Geoinformatica (2011) 15:665–697
DOI 10.1007/s10707-010-0119-y

Spatial skyline queries: exact and approximation algorithms

Mu-Woong Lee · Wanbin Son · Hee-Kap Ahn · Seung-won Hwang

Received: 15 March 2010 / Revised: 28 August 2010 /
Accepted: 18 November 2010 / Published online: 4 December 2010

Abstract As more data-intensive applications emerge, advanced retrieval semantics, such as ranking and skylines, have attracted the attention of researchers. Geographic information systems are a good example of an application using a massive amount of spatial data. Our goal is to efficiently support exact and approximate skyline queries over massive spatial datasets. A spatial skyline query, consisting of multiple query points, retrieves data points that are not father than any other data points, from all query points. To achieve this goal, we present a simple and efficient algorithm that computes the correct results, also propose a fast approximation algorithm that returns a desirable subset of the skyline results. In addition, we propose a continuous query algorithm to trace changes of skyline points while a query point moves. To validate the effectiveness and efficiency of our algorithm, we provide an extensive empirical comparison between our algorithms and the best known spatial skyline algorithms from several perspectives.

Keywords Spatial databases · Skyline queries

M.-W. Lee · W. Son · H.-K. Ahn · S.-w. Hwang (✉)
Department of Computer Science and Engineering,
Pohang University of Science and Technology, Pohang, Republic of Korea
e-mail: swhwang@postech.ac.kr

M.-W. Lee
e-mail: sigliel@postech.ac.kr

W. Son
e-mail: mnbiny@postech.ac.kr

H.-K. Ahn
e-mail: heekap@postech.ac.kr

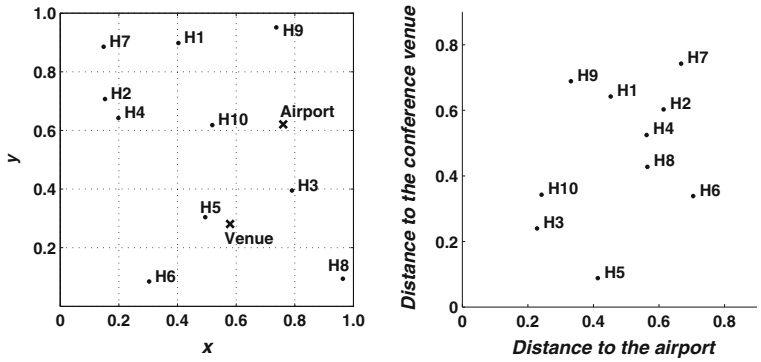
1 Introduction

Since the advent of data-intensive applications, advanced query semantics, that enable efficient and intelligent access to large scale datasets, have been an active study area. Geographic information systems (GISs) are a good example of this type of application, and it aims to support efficient access to massive spatial datasets, as Example 1 illustrates.

Example 1 Consider a hotel search scenario for a business trip to San Francisco, where the user marks two locations of interest, e.g., a conference venue and an airport, as Fig. 1a illustrates. Given these two query locations, one option is to identify hotels that are close to both locations. To better illustrate this problem, Fig. 1b rearranges the hotels with respect to the distance to each query point. From this figure, we can claim that hotel H3 is more desirable than H10, because H3 is closer to both query points than H10 is. This kind of advanced retrieval, based on *ranking* the hotels using the aggregate distance to the given query points, or based on finding *spatial skyline* hotels, will enable intelligent access to the underlying hotel datasets.

In this paper, we study *skyline queries* [1–5] to identify objects that are "not dominated" by any other objects. Specifically, we focus on finding *spatial skyline objects* [6] that are not "spatially dominated" by any other objects, i.e., no other object is closer to all of the given query points simultaneously. For instance, in Fig. 1b, H3 is a skyline object, while H10 is dominated by H3 and does not qualify as a skyline object.

Skyline queries have gained a lot attention lately, as formulating the queries is a highly intuitive process, compared to ranking where users need to identify the ideal distance functions to minimize. However, most existing skyline algorithms have not been designed with spatial data in mind and thus do not consider the spatial relationships between objects. To apply existing algorithms for general skyline queries to our problem, we generally need to transform the given location data into the distance data, as illustrated in Fig. 1. However, owing to the dynamic nature of



(a) Location (b) Distance

Fig. 1 Hotel search scenario

transformed data, the algorithms require to transform all data in advance, for each different query.

Our goal is to efficiently support skyline queries on spatial data, without such transformation. Sharifzadeh and Shahabi [6] first studied this problem, and they presented two algorithms for the problem, B^2S^2 and VS^2. Their experiments show that VS^2 outperforms B^2S^2, however, we claim that VS^2 may fail to identify the correct results. Sharifzadeh et al. [7] lately fixed this incompleteness and proposed a new version of VS^2, however, it is computationally expensive than the former algorithm. In contrast, we propose an algorithm for the problem that can identify the exact results in $O(|P|(|S|\log|\mathcal{CH}(Q)|+\log|P|))$ time, for a given set P of data points, a set Q of query points, a set S of spatial skyline points, and the *convex hull* of Q, denoted by $\mathcal{CH}(Q)$.

We also propose an efficient approximation algorithm that retrieves a subset of the exact results at a significantly lower cost, i.e., $O((|S|+|\mathcal{CH}(Q)|)\log|P|)$. Our proposed algorithm controls the quality of the approximate results by approximating the "most representative" [8] subset.

Furthermore, by extending the approximation algorithm, we propose a continuous algorithm to trace changes of skyline points, for a case of one query point moves. In a real world application scenario of using a mobile devise to establish a query, we can assume that we know the movement of one query point, the query inquirer himself. To efficiently support this scenario, we devise an algorithm tracing changes of skyline results while one query point moves in one direction, without computing the whole results repeatedly.

Our contributions can be summarized as follows:

- We study the spatial skyline query processing problem, that enables intelligent and efficient access to massive spatial datasets.
- We show that the best known algorithm is incomplete in the sense that it may not return all the skyline points.
- We propose exact and approximate spatial skyline query processing algorithms. We analyze the complexity of the proposed algorithms and the quality of the results.
- We extend our approximation algorithm to trace skyline changes, for a case of one query point moves in one direction.
- We extensively evaluate our framework using synthetic and real-life data and validate its effectiveness and efficiency.

The remainder of this paper is organized as follows. In Section 2, we provide a brief survey of related work. In Section 3, we observe the drawbacks of the best known algorithms, and propose new exact and approximation algorithms in Section 4 and Section 5 respectively. We propose a continuous algorithm tracing skyline changes in Section 6. Section 7 discusses the details of our implementation of the algorithms. In Section 8, we report on our evaluation results, and finally this paper is concluded in Section 9.

2 Related work

This section provides a brief survey of research work related to (1) skyline query processing, and (2) spatial query processing.

2.1 Skyline computation

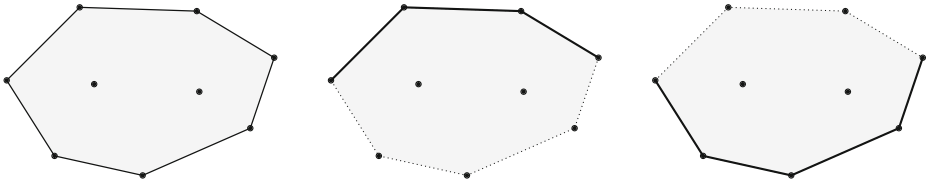
Skyline queries were first studied as maximal vectors [1]. Later, Börzsönyi et al. [2] introduced skyline queries for database applications. A number of different algorithms for skyline computation have been proposed, including progressive skyline computation using auxiliary structures [3], the nearest neighbor algorithm for skyline query processing [9], the branch and bound skyline (BBS) algorithm [4], the sort-filter-skyline (SFS) algorithm that leverages pre-sorted lists [5], and the linear elimination-sort for skyline (LESS) algorithm with an attractive average-case asymptotic complexity [10]. Recently, there have been an active research effort to address the "curse of dimensionality" problem for skyline queries [8, 11, 12] using the inherent properties of skyline points including *skyline frequency*, *k-dominant skylines*, and *k-representative skylines*. Meanwhile, for a case of when values change due to the movement of data objects, some research works for tracing skyline changes were proposed [13, 14]. These works exploit spatio-temporal coherence in the changes to further enhance skyline query processing. All of the attempts, however, have not considered the spatial dominance relationships between data points.

2.2 Spatial query processing

The most extensively studied spatial query mechanism involves ranking the neighboring points based on the distance to a single query point [15–17]. This line of research also extensively studied the problem of continuous nearest neighbor query processing on moving objects [18–21]. For multiple query points, Papadias et al. [22] studied ranking by the "aggregate" distance, for a class of monotone functions aggregating the distances to multiple query points. As these nearest neighbor queries require a distance function, which is often cumbersome to define, another line of research studied skyline query semantics which does not require any distance functions. For a spatial skyline query with a single query point, Huang and Jensen [23] studied the problem of finding spatial locations that are not dominated in respect to the *network distance* to the query point. For this type of query with multiple query points, Sharifzadeh and Shahabi [6] proposed two algorithms that identify the skyline locations for the given query points such that no other location is closer to all of the query points. While the proposed solution enables intelligent access to spatial data, we later show that the solution proposed in [6] is incorrect. Sharifzadeh et al. [7] later proposed a new version of VS^2 resolving the incompleteness of the former VS^2, however, this new algorithm is computationally expensive more than the former algorithm.

3 Preliminaries

In this section, we introduce some fundamental geometric concepts and define the problem of spatial skyline queries. Then we discuss how the known algorithm fails to identify exact answers.



(a) $\mathcal{CH}(P)$ (b) Upper chain of $\mathcal{CH}(P)$ (c) Lower chain of $\mathcal{CH}(P)$

Fig. 2 Convex hull of a point set P

3.1 Convex hull

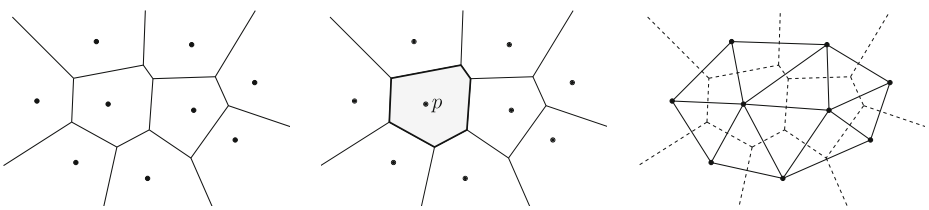
A point set S on a plane is *convex* if and only if for every two points $p, q \in S$ the whole line segment $\overline{pq}$ is contained in S. The *convex hull* $\mathcal{CH}(S)$ of a set S is the intersection of all convex sets that contain S [24]. The *upper chain* of $\mathcal{CH}(S)$ is the part of the boundary of $\mathcal{CH}(S)$ from the leftmost point to the rightmost point in a clockwise order. The *lower chain* is the part of the boundary of $\mathcal{CH}(S)$ from the leftmost point to the rightmost point in a counterclockwise order (Fig. 2).

3.2 Voronoi diagram and Delaunay graph

For a set P of n distinct points on the plane, the Voronoi diagram of P, denoted by Vor(P), is the subdivision of the plane into n cells [24]. Each cell contains only one point of P, which is called the *generator* of the cell. Any point q within a cell is closer to the generator of the cell than any other generator. The Delaunay graph of a point set P is the dual graph of the Voronoi diagram of P [24]. The two points of P have an edge in the Delaunay graph if and only if the Voronoi cells of these points share an edge in Vor(P) (Fig. 3).

3.3 Spatial skyline queries

In the spatial skyline query problem, we are given two point sets: one is a set P of data points, and the other is a set Q of query points. The points in P and Q have d-dimensional coordinate attributes in $\mathbb{R}^d$ space. The distance function $d(p, q)$ returns the Euclidean distance between a pair of points p and q, which obeys the



(a) Vor(P) (b) Voronoi cell $\mathcal{V}(p)$ of p (c) Delaunay graph of P

Fig. 3 Voronoi diagram and Delaunay graphs of a point set P

triangle inequality. We now formally define the spatial skyline query problem, using the following notions.

Definition 1 We say that p_1 *spatially dominates* p_2 if and only if $d(p_1, q) \leq d(p_2, q)$ for every $q \in Q$, and $d(p_1, q') < d(p_2, q')$ for some $q' \in Q$.

Definition 2 A point $p \in P$ is a *spatial skyline point* with respect to Q if and only if p is not spatially dominated by any other point of P.

The goal of the problem is to retrieve all the spatial skyline points from P with respect to Q. We denote S as the set of spatial skyline points of P.

3.4 Existing approaches

Though a lot of work on skyline queries has been published in the literature, little is known about skyline queries for spatial data. Recently, Sharifzadeh and Shahabi [6] studied the spatial skyline query problem and proposed two algorithms that compute S: the Branch-and-Bound Spatial Skyline algorithm (B^2S^2) and the Voronoi-based Spatial Skyline algorithm (VS^2).

In VS^2, they employed two well-known geometric structures, the *Voronoi diagram* of P and the *convex hull* of Q, and claimed that these structures reflect the spatial dominance to some extent, and therefore the algorithm efficiently computes S. In fact, their experiments show that VS^2 outperforms B^2S^2, and VS^2 is considered to be the most efficient solution so far. Figure 4 shows the pseudo-code of VS^2 [6].

VS^2, however, may fail to find all the spatial skyline points.

To verify VS^2, the authors claimed that, for some $p \in P$, if all its Voronoi neighbors and all their Voronoi neighbors are spatially dominated by other points, p is not a spatial skyline. Therefore VS^2 simply marks p as *dominated* and does not consider it afterwards. However, this is not necessarily true.

Fig. 4 Pseudo-code of VS^2 [6] for close comparison purposes

Algorithm VS^2(set Q)
01. compute the convex hull $CH(Q)$
02. set $S(Q) = \{\}$
03. Heap $H = \{(NN(q_1), mindist(NN(q_1), CH_v(Q)))\}$
04. set $Visited = \{NN(q_1)\}$; set $Extracted = \{\}$
05. box $B = MBR(SR(NN(q_1), Q))$
06. while H is not empty
07. (p, key) = first entry of H
08. if $p \in Extracted$
09. remove (p, key) from H
10-11. if p is inside $CH(Q)$ or p is not dominated by $S(Q)$
12. add p to $S(Q)$
13. $B = B \cap MBR(SR(p', Q))$
14. else
15. add p to $Extracted$
16. if $S(Q) = \phi$ or a Voronoi neighbor of p is in $S(Q)$
17. for each Voronoi neighbor of p such as p'
18. if $p' \in Visited$, discard p'
19. if p' is inside B or $\mathcal{V}(p')$ intersects with B
20. add p' to $Visited$
21. add $(p', mindist(p', CH_v(Q)))$ to H
22. return $S(Q)$

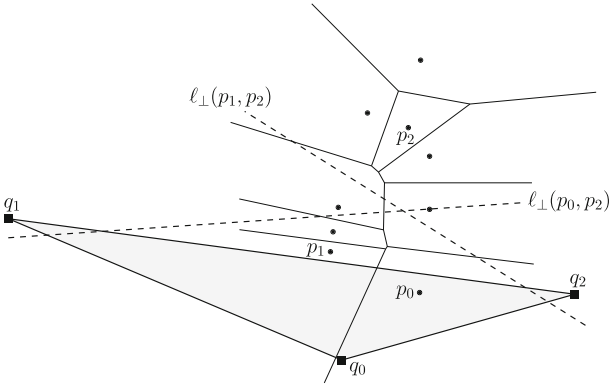
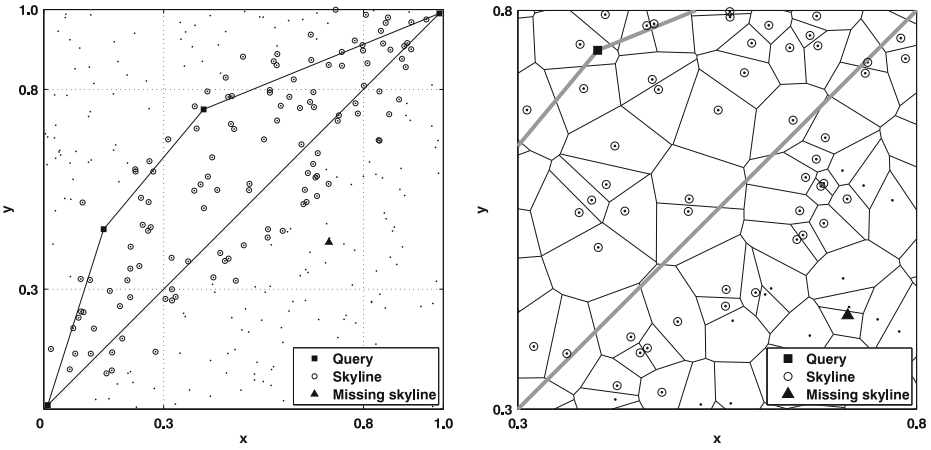


Fig. 5 VS^2 fails to find p_2 even though p_2 is a spatial skyline point

Figure 5 shows a counter example to their claim. There are 3 query points (q_0, q_1, q_2) and 9 data points. Note that all the data points, except for three (p_0, p_1 and p_2), are spatially dominated by p_0 or p_1. This means that, all the Voronoi neighbors of p_2 are spatially dominated, and VS^2 thus simply marks p_2 as "dominated" and does not consider it again. However, p_2 is a spatial skyline point, as the *bisector* $\ell_\perp(p_1, p_2)$ of p_1 and p_2, i.e., a perpendicular line to the line segment $\overline{pq}$, intersects $\mathcal{CH}(Q)$. This implies that there is a query point (q_2) closer to p_2 and therefore p_2 is not spatially dominated by p_1, as we discuss more formally in Lemma 3. Similarly, p_2 is not spatially dominated by p_0, because $\ell_\perp(p_0, p_2)$ intersects $\mathcal{CH}(Q)$. Since every bisecting line of p_2 and other points intersects $\mathcal{CH}(Q)$, we conclude that p_2 is a spatial skyline point.

This observation suggests that if $\mathcal{CH}(Q)$ has a long edge, it is likely that VS^2 would fail. Figure 6a illustrates a failure case for 300 randomly generated data points, with a



(a) Whole space (b) Closer look at the missing skyline point

Fig. 6 Failure case of VS^2 for 300 randomly generated data points

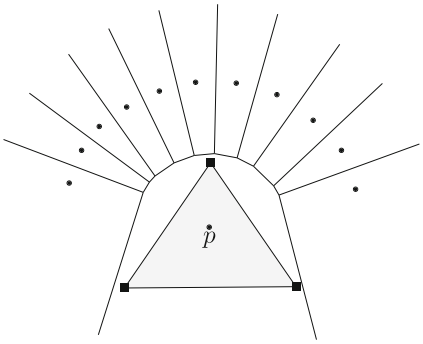


Fig. 7 A point can have many neighbors

query point set of four points, where VS^2 fails to find a skyline point below this edge. Figure 6b provides a closer look at the missing skyline point.

Moreover, the asymptotic time complexity analysis of VS^2 is incorrect. The authors assumed implicitly that VS^2 tests only $O(|S|)$ points and claimed that it finds S in time $O(|S|^2|\mathcal{CH}(Q)| + \sqrt{|P|})$. However, a skyline point p can have at most $O(|P|)$ Voronoi neighbors that are all spatially dominated by p, as Fig. 7 illustrates. Since it also calls $|P|$ heap operations during the iteration, each of which takes $\log |P|$, the correct worst-case time complexity of VS^2 must be $O(|P|(|S||\mathcal{CH}(Q)| + \log |P|))$.

Sharifzadeh et al. [7] lately fixed their own fault and proposed a new version of VS^2. To distinguish, we denote the former algorithm as VS^{2}c, and the new algorithm as VS^{2}j. However, VS^{2}j shows lower performance than VS^{2}c, in the term of query execution time. Figure 8 shows the pseudo-code of VS^{2}j [7].

VS^{2}j maintains a minheap H containing data entries to be processed later. VS^{2}j repeatedly processes and removes the top entry in H, until H becomes empty. When processing the top entry p in H, VS^{2}j tests each Voronoi neighbor p' of p to determine that p' is needed to be inserted to H (lines 15 and 16). This test is done by testing that the Voronoi cell $\mathcal{V}(p')$ of p' is not dominated by any known skyline results, that is, $\mathcal{V}(p')$ is not completely inside the union of dominance regions of all

Fig. 8 Pseudo-code of VS^{2}j [7] for close comparison purposes

Algorithm VS^{2}j(set Q)
01. compute the convex hull $CH(Q)$
02. set $S(Q) = \{\}$
03. minheap $H = \{(NN(q_1), adist(NN(q_1), CH_v(Q)))\}$
04. set $Visited = \{NN(q_1)\}$
05. minheap $HS = \{\}$
06. while H is not empty
07. remove the first entry (p, key) from H
08. if p is inside $CH(Q)$
09. add p to $S(Q)$
10. else if p is NOT dominated by $S(Q) \cup HS$
11. add (p, key) to HS
12. for each Voronoi neighbor of p such as p'
13. if $p' \in Visited$
14. add p' to $Visited$
15. if $\mathcal{V}(p')$ is NOT dominated by $S(Q) \cup HS$
16. add $(p', adist(p', CH_v(Q)))$ to H
17. while HS is not empty
18. remove the first entry (p, key) from HS
19. if p is not dominated by $S(Q)$ add p to $S(Q)$
20. return $S(Q)$

currently known possible skyline points (definite skyline points $S(Q)$ and skyline candidates HS discovered so far). Testing $\mathcal{V}(p')$ thus involves many intersection tests between $\mathcal{V}(p')$ and the dominance region of each possible skyline point. Though VS^{2}j uses a heuristic approach to reduce the cost of each test, the number of tests may become very large as we empirically show in Section 8. This makes VS^{2}j inefficient.

4 Exact algorithm

We first propose a progressive algorithm for the spatial skyline problem, that retrieves all the spatial skyline points of P with respect to Q, and then in Section 7.4, we improve this algorithm by combining it with our approximation algorithm, which is proposed in Section 5.

4.1 Properties of spatial skylines

We assume the dimensionality d of the data and query points is $d = 2$ for now, which can be extended to an arbitrary dimension (as we discuss in Section 9). Before we explain our algorithms, we first discuss some properties of spatial skylines that will be used later on. The following lemmas, though formulated in our spatial setting, are generally true for skyline processing and we omit the proofs.

Lemma 1 (Contraposition of Definition 1) *p_1 does not spatially dominate p_2 if and only if either $d(p_1, q) > d(p_2, q)$ for some $q \in Q$, or $d(p_1, q) = d(p_2, q)$ for every $q \in Q$.*

Lemma 2 (Transitivity) *Let p_1, p_2 and p_3 be three data points. If p_1 spatially dominates p_2 and p_2 spatially dominates p_3, then p_1 spatially dominates p_3.*

We now move on to discuss how to use these properties to reduce (1) the time required for each *dominance test*, i.e., to check if one data point is spatially dominated by another one, and (2) the number of dominance tests needed to produce the desired result.

4.2 Efficient spatial dominance test

Sharifzadeh and Shahabi [6] showed that we can determine the spatial dominance by using just the convex hull of Q instead of all the query points in Q: If $p \in P$ is not dominated by any other point in P with respect to the vertices of $\mathcal{CH}(Q)$, then p is a spatial skyline point. In addition, we can interpret this property in a geometric setting as follows.

Lemma 3 *The bisector of two data points intersects the interior of $\mathcal{CH}(Q)$ if and only if they do not spatially dominate each other.*

Proof If the bisector of two data points intersects the interior of $\mathcal{CH}(Q)$, then for each of the data points, there exists a vertex of $\mathcal{CH}(Q)$ closer to it than the other. For example, in Fig. 9, the bisector of p_1 and p_2 intersects $\mathcal{CH}(Q)$, so at least one

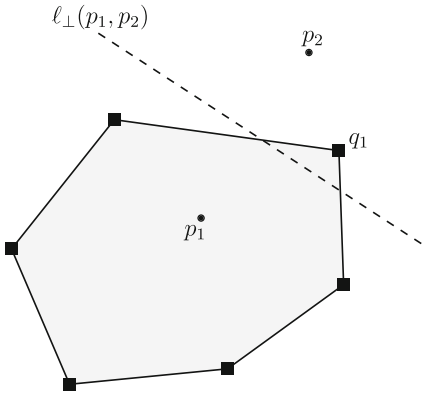


Fig. 9 $\mathcal{CH}(Q)$ intersects the bisector of two data points

query point is closer to one of the data points than the other. Therefore, they do not dominate each other. If the bisector does not intersect the interior of $\mathcal{CH}(Q)$, all the vertices of $\mathcal{CH}(Q)$ (and therefore all the query points) are closer to one data point than the other. This means that one data point spatially dominates the other point. □

As we can determine whether a line intersects the convex hull or not in $O(\log |\mathcal{CH}(Q)|)$ time by using a binary search technique, the dominance test can be done in the same time.

Lemma 4 *When $\mathcal{CH}(Q)$ is given, the dominance test for a pair of data points can be done in $O(\log |\mathcal{CH}(Q)|)$ time.*

4.3 Bounding the number of dominance tests

To make the algorithm faster, we need to reduce the number of dominance tests. To improve the speed, for some vertex q of $\mathcal{CH}(Q)$, we keep the sorted list $\mathcal{A}$ of all the data points in an ascending order of distance from q. With this list, we can determine that, if a data point p_1 is located before p_2 in $\mathcal{A}$, then p_2 does not spatially dominate p_1 using Lemma 1. In addition, together with Lemma 2, if data points p_1 and p_2 are located before p_3 in $\mathcal{A}$, and p_1 spatially dominates p_2, then we do not need to test p_3 with p_2. Therefore, it is sufficient to perform the dominance test on p only with the spatial skyline points that are located before p in $\mathcal{A}$, as we formally state below.

Lemma 5 *For a data point p, if we have the set of all spatial skyline points located before p in $\mathcal{A}$, we can determine whether p is a spatial skyline or not by $O(|S|)$ dominance tests.*

If there are two data points that are the same distance from q, we can break the tie by computing the distances from another vertex of $\mathcal{CH}(Q)$. Since no two points have the same distance from the three vertices of $\mathcal{CH}(Q)$, we only need to do this for at most three times.

We now present our proposed algorithm that retrieves all the spatial skyline points. As we will see, the algorithm is surprisingly simple and easy to follow.

Algorithm *ExactSkyline*
Input: P, Q
Output: S
1. initialize the array $\mathcal{A}$ and the list S
2. compute $\mathcal{CH}(Q)$
3. $\mathcal{A} \leftarrow$the distances from a vertex q of $\mathcal{CH}(Q)$ to every data point
4. sort $\mathcal{A}$ in an ascending order
5. **for** $i \leftarrow 0$ **to** $|P| - 1$
6. **do if** $\mathcal{A}$[i] is not spatially dominated by S
7. **then** insert $\mathcal{A}$[i] to S
8. **return** S

We now analyze the time complexity of *ExactSkyline*. In line 2, the convex hull can be constructed in $O(|Q| \log |Q|)$ time [24]. Line 3 takes $O(|P|)$ time and the sorting in line 4 can be done in $O(|P| \log |P|)$ time. In line 6, we perform the dominance test $O(|S|)$ times, each of which takes $O(\log |\mathcal{CH}(Q)|)$ time. As the **for** loop contained in lines 5–7 repeats $|P|$ times, the entire loop takes $O(|P||S| \log |\mathcal{CH}(Q)|)$ time. Since $|Q| < |P|$ in most realistic skyline queries, the total time complexity is $O(|P|(|S| \log |\mathcal{CH}(Q)| + \log |P|))$. In contrast, if we naively adopt an existing algorithm, i.e., SFS [5], by transforming P as illustrated in Fig. 1, the total time complexity is $O(|P|(|S||Q| + \log |P|))$.

5 Approximation

In this section, we discuss how to quickly compute a "desirable" subset of the skyline results, as quite often (1) it is too expensive to compute all the skyline results or (2) there are too many skyline results to be helpful [10, 25]. To achieve this goal, we first discuss how to define a desirable subset of skyline results, based on a metric studied in previous research (Section 5.1), and then present an approximation algorithm to compute a desirable subset (Section 5.3).

5.1 Desirability property

As we introduced in Section 2, skyline result sets are often too large, which provided the motivation for research efforts to reduce the results into a subset with high "desirability". In particular, the following two metrics, which have complementary strengths, have been widely adopted.

– **Nearest skylines:** One way to quantify the desirability is to favor skyline points that have the smallest *aggregate* distance to the query points [22]. This aggregation can represent various user-specific needs. For instance, a spatial skyline query retrieving the hotels close to three query points of interest, e.g., an airport, a conference venue, and a park, may rank the results by the weighted sum of the distance to the points, with a high weight on important query locations.

– **Representative skylines:** While the above metric enables *personalization* based on the user-specific needs represented in the aggregation function, it is often non-trivial for end-users to articulate the ideal aggregation function for themselves. Another approach is to identify the subset that is "commonly desirable." Specifically, Lin et al. [8] studied how to select a subset of size k that maximizes the total number of points dominated by at least one of these skyline points.

In this paper, we focus on the second metric, as the strength of skyline queries, i.e., not requiring users to specify any ranking function, can be preserved with this metric. However, this metric [8] was not previously studied in the context of spatial dominance, and a basic adoption of the algorithm proposed in the literature [8] for our problem incurs a prohibitive cost. We first briefly describe this basic adoption as a baseline approach and then discuss how we can simplify the goal for a restricted problem scenario, which serves as an efficient approximation for general problem settings (as we empirically evaluate in Section 8).

5.1.1 Baseline

A spatial skyline query problem with $|Q|$ query points can be transformed into a classical skyline query problem by transforming each data point into $|Q|$ numerical attributes representing the distance to each of the $|Q|$ query points, just like the transformation from Fig. 1a to Fig. 1b. With this transformation, the skyline results in $|Q|$-dimensional space are identical to the spatial skyline points. Given this set P' of transformed data points, and skyline points S' in the transformed dataset, we denote as $D(S_i)$ the set of points that are dominated by some subset of the skyline points $S_i \subseteq S'$ with size k, i.e., $|S_i| = k$. With this notation, our goal is to find a subset S_i of the skyline set S' that maximizes $D(S_i)$:

$$S_{\text{approx}} = \underset{S_i \subseteq S'}{\operatorname{argmax}}\, D(S_i) \tag{1}$$

However, this basic adoption is proved to be NP-hard for $|Q| \geq 3$ [8].

5.2 Our approximation

To reduce the high computational overhead, we devise an approximation which does not require data transformation as in the baseline approach. In contrast to the baseline approach, we keep the data in the original spatial domain and develop a solution to take advantage of the spatial locality of the data. Specifically, we simplify the goal in Eq. 1 for a restricted problem scenario with the following assumption:

Assumption 1 (Uniformity) *We assume that the data points are uniformly distributed. Based on this assumption, our goal is to find a subset of skyline points that maximizes the "number" of points dominated by the subset, and can then be reformulated as finding the set that maximizes the "volume" (or "area" in our two-dimensional examples) dominated by the subset.*

Based on the uniformity assumption, the goal of approximate skyline processing is to identify a skyline subset that maximizes the volume of the union of "dominated

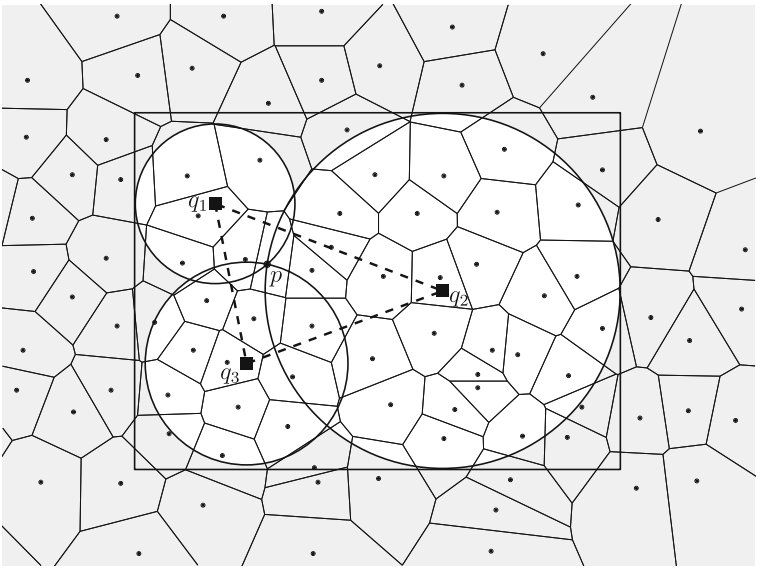


Fig. 10 All points in the shaded region are dominated by p with respect to three query points q_1, q_2, and q_3

regions" of each skyline point. Figure 10 illustrates this type of region for a skyline point p, with respect to $\mathcal{CH}(Q)$ of three query points, q_1, q_2, and q_3. All points in the dominated region, the shaded region in Fig. 10, are dominated by p, as they are always father than p from all query points.

Specifically, we draw m circles (hyper-spheres in higher dimensional spaces) for m query points, using each query point as a center, such that the given skyline point p lies on the circumference of all such circles. Any point inside these circles is not "dominated" by p, as it is closer to at least one query point compared to p. We can thus reformulate our goal statement as, a skyline subset that maximizes the volume of the union of dominated regions, i.e., the shaded area of Fig. 10.

$$S_{\text{approx}} = \underset{S_i \subseteq S}{\text{argmax}} \cup_{p \in S_i} (\text{U} - \text{unionvolume}(p)) \tag{2}$$

where the unionvolume(p) represents the union volume of the d circles with p on each of the circumferences, and U denotes the entire data space.

However, optimizing the above goal function in Eq. 2 is inherently complex, as this type of union region has a complex boundary. Figure 11 illustrates the union region for five query points and S with four skyline points. We thus approximate our objective function into a simpler form, that approximates the union of the dominated regions as the sum of these types of regions, i.e., the sumvolume.

Assumption 2 (Sumvolume) *We approximate the union volume of the dominated regions as the sum volume of these types of regions, where* sumvolume(p) *upper*

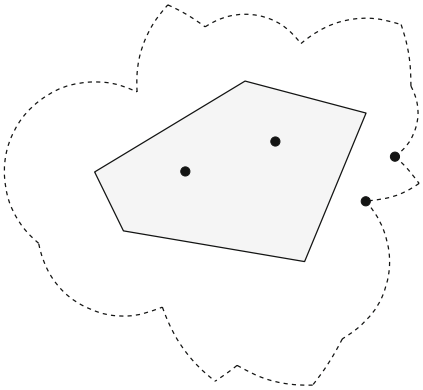


Fig. 11 Dominated regions for sample skylines

bounds unionvolume(p). *Then this approximation function can be reformulated as a function to find a skyline subset that minimizes the sum of* sumvolume.

$$S_{\text{approx}} = \underset{S_i \subseteq S}{\text{argmax}} \sum_{p \in S_i} (\text{U} - \text{sumvolume}(p))$$

$$= \text{argmin}_{S_i \subseteq S} \sum_{p \in S_i} \text{sumvolume}(p) \tag{3}$$

With this approximation, if we draw contour lines of sumvolume, each contour line is simplified into a convex set, as formally shown in Lemma 6. To illustrate, Fig. 12 shows several conour lines of sumvolume from a convex hull. A contour line of a given function, i.e., sumvolume in this case, is a curve connecting all points that have the same constant function value.

Lemma 6 *For any fixed constant c, let*

$$M(c) = \{x \mid \text{sumvolume}(x) \leq c\}.$$

Then $M(c)$ *is a convex set.*

Proof The Euclidean distance from a query point $q = (q_1, \ldots, q_d)$ in $\mathcal{CH}(Q)$ to any point $x = (x_1, \ldots, x_d)$ is defined as:

$$d(q, x) = \sqrt{(q_1 - x_1)^2 + \ldots + (q_d - x_d)^2}.$$

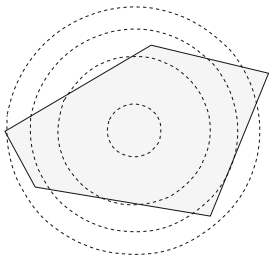


Fig. 12 Several contour lines (*dotted curves*) of sumvolume from a convex hull

This function graph is convex, because the set of points lying above the graph is convex. The volume of a sphere with radius $r = d(q, x)$ is $c_d r^d$ in $\mathbb{R}^d$ space, where c_d is a positive constant that is determined by d. As the power of a convex function is also convex [26], the volume function of the sphere is convex. It is also well known that the sum of two convex functions is also convex [26]. Since sumvolume(x) is the sum of all the dominated regions defined by query points of $\mathcal{CH}(Q)$ to x, the function graph of sumvolume is also convex. □

Furthermore, these types of contour lines are guaranteed not to intersect each other, as illustrated in Fig. 12, which we formally prove below.

Lemma 7 *For two distinct* sumvolume *values, their contour lines do not intersect.*

Proof by contradiction Let c and c' be two distinct sumvolume values. Assume that the contour lines of c and c' intersect at some point x. Then the sumvolume(x) is c, because x is a point on the contour line of c. Also the sumvolume(x) is c', because x is a point on the contour line of c'. As c and c' are distinct, this is a contradiction. □

What Fig. 12 suggests is that (a) a sumvolume is minimized at a point lying inside the convex hull and (b) it monotonically increases for skyline points that are farther from the minimum point. Based on this observation, our proposed algorithm finds a set of "seed" skyline points around the minimum point. Specifically, these are skyline points (a) within the query convex hull and (b) in the Voronoi cells intersecting the boundary of the convex hull. Another desirable property, in addition to a low sumvolume value, is that these points are guaranteed to be skyline points [6].

Theorem 1 (Seed Skyline) *For a given set P of data points and a set Q of query points, if the Voronoi cell $\mathcal{V}(p)$ of $p \in P$ intersects with the boundary of $\mathcal{CH}(Q)$ or $\mathcal{CH}(Q)$ contains $\mathcal{V}(p)$, then p is a skyline point* [6].

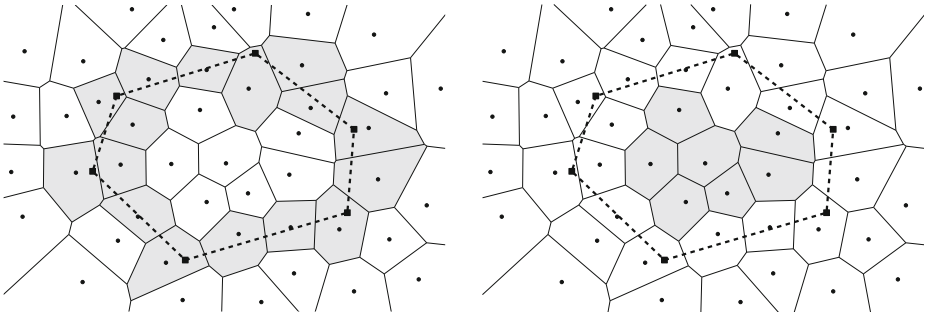
5.3 Approximation algorithm

We now present an efficient algorithm to identify the *seed skylines*, as the "desirable" skyline points to maximize our approximate objective function. To retrieve the points in the subset efficiently, we first find a Voronoi cell that contains a vertex of $\mathcal{CH}(Q)$ by using a typical point location query [24] on Vor(P). From this Voronoi cell, we follow the edges of $\mathcal{CH}(Q)$ and find the Voronoi cells that intersect the edges (Fig. 13a). Then we find Voronoi cells that lie inside $\mathcal{CH}(Q)$ (Fig. 13b) by traversing the Delaunay graph [24].

Our approximation algorithm works as follows. Let $e_i = (q_i, q_{i+1})$ denote the i-th edge along the boundary of $\mathcal{CH}(Q)$.

Note that, we can compute $\mathcal{CH}(Q)$ and Vor(P) in $O(|Q| \log |Q|)$ time (line 2) and in $O(|P| \log |P|)$ time (line 3), respectively, and locate the Voronoi cell $\mathcal{V}(p)$ containing the query point q_0 in $O(\log |P|)$ time by the point location query on Vor(P) (line 4).

To find all the Voronoi cells intersecting an edge $e_0 = (q_0, q_1)$ in line 6, we first compute the intersection r of e_0 with the boundary of $\mathcal{V}(p)$ (illustrated in Fig. 14), which can be done in $O(\log |P|)$ time using a binary search because $\mathcal{V}(p)$ is a convex



(a) Cells intersecting the edges of $\mathcal{CH}(Q)$ (b) Cells lying inside $\mathcal{CH}(Q)$

Fig. 13 Voronoi cells of seed skylines. *Dashed polygons* denote $\mathcal{CH}(Q)$

polygon and since we store its edges sorted along the boundary, as we discuss in Section 7.1. Because r lies on a boundary edge shared by two neighboring Voronoi cells, we can get a pointer to the neighboring Voronoi cell $\mathcal{V}(p')$ in a constant time from the Delaunay graph. We repeat this until we reach the other endpoint q_1. Then we proceed to the next convex hull edge $e_1 = (q_1, q_2)$ and repeat the above process until we find all the Voronoi cells that intersect the boundary of $\mathcal{CH}(Q)$.

Algorithm *ApproximateSkyline*
Input: P, Q
Output: S_{approx}
1. initialize S_{approx}
2. compute $\mathcal{CH}(Q)$
3. compute Vor(P)
4. find a Voronoi cell V(p) containing q_0
5. **for** $i \leftarrow 0$ **to** $|\mathcal{CH}(Q)| - 1$
6. find all the Voronoi cells $\mathcal{V}(p)$ intersecting e_i and insert p to S_{approx}
7. find all the Voronoi cells $\mathcal{V}(p)$ lying in $\mathcal{CH}(Q)$ by traversing the Delaunay graph and insert p into S_{approx}
8. **return** S_{approx}

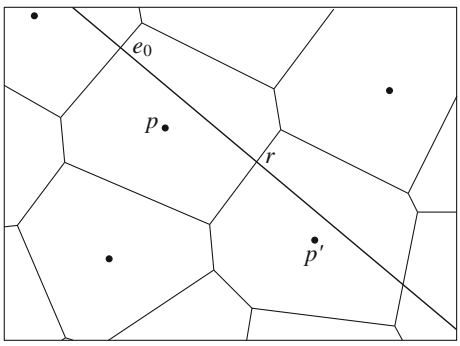


Fig. 14 Two Voronoi cells share an intersection

Note that a Voronoi cell may contain an edge of $\mathcal{CH}(Q)$ in its interior or it may intersect several edges of $\mathcal{CH}(Q)$. The number of intersection tests is thus bounded by the larger of $O(|S|)$ and $O(|\mathcal{CH}(Q)|)$, i.e., at most $O(|S| + |\mathcal{CH}(Q)|)$. By combining the number and cost of intersection tests, the overall worst-case time complexity becomes $O((|S| + |\mathcal{CH}(Q)|) \log |P|)$. Traversing a Delaunay graph can be done in $O(|S|)$ time (line 7). Therefore, the total time complexity of *ApproximateSkyline* is $O((|S| + |\mathcal{CH}(Q)|) \log |P|)$ if $\mathcal{CH}(Q)$ and $\mathrm{Vor}(P)$ are given.

6 Continuous approximate spatial skyline algorithm

In this section, we consider a variation of the problem where one query point m moves along a line while all the other query points remain stationery. Then, during the movement, some data points may change their status. The objective is to figure out for each data point when it becomes a seed skyline and when it becomes a non-seed skyline.

Let $Q_m := Q \setminus \{m\}$. Without loss of generality, we assume that m moves along a horizontal line. For a nonnegative real value t, let $m_t := m + (t, 0)$, that is, the movement of m by t along the x-axis, and let $H_t := \mathcal{CH}(Q_m \cup \{m_t\})$. While m_t moves to the right, some seed skyline point may become a non-seed skyline point, and some non-seed skyline point may become a seed skyline point. However, once a seed skyline point s becomes a non-seed skyline point at t, it remains as a non-seed skyline point for any $t' > t$ as stated in the following Lemma 8.

Lemma 8 *Once a seed skyline point s becomes a non-seed skyline point, it cannot become a seed skyline point again.*

Proof Assume that s becomes a non-seed skyline at time t'.

By definition of seed skylines, $H_{t'}$ does not intersect the Voronoi region $\mathcal{V}s$ of s. Since both $H_{t'}$ and $\mathcal{V}s$ are convex, there is a line ℓ that separates them by the *separation theorem* [27].

It is not difficult to see that m_t lies in the closed halfplane determined by ℓ containing $\mathcal{V}s$ for $t < t'$ but lies in the other open halfplane for all $t \geq t'$. Since m_t moves along a line, for all $t \geq t'$, it does not cross ℓ again and therefore $H_t \cap \mathcal{V}s = \phi$. □

6.1 Events

Without loss of generality, we assume that m_t moves from $t = 0$ to $t = 1$. Since m_t moves along a line, $\mathcal{CH}(Q \cup \{m_1\}) = \bigcup_{0 \leq t \leq 1} H_t$. We call a data point p a *candidate* if p is a seed skyline point for the query set $Q_m \cup \{m_t\}$ for some $0 \leq t \leq 1$. Let S_t denote the set of seed skyline points at time t, and $S := \bigcup_{0 \leq t \leq 1} S_t$. Note that S is the set of all candidates.

The algorithm starts with computing the set S. This can be done by computing all seed skyline points for the query set $Q \cup \{m_1\}$. At the same time, we also find S_0 and S_1. If a point p belongs to both S_0 and S_1, then p is a seed skyline point for any $0 \leq t \leq 1$ by Lemma 8. Therefore we only need to determine when each point in $S \setminus (S_0 \cap S_1)$ becomes a seed skyline point or a non-seed skyline point. To compute

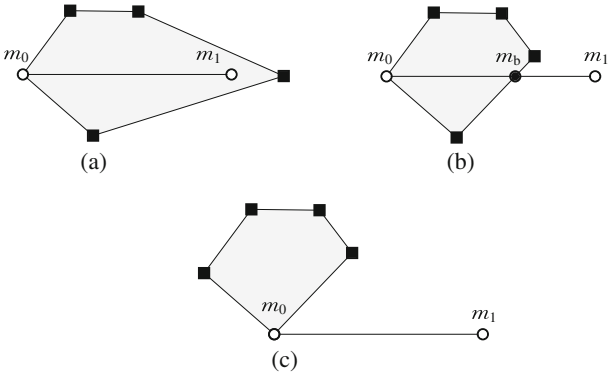


Fig. 15 Three different cases of shape changes of H_t where the gray convex hull is H_0

this efficiently, we consider three different cases of shape changes of H_t: H_t changes while m_t moves along the segment $\overline{m_0 m_1}$.

- case 1: $m_1 \in H_0$ (Fig. 15a).
- case 2: $m_1 \notin H_0$ and the line defined by m_0 and m_1 intersects the interior of H_0 (Fig. 15b).
- case 3: $m_1 \notin H_0$ and the line defined by m_0 and m_1 does not intersect the interior of H_0 (Fig. 15c).

6.1.1 Convex hull events

While we move m_t, there are different types of events at which the combinatorial structure of H_t changes. Let a denote the smallest t that m_t is in $\mathcal{CH}(Q_m)$ and b denote the largest t that m_t is in H_0. Clearly, b is always defined and $0 \leq b \leq 1$. However, a may not be defined in the interval $0 \leq t \leq 1$. If a is defined within the interval, $a < b$.

For case 1, if m_0 is not a vertex of H_0, then H_0 remains the same for $0 \leq t \leq 1$, so there is nothing to do in this case. Now assume that m_0 is a vertex of H_0. This implies that $m_0 \notin \mathcal{CH}(Q_m)$. If $m_1 \in \mathcal{CH}(Q_m)$, a is defined such that m_a is the intersection of the segment $\overline{m_0 m_1}$ and $\mathcal{CH}(Q_m)$. So there is an event at $t = a$. Otherwise, a is not defined. Furthermore, we define another type of events. There is an event at t if a supporting line of $\mathcal{CH}(Q_m)$ crosses $\overline{m_0 m_a}$.

For case 2, b is always defined. If m_0 is a vertex of H_0, we consider the segment $\overline{m_0 m_b}$ and define events as for case 1. We define an event at $t = b$ where the segment

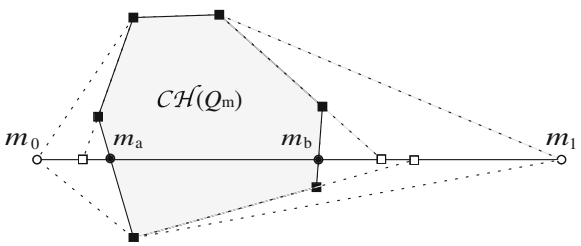


Fig. 16 Events where the combinatorial structure of H_t changes

$\overline{m_0m_1}$ crosses an edge of H_0 at m_b. We also define the following events in the interval $b < t < 1$: there is an event at t if a supporting line of $\mathcal{CH}(Q_m)$ crosses $\overline{m_bm_1}$ at m_t.

For case 3, we define an event at t if a supporting line of $\mathcal{CH}(Q_m)$ crosses $\overline{m_0m_1}$.

Figure 16 illustrates all three cases of events. The gray region is $\mathcal{CH}(Q_m)$. Two points m_a and m_b are events at $t = a$ and at $t = b$, respectively. Three small squares on $\overline{m_0m_1}$ are events where a supporting line of $\mathcal{CH}(Q_m)$ intersects $\overline{m_0m_a}$ or $\overline{m_bm_1}$.

Consider the set of all event points, excluding m_a and m_b. We connect each event point in the set to the closer vertex of its corresponding edge of $\mathcal{CH}(Q_m)$. For m_0 and m_1, we connect them by line segments to vertices of $\mathcal{CH}(Q_m)$ where the segments are tangent to $\mathcal{CH}(Q_m)$. Then we get a set of triangles as shown in Fig. 16. Consider a triangle. The base of the triangle is defined by two event points, say m_i and m_j $(i < j)$, and we denote the triangle by d_i. While m_t moves from $t = i$ to $t = j$, the moving edge of H_t inside the triangle incident to m_t either adds area to H_t or removes area from H_t. If it adds area to H_t, then it is called an *increasing* triangle. Otherwise it is called a *decreasing* triangle.

For simplicity of presentation, we consider triangles that lie above $\overline{m_0m_1}$ only from now on. The triangles lying below $\overline{m_0m_1}$ can be handled analogously. We have the following two lemmas, which are straightforward because all event points are on a line.

Lemma 9 *If a triangle d_j is increasing, then $\mathcal{CH}(d_j \cup \{m_0\})$ contains all increasing triangles d_i with $i < j$. Moreover, $\mathcal{CH}(d_j \cup \{m_0\}) \cap d_k = \phi$ for every increasing triangle with $k > j$.*

Lemma 10 *If a triangle d_j is decreasing, then $\mathcal{CH}(d_j \cup \{m_1\})$ contains all decreasing triangles d_k with $j < k$. Moreover, $\mathcal{CH}(d_j \cup \{m_1\}) \cap d_i = \phi$ for every increasing triangle with $i < j$.*

6.2 Seed skyline algorithm for a moving query

We sort the increasing triangles lying above $\overline{m_0m_1}$ in the order of their base sides along $\overline{m_0m_1}$. We also sort the decreasing triangles lying above $\overline{m_0m_1}$ analogously.

Let p a point that belongs to $S \setminus (S_0 \cap S1)$. Then p belongs to either $S_0 \setminus S_1$ or $S \setminus S_0$. In both cases, we need to find for each p the smallest t where p is a seed skyline, or the largest t where p is a seed skyline, or both. At such t, H_t is tangent to $\mathcal{V}(p)$. Specifically, a moving edge of H_t crosses a vertex of $\mathcal{V}(p)$ or the moving vertex m_t of H_t crosses an edge of $\mathcal{V}(p)$ at such t.

We first handle the case that $p \in S_0 \setminus S_1$. This means that p is a seed skyline at $t = 0$, but p is not a seed skyline at $t = 1$. Our goal is to find the largest t where p is a seed skyline. To do this we test whether $\overline{m_0m_1}$ intersect $\mathcal{V}(p)$. If it does, let t' be such that $m_{t'}$ is the last intersection point. Then we find the last triangle d that intersects $\mathcal{V}(p)$ among decreasing triangles lying above $\overline{m_0m_1}$. By Lemma 10, this can be done using binary search. Within d, we find the vertex of $\mathcal{V}(p)$ where a moving edge of H_t becomes tangent to $\mathcal{V}(p)$. This can also be done by binary searching on the vertices of $\mathcal{V}(p)$ that lie in d, because $\mathcal{V}(p)$ is convex. Let t'' be such that an edge of $H_{t''}$ is tangent to $\mathcal{V}(p)$ within d. Note that there may be no such t'', and if this is the case we set $t'' = 0$. Then $\max\{t', t''\}$ is the largest t where p is a seed skyline.

Consider now the second case that $p \in S \setminus S_0$. This means that p is not a seed skyline at $t = 0$, but p becomes a seed skyline at some $t > 0$. Possibly, it may become a non-seed skyline again later. Our goal is to find the smallest t where p is a seed skyline and, if exists, the largest t where p is a seed skyline. To do this we test whether $\overline{m_0 m_1}$ intersects $\mathcal{V}(p)$. If it does, let t' be such that $m_{t'}$ is the first intersection point. Then we find the first triangle d that intersects $\mathcal{V}(p)$ among increasing triangles lying above $\overline{m_0 m_1}$. By Lemma 9, this can be done by binary searching. Within d, we find the vertex of $\mathcal{V}(p)$ where a moving edge of H_t becomes tangent to $\mathcal{V}(p)$ for the first time, as for the first case. Let t'' be such that an edge of $H_{t''}$ is tangent to $\mathcal{V}(p)$ within d for the first time. Note that there may be no such t'', and if this is the case we set $t'' = 1$. Then $\min\{t', t''\}$ is the smallest t where p is a seed skyline. As mentioned earlier, p may become a non-seed skyline again later. If this is the case, we need to find the largest t where p is a seed skyline. This can be handled as for the first case.

Now we analyse the time complexity of the algorithm. Computing S, S_0 and S_1 takes $O((|S| + |Q|) \log |P|)$ time. Once computing $\mathcal{CH}(Q_m)$ in $O(|Q| \log |Q|)$ time, we can find the the intersection of $\overline{m_0 m_1}$ with $\mathcal{CH}(Q_m)$ in $O(\log |\mathcal{CH}(Q_m)|)$ time. Then increasing and decreasing triangles can be constructed in time $O(|\mathcal{CH}(Q_m)|)$, by traversing the edges of $\mathcal{CH}(Q_m)$ along its boundary and computing the intersection points of their supporting lines with $\overline{m_0 m_1}$. This gives sorted lists of increasing and decreasing triangles. Binary searching on a list of triangles takes $O(\log |\mathcal{CH}(Q_m)| \cdot \log |\mathcal{V}(p)|)$ time for a candidate $p \in S$, so in total $O(|S| \cdot \log |\mathcal{CH}(Q_m)| \cdot \log |P|)$ time for all candidates in S. Within a triangle, binary search on the vertices of $\mathcal{V}(p)$ can be done in $O(\log |\mathcal{V}(p)|)$ time for p, so in total $O(|S| \cdot \log |P|)$ time for all candidates in S. The intersection of $\mathcal{V}(p)$ and $\overline{m_0 m_1}$ can also be done in $O(\log |\mathcal{V}(p)|)$ time for p, so in total $O(|S| \cdot \log |P|)$ time for all candidates in S.

In the worst case $O(|\mathcal{V}(p)|) = O(|P|)$, so the total time complexity for our algorithm is $O((|S| \log |\mathcal{CH}(Q_m)| + |Q|) \log |P|)$.

7 Implementation

In this section, we discuss the implementation details of the proposed algorithms, including how to compute and store the Voronoi diagram (Section 7.1) and the query convex hull (Section 7.2) to optimize the implementation of our proposed algorithms.

7.1 Voronoi diagram

First, we discuss how we construct the Voronoi diagram and the Delaunay graph for the data points. As both structures have been studied extensively, many algorithms and implementations are already available, including 'Qhull' [28] which we adopt for our implementation. However, it is challenging to store the resulting diagram and graph in such a way that the spatial skyline query computation can be optimized. To achieve the goal, we store the Voronoi cells and Delaunay graph edges as follows:

– **cells:** As each Voronoi cell is a convex region, we take advantage of the convexity and store the vertices of each cell in an increasing angular order from one point, which preserves the adjacency of vertex pairs in the cell.

– **edges:** Every edge of a Voronoi cell is shared by a neighboring Voronoi cell. To represent the Delaunay graph, for each edge $\overline{v_i v_{i+1}}$, from a vertex v_i of a Voronoi cell, we need to store the pointer to the neighboring cell that share the edge.

Using this structure, we can exploit the convexity of a Voronoi region and the Delaunay graph discussed above, by reading only one Voronoi cell block from the file. To find a specific Voronoi cell block, we maintain a file pointer for each Voronoi cell block.

7.2 Convex hull

To compute the convex hull $\mathcal{CH}(Q)$, we use *Graham's scan algorithm* [24]. By using a binary search technique, the dominance test can be done in $O(\log |\mathcal{CH}(Q)|)$ time, as discussed in Lemma 4. We implement the test as follows.

Remember that we denote the bisector of two data points, p_1 and p_2, by $\ell_\perp(p_1, p_2)$. As discussed in Section 4.2, we can determine the dominance of two data points by testing whether $\ell_\perp(p_1, p_2)$ intersects $\mathcal{CH}(Q)$ or not. If $\ell_\perp(p_1, p_2)$ intersects $\mathcal{CH}(Q)$, at least one vertex of the upper chain of $\mathcal{CH}(Q)$ lies above $\ell_\perp(p_1, p_2)$, and one vertex of the lower chain of $\mathcal{CH}(Q)$ lies below $\ell_\perp(p_1, p_2)$ (Fig. 9). Let e_i and e_{i+1} be two edges of the upper chain that share a vertex q_i such that $\ell_\perp(p_1, p_2)$ has a slope in between the maximum and the minimum of the slopes of e_i and e_{i+1}. If $\ell_\perp(p_1, p_2)$ intersects $\mathcal{CH}(Q)$, then q_i lies strictly above $\ell_\perp(p_1, p_2)$ by convexity of $\mathcal{CH}(Q)$. We can use a similar argument for the lower chain of $\mathcal{CH}(Q)$. Because the upper and the lower chain of CH(Q) is sorted in the increasing order of the slopes of the edges, we can find these two vertices by using a binary search on the slopes of edges. After finding these two vertices in $O(\log |\mathcal{CH}(Q)|)$, we can determine the dominance in constant time. When $\mathcal{CH}(Q)$ is small, a linear search may outperform a binary search, and we use a linear search in this case.

7.3 VS^{2}c and VS^{2}j

As baselines to compare with our proposed algorithms, we use both VS^{2}c [6] and VS^{2}j [7]. We implement the algorithms using the same implementation of the R*-tree [29] and the Voronoi diagram we use to implement our proposed algorithms, to ensure fairness in empirical comparisons. To construct the convex hull, we use the same implementation as for our proposed algorithms, except that, to accommodate the complexity $O(|\mathcal{CH}(Q)|)$ dominance test discussed in the papers proposing the both algorithms [6, 7], we use a linear scan.

In our implementation, an R*-tree is used to find the closest point to one query point. The leaves of the R*-tree contain Voronoi cells that are each packed by MBRs, so that we can easily obtain candidate Voronoi cells containing a query point.

However, as shown in Section 3.4, VS^{2}c may fail to find all the spatial skyline points in some cases. Our implementation of VS^{2}c is revamped to eliminate these cases. Specifically, we remove one condition. For some $p \in P$, if all its Voronoi neighbors and all their Voronoi neighbors are spatially dominated by other points, then the original VS^{2}c does not test $p \in P$, but we implement VS^{2}c to test this point to find all skyline points.

7.4 Exact spatial skyline (*ES*)

Our implementation of the exact algorithm does not strictly follow Algorithm *Exact-Skyline*. Instead, it works in a hybrid manner involving Algorithms *ExactSkyline* and *ApproximateSkyline* to further reduce the computational cost.

7.4.1 Bypassing the dominance tests using the Voronoi diagram

By combining Algorithms *ExactSkyline* and *ApproximateSkyline*, we can retrieve all of the spatial skyline points more efficiently than with *ExactSkyline* alone. Instead of testing the dominance for all data points, we can find seed skylines using *ApproximateSkyline* efficiently, and then find the other skyline points using *ExactSkyline*. We present the combined *EnhancedExactSkyline* algorithm that results from this idea as follows:

Algorithm *EnhancedExactSkyline* **Input:** P, Q
Output: S
1. initialize the array $\mathcal{A}$ and the list S
2. compute $\mathcal{CH}(Q)$
3. $S \leftarrow$*ApproximateSkyline*(P, Q)
4. $\mathcal{A} \leftarrow$the distances from a vertex q of $\mathcal{CH}(Q)$ to every data point
5. sort $\mathcal{A}$ in an ascending order
6. **for** $i \leftarrow 0$ **to** $|P| - 1$
7. **do if** $\mathcal{A}$[i] is not in S
8. **then if** $\mathcal{A}$[i] is not spatially dominated by S
9. **then** insert $\mathcal{A}$[i] to S
10. **return** S

The asymptotic time complexity of *EnhancedExactSkyline* is the same as that of *ExactSkyline*. In practice, however, by bypassing the dominance test for the seed skylines, it performs much better than *ExactSkyline*.

Consequently, *ES* works as follows. First, it computes the Voronoi cells intersecting the boundary of the query convex hull and finds all the Voronoi cells lying in the convex hull by traversing the Delaunay graph. In this process, we restrict the *search region* for the rest of the skyline points to the bounding box containing $|Q|$ circles for $|Q|$ query points (Fig. 10). More precisely, we set the bounding box as the intersection of all bounding boxes defined by the skyline subset found so far. After that, we get a list of the candidates in this bounding box by using the R*-tree. We sort the list in an ascending order of the candidates' distances to a query point and process them one by one in this order. When we find a new skyline point, we reduce the size of the bounding box by taking the intersection of the current bounding box with the bounding box of this new skyline point. During the process, if some candidate point is not contained in the bounding box then we can simply skip the dominance test.

7.5 Approximate spatial skyline (*AS*)

We implement *ApproximateSkyline* as follows. We first precompute the Voronoi diagram and the Delaunay graph of the data points. We then store them in the form of the file mentioned in Section 7.1. We use the R*-tree to find the point closest to one

query point. As we only need to see each Voronoi cell at most once while traversing the Delaunay graph of data points, we read it from the file when it is required and deallocate it from memory after passing it by. After finding all the Voronoi cells intersected by the boundary of $\mathcal{CH}(Q)$, as discussed in Section 5.3, we find all the Voronoi cells lying in $\mathcal{CH}(Q)$ by using a breadth-first search on the Delaunay graph.

7.6 Continuous approximate spatial skyline (*CAS*)

We also implement the continuous approximate spatial skyline algorithm, *CAS*, proposed in Section 6. Our implementation of *CAS* is mostly based on that of *AS*, as *CAS* uses *AS* as its substructure.

8 Experiments

In this section, we outline our experimental settings, and present evaluation results to validate the efficiency and effectiveness of our framework. We compare our algorithms for exact spatial skylining and approximate spatial skylining with VS^2c and VS^2j, and we also evaluate our continuous approximation algorithm *CAS*. As datasets, we use both synthetic datasets and a real dataset of points of interest (POI) in California.[1] We carried out our experiments on a Pentium IV PC running on Linux with a Pentium IV 3.2 GHz CPU and 1 GB of memory, and all the algorithms were coded in C++.

8.1 Experiment settings

8.1.1 Synthetic dataset

A synthetic dataset contains up to one million uniformly distributed random locations in a 2D space. The space of the datasets is limited to the unit space, i.e., the upper and lower bound of all points are 0 and 1 for each dimension, respectively. Specifically, we use five synthetic datasets with 50 K, 100 K, 200 K, 500 K, and 1 M uniformly distributed points.

Using the synthetic datasets, we investigate the effect of the number of points in a query $|Q|$, distribution of the points in a query σ, and dataset cardinality $|P|$. The parameters used in the experiments are summarized in Table 1.

The queries are generated using the following steps: (1) We randomly generate a *reference point* then (2) generate the query points, normally distributed around the reference. Specifically, we generate points that are normally distributed, with the mean as the reference point and the deviation (distance from the reference) as a user-specified parameter σ, which varies between 0.02 and 0.10. That is, high σ produces points scattered over a wide area, low σ produces points clustered in a small area. Each query consists of up to 40 points to test scalability, though queries of a small number of points may useful in practice. We generate one hundred queries for each setting and measured the average execution times of all algorithms.

[1] Available at http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm

Table 1 Parameters used for synthetic datasets

Parameter	Setting (default value is underlined)
Dimensionality	2
Dataset cardinality	50 K, 100 K, 200 K, 500 K, 1 M
The number of points in a query	3, 5, 10, 15, 20, 40
Standard deviation of points in a query	0.02, 0.04, 0.06, 0.08, 0.10

8.1.2 POI dataset

We also validate our proposed framework using a real-life dataset. Specifically, we use a POI dataset, which consists of 104,770 locations with 63 different categories in California. Figure 17 shows the characteristics of this POI dataset.

For this POI dataset, we investigate the effect of $|Q|$ and σ. We generate the queries similarly, by randomly picking one data point as a reference point and generating query points to be normally distributed around the reference point, in the same way we generate queries for synthetic datasets. The reason we pick the reference point from the data points, instead of generating a random point, is to avoid generating queries to regions with no data points (such as the blank regions in Fig. 17). We generate one hundred queries for each setting, by varying the number of query points in a range from 3 to 40 and a standard deviation between 0.02 to 0.10, just as in our synthetic data point generation.

8.2 Efficiency of *ES* and *AS*

We first validate the efficiency of *ES* and *AS*, over varying values of $|P|$, $|Q|$, and σ. Figure 18a and b show the effect of dataset cardinality on the query execution time, and the number of dominance tests.

In Fig. 18a, our proposed algorithm *ES* outperforms both VS^{2}c and VS^{2}j. Precisely, *ES* performs up to 80 times faster than VS^{2}j. Meanwhile, our approximation algorithm *AS* outperforms *ES* several times. In a similar fashion to this, in Fig. 18b, *ES* performs a significantly smaller number of dominance tests than VS^{2}c does,

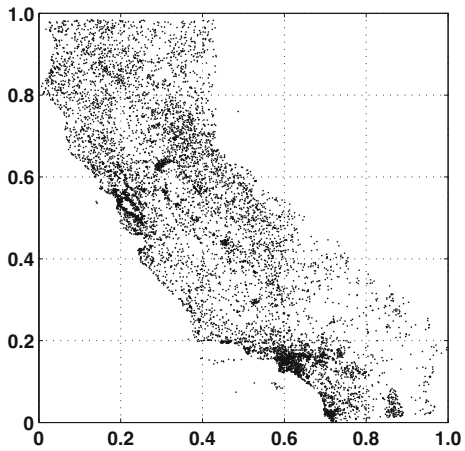
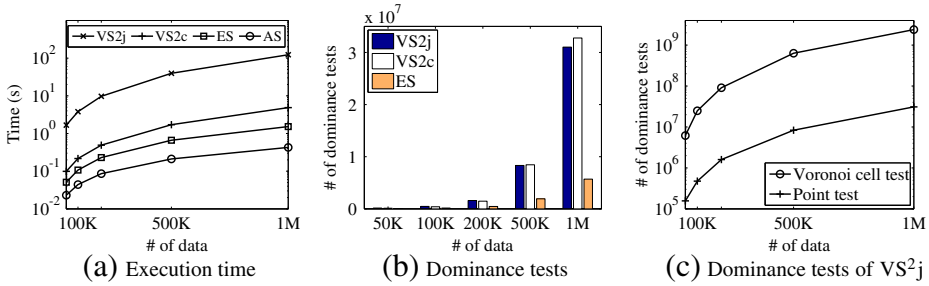


Fig. 17 10,000 sampled points from the California's POI dataset



(a) Execution time (b) Dominance tests (c) Dominance tests of VS^{2}j

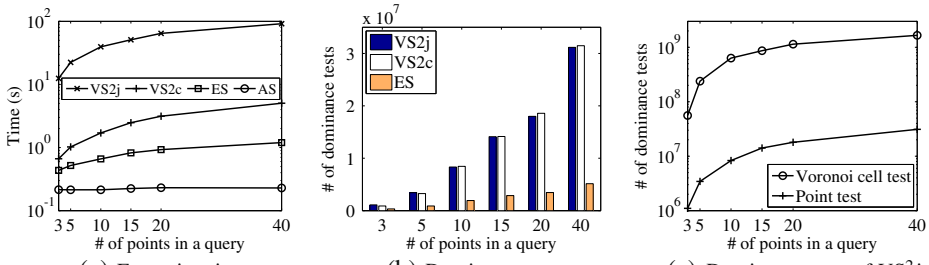
Fig. 18 Effect of the dataset cardinality for synthetic datasets

by bypassing dominance tests for skyline points whose Voronoi cells intersect the boundary of $\mathcal{CH}(Q)$. This type of saving is more prominent between skyline points, as the number of the dominance tests for skyline points is significantly higher. Note that *AS* is not reported in the graph, as it does not perform any dominance test.

We can also observe from Fig. 18b that VS^{2}j performs a slightly smaller number of dominance tests than VS^{2}c does. For VS^{2}j in Fig. 18b, we only report the number of spatial dominance tests for data points, except the tests for Voronoi cells discussed in Section 3.4. Meanwhile, Fig. 18c shows the number of spatial dominance tests for data points as well as Voronoi cells, performed by VS^{2}j. As shown in Fig. 18c, VS^{2}j performs a hundred-fold larger number of tests for Voronoi cells, than data points. Even though a spatial dominance test for a Voronoi cell is not quite expensive than for a point, this large number of tests incurs low performance of VS^{2}j, as reported in Fig. 18a.

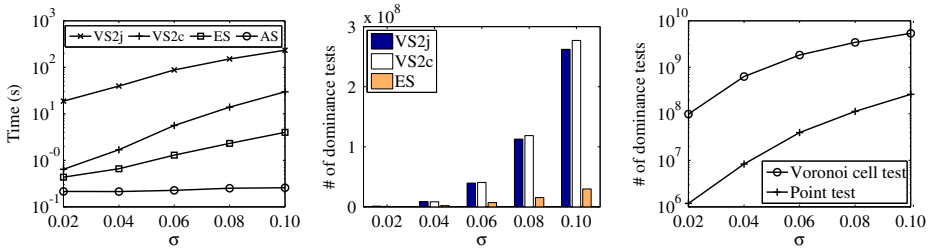
Figure 19 shows the effect of $|Q|$ on the query execution time, and the number of dominance tests. We observe similar trends as those shown in Fig. 18, except that the execution time and the number of dominance tests scale more gracefully over increasing $|Q|$ values. This can be explained by the fact that all four algorithms use $\mathcal{CH}(Q)$, instead of Q itself, the size of which grows much more gradually than that of Q. For instance, even when $|Q|$ doubles, the size of the convex hull may not change much, if the deviation σ stays the same. Observe that, *AS* scales much gracefully over $|Q|$, by eliminating the dominance tests.

Figure 20 shows the influence of σ. In a similar fashion to previous results, *ES* and *AS* significantly outperform VS^{2}c and VS^{2}j in terms of execution time and



(a) Execution time (b) Dominance tests (c) Dominance tests of VS^{2}j

Fig. 19 Effect of the number of query points for synthetic datasets



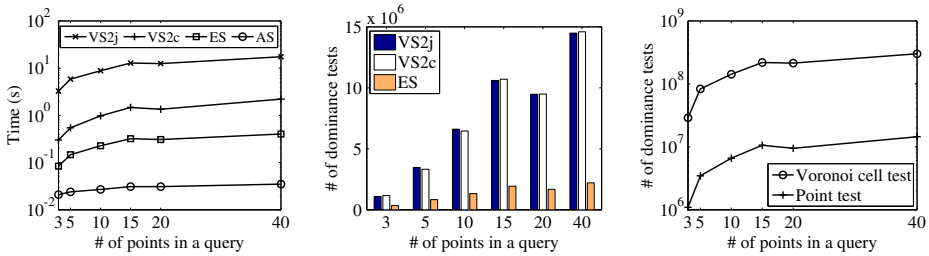
(a) Execution time (b) Dominance tests (c) Dominance tests of VS^{2}j

Fig. 20 Effect of σ of a query for synthetic datasets

dominance tests. In this case, however, we note that the execution time is more sensitive to σ. The number of dominance tests increases much faster as σ increases, than $|Q|$, because the size of $\mathcal{CH}(Q)$ may increase quadratically as σ increases. For example, when σ increases from 0.04 to 0.08 (two-fold), the circle area containing the points within the 95% confidence interval increases four-fold (i.e., quadratically) and so does the area of $\mathcal{CH}(Q)$ (and the points within it). As these types of points are guaranteed to be skyline points, this observation suggests a reason for why the number of dominant tests increases faster.

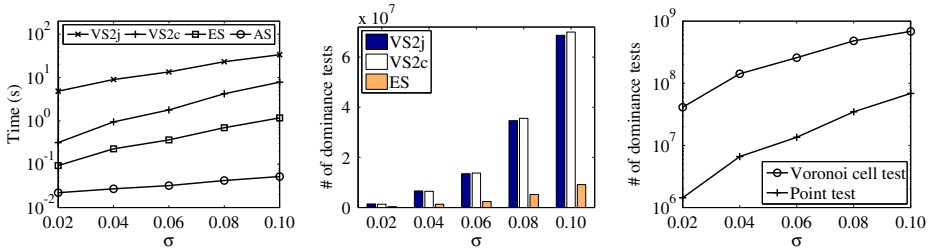
In other words, it also can be explained by imagining a transformed $|\mathcal{CH}(Q)|$-dimensional space, where $|\mathcal{CH}(Q)|$ means the number of vertices of $\mathcal{CH}(Q)$, and attribute values are distances from the vertices. Then the transformed data tend to be "anti-correlated," as σ increases. For skyline query processing, anti-correlation negatively affects the overall performance, because the size of skyline results increases considerably. Meanwhile, the performance gaps between *AS* and others increase faster as σ increases, because *AS* eliminates the dominance tests that increase exponentially over σ.

We perform the same set of experiments on the POI dataset, varying the size of the query and σ, as displayed in Figs. 21 and 22, respectively. Our observations of these evaluations are consistent with the corresponding evaluation for synthetic datasets.



(a) Execution time (b) Dominance tests (c) Dominance tests of VS^{2}j

Fig. 21 Effect of the number of query points for the POI dataset



(a) Execution time (b) Dominance tests (c) Dominance tests of VS^{2}j

Fig. 22 Effect of σ of a query for the POI dataset

8.3 Effectiveness of *AS*

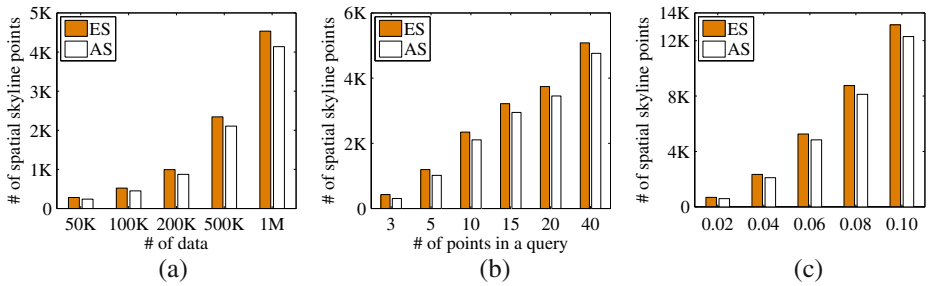
In this section, we validate the effectiveness of *AS*, which is the most efficient algorithm of the four algorithms in our efficiency evaluation discussed in Section 8.2.

To quantify the quality of the approximate results, we first compared the results of *ES* and *AS* in Figs. 23 and 24. Recall that, the results from *AS* are guaranteed to be skyline points. From the figures, we can observe that *AS* identifies 91.9% of the skyline points on average, with a much lower computational cost than *ES*.

Secondly, we compared the results of *AS* with the k most representative skyline points maximizing Eq. 1, which were obtained by setting k as the number of results from our approximation algorithm. Due to the computational overhead of computing these k representative skyline points, we only conducted small scale experiments on the approximation quality of our algorithm. In this set of experiments, we evaluated 15 queries with three query points on a dataset containing 10 thousand randomly generated points. Five queries for each were generated to acquire 20, 25, or 30 skyline results. Table 2 shows the results of these experiments. Precision was computed as

$$\text{Precision} = \frac{|\text{optimal sub-skyline} \cap \text{approximate skyline}|}{|\text{approximate skyline}|}.$$

Recall is identical to precision as $|\text{optimal sub-skyline}| = |\text{approximate skyline}| = k$.



(a) (b) (c)

Fig. 23 Quality of approximate skylines in synthetic datasets

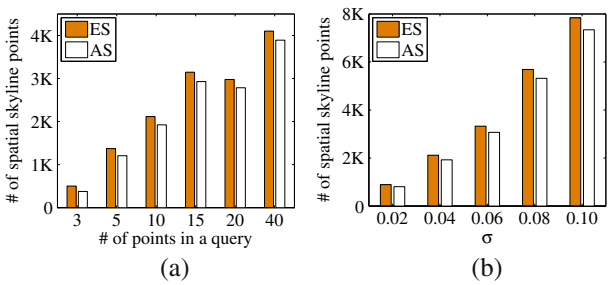


Fig. 24 Quality of approximate skylines in the POI dataset

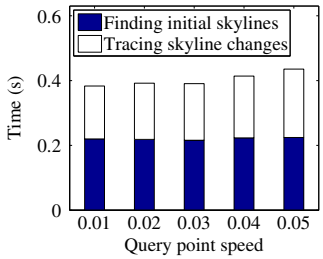
8.4 Efficiency of *CAS*

To validate the efficiency of *CAS*, we perform a set of experiments over varying speed $|v_q|$ of a query point. For this set of experiments, the initial locations of queries are generated using the same process used for generating queries for synthetic data in Section 8.1. For each setting of $|v_q|$, we generate one hundred queries consisting of 10 points for each, and $\sigma = 0.04$. For each query, we pick a point to move, and randomly choose the direction of the point picked. Then the speed of the point is set to $|v_q|$. We assume that all queries expire after a unit time, that is, the point moves $|v_q|$ length in the chosen direction. $|v_q|$ varies between 0.01 and 0.05. Figure 25 reports the execution time and the number of skyline changes over varying $|v_q|$, using a synthetic dataset.

Figure 25a shows the time for finding initial skyline points, the skyline points at the time when the query is established, as well as the time for tracing all skyline changes until the query expires. Figure 25b shows the average number of skyline changes for each setting. To combine the both results, Fig. 25c shows the average time to handle one skyline change. The average time is computed both including and excluding the time for finding initial skyline points. Observe from the figure that, *CAS* is not sensitive to $|v_q|$, which suggests that *CAS* would be robust to fast moving queries, or long query life-times, i.e., query points moves long distance until

Table 2 Approximation effectiveness

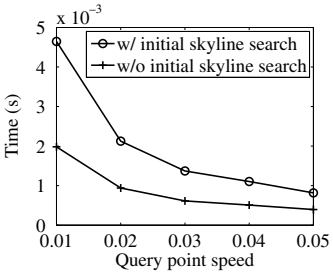
|skyline|	k = |approximate skyline|	Precision (recall)
20	10	1.000
	13	1.000
	14	1.000
	16	1.000
	17	1.000
25	11	0.727
	16	1.000
	17	1.000
	21	0.952
	21	1.000
30	17	0.882
	18	0.889
	21	0.905
	21	0.952
	23	1.000



(a) Time to trace all changes

$\|v_q\|$	# of changes
0.01	82.50
0.02	184.51
0.03	285.13
0.04	374.83
0.05	533.66

(b) Number of changes



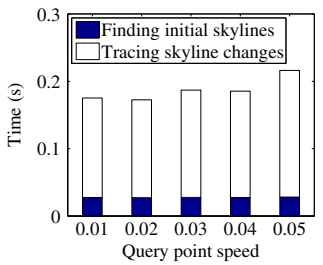
(c) Time per one change

Fig. 25 Efficiency of *CAS* over varying speed of a query point. 500 K points synthetic dataset

the queries expire. In addition, when $|v_q| = 0.05$, *CAS* only took twice longer time in total than the time for finding initial skyline points, though the skyline results change more than 500 times. To put it simply, to obtain the same results of *CAS* using only *AS*, we need to execute *AS* more than 500 times. This suggests that *CAS* is hundreds-fold efficient than a naive adoption of *AS*, i.e., executing *AS* whenever the skyline results may change.

We also perform the same set of experiments on the POI dataset, and Fig. 26 reports the results. For these experiments, the initial locations of queries are generated in the same way used for the POI dataset in Section 8.1, and the movements of queries are decided in the way as we do for the synthetic dataset. Our observation of this evaluation is roughly consistent with the evaluation for the synthetic dataset.

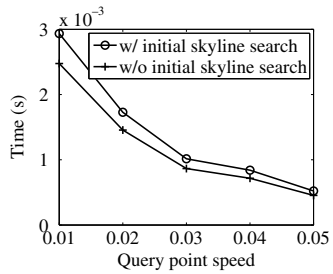
Compared to Fig. 25, the time for finding initial skylines takes a relatively smaller portion in Fig. 26a. The reason is that the cardinality of the POI dataset is fairly smaller than that of the synthetic dataset used in Fig. 25. This leads to reduce the time for finding initial skylines considerably, however, the time for tracing skyline changes has not changed much because skyline changes still occur frequently, like the case of the synthetic data.



(a) Time to trace all changes

$\|v_q\|$	# of changes
0.01	59.70
0.02	99.81
0.03	184.11
0.04	220.52
0.05	414.33

(b) Number of changes



(c) Time per one change

Fig. 26 Efficiency of *CAS* over varying speed of a query point. POI dataset

9 Conclusion and discussion

We have studied spatial skyline query processing and presented an efficient and correct algorithm. We showed that our algorithm can identify the correct result in

$$O(|P|(|S| \log |\mathcal{CH}(Q)| + \log |P|))$$

time. We also developed an approximation algorithm, and extended the approximation algorithm to trace skyline changes while a query point moves. Lastly, we empirically validated our proposed algorithms. Our exact spatial skyline algorithm, *ES*, outperforms VS^{2}j, up to nearly a hundred-fold.

So far we have assumed that the points lie in 2-dimensional space, and shown how to efficiently retrieve the spatial skyline points using some geometric structures such as the convex hull and the Voronoi diagram of points in the plane. We will now turn our attention to higher dimensional skyline queries. All the definitions, lemmas, and algorithms described in this paper generalize for higher dimensions: For a set of n points in a d-dimensional space, the Voronoi diagram of the points has $\Theta(n^{\lceil d/2 \rceil})$ combinatorial complexity [30] and can be computed in $O(n \log n + n^{\lceil d/2 \rceil})$ time [31–33]. The convex hull of those points has $\Theta(n^{\lfloor d/2 \rfloor})$ combinatorial complexity (by the so-called *Upper Bound Theorem*) and can be computed in $\Theta(n^{\lfloor d/2 \rfloor})$ expected time [24]. The dominance test, which the intersection query of a line with a convex polygon used in Section 4.2, can be generalized for higher dimensions, as an intersection query of a hyperplane with a convex polyhedron in higher dimensions. Similarly, the intersection of an edge with the Voronoi diagram can also be generalized as the intersection of a $d-1$-face with the Voronoi diagram in d-dimensional space.

Acknowledgement This research was supported by the National IT Industry Promotion Agency (NIPA) under the program of Software Engineering Technologies Development.

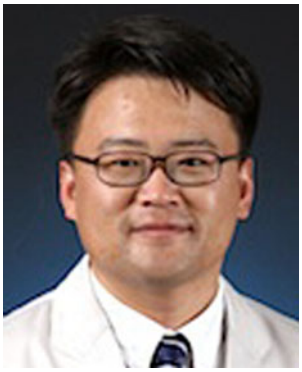
References

1. Kung HT, Luccio F, Preparata FP (1975) On finding the maxima of a set of vectors. J ACM 22(4):469–476
2. Börzsönyi S, Kossmann D, Stocker K (2001) The skyline operator. In: ICDE '01: Proceedings of the 17th international conference on data engineering. Washington, DC, USA. IEEE Computer Society, New York, pp 421–430
3. Tan K-L, Eng P-K, Ooi BC (2001) Efficient progressive skyline computation. In: VLDB '01: Proceedings of the 27th international conference on very large data bases. San Francisco, CA, USA. Morgan Kaufmann, San Mateo, pp 301–310
4. Papadias D, Tao Y, Fu G, Seeger B (2003) An optimal and progressive algorithm for skyline queries. In: SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on management of data. New York, NY, USA. ACM, New York, pp 467–478
5. Chomicki J, Godfery P, Gryz J, Liang D (2003) Skyline with presorting. In: ICDE '03: Proceedings of the 19th international conference on data engineering. IEEE Computer Society, New York, pp 717–816
6. Sharifzadeh M, Shahabi C (2006) The spatial skyline queries. In: VLDB '06: Proceedings of the 32nd international conference on very large data bases. VLDB Endowment, pp 751–762
7. Sharifzadeh M, Shahabi C, Kazemi L (2009) Processing spatial skyline queries in both vector spaces and spatial network databases. ACM Trans Database Syst 34(3):1–45
8. Lin X, Yuan Y, Zhang Q, Zhang Y (2007) Selecting stars: the k most representative skyline operator. In: ICDE '07: Proceedings of the 23rd international conference on data engineering, pp 86–95

9. Kossmann D, Ramsak F, Rost S (2002) Shooting stars in the sky: an online algorithm for skyline queries. In: VLDB '02: Proceedings of the 28th international conference on very large data bases. VLDB Endowment, pp 275–286
10. Godfrey P, Shipley R, Gryz J (2005) Maximal vector computation in large data sets. In VLDB '05: Proceedings of the 31st international conference on very large data bases. VLDB Endowment, pp 229–240
11. Chan CY, Jagadish HV, Tan K-L, Tung AKH, Zhang Z (2006) On high dimensional skylines. In: EDBT '06: Proceedings of the 10th international conference on extending database technology, pp 478–495
12. Chan C-Y, Jagadish HV, Tan K-L, Tung AKH, Zhang Z (2006) Finding k-dominant skylines in high dimensional space. In: SIGMOD '06: Proceedings of the 2006 ACM SIGMOD international conference on management of data. New York, NY, USA. ACM, New York, pp 503–514
13. Huang Z, Lu H, Ooi BC, Tung AKH (2006) Continuous skyline queries for moving objects. IEEE Trans Knowl Data Eng 18(12):1645–1658
14. Lee M-W, Hwang S-w (2009) Continuous skylining on volatile moving data. In: ICDE '09: Proceedings of the 2009 IEEE international conference on data engineering. Washington, DC, USA. IEEE Computer Society, New York, pp 1568–1575
15. Roussopoulos N, Kelley S, Vincent F (1995) Nearest neighbor queries. SIGMOD Rec 24(2): 71–79
16. Berchtold S, Böhm C, Keim DA, Kriegel H-P (1997) A cost model for nearest neighbor search in high-dimensional data space. In: PODS '97: Proceedings of the sixteenth ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems. New York, NY, USA. ACM, New York, pp 78–86
17. Beyer KS, Goldstein J, Ramakrishnan R, Shaft U (1999) When is "nearest neighbor" meaningful? In: ICDT '99: Proceedings of the 7th international conference on database theory. London, UK. Springer, Berlin, pp 217–235
18. Song Z, Roussopoulos N (2001) K-nearest neighbor search for moving query point. In: SSTD '01: Proceedings of the 7th international symposium on advances in spatial and temporal databases. London, UK. Springer, Berlin, pp 79–96
19. Benetis R, Jensen CS, Karciauskas G, Saltenis S (2002) Nearest neighbor and reverse nearest neighbor queries for moving objects. In: IDEAS '02: Proceedings of the 2002 international symposium on database engineering & applications. Washington, DC, USA. IEEE Computer Society, New York, pp 44–53
20. Tao Y, Papadias D, Shen Q (2002) Continuous nearest neighbor search. In: VLDB '02: Proceedings of the 28th international conference on very large data bases. VLDB Endowment, pp 287–298
21. Raptopoulou K, Papadopoulos AN, Manolopoulos Y (2003) Fast nearest-neighbor query processing in moving-object databases. Geoinformatica 7(2):113–137
22. Papadias D, Tao Y, Mouratidis K, Hui CK (2005) Aggregate nearest neighbor queries in spatial databases. ACM Trans Database Syst 30(2):529–576
23. Huang X, Jensen CS (2004) In-route skyline querying for location-based services. In: Proceedings of the international workshop on web and wireless geographical information systems (W2GIS), pp 120–135
24. de Berg M, Cheong O, van Kreveld M, Overmars M (2008) Computational geometry: algorithms and applications, 3rd edn. Springer, Berlin
25. Bentley JL, Kung HT, Schkolnick M, Thompson CD (1978) On the average number of maxima in a set of vectors and applications. J ACM 25(4):536–543
26. Rockafellar RT (1996) Convex analysis. Princeton University Press, Princeton
27. Matoušek J (2002) Lectures on discrete geometry. Springer, Berlin
28. Barber B (1995) Qhull code for convex hull, delaunay triangulation, voronoi diagram, and halfspace intersection about a point. http://www.qhull.org/
29. Beckmann N, Kriegel H-P, Schneider R, Seeger B (1990) The r*-tree: an efficient and robust access method for points and rectangles. SIGMOD Rec 19(2):322–331
30. Klee V (1980) On the complexity of d-dimensional Voronoi diagrams. Arch Math 34:75–80
31. Chazelle B (1991) An optimal convex hull algorithm and new results on cuttings (extended abstract). In: SFCS '91: Proceedings of the 32nd annual symposium on foundations of computer science. Washington, DC, USA. IEEE Computer Society, New York, pp 29–38
32. Clarkson KL, Shor PW (1989) Applications of random sampling in computational geometry, II. Discrete Comput Geom 4(5):387–421
33. Seidel R (1991) Small-dimensional linear programming and convex hulls made easy. Discrete Comput Geom 6(5):423–434

Mu-Woong Lee is a Ph.D. candidate student in the Department of Computer Science and Engineering at POSTECH, Korea.

Wanbin Son is a Ph.D. candidate student in the Department of Computer Science and Engineering at POSTECH, Korea.



Hee-Kap Ahn is an assistant professor in the Department of Computer Science and Engineering at POSTECH, Korea.

Seung-won Hwang is an assistant professor in the Department of Computer Science and Engineering at POSTECH, Korea.