Range-Based Skyline Queries in Mobile Environments

Xin Lin, Jianliang Xu, *Senior Member*, *IEEE*, and Haibo Hu

Abstract—Skyline query processing for location-based services, which considers both spatial and nonspatial attributes of the objects being queried, has recently received increasing attention. Existing solutions focus on solving point- or line-based skyline queries, in which the query location is an exact location point or a line segment. However, due to privacy concerns and limited precision of localization devices, the input of a user location is often a spatial range. This paper studies a new problem of how to process such range-based skyline queries. Two novel algorithms are proposed: one is index-based (I-SKY) and the other is not based on any index (N-SKY). To handle frequent movements of the objects being queried, we also propose incremental versions of I-SKY and N-SKY, which avoid recomputing the query index and results from scratch. Additionally, we develop efficient solutions for probabilistic and continuous range-based skyline queries. Experimental results show that our proposed algorithms well outperform the baseline algorithm that adopts the existing line-based skyline solution. Moreover, the incremental versions of I-SKY and N-SKY save substantial computation cost, especially when the objects move frequently.

Index Terms—Location-based services, query processing, skyline queries, moving objects

1 Introduction

The combination of personal locator technologies, global positioning systems, and wireless communication technologies has flourished location-based services (LBSs) in recent years [4], [8], [14]. The main usage of LBSs is to provide mobile users with timely information at the right place for their decision making. Conventional LBSs focus on processing proximity-based queries, including the range query and nearest neighbor (NN) query [8], [24]. However, these queries are not sufficient for the applications that need to consider both spatial and nonspatial attributes of the objects being queried. A typical scenario is finding a nearby car park with cheap parking fee, in which distance is a spatial attribute and parking fee is a nonspatial attribute. Clearly, here a multicriterion query is more appealing than a conventional spatial query that considers the distance only. Among various multicriterion queries, the skyline query is considered as one of the most classical ones and receives the most attention in LBS research.

However, the dynamic nature of the spatial attribute makes skyline queries in LBSs unique and challenging. Take the above park-finding scenario, for example. At different locations, the distances from the user to the car parks are not the same. As a consequence, the skyline query results differ for different locations. To efficiently compute location-based skyline results, a number of algorithms have been proposed for one-shot queries [15], [18] and continuous queries [12], [28]. Nevertheless, these previous studies have limitations as they simply assume that the query location is an exact location point or a line segment. In this paper, we relax this assumption and propose a more general skyline query—*range-based skyline query* (RSQ), which takes a spatial range as the input of user location, as opposed to a point or a line in existing LBS skyline studies. Compared to existing skyline queries, the range-based skyline query might be more practical for several reasons:

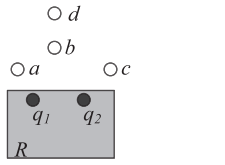
- The location of the query issuer could be shaped as a spatial region (e.g., a region that covers a complex spatial object or a group of users).
- Due to limited precision of localization devices, the query issuer does not have accurate knowledge about his/her exact location.
- For privacy reasons, the query issuer may not want to expose his/her exact location to the service provider. And a widely used solution is to blur the location into a cloaking region so that the adversary has no clue about where the query issuer is exactly located [6], [11], [26].

To study the range-based skyline problem, Fig. 1a shows an example of the above park-finding scenario, where the hollow points represent the parks' locations and the rectangle R represents the input range of the query. The nonspatial attributes of the parks are shown in Fig. 1b, where low parking fee and high service quality are preferred. Given two parks p_1 and p_2, if p_1 is no worse than p_2 in nonspatial attributes and p_1 is closer to the query point than p_2,[1] we say p_1 dominates p_2. A skyline query is to find all parks that are not dominated by any other park. The skyline results depend

- *X. Lin is with the Department of Computer Science and Technology, East China Normal University 500 Dongchuan Road, Shanghai 200241, China, and the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, KLN, Hong Kong. E-mail: xlin@cs.ecnu.edu.cn.*
- *J. Xu and H. Hu are with the Department of Computer Science, Hong Kong Baptist University, Kowloon Tong, KLN, Hong Kong. E-mail: {xujl, haibo}@comp.hkbu.edu.hk.*

Manuscript received 9 Dec. 2010; revised 15 Oct. 2011; accepted 23 Oct. 2011; published online 10 Nov. 2011.
For information on obtaining reprints of this article, please send e-mail to: tkde@computer.org, and reference IEEECS Log Number TKDE-2010-12-0654. Digital Object Identifier no. 10.1109/TKDE.2011.229.

1. Throughout this paper, we assume that the distances from a user to any two objects being queried are always not equal to each other, given sufficient precision in location representation.



Parks	Park fees	Service quality
a	15	4
b	13	5
c	14	5
d	18	3

(b)

Fig. 1. Example of range-based skyline query.

on the location of the query point. For example, if the query point is q_1, the skyline result set is $\{a, b\}$ because c is dominated by b while d is dominated by both a and b; if the query point is q_2, the skyline result set is $\{b, c\}$ because a and d are both dominated by c. Thus, given a range-based skyline query, the service provider should return a collective set of skyline results for every possible query point of the user in R. Unfortunately, this is not easy as the number of possible points in R is infinite. To address this issue, we propose a novel index-based algorithm, called I-SKY. The idea is to precompute the *skyline scopes* for all objects by their dominance relations. By indexing the skyline scopes, a range-based skyline query can be efficiently processed through a root-to-leaf traversal of the index tree.

The range-based skyline problem becomes even more challenging for dynamic data sets where the objects being queried can move and update their spatial attributes frequently. Processing skyline queries over moving objects has numerous applications, such as object tracking and monitoring, location-aware computing, virtual environments, and computer games [9], [15], [27]. For example, in a taxi dispatching application, the taxis notify their locations to the dispatcher from time to time (e.g., every 5 minutes). To dispatch taxis, the dispatcher considers the location identified and the last time the location was reported, and selects the taxis that are near to the customers with low volatility (e.g., either locations are recently updated or taxis are located at highly predictable regions with no traffic). As another example, in battlefields, to send critical supplies such as ammunitions and medical kits, the best locations can be dynamically determined by the positions of ground troops that are on the move and other nonspatial factors such as the current equipments and the number of wounded persons.[2] In these highly dynamic scenarios, the index-based I-SKY algorithm may not be efficient, as the index maintenance cost is expensive. To overcome this drawback, we further propose a nonindex algorithm, called N-SKY. It reduces a range-based skyline query into several segment-based skyline queries (SSQ), for which we develop efficient query processing techniques.

Our contributions made in this paper are as follows:

- We identify a new problem of range-based skyline queries arising in LBS applications. To the best of our knowledge, this is the first work that studies this problem.
- To process range-based skyline queries, we propose an index-based algorithm called I-SKY for static data sets and a nonindex algorithm called N-SKY for highly dynamic data sets.
- We further extend the problem to probabilistic and continuous range-based skyline queries. To process probabilistic top-k queries, we propose pruning techniques to improve computational efficiency. To process continuous queries, we propose efficient methods to compute the valid scope of each skyline object.
- We conduct extensive experiments to evaluate the performance of the proposed algorithms. The results show that our algorithms perform well under various system settings. In particular, the incremental algorithms save substantial computation cost for highly dynamic data sets.

The rest of this paper is organized as follows: Section 2 reviews related work on skyline queries and range-based queries. Section 3 gives some preliminaries of the problem. We present the index-based algorithm I-SKY and the nonindex algorithm N-SKY in Sections 4 and 5, respectively. In Section 6, we extend the algorithms to probabilistic top-k and continuous range-based skyline queries. The proposed algorithms are experimentally evaluated in Section 7. Finally, this paper is concluded in Section 8.

2 Related Work

In essence, a range-based skyline query inherits the characteristics of a skyline query and a range-based query. As such, we review the existing work on these two queries.

Skyline query processing. Skyline query processing was first introduced into the database community by Borzonyi et al. [3]. A number of algorithms have been proposed from then on. These algorithms can be divided into two categories. The first category is nonindex algorithms and the representatives are Black Nested Loop (BNL) and Divide-and-Conquer (D&C) [3]. BNL scans the data set sequentially and compares each new object to all skyline candidates kept in the memory. D&C partitions the data set into several parts, processes each part in the memory, and, finally, merges all partial skylines together. SFS [5] improved BNL by presorting the data sets. In the Bitmap approach [23], each data point is encoded in a bit string and the skyline is computed by some efficient operations on the bit matrix of all data points. The other category of skyline algorithms is index-based. In [23], a high-dimensional data set is converted into a 1D data set and a B+-tree is built to accelerate query processing. In [16], an algorithm called NN was proposed based on the depth-first nearest neighbor search via R*-tree. Papadias et al. [18], [19] proposed an improved algorithm, called Branch-and-Bound Skyline (BBS), which was based on the best-first nearest neighbor search. By accessing only the nodes that contain skyline points and employing effective pruning techniques, BBS achieves the optimal I/O access. More recently, in [25], a subset of skyline points are collected to approximately represent the distribution of an entire set of skyline points. Lee et al. [13] proposed a new index

2. More examples are provided in Appendix A, which can be found on the Computer Society Digital Library at http://doi.ieeecomputersociety.org/10.1109/TKDE.2011.229.

structure called ZBtree to index and store data points based on Z-order curve, and developed a novel algorithm ZSearch to process skyline queries.

Huang et al. [12] introduced the skyline query problem in the context of LBSs and proposed a continuous skyline query processing algorithm called CSQ for moving clients. Assuming a linear movement model, CSQ processes the skyline query at the starting point of the query segment and tries to predict the possible changes to the answer set when the client moves. This avoids continuously computing the skyline results from scratch. Zheng et al. [28] introduced a notion of valid scope for LBS skyline queries, which saves the recomputation if the next query point is still inside the valid scope. Sharifzadeh and Shahabi [22] defined a variant of skyline query in LBSs by considering the distance between an object and a set of query points. Our work is inspired by these prior point-based or line-based skyline algorithms, but focus on range-based skyline queries. Obviously, range-based skyline queries cannot be processed by simply applying the existing algorithms because the number of query points/line segments in a range is infinite.

Range-based query processing. Range-based query processing has recently received notable attention as the location privacy issue is becoming increasingly important. For privacy reasons, mobile clients tend to blur their exact locations into an uncertain range so that the service provider cannot find where they are exactly located. The service provider then returns a superset of candidate results for every possible query point in the range. Finally, the clients filter these results and obtain the true result by their exact locations. Existing range-based query algorithms studied only range-based kNN (RkNN) query. Hu and Lee proposed the first RkNN solution for rectangular ranges [10]. Ku et al. studied the same problem in spatial networks [17]. Complementally, Xu et al. developed an RkNN algorithm for circular ranges [26]. To the best of our knowledge, there is no work that has studied range-based skyline queries.

3 Preliminaries

3.1 Problem Definition

Before we present the detailed algorithms for processing range-based skyline queries, in this section we give some preliminaries of the problem. We consider a data set of objects O. Each object $o \in O$ is associated with one spatial (i.e., location) attribute and several other nonspatial attributes (e.g., parking fee and service quality).

Definition 1 (Nonspatial Dominance). *Given two objects o and o', if o' is no worse than o in all nonspatial attributes, then we say o'* **nonspatially dominates** *o. And o' is a* **nonspatial dominator object** *of o, and o is a* **nonspatial dominance object** *of o'. Formally, it is denoted as $o' \lhd o$. The set of o's nonspatial dominator objects is denoted as* **Dom**(o)*, i.e., o is dominated by any object in Dom(o) on nonspatial attributes.*

Definition 2 (Dominance). *Given a query point q and two objects o and o', if 1) o' nonspatially dominates o, and 2) o' is closer to q than o (i.e., o' also* **spatially dominates** *o), then we say o'* ***dominates*** *o w.r.t. the query point q. Formally, it is denoted as $o' \lhd_q o$.*

TABLE 1
Geometrical Notations

PerBis($\overline{ao}$)	Perpendicular bisector of line segment $\overline{ao}$.
$H(o, m)$	The open half-plane on o side of PerBis($\overline{om}$).
Cir(o, a)	Interior area of the circle centered at o and with a radius of $\|oa\|$.
Line(l)	The line on which line segment l is located.
CSP$\langle a, o, l\rangle$	Intersection point of PerBis($\overline{ao}$) and Line(l) for two objects a and o satisfying $a \lhd o$.

Definition 3 (Point-Based Skyline Query (PSQ)). *Given a data set O, the* **point − based skyline** *of a query point q returns a subset of O in which each object is not dominated by any other object in O w.r.t. q. We call this subset* skyline set *and denote it as $PSQ(q, O)$.*

Based on the above definitions, we formally define the range-based skyline query as follows:

Definition 4 (Range-Based Skyline query) *Given a data set O and a query range R, the* **range-based skyline query** *returns a superset of objects that appear in the skyline set of some point in R. Formally, it is denoted as $RSQ(R, O)$, and $RSQ(R, O) = \{o | \exists q \in R, o \in PSQ(q, O)\}$.*

In the example of Fig. 1, objects a and b are members of $PSQ(q_1, O)$, c is a member of $PSQ(q_2, O)$, and d is dominated by b w.r.t. any point in R. Hence, $RSQ(R, O)$ is $\{a, b, c\}$.

3.2 Assumptions and Notations

In this paper, we are interested in how to efficiently compute RSQ(R, O) given the query range R and the data set O. To facilitate query processing, we make the following assumptions:

- Both the mobile user and the objects are located in a 2D plane and the distance metric is euclidean distance.
- In a dynamic environment, both the mobile user and the objects may move and update their spatial locations.
- The values of nonspatial attributes remain constant throughout the query period.
- The query range R is rectangular.
- Each object has a different location. And the distances from the mobile user to any two objects are always not equal to each other.

The geometrical notations that will be used in this paper are listed in Table 1.

4 Index-Based RSQ Algorithm: I-SKY

We now present the algorithms for processing the range-based skyline query. We first consider the basic one-shot RSQ in this and next section. We will extend it to the probabilistic RSQ and continuous RSQ (C-RSQ) in Section 6.

This section introduces an index-based RSQ algorithm, called I-SKY. In Section 4.1, we first propose a notion of *skyline scope* for each object. By precomputing and indexing such skyline scopes, the RSQ query can be easily processed. To minimize the cost in computing the skyline scopes, we also propose an incremental version of the skyline scope construction algorithm in Section 4.2.

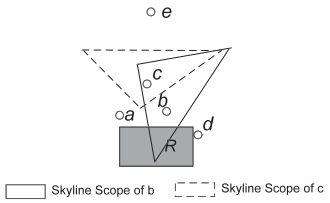


Fig. 2. Processing RSQ by skyline scopes ($\mathrm{Dom}(b) = \{a, d, e\}$, $\mathrm{Dom}(c) = \{a, b, d, e\}$).

4.1 Index Construction and Query Processing

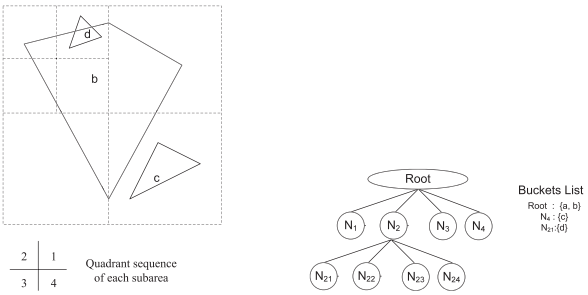
First, we introduce a notion of skyline scope for each object o. If Dom(o) is empty, i.e., o has no nonspatial dominator object, then o must be a skyline member of any query point q. Otherwise, o will not be a skyline member of a query point q if it is farther away from q than any of its nonspatial dominator objects in Dom(o); in other words, o will be a skyline member of q if it is closer to q than all its dominators in Dom(o). Therefore, we define the skyline scope of each object o as a region in which for any point q, o is closer to q than any object in Dom(o).

Definition 5 (Skyline Scope). *For an object $o \in O$, its skyline scope in a 2D plane P is denoted as $SS(o) = \{q | q \in P \wedge o \in PSQ(q, O)\}$, where $o \in PSQ(q, O)$ means $\forall m \in Dom(o), dist(o, q) < dist(m, q)$.*

To compute the skyline scope of each object o, we borrow the concept of Voronoi cell [2].

Definition 6 (Voronoi Cell). *Given a data set O, the Voronoi cell of an object $o \in O$, denoted as $V(o, O)$, is a convex hull in which for any point q, o is the nearest object in O. $V(o, O)$ can be directly obtained by the expression: $V(o, O) = \bigcap_{m \in O, m \neq o} H(o, m)$, where $H(o, m)$ is the open half-plane containing o, cut by the perpendicular bisector of line segment $\overline{om}$, i.e., PerBis($\overline{om}$). The intersection area of the half planes can be computed using a divide-and-conquer algorithm, with a time complexity of $O(|O| log |O|)$ [2].*

Obviously, the skyline scope of an object o can be obtained by computing the Voronoi cell of o with the object subset containing o and its nonspatial dominator objects, i.e., $\{o\} \cup \mathrm{Dom}(o)$. Then, a range-based skyline RSQ(R, O) can be computed by finding the objects whose skyline scopes intersect with R. We remark that as each object decides its own skyline scope with a different object subset, the skyline scopes of different objects may overlap. One should not take the union of all objects' skyline scopes as a *Voronoi diagram* [2]. Fig. 2 shows an example. Suppose a, d, and e have no nonspatial dominator object and we have the nonspatial dominance relations as $a \lhd b \lhd c$, $d \lhd b \lhd c$, and $e \lhd b \lhd c$. That is, $\mathrm{Dom}(a) = Dom(d) = \mathrm{Dom}(e) = \emptyset$, $\mathrm{Dom}(b) = \{a, d, e\}$, and $\mathrm{Dom}(c) = \{a, b, d, e\}$. Thus, the skyline scopes of a, d, and e cover the whole space. On the other hand, the skyline scopes of b and c are obtained by their Voronoi cells with object subsets $\{b\} \cup \{a, d, e\}$ and $\{c\} \cup \{a, b, d, e\}$, respectively, as shown in the figure. Given an RSQ query with R as the input of the query range (see Fig. 2), since R intersects with the skyline scopes of a, b, d, and e, the skyline result set is $\{a, b, d, e\}$.



(a) Space decomposition (b) Nodes and associated objects

Fig. 3. MX-CIF quadtree index.

As the skyline scopes do not depend on the query, they can be precomputed and indexed in advance for fast retrieval. In this paper, we use the MX-CIF quadtree [21] for indexing since it is considered more efficient to index overlapping spatial objects than an R-tree. Specifically, the MX-CIF quadtree recursively decomposes the underlying space into four equal-sized subspaces such that the skyline scope of each object is fully enclosed by a minimal subspace. Each subspace corresponds to an index node in the MX-CIF quadtree. Each object is associated with the index node of the minimal subspace. For example, in Fig. 3a, since the skyline scopes of a and b cannot be enclosed by any single quadrant of the whole space, they are associated with the root in Fig. 3b; for the skyline scopes of c and d, their minimal enclosing nodes are N_1 and N_{21}, respectively. Algorithm 1 summarizes the procedure of constructing the skyline scope index.

Algorithm 1. Construction of skyline scope index
1: **for** each object $o \in O$ **do**
2: compute its non-spatial dominator objects Dom(o);
3: **if** $Dom(o) = \emptyset$ **then**
4: SS(o) = the whole space;
5: **else**
6: SS(o) = o's Voronoi cell in $\{o\} \cup \mathrm{Dom}(o)$;
7: build an MX-CIF quadtree over the skyline scopes of all objects;

The search over the index is straightforward. Given a query range R, we want to find out the skyline scopes that have intersection with R. Thus, we recursively traverse the MX-CIF quadtree from the root all the way down to the leaf nodes. For any index node whose corresponding subspace intersects with R, the skyline scope of every associated object is retrieved and checked. If it has intersection with R, the corresponding object is added to the skyline result set.

4.2 Incremental Skyline Scope Computation

The skyline scopes may change drastically as the objects move. To avoid recomputing all skyline scopes from scratch, in this section we introduce an incremental algorithm that efficiently updates skyline scopes when the objects move.[3] As we will explain soon, the incremental algorithm can also be used to compute the initial skyline scopes (i.e., Line 6 of Algorithm 1 can be modified to incrementally compute the skyline scopes).

3. Note that the existing update algorithms on Voronoi cell are all based on Voronoi diagram and hence are not suitable to our scenario where skyline scopes are computed individually (with different object subsets).

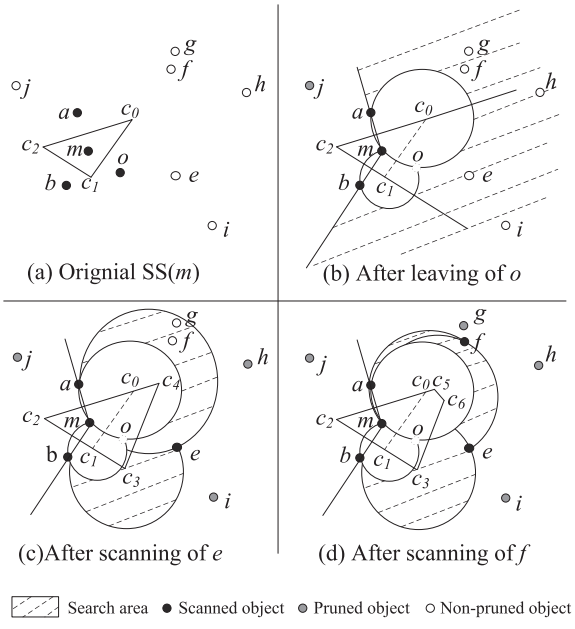


Fig. 4. Recomputation of skyline scope.

Any movement of an object can be decomposed into two operations: *leaving* the data set first and *joining* again. The solution to the object's joining is straightforward. Suppose that an object o joins the data set. Its skyline scope with the current data set can be computed and inserted into the index tree. Additionally, for each o's nonspatial dominance object m, we should check whether PerBis($\overline{mo}$) intersects with the original skyline scope of m. If it does, the new skyline scope of m should be updated to the intersection area of $H(m,o)$ and the original skyline scope of m. Thus, the initial skyline scopes in Algorithm 1 can be also incrementally computed as if they join the data set one by one.

The solution to the object o's leaving is more complex. The first step is deleting the skyline scope of o from the index tree. Then, for each o's nonspatial dominance object m, we should check whether o has contributed to the boundary of m's skyline scope: if it has not, the skyline scope of m does not change. Otherwise, it should be recomputed.

Fig. 4 illustrates such a recomputation process. We assume that the original skyline scope of m is shown as the triangle $<c_0, c_1, c_2>$ in Fig. 4a and then the object o leaves. Without the bounding of PerBis($\overline{mo}$) (i.e., $\overline{c_0c_1}$), m's skyline scope will be temporarily extended and bounded by half-lines $\overrightarrow{c_2c_0}$ and $\overrightarrow{c_2c_1}$, as shown in Fig. 4b. It is an *intermediate* skyline scope. The essence of skyline scope recomputation is finding the objects in Dom(m) whose perpendicular bisectors with m "cut" the intermediate skyline scope and form the final new skyline scope. However, not every object in Dom(m) has the chance of cutting and some can be pruned to reduce the computation cost. To facilitate the pruning, we introduce a notion of ***search area*** in the recomputation, which is defined as an area that any object located outside has no chance to contribute to the new skyline scope and can be pruned from further consideration. For example, the search area is marked by dashed lines in Figs. 4b, 4c, and 4d. And then we repeatedly choose the objects in the search area (e.g., object e in Fig. 4c and object f in Fig. 4d) from near to far to perform such cutting. After each step of cutting, the intermediate skyline scope may shrink. In Fig. 4c, it becomes polygon $\langle c_2, c_0, c_4, c_3, c_1\rangle$ and in Fig. 4d it further becomes polygon $\langle c_2, c_0, c_5, c_6, c_3, c_1\rangle$. The search area will also shrink accordingly. When no more object is found in the search area, the final skyline scope is obtained.

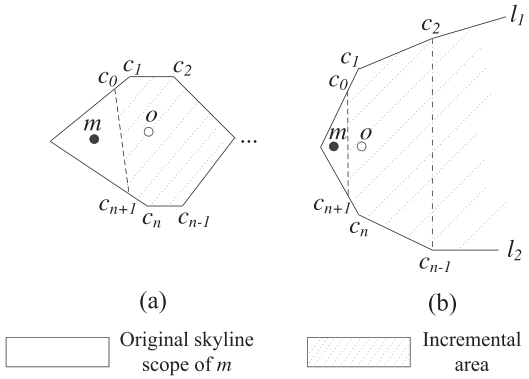


Fig. 5. Intermediate skyline scopes.

In the following, we explain in detail how to update the search area from an intermediate skyline scope. An intermediate skyline scope may be a close or an open area, as shown in Figs. 5a and 5b, respectively. In both cases, it can be divided into two parts: the original skyline scope of m and the other area that shares its border PerBis($\overline{mo}$) (marked by dotted lines in Figs. 5a and 5b). We call the second part an *incremental area*. With this notion, if an object has a chance to contribute to the final new skyline scope of m, its perpendicular bisector with m should only intersect with the incremental area, and not with the original skyline scope of m. This principle lays the foundation for object pruning. If an object does contribute to the new skyline scope, the shape of new search area can be obtained by Theorem 1.

Theorem 1. *An intermediate skyline scope has two possible shapes, the close one (Fig. 5a) and the open one (Fig. 5b). Without loss of generality, the incremental area of the close one is represented by* $\langle c_0, c_1, \ldots, c_n, c_{n+1}\rangle$, *where* $c_i (0 \le i \le n+1)$ *is a vertex of the incremental area. Then, the search area of this case is* $\bigcup_{i=1}^{n} Cir(c_i, m) - (Cir(c_0, m) \cup Cir(c_{n+1}, m))$, *where* $Cir(o, a)$ *represents the interior area of a circle centered at* o *and with a radius of* $|oa|$ *(see Fig. 6a).*

Similarly, without loss of generality, the incremental area of the open one is represented by $\langle c_0, c_1, \ldots, c_i, l_1, l_2, c_{i+1}, \ldots, c_n, c_{n+1}\rangle$, where $c_i (0 \le i \le n+1)$ is a vertex of the incremental area and l_1 and l_2 are the two half-lines on the boundary. Then, the search area of this case is $\bigcup_{i=1}^{n} Cir(c_i, m) \cup \partial(m, l_1) \cup \partial(m, l_2) - (Cir(c_0, m) \cup Cir(c_{n+1}, m))$ *(see Fig. 6b)*, where $\partial(o, r)$ is defined below:

Definition 7. $(\partial(o, r))$ *Given a point* o *and a half-line* r *starting from point* a, *let* o' *denote the symmetrical point of* o *with respect to* r, *and* $H_{oo'}(r)$ *denote the half plane on* r*'s infinite-direction side of Line*($\overline{oo'}$) *(i.e., the right side of Line*($\overline{oo'}$) *in Fig. 7).* $\partial(o, r) = H_{oo'}(r) - Cir(a, o)$, *no matter whether Line*($\overline{oo'}$) *intersects with* r, *as shown in Fig. 7. Note that for any point* p *in* $\partial(o, r)$, *PerBis*($\overline{po}$) *will intersect with* r.

The proof of this theorem is given in Appendix B, available in the online supplemental material.

With the notion of search area, we present the index update algorithm in Algorithm 2. In the object leaving

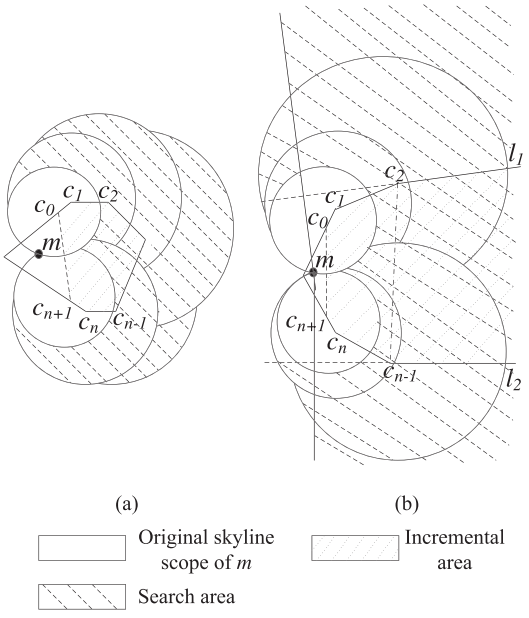


Fig. 6. Illustration of search area for the two cases.

scenario, we check the nonspatial dominator objects from near to far (Lines 11 and 13), because the nearer dominator objects are more likely to contribute to the boundary of the final skyline scope. Note that the search area is the union of some circular areas (see Fig. 6). We also set a stop condition—the search bound $maxDist$—for the cutting. This bound is set to two times of the distance between m and the farthest vertex of the incremental area (Lines 15-19).

Algorithm 2. Update of skyline scope index for I-SKY

```
1: if object o joins then
2:     compute SS(o) and insert it into the index tree;
3:     for each o's non-spatial dominance object m do
4:         if PerBis(m̄o) intersects with SS(m) then
5:             SS(m) = SS(m) ∩ H(m, o);
6:             replace SS(m) in the index tree;
7: if object o leaves then
8:     remove SS(o) from the index tree;
9:     for each o's non-spatial dominance object m do
10:        if o has contributed to m's skyline scope then
11:            sort m's non-spatial dominator objects into
               DomQueue ordered by their distances to m;
12:            initialize the search area and incremental area
               by Theorem 1;
13:            while DomQueue is not empty do
14:                x ← DomQueue.pop();
15:                if the incremental area is open then
16:                    maxDist ← ∞;
17:                else maxDist ← the distance from the
                   farthest vertex of the incremental area to m;
18:                if dist(x, m) > 2 × maxDist then
19:                    break;
20:                if x is in the search area then
21:                    cut the intermediate skyline scope by
                       PerBis(x̄m);
22:                update the incremental area and search area
                   by Theorem 1;
23:        replace SS(m) in the index tree;
```

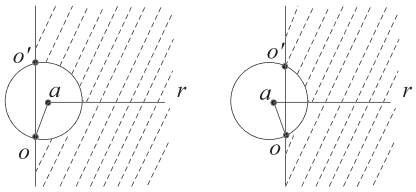


Fig. 7. $\partial(o, r)$ (shaded area).

Considering the complexity of handling object leaving in Algorithm 2, the most complex operations are sorting $DomQueue$ (Line 11) and cutting the intermediate skyline scope (Lines 20-21). The time complexity of the sorting operation is $O(|Dom(m)| \cdot \log|Dom(m)|)$, where $|Dom(m)|$ is the number of m's nonspatial dominator objects.[4] In each cutting operation, the new perpendicular bisector will be intersected with the existing edges of intermediate skyline scope and the number of such edges will not exceed that of the final skyline scope (denoted by $|SS(m)|$). Since $|SS(m)|$ is usually a very small constant (< 10), the time complexity of cutting operations for all m's nonspatial dominator objects is $O(|Dom(m)|)$. Thus, the worse case time complexity of handling object leaving in Algorithm 2 is $O(|Dom'(o)| \cdot |Dom(m)| \cdot \log|Dom(m)|)$, where $|Dom'(o)|$ is the number of o's nonspatial dominance objects. As both $|Dom(o)|$ and $|Dom'(o)|$ can be estimated as $|O|/2^d$, where $|O|$ is the data set size and d is the dimensionality of nonspatial attributes, this complexity can be rewritten as $O(|O|^2 \log|O|/4^d)$.

5 Nonindex RSQ Algorithm: N-SKY

I-SKY indexes the skyline scopes, which accelerates the processing of range-based skyline queries. However, the maintenance cost of the skyline scope index would be high if the objects update their locations frequently. Although we have developed an incremental index update algorithm, frequent object location updates may still cause many unnecessary index updating operations on the skyline scopes that are not being queried.

To avoid the high update cost of I-SKY for the scenarios where the objects move frequently and fast, in this section we propose a nonindex algorithm N-SKY. First, we prove that a range-based skyline query can be reduced to several segment-based skyline queries. Then, we present an efficient algorithm for processing segment-based skyline queries.

5.1 Reducing RSQ to SSQs

According to the definition of RSQ (Definition 4 in Section 3.1), any object $o \in O$ located inside the query range R must be a result object of the query, i.e., a member of PSQ(o, O), since o is not spatially dominated by any other object with respect to the query point occupied by o. Such objects can be obtained by a range query of R. In the following, we focus on finding the set of result objects located outside R.

We start by defining the segment-based skyline query.

4. In the actual implementation, we employ a grid-based index to reduce the cost, where we only need to sort the objects in the grid cells that intersect with the search area in an incremental manner.

Definition 8 (Segment-based Skyline Query). *Given a data set O and a line segment l, the segment-based skyline returns a superset of objects that appear in the skyline set of any point on l. Formally, it is denoted as $SSQ(l, O)$, and $SSQ(l, O) = \{o | \exists\ q \in l, o \in PSQ(q, O)\}$.*

An important observation is that, any result object of $RSQ(R, O)$ must be a member of the skyline set of some point on the boundary of R. Theorem 2 proves the correctness of this observation, which can then be used to reduce the RSQ problem to SSQ problem.

Theorem 2. *If an object o is a result of $RSQ(R, O)$ and o is outside R, o must be a member of the skyline set w.r.t. some query point on the boundary of R. Formally, this theorem can be expressed as: $o \in RSQ(R, O) \Rightarrow \exists q, q \in Boundary(R) \wedge o \in PSQ(q, O)$.*

The proof of this theorem is given in Appendix C, available in the online supplemental material.

By Theorem 2, the RSQ problem can be reduced to a range query plus several SSQs based on the boundary of the query range. As the query range is rectangular, its boundary consists of four line segments. This reduction effectively decreases the dimensionality of the problem from 2D to 1D, as stated in Theorem 3.

Theorem 3. *Given a data set O and a rectangular range R whose boundary consists of four line segments l_1 through l_4, the RSQ problem can be reduced to a range query of R and four SSQs of $SSQ(l_1, O)$, $SSQ(l_2, O)$, $SSQ(l_3, O)$, and $SSQ(l_4, O)$.*

The range query of R can be easily evaluated using a traditional method. In the following, we discuss how to efficiently compute SSQ(l, O) given a line segment l and a data set O.

5.2 SSQ Algorithm

5.2.1 Overview

SSQ(l, O) consists of the skyline of each point q on the line segment l. The basic idea of evaluating an SSQ is to move q from l's left end to its right end, and look for positions where the skyline set changes. The changes can be divided into two cases: an object enters the skyline set and an object leaves the skyline set. Intuitively, the former case means no any other object dominates the entering object any longer, and the latter case means another object is about to dominate the leaving object. The corresponding points on the line segment are called *enter-in* and *leave-out* points, respectively. To efficiently compute the enter-in point and leave-out point of each object o, we observe a prerequisite as stated in Lemma 1.

Lemma 1. *Regarding q as the query point, if there is no current skyline object $o' \in$ PSQ(q, O) dominating object o (both nonspatially and spatially), o will not be dominated by any other object in O, and hence should enter the skyline set; on the other hand, if there is some skyline object $o' \in$ PSQ(q, O) dominating object o, o is no longer a skyline member of q.*

However, the challenge is that the skyline set will change as the query point moves. As a result, it is not easy to precompute the enter-in points and leave-out points of all objects in advance. To resolve this issue, we maintain the candidate enter-in points and leave-out points for each nonskyline object and skyline object, respectively, based on the current skyline set. As the skyline set changes, we update the enter-in and leave-out points that are affected, and then choose the next nearest point for updating the skyline set. In the following, we give the details of this SSQ algorithm.

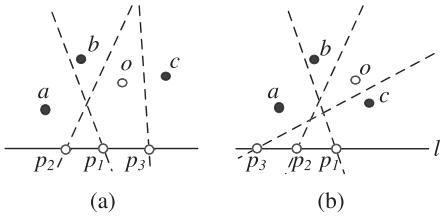


Fig. 8. Illustration of enter-in point.

5.2.2 Data Structures of the SSQ Algorithm

To facilitate our discussion, we introduce a coordinate system along the line segment. We use the name of a point and its coordinate interchangeably, i.e., a means both the point a and its coordinate. Hence, $a < b$ (resp. $a > b$) means a lies to the left (resp. right) of b.

For each object $o \in O$ and a point p on the line segment, if no object in the skyline set PSQ(p, O) dominates o w.r.t. p, p is called a ***free point*** of o. For each nonskyline object o, the left-most free point of o is called o's ***enter-in point*** (denoted as $In[o]$). For each skyline object o, the right-most free point of o is called o's ***leave-out point*** (denoted as $Out[o]$). Meanwhile, we say o is the corresponding object of its enter-in point or leave-out point. Obviously, as the skyline set changes, the enter-in and leave-out points of some objects will change as well.

Fig. 8 illustrates a method to compute the enter-in and leave-out points, where l is a boundary line of the query region. We first find out all nonspatial dominator objects of o in the skyline set, e.g., a, b, and c as shown in Fig. 8a. According to the relationship between their projections on the line segment l and o's projection, these objects can be divided into two subsets. If the projection of an object lies to the left of o's projection, it is called o's ***left nonspatial dominator***, e.g., a and b. Meanwhile, we say a and b ***left dominate*** o. For any object o and its nonspatial dominator object a, the intersection point of PerBis($\overline{ao}$) and Line(l) is called $CSP\langle a, o, l\rangle$. The CSP of o's left nonspatial dominator and o is called o's ***left non-spatial dominate point (LDP)***, e.g., points p_1 and p_2. On the other hand, if the projection of an object lies to the right of o's projection, it is called o's right nonspatial dominator, e.g., c. Meanwhile, we say c ***right dominates*** o. The CSP of o's right dominator and o is called o's ***right nonspatial dominate point (RDP)***, e.g., point p_3. To get the enter-in point $In[o]$, we find out the rightmost LDP (RM-LDP) and the leftmost RDP (LM-RDP). If the RM-LDP lies on the left side of the LM-RDP (e.g., point p_1 versus p_3 in Fig. 8a), the RM-LDP is obviously the point of $In[o]$, as starting from this point to the LM-RDP, o is not spatially dominated by any of its other nonspatial dominator object. Otherwise, if the RM-LDP lies on the right side of the LM-RDP (e.g., point p_1 versus p_3 in Fig. 8b), it means that o will always be spatially dominated by some of

its nonspatial dominator objects. Hence, o would never enter the skyline set and thus $In[o]$ is left as empty. Similarly, $Out[o]$ for a skyline member o can be obtained from the LM-RDP.

We introduce a priority list *UpdateQueue* to store the enter-in and leave-out points from left to right. As the query point moves, we repeatedly pop up the first element of *UpdateQueue* and execute the corresponding enter-in or leave-out operation. As the skyline set changes, the enter-in and leave-out points of some object(s) will change as well and the corresponding elements in *UpdateQueue* should be adjusted. As will be proved in Theorem 4, the leave-out operation will not affect the enter-in or leave-out point of any other object. The enter-in operation of an object o will affect the enter-in and leave-out points of o's nonspatial dominance objects only. Specifically, the enter-in point(s) of some object(s) may move rightwards and the leave-out point(s) of some object(s) may move leftwards. Thus, the corresponding elements in *UpdateQueue* must change accordingly. In particular, the new enter-in point of some object may lie to the right of its the leftmost RDP. Since in this case it has no chance for the object to enter the skyline set any longer, such an enter-in point should be removed from *UpdateQueue*. Algorithm 3 gives a formal description for the dynamic update of *UpdateQueue* when a new object o enters the skyline set. The following theorem justifies the leave-out operation and thus the correctness of this algorithm.

Algorithm 3. Dynamic update of *UpdateQueue* (when object o enters the skyline set)

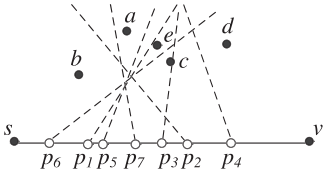
1: $NON\_SKY \leftarrow \{m | o \triangleleft m \wedge m \notin \text{PSQ}(In[o], O)\}$;
2: **for** each object m in NON_SKY **do**
3: **if** o left dominates m at p and p lies on the right side of $In[m]$ **then**
4: $In[m] \leftarrow p$;
5: update $In[m]$ in *UpdateQueue*;
6: **if** o right dominates m at p and p lies on the left side of $In[m]$ **then**
7: remove $In[m]$ from *UpdateQueue*;
8: $SKY \leftarrow \{m | o \triangleleft m \wedge m \in \text{PSQ}(In[o], O)\}$;
9: **for** each object m in SKY **do**
10: **if** o right dominates m at p and p lies on the left side of $Out[m]$ **then**
11: $Out[m] \leftarrow p$;
12: update $Out[m]$ in *UpdateQueue*;
13: compute $Out[o]$ based on the skyline set;
14: insert $Out[o]$ into *UpdateQueue*;

Theorem 4. *In the SSQ(l, O) problem, if an object $m \in O$ leaves the skyline set at point q on the line segment l, the enter-in or leave-out point of other objects in O will not be affected.*

The proof of this theorem is given in Appendix D, available in the online supplemental material.

5.2.3 SSQ Algorithm

With the data structures of enter-in points, leave-out points, and *UpdateQueue*, Algorithm 4 gives the pseudocode of the complete SSQ algorithm. We illustrate the algorithm using an example shown in Fig. 9. Fig. 9a shows the positions of five objects a-e and the line segment l, and Fig. 9b shows the two nonspatial attributes of each object. Assuming that lower attribute values are preferred, we can get seven nonspatial dominance pairs, i.e., $a \triangleleft c, a \triangleleft b, a \triangleleft d, c \triangleleft d, a \triangleleft e$, $c \triangleleft e, c \triangleleft b$. Their corresponding CSPs are p_1-p_7 as shown in Fig. 9a.



p_1: CSP<a,c,l> p_2: CSP<a,b,l> p_3: CSP<a,d,l>
p_4: CSP<c,d,l> p_5: CSP<a,e,l> p_6: CSP<c,e,l>
p_7: CSP<c,b,l>

(a)

	A1	A2
a	1	0
b	3	2
c	1	2
d	1	4
e	2	3

(b)

(1) Query point q is initialized as the starting point s:

Skyline Set	{a, b}
Enter-in Point	$In[c]=p_1$, $In[e]=p_5$, $In[d]=p_3$
Leave-out Point	$Out[b]=p_2$
UpdateQueue	<$In[c]=p_1$, $In[e]=p_5$, $In[d]=p_3$, $Out[b]=p_2$>

(2) After q passes p_1, c enters the skyline set:

Skyline Set	{a, b, **c**}
Enter-in Point	~~$In[e]=p_5$~~, $In[d]=$ ~~p_3~~ **p_4**
Leave-out Point	$Out[b]=$ ~~p_2~~ **p_7**
UpdateQueue	<$Out[b]=$**p_7**, $In[d]=$**p_4**>

(3) After q passes p_7, b leaves the skyline set:

Skyline Set	{a, ~~b~~, c}
Enter-in Point	$In[d]=p_4$
Leave-out Point	-
UpdateQueue	<$In[d]=p_4$>

(4) After q passes p_4, d enters into the skyline set

Skyline Set	{a, c, **d**}
Enter-in Point	-
Leave-out Point	-
UpdateQueue	-

(c)

Fig. 9. Illustration of SSQ algorithm.

Algorithm 4. Algorithm for SSQ(l, O)

1: initialize $PSQ \leftarrow$ PSQ(s, O), where s is the starting point of l;
2: **while** *UpdateQueue* is not empty **do**
3: $next\_q \leftarrow$ leftmost point of *UpdateQueue*;
4: **if** $next\_q$ is not within l **then**
5: break;
6: $o \leftarrow$ corresponding object of $next\_q$;
7: remove $next\_q$ from *UpdateQueue*;
8: **if** $next\_q$ is a leave-out point **then**
9: $PSQ \leftarrow PSQ - \{o\}$;
10: **if** $next\_q$ is an enter-in point **then**
11: $PSQ \leftarrow PSQ \cup \{o\}$;
12: invoke Algorithm 3 to update *UpdateQueue*;

Initially, the skyline set at the starting point s is computed by a point-based skyline algorithm (e.g., BBS [19]) or reused from the results of an adjacent line segment if available. The result set for this example is $\{a, b\}$ (see Fig. 9c). Then, the enter-in point and leave-out point of each object is computed by the method presented in the last section, i.e., $In[c] = p_1$, $In[e] = p_5$, and $In[d] = p_3$. Since $a \lhd b$, $Out[b] = p_2$. As no object dominates a, a will never leave the skyline set, and hence there is no $Out[a]$. After sorting these enter-in and leave-out points from left to right, *UpdateQueue* is initialized as $\langle In[c], In[e], In[d], Out[b]\rangle$. Next, we pop up the first element from *UpdateQueue* and perform the corresponding update of the skyline set, that is, object c enters the skyline set when the query point q passes p_1. Because objects b, d, and e are c's nonspatial dominance objects, we should check whether their enter-in and leave-out points will be affected. Since c right dominates e w.r.t. point p_6 and p_6 lies to the left of e's enter-in point p_5, e has no chance to enter the skyline set. Its enter-in point should be removed from *UpdateQueue*. Since c left dominates d w.r.t. point p_4, which lies to the right of d's original enter-in point p_3, $In[d]$ should be updated as p_4. Since c right dominates b w.r.t. p_7, which lies to the left of b's original leave-out point p_2, $Out[b]$ should be updated as p_7. After updating the enter-in and leave-out points, two elements remain in *UpdateQueue*. After that, they are popped up and the corresponding updates are performed, that is, b leaves the skyline set and d enters the skyline set. Finally, *UpdateQueue* becomes empty and the algorithm is terminated.

The union of the skyline sets generated during query processing forms the final results of SSQ(l, O). In the above example, SSQ(l, O) can be obtained as $\{a, b, c, d\}$. By the SSQ algorithm, we can also record the *sky_interval* of each SSQ result object, i.e., the part of the line segment in which the object is in the skyline set.

The time complexity of the SSQ algorithm (Algorithm 4) is analyzed as follows: For each result object of SSQ(l, O), Algorithm 3 is invoked once. The size of the SSQ result set can be estimated as $O((\ln|O|)^d)$, where d is the number of nonspatial attributes [1]. Algorithm 3 needs to handle each nonspatial dominance object of the new skyline member. The average number of such dominance objects is $O(|O|/2^d)$. For each nonspatial dominance object, the possible operations are modifying their enter-in and leave-out points and updating their positions in *UpdateQueue*. The complexity of the former is linear and that of the latter is $O(\log|O|)$. As a result, the overall complexity of Algorithm 4 is $O((\ln|O|)^d|O|\log|O|/2^d)$.

6 Extensions

In this section, we extend the RSQ problem to the probabilistic RSQ query in Section 6.1 and the continuous RSQ query in Section 6.2.

6.1 Probabilistic RSQ Problem

As the RSQ query considers a spatial range as the query input, instead of a location point, the result set size might be too large for the user in some applications. To address this problem, we propose a concept of *skyline probability* to rank the skyline results of the RSQ query.

Definition 9 (Skyline Probability). *For a range-based skyline query RSQ(R, O) and a result object $o \in O$, the skyline probability of o is defined as the portion (in percentage) of the query range R in which any point q satisfies that o is a skyline result of PSQ(q, O).*

Thus, we can reduce the result set by returning only the top-k results with the highest skyline probabilities, which is termed as ***probabilistic top-k RSQ***. To answer such queries, we first consider how to extend the I-SKY algorithm. According to Definitions 5 and 9, the skyline probability of an object o can be obtained by the intersection area of the query range R and o's skyline scope $SS(o)$ (denoted as $inter\_area(R, o)$). Recall that in the index tree of skyline scopes (Section 4.1), the subspace of each index node serves as a minimal bound of the associated skyline scopes. In other words, if o's skyline scope is associated with an index node n, we must have $inter\_area(R, o) \subseteq inter\_area(R, n)$. Based on this observation, we develop Algorithm 5 to prune the search space for top-k query processing. We dynamically maintain a priority queue $\mathcal{H}$ for index nodes and data objects, ordered by their $inter\_area$ values, while traversing the index tree (Lines 3-4 and 7-8). An object enqueued from $\mathcal{H}$ may become a top-k result (Lines 9-13). The top-k query processing proceeds until we encounter an index node or an object whose intersection area is no larger than the kth result obtained so far (Line 6). This is because all the remaining objects could not have an intersection area (skyline probability) larger than the current kth result. Thus, they can be pruned from consideration as top-k results.

Algorithm 5. Probabilistic top-k RSQ processing (I-SKY)

```
1: initialize the top-k result set S ← ∅
2: denote the query range as R, the k-th result in S as s_k
3: insert the root of skyline-scope index iRoot into a
   priority queue H, sorted in descending order of
   inter_area
4: while H is not empty do
5:   pop up the top element e from H
6:   if |S| = k and inter_area(R, e) ≤ inter_area(R, s_k)
     then break;
7:   if e is an index node then
8:     insert e's children and associated objects into H
9:   if e is a data object then
10:    if |S| < k then
11:      insert e into S and compute s_k if available
12:    else if inter_area(R, e) > inter_area(R, s_k) then
13:      replace s_k with e in S and recompute s_k
```

Next, we discuss how to extend the N-SKY algorithm for processing the probabilistic top-k RSQ query. In Theorem 3, we have showed that the RSQ problem can be reduced to a range query of R and four SSQs of the range boundary. Recall in the proposed SSQ algorithm (Section 5.2), the skyline results of a line segment l are obtained by first computing the skyline set PSQ(s, O) of l's starting point s and then dynamically updating it by moving the query point along l to its ending point. A significant amount of computation lies in updating the skyline set based on the enter-in/leave-out points of the objects when the query point moves. To prune the computation for probabilistic top-k processing, we prove

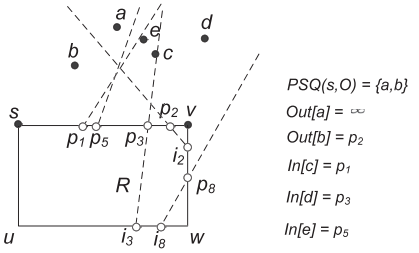


Fig. 10. Illustration of *max-prob* bound.

in Theorem 5 that if an object is a final SSQ result, it must be a skyline object in PSQ(s, O) or its initial enter-in point must lie on l. Thus, we can quickly obtain a candidate set of skyline results through the initial processing.

Theorem 5. *For SSQ(l, O), if an object is a final SSQ result, it must be in PSQ(s, O) or its initial enter-in point must lie on l, where s is the starting point of l.*

The proof of this theorem is given in Appendix E, available in the online supplemental material.

Then, we develop an upper bound of skyline probability for each object in the candidate set in order to prioritize further processing. For a skyline object o in PSQ(s, O), its leave-out point $Out[o]$ can be computed as discussed in Section 5.2.2. Denote o's right nonspatial dominator object corresponding to $Out[o]$ as o'. The perpendicular bisector PerBis($\overline{oo'}$) divides the whole space into two open-half planes. Clearly, o will not be a skyline object for any query point q located in the open-half plane containing o', since o will be (at least) dominated by o' w.r.t. such q. Thus, we can derive the upper bound of o's skyline probability as *max-prob*$(o) = Area(H(o, o') \cap R)/Area(R)$, where $H(o, o')$ is the open-half plane containing o cut by PerBis($\overline{oo'}$). For example, consider the segment $\overline{sv}$ in Fig. 10, where b is an initial skyline object and $Out[b] = p_2$; hence, *max_prob*(b) is the portion of the area enclosed by the polygon $\langle s, p_2, i_2, w, u\rangle$.

Similarly, for a nonskyline object o, we can compute its enter-in point $In[o]$. Denote its left nonspatial dominator object (an initial skyline result) corresponding to $In[o]$ as o''. It is possible for this object o to become a skyline result only if the query point q is located in the open-half plane containing o cut by PerBis($\overline{oo''}$), since otherwise o will be dominated by o''. Thus, the upper bound of o's skyline probability can be derived as *max-prob*$(o) = Area(H(o, o'') \cap R)/Area(R)$. In the example of Fig. 10, for the nonskyline object d, *max_prob*(d) is the portion of the area enclosed by the polygon $\langle p_3, v, w, i_3\rangle$, since $In[d] = p_3$.

The *max-prob* bound can be further tightened when we consider more subsequent line segments. In Fig. 10, suppose d is an initial skyline result for the segment $\overline{vw}$ and $Out[d] = p_8$. Thus, *max_prob*(d) can be further reduced to the portion of the polygon $\langle p_3, v, p_8, i_8, i_3\rangle$ when $\overline{vw}$ is considered.

Based on the notion of *max-prob*, we develop the probabilistic top-k RSQ algorithm for N-SKY in Algorithm 6. In the first stage, we compute the skyline results of range query R (Line 3). After getting their skyline probabilities, the top-k result set S is initialized (Line 4). Next, we consider the four SSQ queries and compute a candidate result set *SKY_CAND* (Lines 5-10). For each object in *SKY_CAND*, we compute its *max-prob* bound and insert it into a priority queue $\mathcal{H}$ in the descending order of *max-prob* (Lines 11-12). Finally, we iteratively compute the skyline probability for each object in $\mathcal{H}$ and dynamically update S, until the kth object obtained so far has a skyline probability higher than the *max-prob* bound of the next object (Lines 13-19).

To obtain the skyline probability for a candidate result (Lines 4 and 18), we need to compute its skyline scope at runtime. A simple solution is to use the generic method described in Section 4.1. We remark that this solution can be further optimized by pruning the candidate objects based on the query range during skyline scope computation, as detailed in Appendix F, available in the online supplemental material.

Algorithm 6. Probabilistic top-k RSQ processing (N-SKY)

```
1: initialize the top-k result set S ← ∅
2: denote the query range as R, the k-th result in S as s_k
3: RQ ← the result set of range query R over data set O
4: compute the skyline probability of each object in RQ
   and insert the top-k objects into S
5: initialize the candidate skyline set SKY_CAND ← ∅
6: for each line segment l of R do
7:    SKY_CAND ← SKY_CAND ∪ PSQ(s, O), where s is
      the starting point of l
8:    for each object m not in PSQ(s, O) do
9:       if In[m] ∈ l then
10:         SKY_CAND ← SKY_CAND ∪{m}
11: for each object m in SKY_CAND but not in RQ do
12:    compute m's max-prob bound and insert m into
       priority queue H in the descending order of max-prob
13: while H is not empty do
14:    pop up the top element e from H
15:    if |S| = k and s_k's skyline probability ≥ max-prob(e)
       then break
16:    if |S| < k then
17:       insert e into S and compute s_k if available
18:    else if e's skyline probability > s_k's then
19:       replace s_k with e in S and recompute s_k
```

6.2 Continuous RSQ Problem

So far we have studied the one-shot RSQ query. However, in location-based services, the user may sometimes prefer that the query is issued once whereas its result is monitored continuously. For example, a driver may issue an RSQ query "finding nearby gas stations with cheap gas prices" on her route from one place to another; a tourist may issue an RSQ query "monitoring nearby taxis with low volatility" while she is walking on a busy street. In this section, we study the continuous RSQ problem which computes the RSQ results for a moving query. In addition, an incremental C-RSQ algorithm for moving objects is presented in Appendix H, available in the online supplemental material.

The C-RSQ problem is defined as follows:

Definition 10 (Continuous Range-Based Skyline Query (C-RSQ)). *Given a data set O and a linearly moving query range from R to R' (see Fig. 11), where we assume the moving path is known in advance, the C-RSQ query returns the set of objects that are results of RSQs for some query range between R and R', together with the valid scope of each result object, denoting the duration when the object is a skyline result.*

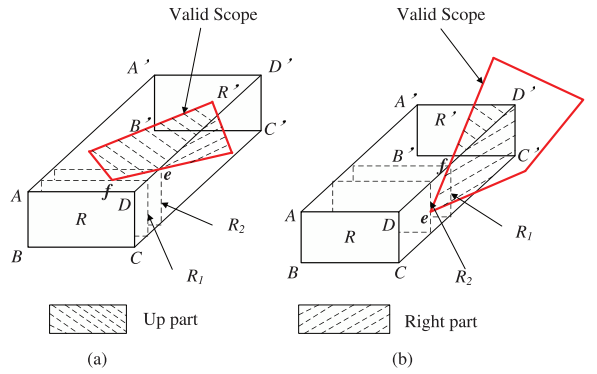


Fig. 11. Continuous RSQ problem.

Without loss of generality, we assume that the query range moves from bottom left to upper right. If a range R_1 is nearer to the starting range than another range R_2, we say R_1 is *earlier* than R_2 and R_2 is *later* than R_1. For any result object of C-RSQ, the earliest range and latest range in its valid scope is called start RSQ range (SRR) and end RSQ range (ERR), respectively.

To process C-RSQ, we first compute the possible results of C-RSQ by the one-shot RSQ algorithm. Specifically, this superset is the RSQ result of hexagon $ABCC'D'A'$. To identify the valid scope of each result object, the intersection areas of their skyline scopes and the hexagon $ABCC'D'A'$ are computed. Then, the SRR of an object is when the moving query range touches the skyline scope of the object. Obviously, if the skyline scope intersects with R, the initial range $ABCD$, then the SSR will be R. Otherwise, the derivation of SSR can be divided into two cases as shown in Fig. 11. In Fig. 11a, the skyline scope will first reach the moving edge of $\overline{AD}$, whereas in Fig. 11b, it will first reach the moving edge of $\overline{CD}$. The skyline scope is then split by the diamond $ADD'A'$ or $DCC'D'$. The two split parts are called the *upper part* and the *right part* of the skyline scope. The lowest point of the upper part is the earliest intersection point of moving edge $\overline{AD}$ and the skyline scope, which is f in Fig. 11. Similarly, the leftmost point of the right part is the earliest intersection point of moving edge $\overline{CD}$ and the skyline scope, which is e in Fig. 11. The range whose edge $\overline{AD}$ contains f is denoted as R_1. The range whose edge $\overline{CD}$ contains e is denoted R_2. Then, SRR is then the earlier range of these two. The ERR can be obtained by a similar method. SRR and ERR determine the valid scope of an object. Theorem 6 proves the correctness of the above method.

Theorem 6. *The valid scope computed by the above method is correct.*

The theorem is proved in Appendix G, available in the online supplemental material.

TABLE 2
Index Size and Index Construction Time

Dataset cardinality	10K	100K	1000K
Index size	25 MB	238 MB	2 GB
Divide-and-conquer	4.5 mins	1.5 hrs	23 hrs
Incremental algorithm	1.5 mins	22 mins	5.3 hrs

TABLE 3
Parameter Settings

Parameter	Value Range	Default Setting
Dataset cardinality	[10K, 1,000K]	100K
# non-spatial attributes	2, 4, 6 ,8	2
Side length of query range	[500, 2,000]	2,000 Units
Buffer size	[1%, 20%]	5% of dataset size
Average movement speed	$[2m/s, 20m/s]$	0
Non-spatial attribute values	[0, 100,000]	

7 Performance Evaluation

7.1 Experiment Setup

In this section, we evaluate the performance of our proposed algorithms through simulations. The spatial data set used in the experiments contains 2,249,727 objects representing the centroids of the street segments in California [20]. A subset of these objects are randomly chosen to form the testing data set. The nonspatial attribute values of these objects are synthesized with a uniform distribution in the interval [0, 100,000]. The data space is normalized to a 100,000 Unit $\times$ 100,000 Unit square, where 1 Unit represents about 1 meter. We index the nonspatial attributes of the objects by an R-tree (with a page fanout of 200 and a page occupancy of 70 percent). The page size is 4 K bytes and the size of each object is 320 bytes. We simulate the object movement by following a well-known random way-point model [7]. As for the location update strategy, we adopt a common deviation-based one as follows: an object will update when and only when it is 100 meters away from its last reported location. Obviously, the location update frequency is proportional to the speed of the moving object.

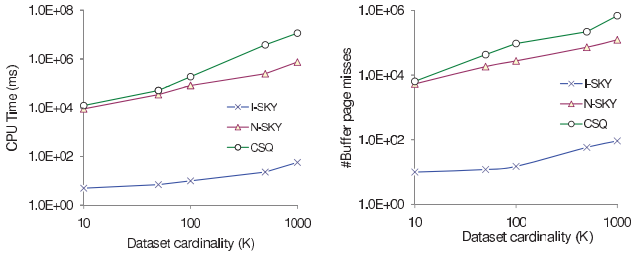
We conducted our experiments on a workstation (Intel Xeon E5440 2.83 GHz CPU) running on Ubuntu Linux Operating System. The simulation codes were written in Java (JDK 1.6). For simplicity, the query ranges are randomly generated as squares. We measure the performance with two metrics: CPU time and I/O cost. In each I/O cost experiment, all objects and indexes are stored on a secondary-storage disk and a buffer in main memory is simulated. The number of buffer misses is an indicator of I/O cost. In each CPU time experiment, all objects and indexes are stored in the memory to exclude the overhead of disk accesses. Each measurement is the average result over 100 queries.

For I-SKY, we assume that the skyline scope of each object has been precomputed and indexed by the MX-CIF quadtree. Table 2 shows the index size and the index construction time under the divide-and-conquer algorithm [2] and the incremental algorithm proposed in Section 4.2. While the index size and construction time are proportional to the data set cardinality, the proposed incremental algorithm clearly outperforms the divide-and-conquer algorithm.

The experiments are divided into two parts: one-shot RSQ and continuous RSQ. The default settings and value ranges of the system parameters are summarized in Table 3.

7.2 One-Shot RSQ Results

For one-shot RSQ queries, we compare the CPU time and I/O cost of three algorithms, i.e., CSQ [12], I-SKY, and N-SKY. We investigate the effect of data set cardinality, dimensionality of nonspatial attributes, query range size, and buffer size. With CSQ, the RSQ problem is also reduced to the SSQ



(a) CPU time cost (b) I/O cost

Fig. 12. Effect of data set cardinality.

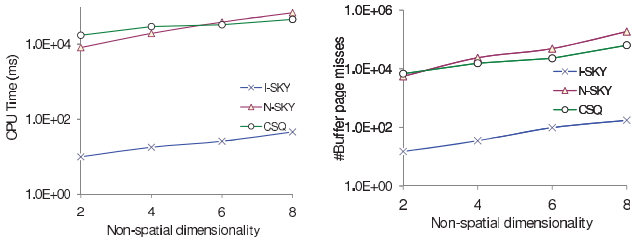
problem as a first step. It then imagines the query point moving along the query range boundary. When the distance relationship of some dominance pairs changes (called ***events*** in [12]), it will check whether there is any object entering or leaving the skyline set. The main difference between CSQ and N-SKY is that N-SKY only needs to check the nonspatial dominance pairs in which the dominator objects appear in the skyline set, which, as will be shown in Figs. 12, 13, and 14, greatly reduces the CPU time and I/O cost.

7.2.1 Effect of Data Set Cardinality

Fig. 12 shows the effect of data set cardinality in terms of the CPU time and I/O cost. We vary the data set cardinality from 10K to 1,000K. Obviously, I-SKY is overwhelmingly better than the other two algorithms because its query cost only depends on the height of the skyline-scope index tree. Assuming that the average disk access time is 10 ms, even for 1,000K objects, the processing time of I-SKY is only about 1 second. CSQ has a poor performance in terms of both the CPU time and I/O cost, especially on a larger data set. This is because CSQ needs to monitor the distance relationship between the skyline objects and their nonspatial dominance objects. With a larger data set, the change of distance relationship happens more frequently, which resulted in a higher cost.

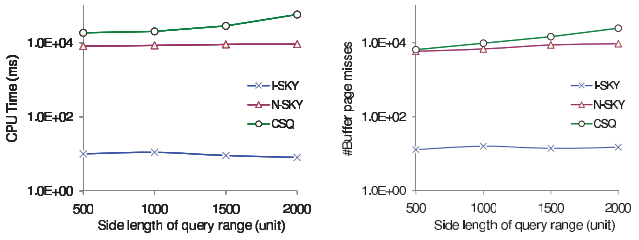
7.2.2 Effect of Nonspatial Dimensionality

In this experiment, we vary the nonspatial dimensionality of the objects from 2 to 8. Fig. 13 shows the result as the nonspatial dimensionality changes. In all cases tested, the performance of I-SKY, again, is much better than the other two algorithms. It is interesting to see that CSQ gradually becomes better than N-SKY as the nonspatial dimensionality grows. This is mainly because that the probability of an object being nonspatially dominated by another object becomes lower as more dimensions are involved, which



(a) CPU time cost (b) I/O cost

Fig. 13. Effect of nonspatial dimensionality.



(a) CPU time cost (b) I/O cost

Fig. 14. Effect of query range size.

effectively reduces the number of events in CSQ as well as its processing cost. On the other hand, the number of SSQ results in N-SKY becomes larger as the nonspatial dimensionality increases and *UpdateQueue* is adjusted more frequently in Algorithm 4. This degrades the performance of N-SKY in high dimensionality.

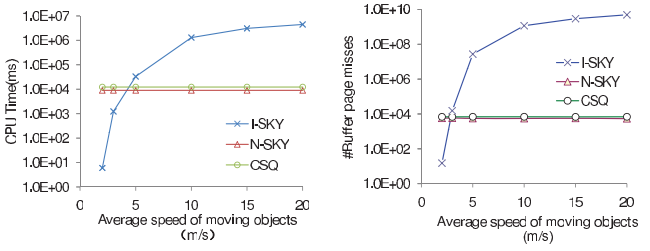
7.2.3 Effect of Query Range Size

In this experiment, we investigate the impact of query range size. The size of query range varies by changing the side length of the query range. We set the side length of query range from 500 to 2,000 Units, while the side length of the whole space is 100,000 Units. As shown in Fig. 14, the performance curves of I-SKY and N-SKY do not fluctuate very much as the query range size increases. For I-SKY, the reason is that the height of the skyline-scope index tree does not change. And for N-SKY, the increase of side length only makes more objects enter the skyline set, but this number is far less than the total number of SSQ results. However, as the side length increases, the performance of CSQ gets worse more rapidly. This is because the number of its dominance events grows linearly with the side length.

7.2.4 Effect of Location Updates

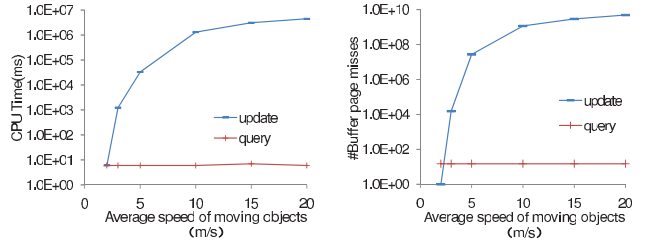
This set of experiments investigates the impact of location updates on the CPU time and I/O cost. We vary the average movement speed from 2 to 20 m/s. The higher the moving speed, the more the location updates of the objects. We compare the buffer misses of three algorithms: CSQ, the I-SKY with incremental index maintenance (Section 4.2), and the N-SKY that computes RSQ results from scratch (Section 5).

As shown in Fig. 15, the performance of N-SKY and CSQ does not change very much because the updates in these two algorithms would not affect the query performance. When the objects move slowly (below 3 m/s), I-SKY is better. As the movement speed increases, more updates are reported and the performance of I-SKY degrades rapidly. N-SKY is better when the movement speed is higher than 5 m/s. To understand why I-SKY degrades significantly as the objects move faster, we recorded the CPU time and buffer misses for processing index updates (including recomputing the skyline scopes) and queries separately (see Fig. 16). We can observe that most of the CPU time and buffer misses are spent on processing index updates, which increases the total cost. In all, N-SKY is more suitable when the location update rate is very high.



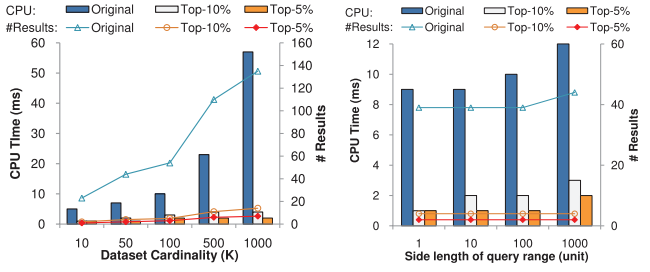
(a) CPU time cost (b) I/O cost

Fig. 15. Effect of movement speed of the objects.



(a) CPU time cost (b) I/O cost

Fig. 16. Costs for processing index updates and queries (I-SKY).



(a) Effect of dataset cardinality (b) Effect of query range size

Fig. 17. Probabilistic RSQ with I-SKY.

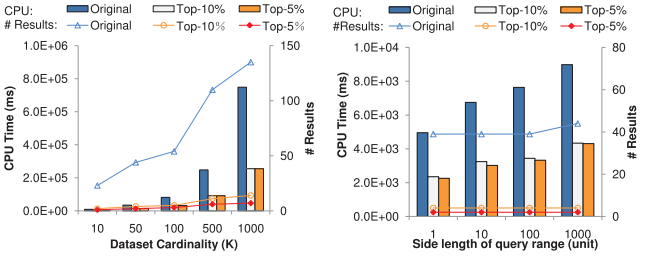
7.2.5 Probabilistic RSQ Results

In this section, we investigate the performance of probabilistic top-k RSQ processing. Fig. 17 shows the CPU time and the number of skyline results for I-SKY, when k is set at 5 and 10 percent of the original result set size. The original I-SKY algorithm (labeled by Original) is also included for comparison. We can see that our probabilistic top-k algorithm (Algorithm 5) saves 70-97 percent of the CPU time of the original I-SKY, while significantly reducing the result set size, thanks to the powerful pruning technique developed. A similar performance trend is also observed for N-SKY. As shown in Fig. 18, our probabilistic top-k N-SKY algorithm (Algorithm 6) saves 50-65 percent of the CPU time of the original N-SKY.

7.3 Continuous RSQ Results

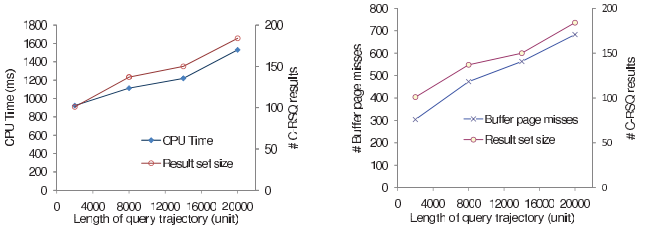
7.3.1 Effect of Query Trajectory Length

We now report the results for C-RSQ queries. We first vary the length of query movement trajectory and measure the CPU time and I/O cost over the default static data set. As shown in Fig. 19, the CPU time and I/O cost increase as the trajectory length grows, but the rate is slower than the growth rate of the trajectory length. To take a deeper look into the C-RSQ algorithm, we further analyze the costs of the two steps involved in this algorithm. The first step is to compute all C-RSQ results,



(a) Effect of dataset cardinality (b) Effect of query range size

Fig. 18. Probabilistic RSQ with N-SKY.



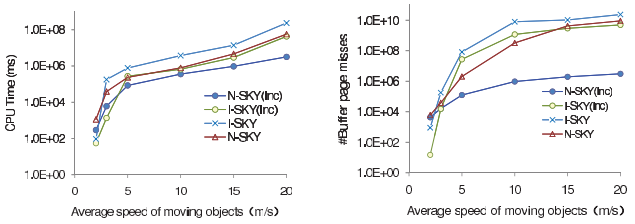
(a) CPU time cost (b) I/O cost

Fig. 19. Effect of query trajectory length.

which is the same as the one-shot RSQ algorithm (we used I-SKY here). As already shown in Fig. 14, the CPU time and I/O cost of this step do not change much as the query range grows. The second step is to identify the valid scope of each C-RSQ result object. According to Section 6.2, this cost is proportional to the size of C-RSQ result set, which is also plotted in Fig. 19. We can see that the C-RSQ results only get doubled as the query length grows 10 times. Combining these two steps, it can be concluded that our C-RSQ algorithm is scalable as the total cost does not degrade sharply as the query trajectory length grows.

7.3.2 Effect of Object Movement Speed

Similar to Section 7.2.4, we vary the average speed of the objects from 2 to 20 m/s and employ the deviation-based location update strategy. Four algorithms are compared, which are different at the first step of C-RSQ processing, i.e., how the results of C-RSQ are computed and changed with location updates of the objects. The first algorithm uses the incremental version of N-SKY, which is detailed in Appendix H, available in the online supplemental material, and denoted as N-SKY (Inc). The second and third ones both use I-SKY to evaluate an RSQ query of hexagon-shaped range. The difference is that one recomputes the skyline scopes from scratch by using the divide-and-conquer algorithm (Section 4.1, denoted as I-SKY) and the other recomputes them incrementally (Section 4.2, denoted as I-SKY(Inc.)). The fourth algorithm repeatedly issues N-SKY queries to evaluate an RSQ query of hexagon-shaped range once a location update of an object is received (denoted as N-SKY). Fig. 20 shows the results. When the movement speed is slow, the algorithms from the best to the worst are: I-SKY (Inc.), I-SKY, N-SKY (Inc.), and N-SKY; when the movement speed is slow, the algorithms from the best to the worst are: N-SKY (Inc.), I-SKY (Inc.), N-SKY, and I-SKY. As is consistent with our complexity analysis, the two incremental algorithms save a lot of unnecessary computation and are more suitable for scenarios where the objects move frequently and fast.



(a) CPU time cost (b) I/O cost

Fig. 20. Effect of object movement speed.

8 Conclusions

In this paper, we have presented a range-based skyline query as an extension to point- and line-based skyline queries. We have proposed index-based (I-SKY) and nonindex (N-SKY) solutions to resolve the range-based skyline problem. To handle the movement of the objects being queried, the incremental construction of the I-SKY index has also been devised. We have also studied the probabilistic range-based skyline problem to reduce both the result set size and computation cost. Additionally, we have extended the range-based skyline query to the continuous domain, and developed query processing algorithms for static and moving objects. The experimental results show that our proposed algorithms outperform than the existing line-based skyline solution in terms of both the CPU time and I/O cost.

As for future work, we will extend the query range to arbitrary shapes that have a closed-form mathematical expression, especially those with arc-like boundaries. Furthermore, we plan to extend our range-based skyline problem to road networks. As the perpendicular-bisector-based method does not work for the network distance, new query processing algorithms need to be developed.

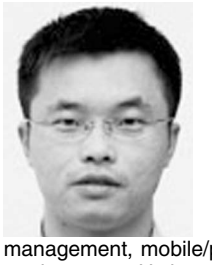
Acknowledgments

The authors are grateful to the editor and the anonymous reviewers for their constructive comments that significantly improved the quality of this paper. This work was supported by the Research Grants Council of Hong Kong (Grants 210811, 211512, and 210612), the Natural Science Foundation of China (Grant 60903169), and the Hong Kong Scholars Program (Grant XJ2011008). Lin Xin's research is also supported by the Fundamental Research Funds for the Central Universities and the Opening Project of Shanghai Key Laboratory of Integrated Administration Technologies for Information Security (No. AGK2008004). Jianliang Xu is the corresponding author.

References

[1] J.L. Bentley, H.T. Kung, M. Schkolnick, and C.D. Thompson, "On the Average Number of Maxima in a Set of Vectors and Applications," *J. ACM*, vol. 25, no. 4, pp. 536-543, 1978.
[2] M. Berg, O. Cheong, and M. Kreveld, *Computational Geometry: Algorithms and Applications*, third ed., chapter 7. Springer, 2008.
[3] S. Borzonyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *Proc. Int'l Conf. Data Eng.*, pp. 421-430, 2001.
[4] Y. Cai and T. Xu, "Design, Analysis, and Implementation of a Large-Scale Real-Time Location-Based Information Sharing System," *Proc. ACM Sixth Int'l Conf. Mobile Systems, Applications, and Services (MobiSys '08)*, pp. 106-117, 2008.
[5] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with Pre-Sorting," *Proc. Int'l Conf. Data Eng.*, 2003.
[6] C.-Y. Chow, M.F. Mokbel, and W.G. Aref, "Casper*: Query Processing for Location Services without Compromising Privacy," *ACM Trans. Database Systems*, vol. 34, no. 4, article 24, Dec. 2009.
[7] B. Fan and H. Ahmed, "A Survey of Mobility Models in Wireless Ad-Hoc Networks," *Wireless Ad-Hoc Networks*, Chapter 1, Kluwer Academic, 2006.
[8] Y. Gao and B. Zheng, "Continuous Obstructed Nearest Neighbor Queries in Spatial Databases," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, pp. 557-590, 2009.
[9] B. Gedik, K.-L. L, P.S. Yu, and L. Liu, "Processing Moving Queries over Moving Objects Using Motion-Adaptive Indexes ," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 5, pp. 651-668, May 2006.
[10] H. Hu and D.L. Lee, "Range Nearest-Neighbor Query," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 1, pp. 78-91, Jan. 2006.
[11] H. Hu, J. Xu, S.T. On, J. Du, and J.K. Ng, "Privacy-Aware Location Data Publishing," *ACM Trans. Database Systems*, vol. 35, no. 3, article 18, July 2010.
[12] Z. Huang, H. Lu, B.C. Ooi, and K.H. Tong, "Continuous Skyline Queries for Moving Objects," *IEEE Trans. Knowledge and Data Eng.*, vol. 18, no. 12, pp. 1645-1658, Dec. 2006.
[13] C.K. Lee, W.-C. Lee, B. Zheng, H. Li, and Y. Tian, "Z-SKY: An Efficient Skyline Query Processing Framework Based on Z-Order," *VLDB J.*, vol. 19, no. 3, pp. 333-362, 2010.
[14] D.L. Lee, W.-C. Lee, J. Xu, and B. Zheng, "Data Management in Location-Dependent Information Services," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 65-72, July 2002.
[15] M.-W. Lee and S.-W. Wang, "Continuous Skylining on Volatile Moving Data," *Proc. IEEE 25th Int'l Conf. Data Eng. (ICDE)*, 2009.
[16] D. Kossmann, F. Ramsak, and S. Rost, "Shooting Stars in the Sky: An Online Algorithm for Skyline Queries," *Proc. 28th Int'l Conf. Very Large Databases (VLDB)*, 2002.
[17] W.-S. Ku, R. Zimmermann, W.-C. Peng, and S. Shroff, "Privacy Protected Query Processing on Spatial Networks," *Proc. ICDE Workshop Privacy Data Management*, pp. 215-220, 2007.
[18] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An Optimal and Progressive Algorithm for Skyline Queries," *Proc. ACM SIGMOD Int'l Conf. Management of Data*, 2003.
[19] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive Skyline Computation in Database Systems," *ACM Trans. Database Systems*, vol. 30, no. 1, pp. 41-82, 2005.
[20] R-Tree Portal, http://www.rtreeportal.org/, 2012.
[21] H. Samet, *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.
[22] M. Sharifzadeh and C. Shahabi, "The Spatial Skyline Queries," *Proc. 32nd Int'l Conf. Very Large Data Bases (VLDB)*, 2006.
[23] K.-L. Tan, P. Eng, and B.C. Ooi, "Efficient Progressive Skyline Computation," *Proc. 27th Int'l Conf. Very Large Data Bases (VLDB)*, 2001.
[24] Y. Tao, D. Papadias, and Q. Shen, "Continuous Nearest Neighbor Search," *Proc. 28th Int'l Conf. Very Large Data Bases (VLDB)*, 2002.
[25] Y. Tao, L. Ding, X. Lin, and J. Pei, "Distance-Based Representative Skyline," *Proc. IEEE 25th Int'l Conf. Data Eng. (ICDE)*, pp. 892-903, 2009.
[26] J. Xu, X. Tang, H. Hu, and J. Du, "Privacy-Conscious Location-Based Queries in Mobile Environments," *IEEE Trans. Parallel and Distributed Systems*, vol. 21, no. 3, pp. 313-326, Mar. 2010.
[27] Z. Zhang, Y. Yang, A.K.H. Tung, and D. Papadias, "Continuous k-Means Monitoring over Moving Objects," *IEEE Trans. Knowledge and Data Eng.*, vol. 20, no. 9, pp. 1205-1216, Sept. 2008.
[28] B. Zheng, C.K. Lee, and W.-C. Lee, "Location-Dependent Skyline Query," *Proc. Int'l Conf. Mobile Data Management*, 2008.

Xin Lin received the PhD degree in computer science from Zhejiang University, China, in 2008. He is currently an assistant professor in the Department of Computer Science, East China Normal University. He is also a visiting scholar in the Database Group at Hong Kong Baptist University (http://www.comp.hkbu.edu.hk/~db/). His research interests include location-based services, spatial databases, and privacy-aware computing.



Jianliang Xu received the BEng degree in computer science and engineering from Zhejiang University, Hangzhou, China, in 1998 and the PhD degree in computer science from Hong Kong University of Science and Technology in 2002. He is an associate professor in the Department of Computer Science, Hong Kong Baptist University. He held visiting positions at Pennsylvania State University and Fudan University. His research interests include data management, mobile/pervasive computing, and networked and distributed systems. He has published more than 100 technical papers in these areas. He has served as a vice chairman of ACM Hong Kong Chapter. He is a senior member of the IEEE.

Haibo Hu received the PhD degree in computer science from the Hong Kong University of Science and Technology in 2005. He is a research assistant professor in the Department of Computer Science, Hong Kong Baptist University. Prior to this, he held several research and teaching posts at HKUST and HKBU. His research interests include mobile and wireless data management, location-based services, and privacy-aware computing. He has published more than 30 research papers in international conferences, journals, and book chapters. He is also the recipient of many awards, including ACM-HK Best PhD Paper Award and Microsoft Imagine Cup.