Australia
E-mail: ying.zhang@uts.edu.au

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, spatial queries have received huge research attention in the past couple of decades. Our focus in this paper is on continuously monitoring moving queries that require continuously updating the query results as the query moves in a two dimensional space. Similar to almost all existing techniques [48,31,11,21,45,10], we use a timestamp-based model where the results are to be updated at each timestamp. The length of the timestamp can be appropriately set depending on the application. For example, a taxi driver may want to get the updated results every minute (in which case the timestamp length may be set to one minute). On the other hand, an intelligent virtual assistant may want to retrieve the result updates every second, e.g., to compute the results of other queries such as loyalty-based queries [36] or to alert the user for only the important updates. Critical applications may want to monitor the results at a much higher frequency (e.g., every millisecond). For example, the alert system in a fighter jet may require continuously monitoring other nearby objects at a very high frequency to detect potentially dangerous situations and raise alarm.

Significant research attention has been given to continuously monitor spatial queries such as k nearest neighbors (kNN) queries [48,31], range queries [48,11] and reverse k nearest neighbors queries [21,45]. Each of these 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 criteria (distance being one of the criteria). We study the problem of continuously monitoring skyline query and top-k query where distance between the objects and the moving query is one of the criteria.

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 criteria. A traditional continuous kNN query monitors the k closest restaurants regardless of their reputations and food prices. In contrast, skyline and top-k queries consider the restaurants' reputations and their prices as well as their distance to the query point. Specifically, 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). Moreover, a top-k query returns k restaurants that have best scores defined according to a user defined monotonic scoring function that involves those three criteria (distance, reputation and price). Since the distances between the car and the restaurants change as the car moves, the skyline and the top-k objects are needed to be updated continuously. In this paper, we present efficient techniques to continuously monitor the moving skyline and the moving top-k queries.

In the past few years, several *safe zone* based approaches have been proposed to monitor various continuous spatial queries (see [35] for a nice survey). 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. The safe zone provides a guarantee to the human users that the results are stable as long as they are in the safe zone (thus providing the safe zone ensures them that they are not missing out on any interesting results as long as they are inside it regardless of the update rate they choose). Due to the effectiveness and popularity of the safe zone based approaches, we also propose efficient safe zone based solutions. Next, we present our contributions for both moving skyline and moving top-k queries.

1.1 Moving Skyline Queries

For the moving skyline queries, we make the following contributions in this paper.

- We design a novel safe zone based algorithm to continuously monitor moving skyline queries. The 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. Note that the lower bound cost of any algorithm to compute 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 [33] 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.
- 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 3.2 to 3.4) 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).
- For a strict evaluation, we compare our algorithm with a specially designed imaginary 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 [33] 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.

1.2 Moving Top-k Queries

We make the following contributions for the moving top-k queries.

- Multicriteria decision making problems are widely studied and, depending on the application needs, different monotonic scoring functions are used such as weighted sum [18,15], weighted product [4,29], weighted distance [43,5], and weighted aggregate sum product assessment (WASPAS) [47] etc. There is no single scoring function that is superior to all other types of scoring functions for all kinds of problems [3]. Since different scoring functions have different characteristics and properties, a decision maker may prefer one scoring function over the other depending on the applications. Thus, there is a need to design techniques that are applicable to a wide variety of scoring functions. To the best of our knowledge, we are the first to present techniques for continuously monitoring top-k queries for arbitrary monotonic scoring functions. We develop generic non-trivial pruning techniques and novel construction algorithm to efficiently compute the safe zone for moving top-k queries.
- We show that our proposed techniques can be generalized to continuously monitor a variety of other queries for multidimensional space and for other distance metrics such as Manhattan distance or network distance in road networks.
- We conduct an extensive experimental study on real data sets for weighted sum, weighted product and weighted distance scoring functions comparing our algorithm against a naïve approach as well as a specially designed imaginary algorithm called *supreme*. The supreme algorithm assumes the existence of an oracle to compute the safe zone without incurring any IO or CPU cost. The results demonstrated that our algorithm is several orders of magnitude faster than the naïve algorithm and its cost is reasonably close to that of the supreme algorithm.

The rest of the paper is organized as follows. We discuss the related work in section 2. In Section 3, we present our algorithm to continuously monitor skyline queries. Section 4 presents our generic algorithm for continuous top-k queries. An extensive experimental study is presented in Section 5. Section 6 concludes the paper.

2 Related Work

2.1 Skyline 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) [8], *divide and conquer* (D&C) [8], *bitmap* [40], *index* [40], *nearest neighbor* (NN) [22] and *branch and bound search* (BBS) [33]. 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. Liue *et. al* [27] propose Skyline diagram which is inspired by Voronoi diagram and helps computing different variants of skyline queries.

Continuous skyline queries have been studied under various settings, e.g., updating skyline in data streams [26, 28], skyline maintenance due to deletions [44] and skyline monitoring for dynamically changing points [19,24, 17]. Below, we discuss the most relevant existing studies on continuous skyline queries.

Shi *et. al* [39] study skyline queries for the case where the objects are continuously moving in the road network and the query is static. Tang *et. al* [41] also study skyline queries over moving objects on dynamic road networks where the edge weights may also change. Chen *et. al* [14] study continuous skyline queries where the dominance is defined using distance and the textual similarity. Papapetrou *et. al* [34] study continuous skyline queries on data streams in a distributed environment. Zheng *et. al* [49] study continuous range skyline queries that aim to continuously find the objects that are not dominated by any other object *and* are within a given distance from the moving query's location. Note that the results of range skyline query may be empty when all objects within the given range are dominated by some other object. Fu *et. al* [16] study range-based skyline queries in road networks that, given a query range, return skyline points for each query location within the range.

Several existing studies [17,23,25] focus on continuous skyline queries where the data objects change their attribute values. These techniques are designed to update the skyline assuming that the number of objects that change their attributes is small and are 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 these techniques unattractive for the problem studied in this paper.

The work that is most closely related to our work is done by Huang *et. al* [19] who propose a kinetic-based data structure to update the skyline results. Lee *et. al* [24] 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.

2.2 Top-k queries

Ihab *et. al* [20] provide an extensive survey on snapshot top-k queries. Below, we briefly discuss some of the most related work on continuous top-k queries.

Continuous top-k queries [30,7,38,37] on data streams has received significant attention. These techniques focus on continuously updating the top-k objects in data streams using a sliding window model. However, the continuous monitoring of queries on data streams is an inherently different problem to that of monitoring moving queries. To the best of our knowledge, the first work on moving top-k query was presented in [43]. However, they study top-k spatial keyword queries. The authors use weighted distance as the scoring function and utilize weighted Voronoi cells to construct the safe zone for each query. They proposed Incremental Border Distance algorithm that prunes objects which cannot contribute to the safe zone.

Huang *et. al* [18] also proposed a safe zone based approach to monitor the result of moving top-k spatial keyword queries focusing on the weighted sum scoring function. One major difference between the above mentioned existing studies and our work is that the existing studies focus on spatial keyword queries where both the distance and the keyword similarity are query dependent. In contrast, in our problem settings, only the distance is query dependent and other attributes (e.g., price, ranking) do not depend on the query. Furthermore, our approach is more generic in that it works on any number of criteria in contrast to the existing studies that are designed for only spatial and keyword similarities. Lastly and most importantly, our proposed techniques are generic and can be immediately applied to all monotonic scoring functions in contrast to the above mentioned studies that focus only on a given type of scoring function.

3 Continuous monitoring of skyline queries

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.

3.2 Formalizing Safe Zone

Throughout this section, we use Figure 1 to explain the concepts. Figure 1(a) shows 5 objects according to their static dimensions (price and rank) and Figure 1(b) shows the same objects according to their location coordinates. We remark that although Figure 1(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 Figure 1(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

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

$o' \preceq_s o$ 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 Figure 1, o_2 completely dominates o_4 because $o_2 \preceq_s o_4$ (see Figure 1(a)) and $dist(q, o_2) < dist(q, o_4)$ (see Figure 1(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 Figure 1, 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 the three objects that statically dominate it (i.e., 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 Figure 1(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 $A(o_5) = \{o_5\}$.

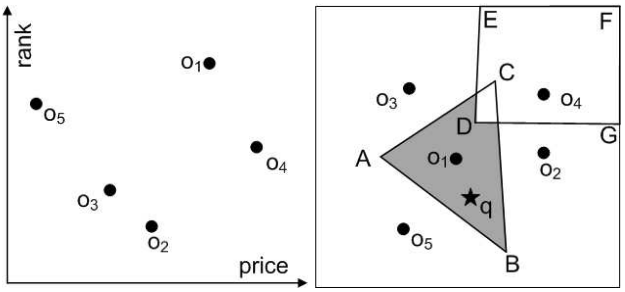
Let x be a point and S be a set of objects. $NN(x, S)$ denotes the nearest neighbor (closest object) of x among the objects in S.

Lemma 2 *An object o is a skyline object if and only if $NN(q, A(o)) = o$.*

The proof is straightforward 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 Figure 1, $NN(q, A(o_1)) = o_1$ 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

Fig. 1 Illustration of safe zone

Assume that we draw a Voronoi diagram [32] 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. 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 [32]).

Example 1 Recall that Figure 1(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 Figure 1(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 Figure 1(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 Figure 1, 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 Figure 1(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 Figure 1. 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 Figure 1(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 1. 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 in [12]. 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).

Algorithm 1 A Basic Algorithm

Input: a set of objects O, the query point q
Output: the set of skyline objects S, the safe zone Z

```
1: initialize safe zone Ze as the whole data universe
2: for each object o ∈ O do
3:     A(o) ← the set consisting of every o' ∈ O s.t. o' ⪯s o
4:     IR(o) = Vor(o, A(o))   # Algorithm 1 in [12]
5:     if q ∈ IR(o) then   # o is a skyline object
6:         Ze ← Ze ∩ IR(o)
7:         add o in S
8:     else   # o is a non-skyline object
9:         Ze ← Ze − IR(o)
10: return set of skyline objects S and safe zone Z = Ze
```

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.

3.4 Optimizations

Algorithm 1 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.

3.4.1 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 1, 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 3.4.2.

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 Figure 2(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 $A(o_3) = PA(o_3) = \{o_2, o_3\}$.

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 Figure 2(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. Figure 2(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 $IR(o_1) = PIR(o_1) = GFCJ$.

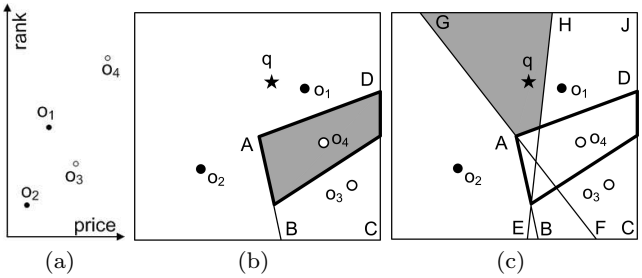


Fig. 2 Pseudo Impact Region

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 Figure 2(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 Figure 2(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 Figure 2(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., $IR(o) \subseteq PIR(o)$), it can be proved that the pseudo-safe zone is always equal to the original safe zone, i.e., $Z = Z_P$. Due to the space limitations, we omit the proof but the interested readers are referred to the earlier version [12] of this paper for the proof.

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 3). 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 4).

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 3 *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 1). Although $PIR(o)$ is always at least as large as $IR(o)$, Lemma 4 shows that such guarantee can be provided even if $PIR(o)$ is used instead of $IR(o)$.

Lemma 4 *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 $q \in IR(o)$ [12]. 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 [33], 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 3 does not hold if this access order is used. Consider the example of Figure 2 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 Figure 2(a)).

Algorithm 2 An Improved Algorithm

```
1: initialize safe zone Z_e 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:       Z_e ← Z_e ∩ PIR(o)
8:       add o in S
9:    else  # o is a non-skyline object
10:      Z_e ← Z_e − PIR(o)
11: return set of skyline objects S and safe zone Z = Z_e
```

An improved algorithm. To summarize the ideas presented so far, Algorithm 2 presents an improved approach to construct the safe zone. The set of skyline objects S is initially empty (line 2). At line 3, we access the objects in the proposed order. The pseudo-affecting set of each object is constructed using S (line 4). An object o is added to S if q lies inside its pseudo-impact region (line 8). 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).

3.4.2 Discussion

Now, we analyse the impact of the optimization used in Algorithm 2. 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 [33]) 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 criteria 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 5 *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 [32]. Furthermore, our experiments demonstrate that the average number of edges of the safe zone is around 5 for all data sets.

3.4.3 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 6 *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 4). 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 7 *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 2, an object o can be pruned if there exist an object $o' \in S$ that satisfies the above condition.

Lemma 8 *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 the conference version [12] of this paper, 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 2).

3.4.4 Branch and Bound Algorithm

Based on the ideas presented earlier in this section, a branch and bound algorithm can be designed. The basic idea is to use a branch and bound data structure (e.g., R-tree) to index the data objects and access the entries iteratively while pruning the entries that cannot affect the shape of the safe zone. The pruning rules presented earlier are extended for the entries of the R-trees. If the accessed entry is an object and cannot be pruned, it is used to update the safe zone. We omit the details due to the space limitations. However, the interested readers can see the complete algorithm along with the extended pruning rules in an earlier version of this paper [12].

4 Continuous monitoring of top-k queries

4.1 Problem Definition

Similar to Section 3.1, we assume a set of $(d+1)$-dimensional objects O with first d static dimensions and $d+1$-th dynamic dimension referring to $dist(q, o)$.

Continuous Top-k Query. A top-k query returns k objects with the lowest scores where the score of each

object o is calculated using a given monotonic scoring function f defined for the $d+1$ dimensions of the objects. An *ordered* top-k query returns the top-k objects in ascending order of their scores. In contrast, an *unordered* top-k query returns the top-k objects but not necessarily in the sorted order. Since the dynamic attribute of each object $dist(q,o)$ changes as the query q changes its location, the results are needed to be continuously updated as the query moves. A continuous ordered (resp. unordered) top-k query continuously reports the ordered (resp. unordered) top-k objects for a moving query q.

The existing techniques for static top-k queries are usually specifically designed for a particular type of scoring function f (e.g., weighted sum) and are not generally applicable for other types of scoring functions. Our focus in this paper is to design a generic solution to continuously monitor ordered and unordered top-k queries for *any* arbitrary monotonic scoring function f. We assume that all attribute values are non-negative. Our experimental study evaluates the proposed techniques for some of the most popular monotonic scoring functions namely weighted sum, weighted product and weighted distance. Next, we define these.

Weighted Sum [18, 15]. In weighted sum scoring function, a weight w is assigned for each attribute. The weight of i-th attribute is denoted as $w[i]$ where $w[i] \geq 0$ and $\sum_{i=1}^{d+1} w[i] = 1$. Here, $w[d+1]$ is the weight of the dynamic attribute (i.e., $dist(q,o)$) and $w[i]$ (for $1 \leq i \leq d$) is the weight of each static attribute $o[i]$. The score of an object o with regards to the query q is denoted as $score(q,o)$ and is computed as follows:

$$score(q,o) = w[d+1] \cdot dist(q,o) + \sum_{i=1}^{d} w[i] \cdot o[i] \quad (1)$$

Weighted Product [42, 4, 47]. Similar to weighted sum, weighted product assigns a weight $w[i]$ to each attribute where $w[i] \geq 0$ and $\sum_{i=1}^{d+1} w[i] = 1$. The score using weighted product is calculated as follows:

$$score(q,o) = dist(q,o)^{w[d+1]} \cdot \prod_{i=1}^{d} o[i]^{w[i]} \quad (2)$$

Weighted Distance [43, 5, 2]. Let s_o denote the static score of an object which corresponds to its score computed using only the static dimensions. The weighted distance computes the score as follows:

$$score(q,o) = \frac{dist(q,o)}{s_o} \quad (3)$$

The static score s_o can be computed using *any* monotonic scoring function (e.g., weighted product of the static dimensions). Weighted distance has been frequently used in spatio-textual queries [43, 2], where the static score corresponds to the textual similarity between the object and the query.

For the ease of presentation, in the rest of the paper, we assume that the scoring function f is a non-decreasing monotonic scoring function [6], i.e., the score of an object does not decrease when its values in the static and dynamic dimensions increase. However, our techniques are directly applicable for other monotonic functions, e.g., when they are non-increasing or when they are non-decreasing w.r.t. some attributes and non-increasing w.r.t. the remaining attributes. We say that a function is non-decreasing (resp. non-increasing) w.r.t. an attribute i when, assuming each $o[j]$ for $i \neq j$ is fixed, the univariate function over i is non-decreasing (resp. non-increasing). Note that weighted distance is non-decreasing w.r.t. $dist(q,o)$ and is non-increasing w.r.t. the static score s_o assuming s_o is always non-negative.

4.2 Formalizing Safe Zone

Definition 7 Safe Zone. Safe zone Z of a query q is an area such that, as long as q remains inside Z, its top-k objects do not change (although their relative order may change).

Once the safe zone Z of a query is computed by the server, it is sent to the client along with the top-k objects. The client can then locally monitor the results as long as q is inside Z. Specifically, if q is inside Z, the top-k objects remain the same and the results for an unordered top-k query do not need to be updated. For the ordered top-k query, the client locally computes the scores of these k objects to determine their relative order. When the query q leaves the safe zone Z, the client sends its new location to the server that computes a new safe zone Z and sends it back to the client along with the top-k objects.

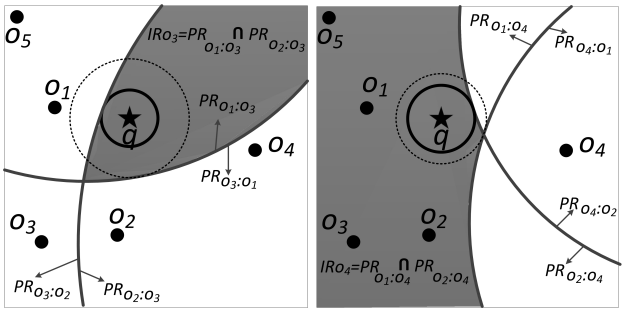
4.2.1 Optimal Safe Zone

The safe zone of a query q is said to be optimal if it is the largest possible safe zone. Before we formalize the optimal safe zone, we define the concept of *preferred region* and *irrelevant region*.

Definition 8 Preferred Region. Given a query q and two objects o_1 and o_2, the preferred region of o_1 with respect to o_2 (denoted as $PR_{o_1:o_2}$) is a region such that, as

long as q is inside $PR_{o_1:o_2}$, $score(q, o_1) \leq score(q, o_2)$, i.e., o_1 is preferable to o_2. Similarly, as long as q is inside the preferred region of o_2 with respect to o_1 (i.e., $PR_{o_2:o_1}$), $score(q, o_2) \leq score(q, o_1)$.

Example 6 Figure 3(a) shows the example of some preferred regions. For simplicity, we do not show the static dimensions of the objects. For the weighted sum scoring function, the preferred region is defined by a hyperbola between the two objects [46,18]. However, in this example, we do not assume any particular scoring function and do not discuss *how* the preferred regions are obtained. Assume that $PR_{o_1:o_3}$ is the region above the boundary shown in Figure 3(a) and $PR_{o_3:o_1}$ is the region below this boundary. As long as q remains inside $PR_{o_1:o_3}$, the score of o_1 remains at most equal to the score of o_3. Figure 3(a) also shows preferred regions $PR_{o_2:o_3}$ (region on the right side of the curve) and $PR_{o_3:o_2}$ (region on the left side of the curve).



(a) Irrelevant region of o_3 (b) Irrelevant region of o_4

Fig. 3 Preferred and Irrelevant regions

Definition 9 Irrelevant region. Irrelevant region IR_p for an object p is the region such that, as long as q is inside this region, p **cannot** be one of its top-k objects. We call it irrelevant region of p because p is irrelevant for the top-k query as long as q is inside this region.

Now, we describe how to compute irrelevant region of an object p using the preferred regions. Let T be the set of top-k objects and p be a non-result object, i.e., $p \in O \setminus T$. For each $o_i \in T$, $PR_{o_i:p}$ defines the space where $score(q, o_i) \leq score(q, p)$. Note that the top-$k$ results are not affected by p as long as q is in $PR_{o_i:p}$ for every $o_i \in T$. In other words, p cannot be one of the top-k objects as long as q is in $\bigcap_{\forall_{o_i \in T}} PR_{o_i:p}$.

$$IR_p = \bigcap_{\forall_{o_i \in T}} PR_{o_i:p} \tag{4}$$

Example 7 Consider the example of Figure 3 that shows a top-2 query and five objects o_1 to o_5. The top-2 objects are $T = \{o_1, o_2\}$ and the non-result objects are o_3, o_4 and o_5. Figure 3(a) shows the irrelevant region of o_3 (see the shaded area) which corresponds to $PR_{o_1:o_3} \cap PR_{o_2:o_3}$. Note that o_3 cannot be among the top-2 objects as long as q is inside its irrelevant region. Figure 3(b) shows the irrelevant region for o_4 (shaded area) which corresponds to $PR_{o_1:o_4} \cap PR_{o_2:o_4}$. For the object o_5, assuming that $PR_{o_1:o_5}$ is the whole data space, the irrelevant region of o_5 is $PR_{o_1:o_5} \cap PR_{o_2:o_5} = PR_{o_2:o_5}$ (see Figure 4(a)).

Now, we are ready to formalize the safe zone using irrelevant regions. Safe zone is the area such that, as long as q is inside it, every non-result object p cannot be one of the top-k objects. Assume that we have computed the irrelevant region for each non-result object $p_j \in O \setminus T$. The safe zone can be obtained as the intersection of all these irrelevant regions.

$$Z = \bigcap_{\forall_{p_j \in O \setminus T}} IR_{p_j} = \bigcap_{\forall_{o_i \in T}, \forall_{p_j \in O \setminus T}} PR_{o_i:p_j} \tag{5}$$

Example 8 Figure 4(a) shows the safe zone (shaded area) for the query q which is the intersection of the irrelevant regions of o_3, o_4 and o_5, i.e., $Z = IR_{o_3} \cap IR_{o_4} \cap IR_{o_5} = PR_{o_1:o_3} \cap PR_{o_2:o_3} \cap PR_{o_1:o_4} \cap PR_{o_2:o_4} \cap PR_{o_1:o_5} \cap PR_{o_2:o_5}$.

The following lemma shows that the safe zone defined above is correct and optimal.

Lemma 9 *Assume that it is possible to correctly compute the largest possible preferred region $PR_{o_i:p_j}$ for each pair of objects o_i and p_j. The safe zone Z defined in Eq. (5) is the largest possible area such that: (a) as long as q is inside Z, the set of its top-k objects T remains the same; and (b) as soon as q leaves Z, T is guaranteed to change.*

Proof. It is easy to see that (a) is correct because q is in $PR_{o_i:p_j}$ for every $o_i \in T$ and every $p_j \in O \setminus T$ which implies that the score of each object o_i in T is no worse than the score of each non-result object p_j as long as q is in Z. To prove (b), we show that, as soon as q leaves Z, there exists at least one non-result object p_j and one top-k object o_i for which $score(q, p_j) < score(q, o_i)$. When q leaves Z, there exists at least one pair of objects o_i and p_j for which q is outside $PR_{o_i:p_j}$. Since q has moved out of $PR_{o_i:p_j}$, $score(q, o_i) \not\leq score(q, p_j)$. □

Hereafter, we refer to the safe zone defined in Eq. (5) as the optimal safe zone and denote it as Z_{opt}. In our solution, we do not compute the optimal safe zone Z_{opt}

due to the following challenges/problems involved: 1) while it may be possible to compute preferred regions for some well defined scoring functions (e.g., using hyperbola for weighted sum), computing the preferred regions for arbitrary monotonic functions may be quite complicated or even impossible; 2) even if preferred regions can be obtained for arbitrary monotonic scoring functions, the optimal safe zone, which is the intersection of many preferred regions, becomes an arbitrarily complicated shape and, as a result, the complexity and computation cost of computing the optimal safe zone may outweigh the advantages of using it for monitoring moving top-k queries; 3) since the optimal safe zone may be an arbitrarily complicated shape, it may not be easy for the client to check whether the query is still inside the safe zone or not.

To address the challenges above, we underestimate the optimal safe zone using a circle (Section 4.2.2) and then present a relaxed definition of safe zone (Section 4.2.3) that guarantees the correctness of results but increases the size of the safe zone at the expense of a higher computation cost at client side.

4.2.2 Circular Safe Zone

We underestimate the safe zone using a circle, called circular safe zone, which also guarantees that the top-k objects remain the same as long as q is in it. Consider the example of Figure 4(a) where the circle underestimates the safe zone. The circular safe zone is not only easier to obtain for arbitrary monotonic functions but it also makes it easy for the client to check whether q is inside this safe zone or not. Below, we formally define the circular safe zone using preferred/irrelevant circles.

Preferred circle. Preferred circle of a top-k object o_i with respect to a non-result object p is a circle centred at q that completely lies within $PR_{o_i:p}$. Preferred circle is denoted as $C_{o_i:p}$ and its radius is denoted as $C^{rad}_{o_i:p}$.

Irrelevant circle. Irrelevant circle of a non-result object p is a circle centred at q that completely lies within its irrelevant region IR_p. Irrelevant circle is denoted as C_p and its radius is denoted as C^{rad}_p.

Note that the irrelevant circle of p is its smallest preferred circle, i.e., $C^{rad}_p = min_{\forall_{o_i \in T}} C^{rad}_{o_i:p}$. As long as q is in the irrelevant circle C_p, p cannot be one of the top-k objects, i.e., $\forall_{q \in C_p, o_i \in T} score(q, o_i) \leq score(q, p)$.

Example 9 In Figure 3(a) the smaller circle is the preferred circle $C_{o_2:o_3}$ as it completely lies within $PR_{o_2:o_3}$. The bigger circle in Figure 3(a) is $C_{o_1:o_3}$ and completely lies within $PR_{o_1:o_3}$. The irrelevant circle C_{o_3} is the smaller circle in Figure 3(a) and it lies completely within the irrelevant region of o_3. Figure 3(b) shows the two preferred circles for o_4 and the smaller circle is the irrelevant circle for o_4.

In the rest of the paper, for any circle C centered at q, we use C and C^{rad} interchangeably and use comparison operators between them whenever clear by context, e.g., $C_1 < C_2$ means that the radius of circle C_1 is smaller than the radius of circle C_2 implying that C_1 is contained in C_2.

Circular Safe Zone. A circular safe zone Z_c corresponds to the smallest irrelevant circle among the irrelevant circles of *all* non-result objects.

$$Z_c = min_{\forall_{p \in O \setminus T}} C_p = min_{\forall_{o_i \in T, p_j \in O \setminus T}} C_{o_i:p} \quad (6)$$

Figure 4 shows the irrelevant circles of o_3, o_4 and o_5. Note that $C_{o_3} < C_{o_4} < C_{o_5}$ and the circular safe zone corresponds to C_{o_3}.

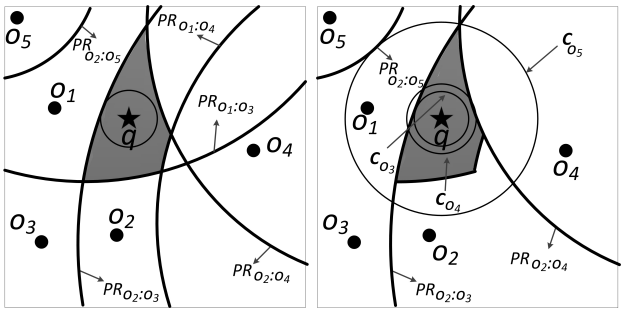
4.2.3 m-relaxed Safe Zone

Note that the circular safe zone may be significantly smaller than the optimal safe zone. In this section, we present a simple idea to relax the safe zone but still guaranteeing the correctness of results.

Definition 10 m-relaxed Safe Sone (Z_m). Given a positive integer m and a set of objects $M \subseteq O$ that contains $k + m - 1$ objects, an m-relaxed safe zone Z_m corresponds to the area such that, as long as q is inside this area, the top-k objects of the query are guaranteed to be present in M.

Note that the safe zone defined earlier (Definition 7) is the 1-relaxed safe zone where M contains only the top-k objects. The optimal safe zone Z_{opt} and the circular safe zone defined in the previous section are two examples of the 1-relaxed safe zone. Recall that the circular safe zone corresponds to the smallest irrelevant circle among the irrelevant circles of all objects in $O \setminus T$. It can be shown that the m-relaxed safe zone corresponds to the m-th smallest irrelevant circle where M consists of *all* top-k objects and the objects related to the $(m-1)$ smallest circles.

Example 10 Figure 4(b) shows the irrelevant circles of o_3, o_4 and o_5 for our running example of a top-2 query. For simplicity, we only show the preferred regions that define these irrelevant circles. The shaded area is the optimal safe zone. The smallest irrelevant circle C_{o_3} is 1-relaxed safe zone Z_1 and its corresponding M contains only the top-2 objects o_1 and o_2. The 2-relaxed safe zone Z_2 corresponds to the second smallest circle C_{o_4} where M consists of the top-2 objects (o_1 and o_2) and the



(a) Safe zone (b) m-relaxed Safe zone

Fig. 4 Safe zone and m-relaxed safe zone

object related to the smallest irrelevant circle, i.e., o_3. Note that, as long as q is inside Z_2, the top-2 objects are among o_1, o_2 and o_3. The 3-relaxed safe zone Z_3 corresponds to the third smallest irrelevant circle C_{o_5} where M consists of the top-2 objects (o_1 and o_2) and the two objects (o_3 and o_4) related to the two smallest irrelevant circles. It is easy to see that the top-2 objects are among these four objects as long as q remains inside Z_3. The 4-relaxed safe zone Z_4 corresponds to the whole data space and M consists of all objects.

In our solution, the m-relaxed safe zone Z_m is computed and sent to the client along with the set of objects M. The client then can locally monitor the top-k objects by computing the scores of these $k+m-1$ objects at each timestamp. When the client leaves Z_m, it requests the server to send it a new Z_m along with the set of objects M. Note that a larger value of m results in a bigger safe zone thus requiring the server to recompute the safe zone less often and potentially reducing the overall continuous monitoring time at the server. However, this also results in an increase in the computation cost at the client side as the client needs to compute the scores of $k+m-1$ objects at each timestamp. We do not discuss how to choose an appropriate value of m and treat it as a user defined parameter because it is application specific and depends on the processing power of the client device and whether the goal is to reduce the computation cost at the server or at the client.

4.3 Computing m-relaxed Safe Zone

One major challenge in computing the m-relaxed safe zone Z_m is how to efficiently compute the preferred and irrelevant circles for arbitrary monotonic scoring functions. In Sections 4.3.2, 4.3.3 and 4.3.4, we present our novel techniques to efficiently compute the preferred and irrelevant circles. However, first we present our algorithm to compute Z_m in order to provide a high level overview to the readers. Hereafter, we use safe zone to refer to m-relaxed safe zone whenever clear by context.

4.3.1 Algorithm

A straightforward approach to compute Z_m is to compute the irrelevant circles for all non-result objects and maintaining the m-smallest circles. In this section, we present an efficient safe zone computation algorithm that does not require computing the irrelevant circles for all non-result objects. The algorithm assumes that the set of objects O is indexed using a branch-and-bound data structure such as R-tree. First, we present a simple pruning rule that allows pruning an entry e of the R-tree.

Definition 11 Minimum (resp. maximum) score. Given a query q, an entry e of the R-tree, and a circle C, the minimum (resp. maximum) score of e with respect to C, denoted as $minScore(e, C)$ (resp. $maxScore(e, C)$), is the smallest (resp. largest) possible score of any object $p \in e$ considering that location of the query q can be anywhere in C. Since the scoring function is non-decreasing monotonic, $minScore(e, C)$ (resp. $maxScore(e, C)$) is obtained using smallest (resp. largest) static value of e in each dimension and considering distance to be the minimum (resp. maximum) distance between e and C denoted as $mindist(e, C)$ (resp. $maxdist(e, C)$).

Lemma 10 *Let e be an R-tree entry e, C be a circle containing q, and o_i be a top-k object. If $maxScore(o_i, C) \leq minScore(e, C)$ then, for every object $p \in e$, $score(q, o_i) \leq score(q, p)$ as long as q is inside the circle C.*

Proof is straightforward and is omitted. In our algorithm, we initialize the safe zone Z_m to be the whole data space and iteratively update it as we compute the irrelevant circles of objects. The following lemma formalizes a condition to prune an entry e of the R-tree.

Lemma 11 *Let e be an R-tree entry e, Z_m be the current safe zone and T be the set of top-k objects. No object $p \in e$ can affect the shape of Z_m (i.e., e can be safely pruned) if $\forall_{o_i \in T} maxScore(o_i, Z_m) \leq minScore(e, Z_m)$.*

Proof. We prove this by showing that, for every $p \in e$, the irrelevant region IR_p completely contains Z_m which implies that the safe zone Z_m can be correctly computed without considering p. Note that for every object $o_i \in T$ and every location of q in Z_m, we have $score(q, o_i) \leq score(q, p)$ (Lemma 10). This implies that every location $q \in Z_m$ is contained in the irrelevant region IR_p. □

Now, we present our safe zone computation algorithm (Algorithm 3). The algorithm initializes Z_m to be the whole data space (line 1). The set M and set of top-k objects T are both initialized to be empty (line 2). A min-heap h is initialized with the root entry of the R-Tree (line 3). For each de-heaped entry e, we check if it can be pruned using Lemma 11 (line 6). If e is not pruned and is an intermediate or leaf node of the R-tree, we insert each of its children c in the min-heap h (line 9). The key for each child c is set to be its minimum score $minScore(q, c)$ where minimum score is computed using the minimum distance from q to c and the smallest static value of the entry c in each dimension. This allows accessing objects in ascending order of their scores which has the following advantages: 1) It guarantees that the first k objects retrieved from the min-heap h are the top-k objects; 2) The non-result objects with smaller scores are expected to have smaller irrelevant circles because they have scores closer to the scores of top-k objects thus are expected to have smaller preferred regions. Therefore, accessing the non-result objects in ascending order of minimum scores is expected to shrink Z_m quicker and thus allowing more effective pruning of the entries in R-Tree using Lemma 11.

Algorithm 3 getSafeZone(q,O,m)

Input: query q; a set of objects O indexed using an R-Tree; a parameter m
Output: Safe zone Z_m and a set of objects M

```
1: Z_m^rad ← ∞
2: T ← φ; M ← φ
3: initialize a min-heap h with root of R-tree
4: while h is not empty and Z_m^rad ≠ 0 do
5:    de-heap an entry e
6:    if e cannot be pruned then  # Lemma 11
7:       if e is a node of R-tree then
8:          for each child c of e do
9:             insert c in h with key minScore(q, c)
10:      else  # e is an object
11:         if T contains less than k objects then
12:            insert e in both T and M
13:         else
14:            C_e ← getIrrelevantCircle(e) # Algorithm 6
15:            if C_e^rad < Z_m^rad then
16:               update m smallest irrelevant circles and M
17:               Z_m ← m-th smallest irrelevant circle
18:               MScore ← argmax_{o_i∈T} maxScore(o_i, Z_m)
19: return Z_m and M
```

If the de-heaped entry e cannot be pruned and is an object, we insert it in the set of top-k objects T and M if T contains less than k objects (line 12). This is because the first k objects retrieved from the min-heap h are guaranteed to be the top-k objects. If e is not a top-k object, we compute its irrelevant circle by calling Algorithm 6 presented later in Section 4.3.4 (line 14). If the irrelevant circle of e is smaller than the current safe zone Z_m, we update M and the m smallest irrelevant circles as required (line 16). The safe zone Z_m is updated to be the m-th smallest circle (line 17). The algorithm terminates when the heap becomes empty and returns the safe zone Z_m and the set of objects M (line 19).

Note that pruning an entry e using Lemma 11 takes $O(k)$ as it requires comparing $minScore(e, Z_m)$ with $maxScore(o_i, Z_m)$ for each o_i in the top-k objects. To reduce the cost to $O(1)$, whenever the safe zone Z_m is updated, we compute $\text{argmax}_{o_i \in T}\, maxScore(o_i, Z_m)$ called $MScore$ (line 18). While computing $MScore$ takes $O(k)$, once $MScore$ is computed, Lemma 11 can be applied in $O(1)$ by comparing $minScore(e, Z_m)$ with $MScore$. Since the number of times the safe zone Z_m is updated (line 17) is significantly smaller than the number of entries for which Lemma 11 is applied (line 6), this reduces the overall cost. The algorithm terminates when the heap h becomes empty or when the safe zone radius is determined to be zero (which may happen if all m objects have irrelevant circles with zero radii).

As stated earlier, the key challenge is to compute the irrelevant circle for an object p (line 14) for arbitrary monotonic functions. In Section 4.3.3 and Section 4.3.4, we present the details of how to efficiently compute the preferred circles and irrelevant circle of an object, respectively. However, first we present an algorithm to trim a rectangle using a preferred region which is a key module to compute preferred and irrelevant circles.

4.3.2 Trimming Rectangle Using a Preferred Region

Our algorithms to compute preferred/irrelevant circles rely heavily on a function that trims a rectangle R by pruning the part of R that is guaranteed to lie within a preferred region. Formally, given a rectangle R, a top-k query q with an arbitrary monotonic scoring function, and two objects o_i and p, the function `trimRectangle` prunes the part of the rectangle that is guaranteed to be contained inside $PR_{o_i:p}$ and returns a trimmed rectangle R_t which is a minimum bounding rectangle (MBR) of the unpruned area of R.

Let $minScore(o, R)$ and $maxScore(o, R)$ be defined as in Definition 11 except that the circle C is replaced with a rectangle R and the R-tree entry e is replaced with an object o. Lemma 12 defines a condition that, if satisfied, prunes the whole rectangle R, i.e., an empty rectangle is returned by `trimRectangle`.

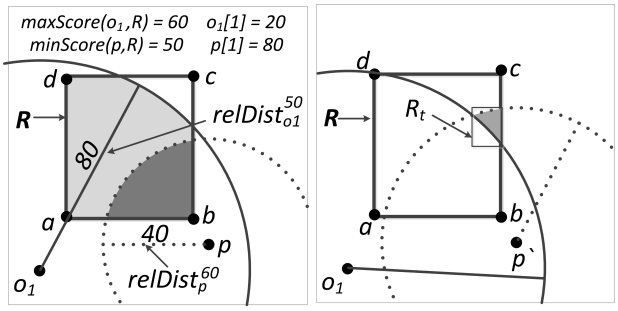
Lemma 12 *Given a rectangle R and two objects o_i and p, R completely lies inside $PR_{o_i:p}$ if $maxScore(o_i, R) \leq minScore(p, R)$.*

Proof. Note that $\forall_{q \in R} score(q, o_i) \leq score(p, R)$ implying that each location $q \in R$ is in $PR_{o_i:p}$. □

Lemma 12 either prunes the whole rectangle R or does not prune any part of it.

Example 11 Consider two objects o_1 and p (see Figure 5(a)) having only one static dimension with values $o_1[1] = 20$ and $p[1] = 80$. Assume that the scoring function is a weighted sum where weight for both static and dynamic dimensions is 0.5, i.e., $score(q, o) = o[1] \times 0.5 + dist(q, o) \times 0.5$. Assume $maxdist(o_1, R) = 100$ and $mindist(p, R) = 20$. Then, $maxScore(o_1, R) = 20 \times 0.5 + 100 \times 0.5 = 60$ and $minScore(p, R) = 80 \times 0.5 + 20 \times 0.5 = 50$. Lemma 12 cannot prune any part of R because $maxScore(o_1, R) > minScore(p, R)$.

Note that $maxScore(o_i, R)$ is computed assuming q is located at the furthest corner of R from o_i (e.g., corner c for o_1 in Figure 5(a)) whereas $minScore(p, R)$ is computed assuming q is located at the closest point in R from p (e.g., corner b for p in Figure 5(a)). In other words, the pruning condition in Lemma 12 is loose because it potentially assumes q to be at two different locations for o_i and p. An optimal containment check will consider the scores for both o_i and p assuming the same location of q in R, i.e., $\forall_{q \in R} score(o_i, p) < score(p, q)$. Although developing an optimal check for generic monotonic functions may be impossible, below we present an improved and more effective check which helps pruning the part of R (or even the whole R) contained in $PR_{o_i:p}$. First, we define relative distance.



(a) Prunes whole R (b) Shaded area is unpruned

Fig. 5 Illustration of trimming a rectangle

Definition 12 Relative Distance. Given a value s, an object o and a query q with a monotonic scoring function, the relative distance of o w.r.t. s denoted as $relDist_o^s$ is the distance such that $score(q, o) = s$ if $dist(q, o) = relDist_o^s$.

Example 12 In Figure 5(a), relative distance of o_1 w.r.t. score 50 is $relDist_{o_1}^{50} = 80$. This is because if $dist(q, o_1) = 80$, $score(q, o_1) = 20 \times 0.5 + 80 \times 0.5 = 50$. Similarly, relative distance of p w.r.t. score 60 is $relDist_p^{60} = 40$ and, if $dist(q, p) = 40$, $score(q, p) = 80 \times 0.5 + 40 \times 0.5 = 60$.

Lemma 13 *Given an object o, a value s and a query q with a non-decreasing monotonic scoring function, $score(q, o) \geq s$ for every location of q for which $dist(q, o) \geq relDist_o^s$. Similarly, $score(q, o) \leq s$ for every location of q for which $dist(q, o) \leq relDist_o^s$.*

The proof is straightforward and is omitted. In Figure 5(a), $relDist_p^{60} = 40$ and the dotted circle is centered at p with radius 40. If q lies outside the dotted circle, $score(q, p) \geq 60$ and, if q lies inside the dotted circle, $score(q, p) \leq 60$. Recall from Example 11 that $maxScore(o_1, R) = 60$. This implies that, for every query location $q \in R$ that lies outside the dotted circle, $score(q, o_1) \leq maxScore(o_1, R) \leq score(q, p)$. In other words, the part of R that lies outside the dotted circle is contained in $PR_{o_i:p}$. Therefore, the part of R that lies outside the dotted circle of p can be pruned and the rest of the rectangle R remains unpruned by p (denoted as $unpruned(p)$). In Figure 5(a), $unpruned(p)$ is the dark shaded area. Below, we formally present this observation.

Lemma 14 *Given a query q with a non-decreasing monotonic scoring function and two objects o_i and p, every point $x \in R$ for which $dist(p, x) \geq relDist_p^{maxScore(o_i,R)}$ is guaranteed to lie inside $PR_{o_i:p}$, i.e., if q is located at x, $score(q, o_i) \leq score(q, p)$.*

The proof is omitted as it is straightforward given Lemma 13. While Lemma 14 prunes the rectangle using the object p, the next lemma prunes it using the object o_i.

Lemma 15 *Given a query q with a non-decreasing monotonic scoring function and two objects o_i and p, every point $x \in R$ for which $dist(o_i, x) \leq relDist_{o_i}^{minScore(p,R)}$ is guaranteed to lie in $PR_{o_i:p}$, i.e., if q is located at x, $score(q, o_i) \leq score(q, p)$*

Proof. As per Lemma 13, if q is located at x, $score(q, o_i) \leq minScore(p, R)$. Thus, $score(q, o_i) \leq score(q, p)$. □

In Figure 5(a), the circle shown in solid line has the radius equal to $relDist_{o_i}^{minScore(p,R)} = relDist_{o_1}^{50} = 80$. For every location $q \in R$ which is inside this circle, $score(q, o_i) \leq 50 \leq minScore(p, R)$ implying that it lies inside $PR_{o_i:p}$. Therefore, the shaded (both light and dark) part of R lies inside $PR_{o_i:p}$ and can be pruned by object o_i. Recall that the dark shaded area is $unpruned(p)$, the area that was unpruned by p. The

unpruned(p) is pruned by o_i. In other words, the whole rectangle is pruned by either p or o_i.

Algorithm 4 trimRectangle(R,p, o_i)

Input: two objects p and o_i; a rectangle R
Output: A trimmed rectangle R_t

1: **if** $maxScore(o_i, R) \leq minScore(p, R)$ **then**
2: return ϕ
3: $Cir_p \leftarrow$ circle of p with radius $relDist_p^{maxScore(o_i,R)}$
4: $Cir_{o_i} \leftarrow$ circle of o_i with radius $relDist_{o_i}^{minScore(p,R)}$
5: $unpruned \leftarrow (R \cap Cir_p) - Cir_{o_i}$
6: $R_t \leftarrow$ MBR of *unpruned*
7: return R_t

Algorithm 4 illustrates how we apply the ideas presented in this section and return a trimmed rectangle R_t that cannot be pruned. First, we apply Lemma 12 to prune the whole rectangle and return ϕ (line 2) if $maxScore(o_i, R) \leq minScore(p, R)$. If Lemma 12 does not prune the rectangle, we compute the circles of p and o_i with radii equal to the relative distances (lines 3 and 4). Note that $(R \cap Cir_p)$ represents *unpruned*(p), the area that is not pruned by p (e.g., the dark shaded area in Figure 5(a)). The part of *unpruned*(p) that lies inside Cir_{o_i} can be pruned (as described earlier). Therefore, the unpruned area can be obtained as $(R \cap Cir_p) - Cir_{o_i}$ (line 5). In Figure 5(a), the unpruned area is empty. On the other hand, in Figure 5(b) where a different object p' is used, the unpruned area of R is the area shown shaded. The algorithm creates an MBR of the unpruned area and returns it (see R_t in Figure 5(b)).

Note that if the trimming results in reducing the size of the rectangle R (i.e., if $R_t \neq R$), we can recursively use the ideas presented above to further trim the rectangle R_t. This is because minimum and maximum distances from p and o_i to R_t (and consequently minimum/maximum scores and relative distances) may have changed resulting in further pruning possible. For example, in Figure 5(b), if Lemma 12 is used to pruned R_t, it may prune the whole rectangle R_t. However, we do not use this idea of recursively trimming the rectangle in Algorithm 4 because, as we show later, our algorithm to compute preferred circle iteratively calls `trimRectangle` to prune smaller rectangles as required.

We remark that obtaining the relative distance is typically straightforward for most of the monotonic scoring functions. However, if it is not easily obtainable for some complex monotonic scoring functions, the algorithm can be modified to use only Lemma 12.

4.3.3 Preferred Circle Computation

Note that radius of a valid preferred circle $C_{o_i:p}$ is at most equal to the minimum distance from q to the preferred region $PR_{p:o_i}$ denoted as $mindist(q, PR_{p:o_i})$, e.g., in Figure 3(a), the preferred circle $C_{o_2:o_3}$ is the smaller circle shown in thick line and its radius is equal to the minimum distance from q to the boundary of $PR_{o_3:o_2}$. Any circle centered at q with radius smaller than or equal to $mindist(q, PR_{p:o_i})$ is a valid preferred circle. If the monotonic scoring function is well defined (e.g., weighted sum), the largest possible preferred circle can be computed by setting its radius to be $mindist(q, PR_{p:o_i})$. However, for arbitrary monotonic scoring functions, it may not be possible to compute $mindist(q, PR_{p:o_i})$ because the preferred region is unknown. In this section, we show how to compute a valid preferred circle for any arbitrary monotonic scoring function.

First, we present the intuition using the example of Figure 6(a) that shows a top-k object o_1 and a non-result object p along with the preferred regions $PR_{o_1:p}$ and $PR_{p:o_1}$. Assume that the whole data space is covered by non-overlapping rectangles as shown in Figure 6(a) where each shaded rectangle lies completely inside $PR_{o_1:p}$. Let $\mathbb{R}$ denote the set of rectangles that do not completely lie inside $PR_{o_1:p}$, i.e., they either overlap with the boundary of $PR_{p:o_1}$ or lie completely in $PR_{p:o_1}$ (e.g., the white rectangles in Figure 6(a)). Note that $mindist(q, PR_{p:o_1})$ is bigger than or equal to the smallest minimum distance from q to these white rectangles, i.e., $mindist(q, PR_{p:o_1}) \geq min_{\forall R_i \in \mathbb{R}} mindist(q, R_i)$. This distance can be used to create a valid preferred circle. In Figure 6(a), the minimum distance from q to the rectangle R_1 (the small rectangle shown in thick lines) defines the preferred circle and, as long as q is inside this circle, $score(q, o_1) \leq score(q, p)$, i.e., a circle centred at q with radius equal to $mindist(q, R_1)$ is a valid preferred circle $C_{o_1:p}$.

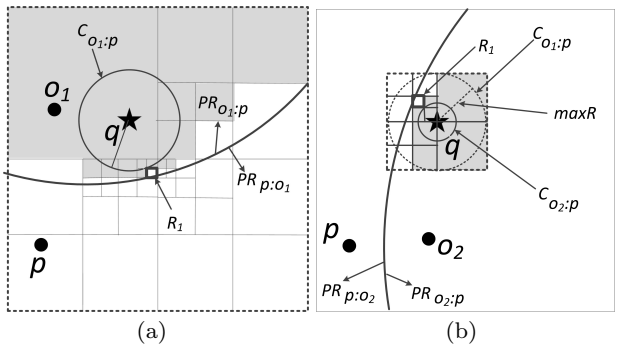


Fig. 6 Computing preferred circles $C_{o_1:p}$ and $C_{o_2:p}$

Using the intuition presented above, we compute the preferred circle by iteratively dividing the data space into smaller rectangles and pruning the rectangles that completely lie within $PR_{o_i:p}$ (shaded rectangles in Figure 6). The unpruned rectangles (white rectangles) are accessed in ascending order of their minimum distances from q. When the number of rectangles processed is greater than a threshold x or an accessed rectangle R_i is sufficiently small (e.g., its side length is at most ϵ), we return its minimum distance from q (i.e., $mindist(q, R_i)$) as the radius of the preferred circle.

Algorithm 5 getPreferredCircle(p, o_i, $maxR$)

Input: two objects p and o_i; maximum radius $maxR$
Output: A circle centered at q with radius at most $maxR$ that lies completely inside the preferred region $PR_{o_i:p}$

```
1:  R_tiny ← MBR of a circle centered at q with radius ε/2
2:  if trimRectangle(R_tiny, p, o_i) = R_tiny then
3:     return C_{o_i:p} with radius 0
4:  R ← MBR of a circle centered at q with radius maxR
5:  R_t ← trimRectangle(R, p, o_i)  # Algorithm 4
6:  if R_t ≠ φ and mindist(q, R_t) < maxR then
7:     initialize a min-heap h by inserting R_t
8:     while h is not empty do
9:        de-heap an entry e
10:       if side length of e ≤ ε or # of iterations > n then
11:          return C_{o_i:p} with radius mindist(q, e)
12:       for each child rectangle R_c of e do
13:          R_t ← trimRectangle(R_c, p, o_i) # Algorithm 4
14:          if R_t ≠ φ and mindist(q, R_t) < maxR then
15:             insert R_t in h with key mindist(q, R_t)
16: return C_{o_i:p} with radius maxR
```

Algorithm 5 shows the details of computing the preferred circle $C_{o_i:p}$. The algorithm takes $maxR$ as one of the arguments which serves as an upper bound on the radius of the preferred circle computed by the algorithm. As we show later in Section 4.3.4, bounding the radius improves the efficiency of the algorithm that computes the irrelevant circle. We illustrate the algorithm using Figure 6 assuming that $maxR$ in Figure 6(a) is infinity and $maxR$ in Figure 6(b) is the radius of the circle shown in broken line. Note that any trimmed rectangle R_t can be pruned if: (i) R_t is empty; or (ii) if $mindist(q, R_t) \geq maxR$. (i) implies that the whole rectangle lies inside $PR_{o_i:p}$ (shaded rectangles in Figure 6) and (ii) implies that the rectangle R_t cannot define the radius of the preferred circle which is bounded from above by $maxR$. For simplicity of illustration, we assume that, for each rectangle R_i in Figure 6, `trimRectangle` either prunes the whole rectangle or does not prune any part of it, i.e., R_t is either empty or equals R_i.

We explain lines 1 to 3 of Algorithm 5 towards the end. The algorithm creates (see line 4) an MBR R of the circle centered at q with radius equal to $maxR$ (see the rectangle shown using broken lines in Figure 6(b)). If $maxR$ is infinity, the MBR is assumed to be the whole data space (as in Figure 6(a)). The rectangle R is then trimmed using Algorithm 4. If R_t can be pruned (i.e., R_t is empty or $mindist(q, R_t) \geq maxR$), the algorithm returns a preferred circle $C_{o_i:p}$ with radius equal to $maxR$ (see lines 6 and 16). If the trimmed rectangle R_t cannot be pruned, a min-heap h is initialized by inserting R_t in it (line 7). The key of each entry in the min-heap is its minimum distance from the query q. For each de-heaped entry e, if its side length is greater than ϵ, it is divided into four equal child rectangles (line 12). Each child rectangle R_c is trimmed and is pruned if R_t is empty or $mindist(q, R_t) \geq maxR$. For example, each shaded rectangle in Figure 6 is pruned. If the trimmed rectangle R_t cannot be pruned, we insert it in the min-heap h with key set to $mindist(q, R_t)$ (line 15). In Figure 6, all the white rectangles are inserted in the min-heap h.

If a de-heaped entry e has side length at most equal to ϵ or if the number of iterations is greater than a threshold n, we return a preferred circle with radius set to $mindist(q, e)$ (line 11). Note that each remaining entry e' in the heap has $mindist(q, e') \geq mindist(q, e)$. Therefore, the returned value corresponds to the smallest distance from q to a rectangle that does not completely lie within $PR_{o_i:p}$. In other words, the algorithm correctly computes the radius of the preferred circle. In each of Figure 6(a) and Figure 6(b), $mindist(q, R_1)$ is the radius of the preferred circle $C_{o_i:p}$. At any stage, if the heap becomes empty, the algorithm returns $C_{o_i:p}$ (line 16) which corresponds to the preferred circle with radius $maxR$.

Let R_{tiny} be a rectangle containing q with side length at most ϵ. If the rectangle cannot be trimmed by `trimRectangle`, the algorithm will eventually return preferred circle with radius $mindist(q, R_{tiny}) = 0$ (at line 11). However, before returning this, the algorithm will need to process all ancestors of this rectangle which results in un-necessary computations. To avoid this, at the beginning, we create R_{tiny} (line 1) and return a preferred circle with radius zero if R_{tiny} cannot be trimmed (line 3).

4.3.4 Irrelevant Circle Computation

A straightforward approach to compute the irrelevant circle for an object p is to compute its preferred circle $C_{o_i:p}$ for each top-k object o_i using Algorithm 5 and maintaining the smallest preferred circle as the irrelevant circle. We present our algorithm to compute irrel-

evant circle (Algorithm 6) that uses some optimizations to improve the efficiency.

Recall that Algorithm 3 iteratively computes irrelevant circles of non-result objects and, when smaller irrelevant circles are found, it updates Z_m to be the m-th smallest irrelevant circle. Therefore, a newly accessed non-result object p does not affect Z_m if its irrelevant circle is bigger or equal to Z_m. Therefore, we bound from above the irrelevant circle of each newly accessed non-result object p to be Z_m (see line 1 of Algorithm 6). We compute the preferred circle $C_{o_i:p}$ for each top-k object o_i and update the irrelevant circle if a smaller preferred circle is found (lines 2 to 4). Specifically, at each iteration, we call Algorithm 5 to compute the preferred circle and pass the radius C_p^{rad} of the current irrelevant circle C_p as the $maxR$ parameter (line 3). This serves as an upper bound for the radius of the preferred circle $C_{o_i:p}$ to be computed by Algorithm 5. This is because C_p corresponds to the smallest preferred circle and we do not need to compute a preferred circle that is bigger than C_p. This reduces the data space for the preferred circle computation algorithm. The irrelevant circle C_p is updated to be $C_{o_i:p}$ (line 4) because $C_{o_i:p}$ computed at line 3 is smaller than or equal to C_p. At any stage, if the radius of C_p becomes zero, the algorithm returns C_p (line 5) without processing other top-k objects.

Example 13 Consider the example of Figure 6 where the irrelevant circle of p is to be computed using two top-k objects o_1 and o_2. Assume that we first compute $C_{o_1:p}$ as shown in Figure 6(a). When we call Algorithm 5 to compute $C_{o_2:p}$, we pass the radius of $C_{o_1:p}$ as $maxR$ parameter (see the circle shown in broken line in Figure 6(b)). This bounds the data space (see the rectangle in Figure 6(b)) and the algorithm may terminate quicker (because it initializes the min-heap using a smaller rectangle).

Algorithm 6 getIrrelevantCircle(p)

1: $C_p \leftarrow Z_m$ `# irrelevant circle initialized to safe zone`
2: **for** each top-k object o_i in descending order of scores **do**
3: $\quad C_{o_i:p} \leftarrow$ getPreferredCircle(p, o_i, C_p^{rad}) `# Algorithm 5`
4: $\quad C_p \leftarrow C_{o_i:p}$
5: $\quad$ **if** $C_p^{rad} = 0$ **then** return C_p
6: return C_p

Finally, note that the top-k objects that have higher scores are expected to generate smaller preferred circles (because they are expected to have smaller preferred regions). Therefore, we compute the preferred circles by accessing the top-k objects in descending order of scores (line 2 in Algorithm 6) so that the smallest preferred circle can be obtained earlier thus reducing the upper bound (i.e., C_p^{rad}) for Algorithm 5 passed as an argument at line 3 of Algorithm 6.

Example 14 In Figure 6, assume that we first compute the preferred circle $C_{o_2:p}$ which is the small circle in Figure 6(b). When we compute the preferred circle $C_{o_1:p}$, we will pass the radius of $C_{o_2:p}$ as $maxR$ parameter to Algorithm 5 which will create a rectangle R as MBR of $C_{o_2:p}$ and trim it (see lines 4 and 5 in Algorithm 5). Although this rectangle R is not shown in Figure 6(a), the readers can see that such rectangle would lie completely inside the preferred region $PR_{o_1:p}$. Therefore, Algorithm 5 will terminate without inserting any rectangle in the heap (and will return a circle with radius $maxR$ which is the radius of $C_{o_2:p}$).

4.4 Discussion

4.4.1 An alternative approach and its limitations

k-skyband [33] consists of every object o that is dominated by at most $k-1$ other objects. k-skyband is a natural extension of skyline, i.e., skyline is 1-skyband. It is well-known that the top-k objects for any monotonic scoring function are guaranteed to be found in the k-skyband [30,13]. Thus, an alternative approach to monitor top-k queries is to extend the techniques presented in Section 3 to create a safe zone for the k-skyband guaranteeing that the k-skyband remains unchanged as long as q remains inside the safe zone. Thus, as long as q is inside, the top-k objects can be locally computed by the client by recomputing the scores of the objects in the k-skyband. However, this approach is inferior to the m-relaxed safe zone and suffers from several limitations as described below.

Firstly, the m-relaxed safe zone guarantees that the top-k objects are within the $k+m-1$ objects, thus, the value of m can be adjusted considering the computational power of a client. On the other hand, the k-skyband cannot guarantee a fixed number of objects to be sent to the client. Consequently, the safe zone constructed using the k-skyband may be ineffective. E.g., in the worst case, the k-skyband may consist of all objects thus making it infeasible for the client to locally update the results.

Secondly, since size of the k-skyband is typically much larger than $k+m-1$, the safe zone for the k-skyband is expected to be much smaller than the m-relaxed safe zone. This is because the safe zone based on the k-skyband needs to guarantee that a larger number of objects remain the result (i.e., k-skyband) of the query. For example, our experimental study shows that the safe zone for the skyline spans around 1 to

2 kms (Figure 11) whereas the m-relaxed safe zone is more than an order of magnitude bigger (e.g., see Figure 13(c)). For this reason, as shown in our experimental study, the cost of monitoring skyline is much higher (e.g., a few milliseconds per timestamp on average) than the cost of monitoring top-k queries using the m-relaxed safe zone (e.g., a few microseconds per timestamp).

Finally, shape of the safe zone for the k-skyband may be a complex polygon requiring higher computational cost for the client to check whether it is still inside the safe zone or not. In contrast, the m-relaxed safe zone is always a circle requiring $O(1)$ to check containment.

4.4.2 Generalizing the proposed solution

Our proposed solution does not only apply to the top-k queries for arbitrary monotonic scoring functions but it also provides a framework that can be used to create relaxed (i.e., larger) safe zones for various other types of queries such as k nearest neighbors (kNN) queries and skyline queries etc. The proposed ideas can also be generalized for other distance metrics such as Manhattan distance, network distance in road networks or for multidimensional space for the locations of query and objects. In the rest of this section, we abuse the concept of a circle C_x centered at x with radius r to refer to the data space such that, for every $p \in C_x$, $dist(x,p) \leq r$ and, for every $p' \notin C_x$, $dist(x,p') > r$ where $dist(x,p)$ is the distance between x and p considering the distance metric used by the query (e.g., Euclidean distance in 3d space, Manhattan distance, or network distance). Below, we briefly describe the generalization needed to adapt our solutions for other settings.

Recall that, for a top-k query, we define preferred region of o_1 w.r.t. o_2 to be a region such that as long as q is in this region, o_1 is preferred over o_2, i.e, $score(q,o_1) \leq score(q,o_2)$. The preferred region can be generalized for other queries by appropriately defining preference of o_1 over o_2. E.g., for kNN queries, o_1 is preferred over o_2 if $dist(q,o_1) \leq dist(q,o_2)$. For skyline queries, o_1 is preferred over o_2 if o_1 dominates o_2. Irrelevant region of a non-result object p is generalized to be a region such that, as long as q is inside it, p cannot be the result of the query. The preferred/irrelevant circle is then immediately generalized based on the generalized definitions of preferred/irrelevant region.

Our algorithm to compute the m-relaxed safe zone (Algorithm 3) can then be easily extended. The pruning rule at line 6 can be applied by appropriately modifying the definitions of minimum and maximum scores $minScore(e,C)$ and $maxScore(e,C)$. The algorithm to compute the preferred circle can be immediately applied except that a hyperrectangle is used for the d-dimensional space and, in each iteration, a hyperrectangle is divided into 2^d child rectangles. This idea is applicable for road networks as long as minimum and maximum network distances to the child rectangles can be computed efficiently. Alternatively, for road networks, a road network branch and bound index (e.g., G-tree [50]) can be used instead of a rectangle. Algorithm 4 which trims an entry (i.e., a rectangle or a G-tree node) can be modified to use only Lemma 12 which either prunes the whole entry or does not prune any part of it. Finally, the algorithm to compute the irrelevant circle is the same where it computes the preferred circle of p w.r.t. each result object of the query.

Let $|T|$ be the number of objects in the query result (e.g., k for a kNN query or skyline size for a skyline query). The m-relaxed safe zone returns the client a set of $|T|+m-1$ objects guaranteeing that the results of the query can be found within these $|T|+m-1$ objects as long as the query is inside the m-relaxed safe zone.

We remark that our algorithm to continuously monitor skyline presented in Section 3 is superior to this generalized algorithm. This is because the generalized algorithm requires the client to process $|T|+m-1$ objects at each timestamp to get the up-to-date skyline. On the other hand, the algorithm in Section 3 does not require any computation at the client as long as q is inside the safe zone because the skyline is guaranteed to be unaffected. However, an advantage of the generalized approach presented in this section is that it provides a single implementation that can be used to monitor various different queries under different setting by using appropriate functions/modules.

5 Experiments

Modern computers have main memories large enough to accommodate typical spatial data sets. However, the IO cost may still be of interest in the applications where the indexed data (e.g., R-trees) is kept in the secondary memory, in which case, an algorithm that only accesses a small part of the indexed data may be preferable. To cover both types of applications, we measure the performance in terms of the CPU cost as well as the number of R-tree nodes accessed.

5.1 Moving skyline queries

5.1.1 Competitors

A naïve approach to monitor moving skyline queries is to compute the safe zone as we described in the basic algorithm (Algorithm 1). Another naïve approach is to

call existing algorithms such as BBS [33] to re-compute the skyline at each timestamp. However, our experiments demonstrated that both of these 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. Specifically, we assume that there exists an oracle that computes the safe zone without incurring any IO or CPU cost. The supreme algorithm uses BBS [33] 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 [33] for skyline queries where the data is indexed by an R-tree, 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.

5.1.2 Experimental setup

Our evaluation framework is similar to the framework used in existing safe zone based techniques [11,48,31] for continuous spatial queries. Table 2 shows the parameters we use in our experiments and the default values are shown in bold.

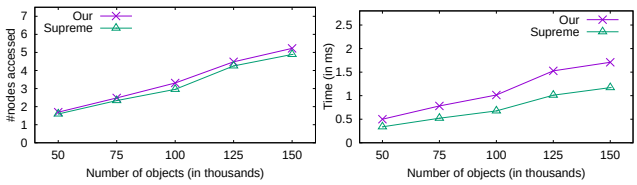
Table 2 System Parameters

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

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) [8]. 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 [9] 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 average cost per timestamp.

5.1.3 Effect of data cardinality

Figure 7 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

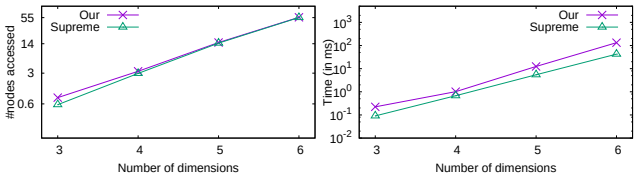
Fig. 7 Effect of data cardinality

5.1.4 Effect of dimensionality

In Figure 8, we change the number of static dimensions from 1 to 4 (the location coordinates are two dimensional). It is well known [33,8] that the cost of skyline computation algorithms significantly increases with the increase in dimensionality. The same can be observed in Figure 8. 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).

5.1.5 Effect of data distribution

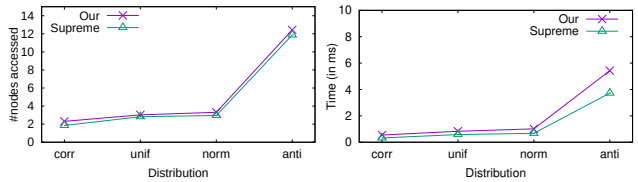
In Figure 9 we study the effect of data distribution. The distribution of location coordinates does not significantly affect the cost of the algorithms. Therefore, we only present the results for the distributions of values on static dimensions. More specifically, Figure 9



(a) Node accesses (b) CPU time

Fig. 8 Effect of dimensionality

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 [33,8], 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.

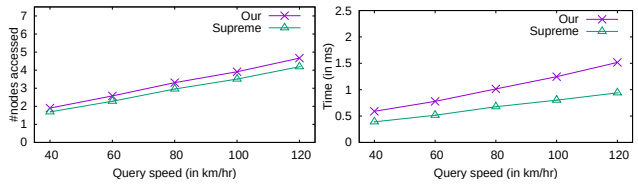


(a) Node accesses (b) CPU time

Fig. 9 Effect of distribution

5.1.6 Effect of query speed

In Figure 10, 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.

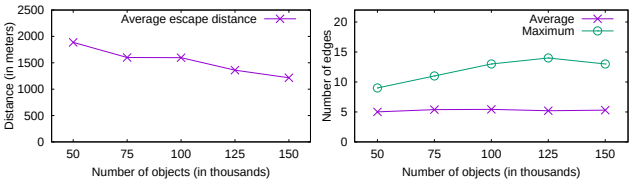


(a) Node accesses (b) CPU time

Fig. 10 Effect of query speed

5.1.7 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 Figure 11(a), we show the average escapee distance which is the average distance the queries travel before leaving their respective safe zones. Figure 11(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.



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

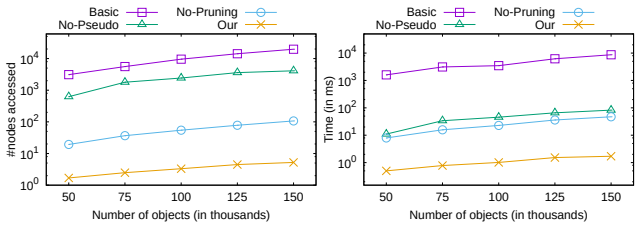
Fig. 11 Effectiveness of safe zone

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 client lies inside the safe zone or not is linear to the number of edges of the polygon. Figure 11(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.

5.1.8 Effectiveness of proposed optimizations

In Figure 12, we evaluate the effectiveness of the optimizations we presented in Section 3.4. More specifically, *Basic* is the basic algorithm (Algorithm 1). *No-Pseudo* is the same as our main algorithm 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.

Figure 12 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.



(a) Node accesses (b) CPU time

Fig. 12 Effectiveness of proposed optimizations

5.2 Moving top-k queries

5.2.1 Competitors

To the best of our knowledge, we are the first to propose a continuous monitoring algorithm for moving top-k queries supporting arbitrary monotonic scoring functions. To evaluate our algorithm, we compared it with a naïve algorithm that recomputes the results at each timestamp. However, naïve algorithm was several orders of magnitude worse than our algorithm. For a strict evaluation, we designed a *supreme* algorithm which assumes that the optimal safe zone (i.e., the largest possible safe zone as described in Section 4.2.1) is known to it at no additional cost. Specifically, at each timestamp, we check if the set of top-k objects are the same as the previous timestamp. If so, we know that the query is still inside the optimal safe zone. Therefore, we assume that the supreme algorithm does not incur any CPU or IO cost in such case. At the timestamp when the set of top-k objects changes, the query must have moved out of the optimal safe zone in which case the CPU and IO cost of the supreme algorithm corresponds to the cost of recomputing the top-k objects which is done using a best-first search on the R-tree. In other words, the cost of the supreme algorithm is the cost of recomputing the top-k objects for only those timestamps when the query is guaranteed to have moved out of the optimal safe zone (and the top-k objects are guaranteed to be changed). Given that the supreme algorithm assumes the existing of an oracle that obtains optimal safe zone without incurring any cost, its IO and CPU cost is guaranteed to be superior than the existing algorithms that compute smaller than the optimal safe zones.

5.2.2 Experimental Setup

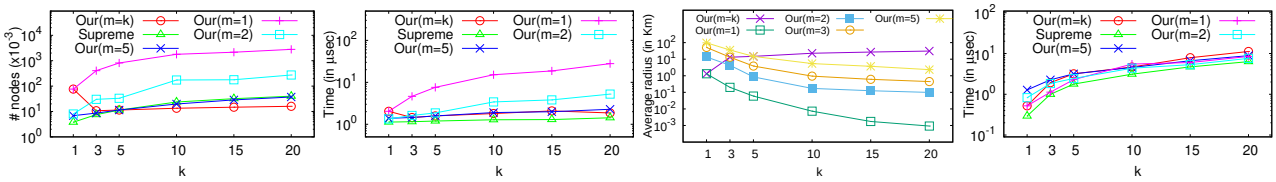
The experimental setup is the same as for the moving skyline queries (Section 5.1.2) and the default values for the relevant parameters are also the same as in Table 2. We vary k from 1 to 20 and the default value of k is 10. The static attributes are obtained from the census data set *House* used in [46] and correspond to the monthly owner costs, electricity cost, and property taxes (normalized from 0 to 1). Our algorithm requires two parameters ϵ and n (see Algorithm 5) which are set to 0.01% of the width of data universe and 100, respectively.

We evaluate the algorithms for the weighted sum, weighted product and weighted distance scoring functions (the default being weighted sum). We assign equal weights to each dimension as the default setting. However, we also show the effect of varying the weights for the dynamic and static dimensions. Note that the score of an object may be zero for the weighted product if even one of the attributes is zero. Similarly, the score may be undefined for the weighted distance function if static score is zero. To overcome these, we add the same constant to each attribute for the weighted product and weighted sum scoring functions. The results of each moving query are monitored for 1 hour (3600 timestamps) and the experiments report the average cost per timestamp.

In our experiments, we evaluate the CPU cost at the server as well as at the client. The CPU cost at the server is the cost of computing the top-k results (for the supreme algorithm) and the cost of computing both the top-k results and the safe zone for our algorithms. At each timestamp, the client checks whether the query is still inside the safe zone or not. If it is still inside the safe zone, it simply recomputes the scores of the objects which were sent by the server and updates the top-k results locally. Otherwise, it sends a request to the server to send a new safe zone along with the top-k results. Therefore, the client CPU time corresponds to the cost of checking whether it is still inside the safe zone or not *and*, if it is still inside the safe zone, the client CPU time includes the cost of recomputing the scores of the objects. For the supreme algorithm, we assume that the cost to check whether the query is still inside the safe zone or not is zero. We show the results for the ordered top-k queries. The server CPU cost for the unordered top-k queries is the same as the ordered top-k queries for both the supreme and our algorithms. The client CPU cost for the unordered top-k queries is zero for the supreme algorithm. The client CPU cost for our algorithm is the same as for the ordered top-k queries.

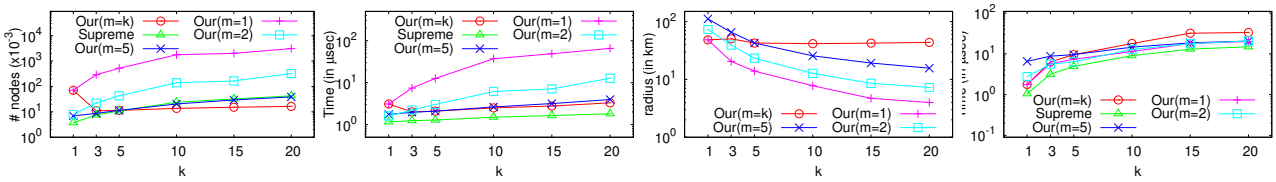
5.2.3 Effect of k and m

Since our algorithm uses m-relaxed safe zone, we evaluate our algorithm for different values of m. Specifically, we vary k and compare the performance of supreme algorithm against our algorithm where our algorithm uses $m = 1$, $m = 2$, $m = 5$ and $m = k$. Throughout



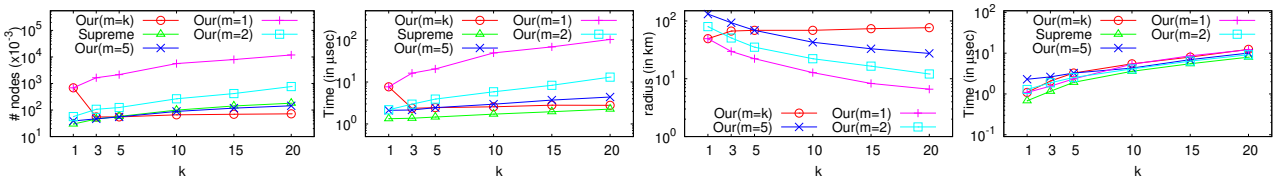
(a) Node accesses (b) Server CPU time (c) Safe zone radius (d) Client CPU time

Fig. 13 Effect of k and m (Weighted Sum Scoring Function)



(a) Node accesses (b) Server CPU time (c) Safe zone radius (d) Client CPU time

Fig. 14 Effect of k and m (Weighted Product Scoring Function)



(a) Node accesses (b) Server CPU time (c) Safe zone radius (d) Client CPU time

Fig. 15 Effect of k and m (Weighted Distance Scoring Function)

this section, we show the average IO cost per timestamp in multiples of 10^{-3} and the average CPU time per timestamp in microseconds.

Figures 13, 14 and 15 show the results for weighted sum, weighted product and weighted distance scoring functions, respectively. The results show that the IO cost and CPU time at the server decrease as the value of m increases. This is because the m-relaxed safe zone gets larger as the value of m increases and, as a consequence, the results of the query are to be recomputed less often. Figures 13(c), 14(c), 15(c) show the average radius of the safe zone for each algorithm and confirm that the radius is larger for bigger values of m. We do not show the radius for the supreme algorithm because the shape of the optimal safe zone is unknown. Note that the radius of the safe zone decreases as the value of k increases which explains the increase in IO and CPU cost for bigger values of k.

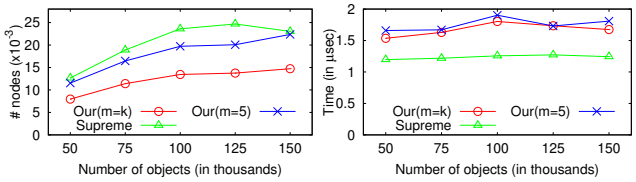
Our algorithm for $m = 5$ and $m = k$ performs reasonably well compared to the supreme algorithm for all scoring functions. Interestingly, the IO cost for our algorithm with $m = k$ and $m = 5$ is smaller than the IO cost of the supreme algorithm for larger k. This is because the supreme algorithm computes the safe zone which is optimal for $m = 1$ (i.e., the largest possible 1-relaxed safe zone) which may be smaller than the m-relaxed safe zone computed by our algorithm for $m > 1$.

The client CPU time increases with k as shown in Figures 13(d), 14(d) and 15(d) because the client needs to recompute the results of more objects at each timestamp. Note that the client CPU cost of our algorithm is pretty similar to the client CPU time of the supreme algorithm especially for smaller values of m.

For the ease of illustration, in the rest of the experiments, we compare the supreme algorithm with our algorithm only for $m = 5$ and $m = k$. Also, we only evaluate the IO cost and the CPU cost at the server for the weighted sum scoring function. The results for the other scoring functions follow a similar trend.

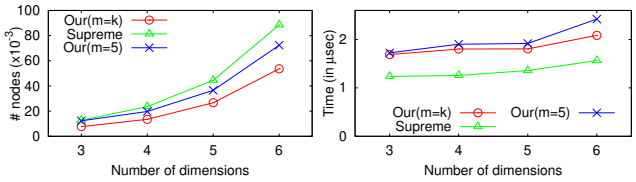
5.2.4 Effect of data cardinality

Figure 16 shows the effect of data cardinality. As expected, the IO cost increases with the increase in data cardinality. Our algorithm has significantly smaller IO cost than the supreme algorithm for $m = k$ which is mainly because of the larger safe zone. The CPU cost of the three algorithms only slightly increase with the increase in data cardinality which is mainly because most of the R-tree entries/objects are pruned. The CPU cost of our algorithms is around 30% more than that of the supreme algorithm. Both of our algorithms are very efficient and incur less than 2 microseconds per timestamp on average.



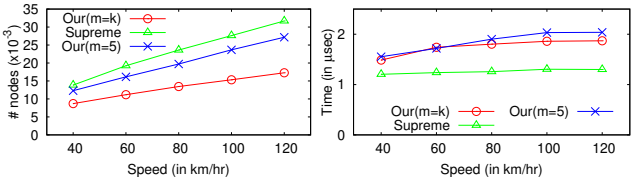
(a) Node accesses (b) Server CPU time

Fig. 16 Effect of data cardinality



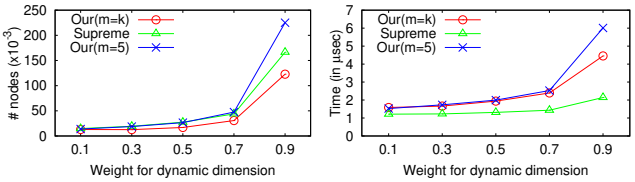
(a) Node accesses (b) Server CPU time

Fig. 17 Effect of dimensionality



(a) Node accesses (b) Server CPU time

Fig. 18 Effect of query speed



(a) Node accesses (b) Server CPU time

Fig. 19 Effect of weights

5.2.5 Effect of dimensionality

Figure 17 shows the effect of data dimensionality on the three algorithms. The IO and CPU cost of each algorithm increase with the dimensionality because the index becomes bigger and the algorithms need to process more nodes and objects to compute the top-k results/safe zone. Our algorithms performs reasonably well compared to the supreme algorithm and have IO costs lower than that of the supreme algorithm.

5.2.6 Effect of query speed

Figure 18 shows the effect of query speed on the three algorithms. We change the average speed of moving queries from 40 km/hr to 120 km/hr. As expected the cost of our algorithms increases with the increase in query speed because the query leaves the safe zone more often. Our algorithms perform reasonably well compared to the supreme algorithm for the CPU time and have lower IO cost.

5.2.7 Effect of weights

In Figure 19, we vary the weight assigned to the dynamic dimension (i.e., distance) and study its effect on the performance of the algorithms. The IO and CPU cost of each algorithm increase as the weight assigned to the distance is increased. This is because the distance between query and an object contributes to its score more significantly when the weight is larger which results in smaller safe zone (as the results become more sensitive to the query movement). The effect is more significant on the CPU time of our algorithms than the supreme algorithm because the m-relaxed safe zones are more severely affected by the increase in weight than the optimal safe zone used by the supreme algorithm.

6 Conclusions

We are the first to present a safe zone based approach to continuously monitor skyline for queries moving in arbitrary fashion. We also present the first approach to continuously monitor top-k queries involving arbitrary monotonic scoring functions. We propose efficient safe zone construction techniques to monitor the skyline queries and top-k queries. Our experiments demonstrate that the cost of our proposed algorithms is close to a lower bound cost and is more than three orders of magnitudes lower than a naïve algorithm. Our proposed approach to monitor top-k queries can be generalized to continuously monitor various other queries for different distance metrics. An interesting direction for future work is to consider continuous queries involving more than one dynamic dimensions.

Acknowledgments. Muhammad Aamir Cheema is supported by Australian Research Council (ARC) FT180100140 and DP180103411. Xuemin Lin is supported by ARC DP180103096 and DP200101338. Wenjie Zhang is supported by ARC DP180103096 and DP200101116. Ying Zhang is supported by ARC FT170100128 and DP180103096.

References

1. `http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm`
2. Abeywickrama, T., Cheema, M.A., Khan, A.: K-SPIN: Efficiently processing spatial keyword queries on road networks. IEEE TKDE (2019)
3. Abrishamchi, A., Ebrahimian, A., Tajrishi, M., Mariño, M.A.: Case study: Application of multicriteria decision making to urban water supply. Journal of Water Resources Planning and Management (2005)

4. Adriyendi, M.: Multi-attribute decision making using simple additive weighting and weighted product in food choice (2015)
5. Aurenhammer, F., Edelsbrunner, H.: An optimal algorithm for constructing the weighted voronoi diagram in the plane. Pattern Recognition **17**(2), 251–257 (1984)
6. Beliakov, G., Pradera, A., Calvo, T., et al.: Aggregation functions: A guide for practitioners, vol. 221. Springer
7. Bohm, C., Ooi, B.C., Plant, C., Yan, Y.: Efficiently processing continuous k-NN queries on data streams. In: ICDE, pp. 156–165 (2007)
8. Börzsönyi, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE, pp. 421–430 (2001)
9. Brinkhoff, T.: A framework for generating network-based moving objects. GeoInformatica **6**(2), 153–180 (2002)
10. Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Continuous monitoring of distance-based range queries. IEEE TKDE **23**(8), 1182–1199 (2010)
11. Cheema, M.A., Brankovic, L., Lin, X., Zhang, W., Wang, W.: Multi-guarded safe zone: An effective technique to monitor moving circular range queries. In: ICDE (2010)
12. Cheema, M.A., Lin, X., Zhang, W., Zhang, Y.: A safe zone based approach for monitoring moving skyline queries. In: EDBT, pp. 275–286 (2013)
13. Cheema, M.A., Shen, Z., Lin, X., Zhang, W.: A unified framework for efficiently processing ranking related queries. In: EDBT, pp. 427–438 (2014)
14. Chen, J., Zheng, J., Jiang, S., Qiu, X.: Distance-based continuous skylines on geo-textual data. In: Asia-Pacific Web Conference, pp. 228–240. Springer (2016)
15. Cong, G., Jensen, C.S., Wu, D.: Efficient retrieval of the top-k most relevant spatial web objects. PVLDB **2**(1), 337–348 (2009)
16. Fu, X., Miao, X., Xu, J., Gao, Y.: Continuous range-based skyline queries in road networks. World Wide Web **20**(6), 1443–1467 (2017)
17. Hsueh, Y., Zimmermann, R., Ku, W.: Efficient updates for continuous skyline computations. In: DEXA (2008)
18. Huang, W., Li, G., Tan, K.L., Feng, J.: Efficient safe-region construction for moving top-k spatial keyword queries. In: CIKM, pp. 932–941 (2012)
19. Huang, Z., Lu, H., Ooi, B.C., Tung, A.K.H.: Continuous skyline queries for moving objects. TKDE (2006)
20. Ilyas, I.F., Beskales, G., Soliman, M.A.: A survey of top-k query processing techniques in relational database systems. ACM Comput. Surv. **40**(4), 11:1–11:58 (2008)
21. Kang, J.M., Mokbel, M.F., Shekhar, S., Xia, T., Zhang, D.: Continuous evaluation of monochromatic and bichromatic reverse nearest neighbors. In: ICDE (2007)
22. Kossmann, D., Ramsak, F., Rost, S.: Shooting stars in the sky: An online algorithm for skyline queries. In: VLDB, pp. 275–286 (2002)
23. Lai, C.C., Akbar, Z.F., Liu, C.M., Ta, V.D., Wang, L.C.: Distributed continuous range-skyline query monitoring over the internet of mobile things. IEEE Internet of Things Journal (2019)
24. Lee, M.W., won Hwang, S.: Continuous skylining on volatile moving data. In: ICDE, pp. 1568–1575 (2009)
25. Li, H., Yoo, J.: An efficient scheme for continuous skyline query processing over dynamic data set. In: BIGCOMP, pp. 54–59 (2014)
26. Lin, X., Yuan, Y., Wang, W., Lu, H.: Stabbing the sky: Efficient skyline computation over sliding windows. In: ICDE, pp. 502–513 (2005)
27. Liu, J., Yang, J., Xiong, L., Pei, J., Luo, J.: Skyline diagram: Finding the voronoi counterpart for skyline queries. In: ICDE, pp. 653–664 (2018)
28. Lu, H., Zhou, Y., Haustad, J.: Continuous skyline monitoring over distributed data streams. In: SSDBM (2010)
29. Mateo, J.R.S.C.: Weighted sum method and weighted product method. In: Multi criteria analysis in the renewable energy industry, pp. 19–22. Springer (2012)
30. Mouratidis, K., Bakiras, S., Papadias, D.: Continuous monitoring of top-k queries over sliding windows. In: ACM SIGMOD (2006)
31. Nutanong, S., Zhang, R., Tanin, E., Kulik, L.: The v*-diagram: a query-dependent approach to moving knn queries. PVLDB (2008)
32. Okabe, A., Boots, B., Sugihara, K., Chiu, S.N.: Spatial Tessellations: Concepts and Applications of Voronoi Diagrams. Wiley (1999)
33. Papadias, D., Tao, Y., Fu, G., Seeger, B.: An optimal and progressive algorithm for skyline queries. In: SIGMOD (2003)
34. Papapetrou, O., Garofalakis, M.: Monitoring distributed fragmented skylines. DAPD **36**(4), 675–715 (2018)
35. Qi, J., Zhang, R., Jensen, C.S., Ramamohanarao, K., He, J.: Continuous spatial query processing: a survey of safe region based techniques. ACM Computing Surveys (CSUR) **51**(3), 64 (2018)
36. Shen, Z., Cheema, M.A., Lin, X.: Loyalty-based selection: Retrieving objects that persistently satisfy criteria. In: CIKM, pp. 2189–2193 (2012)
37. Shen, Z., Cheema, M.A., Lin, X., Zhang, W., Wang, H.: Efficiently monitoring top-k pairs over sliding windows. In: ICDE, pp. 798–809 (2012)
38. Shen, Z., Cheema, M.A., Lin, X., Zhang, W., Wang, H.: A generic framework for top-k pairs and top-k objects queries over sliding windows. IEEE TKDE (2014)
39. Shi, C., Qin, X., Wang, L.: Continuous skyline queries for moving objects in road networks. JSW (2015)
40. Tan, K.L., Eng, P.K., Ooi, B.C.: Efficient progressive skyline computation. In: VLDB, pp. 301–310 (2001)
41. Tang, Y., Chen, S.: Supporting continuous skyline queries in dynamically weighted road networks. Mathematical Problems in Engineering **2018** (2018)
42. Wang, M., Liu, S., Wang, S., Lai, K.K.: A weighted product method for bidding strategies in multi-attribute auctions. J. Systems Science & Complexity (2010)
43. Wu, D., Yiu, M.L., Jensen, C.S., Cong, G.: Efficient continuously moving top-k spatial keyword query processing. In: ICDE, pp. 541–552 (2011)
44. Wu, P., Agrawal, D., Egecioglu, Ö., Abbadi, A.E.: Deltasky: Optimal maintenance of skyline deletions without exclusive dominance region generation. In: ICDE (2007)
45. Xia, T., Zhang, D.: Continuous reverse nearest neighbor monitoring. In: ICDE, p. 77 (2006)
46. Yang, S., Cheema, M.A., Lin, X., Zhang, Y., Zhang, W.: Reverse k nearest neighbors queries and spatial reverse top-k queries. VLDB J. **26**(2), 151–176 (2017)
47. Zavadskas, E.K., Antucheviciene, J., Hajiagha, S.H.R., Hashemi, S.S.: Extension of weighted aggregated sum product assessment with interval-valued intuitionistic fuzzy numbers (waspas-ivif). Applied soft computing **24**, 1013–1021 (2014)
48. Zhang, J., Zhu, M., Papadias, D., Tao, Y., Lee, D.L.: Location-based spatial queries. In: SIGMOD (2003)
49. Zheng, J., Jiang, S., Chen, J., Yu, W., Zhang, S.: Dynamic skyline maintaining strategies for moving query points in road networks. Journal of Internet Technology **20**(5), 1359–1369 (2019)
50. Zhong, R., Li, G., Tan, K.L., Zhou, L.: G-tree: An efficient index for knn search on road networks. In: CIKM, pp. 39–48 (2013)