The Farthest Spatial Skyline Queries

Gae-won You\textsuperscript{a}, Mu-Woong Lee\textsuperscript{a}, Hyeonseung Im\textsuperscript{a}, Seung-won Hwang\textsuperscript{a,*}

\textsuperscript{*}Pohang University of Science and Technology, Republic of Korea

Abstract

Pareto-optimal objects are favored as each of such objects has at least one competitive edge against all other objects, or “not dominated.” Recently, in the database literature, skyline queries have gained attention as an effective way to identify such pareto-optimal objects. In particular, this paper studies the pareto-optimal objects in perspective of facility or business locations. More specifically, given data points $P$ and query points $Q$ in two-dimensional space, our goal is to retrieve data points that are farther from at least one query point than all the other data points. Such queries are helpful in identifying spatial locations far away from undesirable locations, e.g., unpleasant facilities or business competitors. To solve this problem, we first study a baseline algorithm TFSS and propose an efficient progressive algorithm BBFS, which significantly outperforms TFSS by exploiting spatial locality. We also develop an efficient approximation algorithm to trade accuracy for efficiency. We validate our proposed algorithms using extensive evaluations over synthetic and real datasets.

Keywords: Pareto-optimum, skyline query, spatial database

1. Introduction

With the advent of GPS devices, it is becoming important for facility or business location queries to identify an optimal location from massive location data, as we illustrate with the following two real-life examples.

Example 1 (Facility location). Consider a scenario where a desirable construction site for a new park is needed. Among potential locations $P$, we want to avoid locations close to unfavorable sites $Q$ such as chemical plants.

Example 2 (Business location). Consider a scenario of seeking for a desirable location for a new Starbucks franchise. Among potential locations $P$, an optimal location will be those far from the locations of competing branches $Q$. 
For such an optimization problem, we consider a classical notion of “pareto-optimality” [1] in economics such that a point \( p \) is \textit{pareto-optimal}, if no other point \( p' \) is farther from all undesirable locations (or, \( p' \) dominates \( p \)). Figure 1(a) illustrates such a scenario in Example 1. Among 10 potential locations, we aim at finding a pareto-optimal subset of locations, far from two query points represented as two solid rectangles (\textit{i.e.,} a chemical plant and a landfill). An alternative solution is to define “hard” ranges around the two rectangles to avoid such locations, which is hard to define and often require a series of refinements, as observed in [2], with conditions being too strict (or too relaxed). In clear contrast, pareto-optimization is a “soft” query, not burdening users to elaborate such ranges.

Finding pareto-optimal solutions has been actively studied in the database literature as skyline queries [3, 4, 5, 6, 7]. In fact, Figure 1(a) can be restated as a classic skyline problem. Observe from this figure that skyline point \( p_2 \) is more favorable than non-skyline point \( p_4 \), as \( p_2 \) is farther from both unfavorable sites than \( p_4 \), \textit{i.e.,} \( p_2 \) dominates \( p_4 \). We can transform our problem into an equivalent general skyline query problem, as Figure 1(b) illustrates, such that the \( x \)-axis and \( y \)-axis correspond to the distances of data points from the chemical plant and the landfill, respectively. The results of the pareto-optimal query thus correspond to the results of the general skyline query in this transformed space, \textit{i.e.,} \{\( p_1, p_2, p_3, p_8, p_{10} \)\}.

However, though this “transform-then-process” approach qualifies as one naïve solution, such approach is inherently expensive due to the following two reasons. First, as attribute values in the transformed space are determined dynamically with respect to query points, it is infeasible to expect any pre-computed index structure on the transformed space. This restricts from adopting existing state-of-the-art skyline processing algorithms, such as BBS [7], building upon such indices to ensure efficiency. Second, the dimensionality increases up to \(|Q| \) after the transformation, which significantly deteriorates the performance of skyline computation.
In clear contrast, our goal is to develop “sub-linear” algorithms without scanning the whole data space, achieved by fully exploiting spatial locality for efficiency in the original spatial space. The most relevant work is Spatial Skyline Queries (SSQs), aiming at the “opposite goal” of identifying desirable points that are “closer” to locations of interest, e.g., an airport, a beach, and a conference venue, as query points. In this sense, our problem can be named as Farthest Spatial Skyline Queries (FSSQs). These two problems, SSQ and FSSQ, are closely related, when transformed as in Figure 1(b), as a classical skyline algorithm can apply for both problems, for minimizing and maximizing distances respectively. However, such duality ceases to exist in original spatial space, as Figure 2 contrasts the results for SSQ and FSSQ, for 400 uniformly-distributed data points and 10 query points. Unlike the results for SSQ, clustered within and around the query convex hull, the results for FSSQ are scattered across the entire data space. As a result, many geometric properties observed for SSQ problems in [8, 9] no longer hold, and algorithms building upon such properties thus cannot apply to FSSQ, which we will discuss more formally in Section 3.1.

![Figure 2: Why is FSSQ challenging?](image)

We summarize our contributions as follows:

- We formally define the FSSQ problem for pareto-optimization of facility or business locations and propose Algorithm BBFS for FSSQ, which efficiently produces farthest spatial skyline points in a progressive manner.

- For higher efficiency, we also study an approximation algorithm with little loss of accuracy.

- We implement our proposed algorithms and extensively validate them using synthetic and real-life datasets.

The remainder of this paper is organized as follows: Section 2 briefly reviews related work. Section 3.1 discusses the duality between SSQ and FSSQ.
Section 3.2 formally defines our FSSQ problem and Section 3.3 introduces some background theories. We design a baseline algorithm TFSS in Section 4.1, and propose Algorithm BBFS in Section 4.2 and its approximation algorithm in Section 4.3. Section 5 presents experimental results, and Section 6 provides our conclusion.

2. Related Work

Various types of spatial queries have been extensively studied in the database literature. The most well known type is the nearest neighbor query [10, 11], which retrieves the closest point to a single query point in the depth or best first search manner. Later work studies to extend such queries for multiple query points to retrieve the closest point, with respect to the aggregated distance to the query points [12].

However, such a query requires users to elaborate a desirable aggregation function, which is typically too burdensome for end-users. In contrast, SSQ enables users to identify a desirable set of data points that are closer than the remaining data points in any monotonic aggregated distance function without requiring users to elaborate his own distance function.

We first survey existing work on (1) SSQ and then expand our survey to consider (2) general skyline queries.

(1) Spatial skyline computation: Sharifzadeh and Shahabi [8, 13] first introduced the SSQ problem and proposed two algorithms: Algorithm B²S², and Algorithm VS² enhancing B²S² using a Voronoi diagram. Son et al. [9] later proposed Algorithm ES which reduces unnecessary dominance tests and outperforms VS². Meanwhile, while those studies are limited to formulating the skyline queries in Euclidean space called constraint-free network, Deng et al. [14] extended the problem into constraint-based road network, and Chen and Lian [15] defined and solved it in the metric space.

(2) General skyline computation: General skyline query processing has been the subject of more extensive prior work, compared to SSQ. Börzsönyi et al. [3] pioneered skyline queries in database applications, and devised block nested loop (BNL), divide-and-conquer (D&C) and B-tree-based algorithms. Chomicki et al. [4] developed sort-filter-skyline (SFS) algorithm using pre-sorted lists, which improves the BNL algorithm. Tan et al. [5] proposed progressive skyline computation with bitmaps and indices, which quickly returns partial skyline objects. Kossmann et al. [6] improved the D&C algorithm by partitioning data space using the nearest neighbor object, and pruning out the dominated partitioned area. Papadias et al. [7] developed a progressive and I/O optimal branch-and-bound skyline (BBS) algorithm.

We now discuss how our work is different from the above two lines of work.

(1) Unlike typical SSQ [8, 9] which retrieves a set of data points that are “close” to multiple query points, FSSQ retrieves points that are “far” from query points. As illustrated in Section 1, efficient processing for FSSQ is more
challenging than SSQ where the results are more spatially clustered in and around the query convex hull.

(2) FSSQ can be transformed into an equivalent general skyline problem, by transforming each data point into \(|Q|\) numerical attributes such that each of these numerical attributes represents the distance to each query point, as illustrated in Figure 1(b). However, such transformation requires a linear scan, in addition to query processing cost. In clear contrast, we develop an efficient “sub-linear” algorithm with respect to data space, which (a) does not require such transformation, and (b) expedites processing exploiting spatial locality.

3. Preliminaries

Section 3.1 discusses the duality between SSQ and FSSQ. Section 3.2 formally defines FSSQ and Section 3.3 describes backgrounds. Table 1 summarizes our notations to be used hereafter.

<table>
<thead>
<tr>
<th>Notation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(P, Q)</td>
<td>set of data points, set of query points</td>
</tr>
<tr>
<td>(|\cdot, \cdot|)</td>
<td>Euclidean distance</td>
</tr>
<tr>
<td>(FS(Q))</td>
<td>farthest spatial skyline points with respect to (Q)</td>
</tr>
<tr>
<td>(CH(Q))</td>
<td>convex hull of (Q)</td>
</tr>
<tr>
<td>(CH_1(Q), CH_v(Q))</td>
<td>convex polygon and convex points of (Q)</td>
</tr>
<tr>
<td>(\ell_{\perp}(p_1, p_2))</td>
<td>bisecting line of two points (p_1) and (p_2)</td>
</tr>
<tr>
<td>(VD(P), VC(p))</td>
<td>Voronoi diagram of (P), Voronoi cell of (p)</td>
</tr>
</tbody>
</table>

3.1. On Duality between SSQ and FSSQ

In the introduction, we illustrate that the duality between SSQ and FSSQ in the original space does not hold, especially in the scenario depicted in Figure 2. Then, one natural question that follows is whether another two-dimensional space exists where the duality holds. In other words, can data points \(P\) in the original data space be transformed into another set \(P'\) where the distance between two points in \(P\) is reversely defined in \(P'\), i.e., the farthest point pairs in \(P\) are the closest in \(P'\) and vice versa? If this type of transformation is possible, a solution to SSQ can be applied in this transformed space to answer FSSQ efficiently. However, unfortunately, this transformation is not feasible as we prove in the following theorem.

**Theorem 1 (Non-duality between SSQ and FSSQ).** There is no one-to-one transformation \(T\) such that for any given two-dimensional point set \(P\) with \(|P| \geq 3\), the inequality relations of distances among \(T(P) = P'\) are the reverse of \(P\).
If three different points are collinear, then builds on Definition 1 to define spatial skyline points.

3.2. Problem Definition

We first formally define the spatial dominance for FSSQ in Definition 1. Note that \(\|\cdot\|\) denotes the Euclidean distance between two points. Definition 2 then builds on Definition 1 to define spatial skyline points.

**Definition 1 (Dominance for FSSQ).** Given a set of query points \(Q\), a point \(p_1\) "spatially dominates" a point \(p_2\) (denoted \(p_1 \succ_Q p_2\)) if and only if \(\forall q \in Q, \|p_1, q\| \geq \|p_2, q\|\) and \(\exists q' \in Q, \|p_1, q'\| > \|p_2, q'\|\). We denote \(p_1 \not\succ_Q p_2\) to represent that \(p_1\) does not spatially dominate \(p_2\), and \(p_1 \prec_Q p_2\) to represent "incomparability," i.e., \(p_1 \not\succ_Q p_2\) and \(p_2 \not\succ_Q p_1\). These notions can be straightforwardly extended to a set \(R\) of points. That is, \(p \succ_Q R\) denotes that \(\forall p' \in R : p \succ_Q p'\).
Definition 2 (Farthest spatial skyline). Given a set of data points \( P \) and a set of query points \( Q \), a point \( p \in P \) is a “farthest spatial skyline point” if and only if \( p \) is not spatially dominated by any other point, i.e., \( \forall p' \in P\setminus \{p\} : p' \not\succ_{Q} p \). \( FS(Q) \) denotes a set of such farthest spatial skyline points with respect to \( Q \).

3.3. Background Theories

This section describes background theories, based on which we build our proposed algorithms in Section 4. First, we describe what convex hull is and how it can be used for pruning out the query points that do not affect the query results. Second, we describe Voronoi diagram which will be used in Algorithm BBFS in Section 4.2.

3.3.1. Convex Hull

Given a set of points \( P \), a convex hull \( CH(P) \) is the boundary of a minimal convex set containing all points in \( P \). For instance, Figure 4 depicts a convex hull \( CH(P) \) (dashed edges) with respect to 10 data points in \( P \), where convex points \( CH_v(P) \) are 6 vertices on \( CH(P) \) and convex polygon \( CH_I(P) \) represents the area constrained by \( CH(P) \).

![Figure 4: Convex hull and Voronoi diagram](image)

![Figure 5: Collinear points](image)

In two-dimensional space, the convex hull can be computed in \( O(|P| \log |P|) \) using Graham’s scan algorithm [16]. Using the convex hull, we can prove that only the convex query points affect the farthest spatial skyline set in Theorem 2, in the same procedure as proved for SSQ [8]. For such proof, we first develop three lemmas:

**Lemma 2.** Given two sets of query points \( Q' \) and \( Q \) such that \( Q' \subset Q \), \( |Q'| \geq 3 \), and points in \( Q' \) are not collinear, if \( p \in FS(Q') \), then \( p \in FS(Q) \).

**Proof.** There is no point \( p' \in P \setminus \{p\} \) such that \( p' \succ_{Q} p \) because \( p \in FS(Q') \). Thus, for any \( p' \in P \setminus \{p\} \), there exists a query point \( q \in Q' \) such that \( ||p, q|| > ||p', q|| \) because \( Q' \) are not collinear. Since \( Q' \subset Q \), no point \( p' \succ_{Q} p \) by Definition 1, which suggests that \( p \in FS(Q) \).

**Note** that, Lemma 2 assumes that \( |Q'| \geq 3 \) and points in \( Q' \) are not collinear, i.e., there exists no single straight line on which all points in \( Q' \) lie. Figure 5
Given two sets of query points \( \ell \perp q \), let \( p \) and \( p' \) belong to \( \text{FS}(Q') \) since \( \forall q \in Q' : \|p, q\| = \|p', q\| \), \( p \) does not belong to \( \text{FS}(Q) \) since \( \|p', q_4\| > \|p, q_1\| \) and \( p' \succ_q p \).

We now move on to discuss dominance and incomparability. As a preliminary, we first define a bisecting line, or bisector, between two points \( p_1 \) and \( p_2 \). Let \( \ell \perp (p_1, p_2) \) be a bisecting line perpendicular to the segment \( \overline{p_1 p_2} \), on which every point is equidistant from \( p_1 \) and \( p_2 \). \( \ell \perp (p_1, p_2) \) divides a space into two different half spaces, i.e., \( h(p_1, p_2) \), the half space in which every point is closer to \( p_1 \) than \( p_2 \), and \( h(p_2, p_1) \), the other half space. Figure 6 illustrates an example on a bisecting line and its both half spaces.

**Lemma 3 (Dominance using bisector).** Given points \( p_1 \) and \( p_2 \), \( \text{CH}(Q) \subset h(p_2, p_1) \) if and only if \( p_1 \succ_q p_2 \).

**Proof.** We prove if-part and only if-part, respectively. (1) If-part: Since \( p_1 \succ_q p_2 \), \( \forall q \in \text{CH}_v(Q) : \|p_1, q\| \geq \|p_2, q\| \). \( \text{CH}_v(Q) \subset h(p_2, p_1) \) and \( \text{CH}(Q) \subset h(p_2, p_1) \). (2) Only if-part: Since \( \text{CH}(Q) \subset h(p_2, p_1) \), \( \forall q \in Q : \|p_1, q\| \geq \|p_2, q\| \). As we assume points in \( \text{CH}_v(Q) \) are not collinear, \( \exists q' \in Q : \|p_1, q'\| > \|p_2, q'\| \).

Thus, \( p_1 \succ_q p_2 \) by Definition 1. Figure 6(a) illustrates this case.

**Lemma 4 (Incomparability using bisector).** If the bisecting line \( \ell \perp (p_1, p_2) \) of points \( p_1 \) and \( p_2 \) intersects \( \text{CH}(Q) \), then \( p_1 \ll_q p_2 \).

**Proof.** If \( \ell \perp (p_1, p_2) \) intersects \( \text{CH}(Q) \), there exists at least one query point in \( \text{CH}_v(Q) \) farther from \( p_2 \) than \( p_1 \), while all other query points are farther from \( p_1 \), i.e., \( p_1 \ll_q p_2 \) as Figure 6(b) illustrates.

![Figure 6: On Lemmas 3 and 4](image)

**Theorem 2.** Given two sets of query points \( Q' \) and \( Q \) such that \( Q' = \text{CH}_v(Q) \), no query point \( q \in Q \setminus Q' \) affects the set of skyline points \( \text{FS}(Q) \), i.e., \( \text{FS}(Q') = \text{FS}(Q) \).

**Proof.** We prove \( \text{FS}(Q') = \text{FS}(Q) \) by showing \( \text{FS}(Q') \subset \text{FS}(Q) \) then \( \text{FS}(Q') \supset \text{FS}(Q) \). (1) \( \text{FS}(Q') \subset \text{FS}(Q) \): For any \( p \in \text{FS}(Q') \), by Lemma 2, \( p \in \text{FS}(Q) \)
because $Q' \subset Q$. (2) $FS(Q') \supset FS(Q)$: To prove by contradiction, we assume that $p \in FS(Q)$ but $p \notin FS(Q')$. Then, there is a point $p'$ such that $p' \succ Q p$. $CH(Q') \subset h(p,p')$ by Lemma 3, and $CH(Q) \subset h(p,p')$ because $CH(Q') = CH(Q)$. Thus, $p' \succ Q p$, which contradicts the assumption $p \in FS(Q)$.

By Theorem 2, given data points $P = \{p_1, p_2\}$ and query points $Q$, it straightforwardly takes $O(|CH_v(Q)|)$ time to do a dominance test because it computes $\|p_1, q\|$ and $\|p_2, q\|$ for all $q \in CH_v(Q)$. However, we can reduce this complexity into $O(\log |CH_v(Q)|)$ by exploiting Lemma 4 above as shown in [9]. More specifically, if $p$ and $p'$ are incomparable, there exists an edge in the convex hull where one end is above $\ell_{\perp}(p, p')$ and another is below $\ell_{\perp}(p, p')$. Such an edge can be found by binary searching for the angles of edges, as the convex hull keeps the edges sorted.

### 3.3.2. Voronoi Diagram

To exploit spatial locality for optimizing query processing, our proposed algorithm $BBFS$ precomputes a Voronoi diagram. Specifically, the Voronoi diagram $VD(P)$ with respect to $P$ consists of $|P|$ Voronoi cells, where each Voronoi cell $VC(p)$ of a point $p \in P$ is a set of points which have $p$, among all points in $P$, as the closest point. More formally, we can define $VC(p)$ as follows:

$$VC(p) = \bigcap_{p' \in P \setminus \{p\}} h(p,p')$$

(1)

For example, Figure 4 describes a Voronoi diagram. The shaded cell is the Voronoi cell $VC(p)$ of a point $p$. The Voronoi diagram can be computed in $O(|P| \log |P|)$ in two-dimensional Euclidean space [16].

### 4. Algorithms

This section discusses algorithms to solve FSSQ. Section 4.1 extends an existing general skyline algorithm IDS [17] as a baseline. Section 4.2 then proposes a new algorithm outperforming the baseline algorithm exploiting geometric properties on an R-tree index [18]. Finally, Section 4.3 proposes an approximation algorithm to trade accuracy for efficiency.

#### 4.1. Threshold Farthest Spatial Skyline: TFSS

This section develops a baseline algorithm by extending an existing algorithm for general skyline queries. In particular, such existing algorithms can be categorized into the following three categories:

- **No-index**: Algorithms in this category, such as SFS [4] and SaLSa [19], do not assume any index on data. Instead, these algorithms linearly scan or sort data for processing.

- **Index on local data**: Algorithms in this category, such as BBS [7], build upon an index structure on local data, such as an R-tree index [18].
Index on distributed data: Algorithms in this category, such as IDS [17], build upon accesses supported from distributed data sources.

After transforming an FSSQ problem of $n$ query points into an $n$-dimensional general skyline problem, existing algorithms in any of the above categories can apply. However, due to the dynamic nature of transformed data, the algorithms in the first two categories require “linear” scan in order to sort data or build an index structure.

In clear contrast, we aim at a sub-linear algorithm using indices. Specifically, we develop Algorithm TFSS, which is a variation of Improved Distributed Skyline (IDS) [17], building upon standard sets of accesses, such as sorted accesses from distributed sources. We stress that Algorithm TFSS falls into the second category above, though it is based on IDS using distributed sorted access (third category), as we incrementally materialize such sorted accesses exploiting an R-tree index on local data as we explain below.

Sorted accesses retrieve data objects in decreasing order of attribute values, which correspond to the distances from each query point in our problem scenario. Instead of pre-materializing sorted accesses, we propose to incrementally materialize such accesses using a sub-linear distance browsing algorithm [11]. This algorithm, given a query point $q_i$, exploits a priority queue $H_i$ on an R-tree index to incrementally materialize the next farthest point from $q_i$ at each iteration, which we denote as $\text{NextFarthest}(q_i, H_i)$.

Given these incrementally materialized sorted accesses, when $L_i$ indicates an ordered set of points accessed from sorted accesses on $H_i$, IDS terminates as soon as one point $p_{term}$ is seen from all sorted accesses, or more formally,

$$\bigcap_{i=1}^{n} L_i \neq \emptyset.$$  

(2)

At this point, any point yet to be accessed can be safely pruned out, as such a point is dominated by $p_{term}$, i.e., closer to all query points than $p_{term}$.

The cost of IDS, or the number of sorted accesses, varies over how to “schedule” sorted accesses. To illustrate, Figure 7(a) depicts a naïve round-robin
scheduling of sorted accesses, which terminates when \( p_{\text{term}} \) is seen from "equal-depth sorted accesses," represented as a dotted line at the bottom. However, we can easily observe that, such costs can be significantly reduced if we are allowed to perform “uneven accesses” only on gray bars as shown in Figure 7(a). Such scheduling requires heuristics to determine the ideal depth for each sorted access.

We discuss how TFSS addresses the above challenge of (1) candidate selection using the scheduling heuristics. We then discuss (2) filtering step identifying the skyline points among the candidates. Algorithm 1 formally describes two steps of TFSS.

\[ \text{Algorithm 1: TFSS}(Q, T) \]

\[ \text{Input: Set of query points } Q; \text{ R-tree } T \]
\[ \text{Output: Set of skyline points } \text{FS}(Q) \]

\[ Q' \leftarrow CH_v(Q); \text{let } Q' = \{q_1, \ldots, q_n\}; \]
\[ \text{/* compute } CH(Q) \text{ */} \]

1. Initialize \( n \) priority queues \( H_i's \) with \( T \)'s root node
2. \( v_1, \ldots, v_n \leftarrow \text{maximum pairwise distance; } L_1, \ldots, L_n \leftarrow \{\} \)
3. Choose a random point as initial \( p_{\text{prom}} \)
4. \( \forall i \text{ such that } p_{\text{prom}} \notin L_i \) do
5. \( p_{\text{new}} \leftarrow \text{NextFarthest}(q_i, H_i) \)
6. \( L_i \leftarrow L_i \cup \{p_{\text{new}}\}; v_i \leftarrow \|p_{\text{new}}, q_i\| \)
7. \( p_{\text{prom}} \leftarrow p_{\text{new}} \)
8. \( \text{if } \sum_{j=1}^{n} v_j - \|p_{\text{new}}, q_j\| < \sum_{j=1}^{n} v_j - \|p_{\text{prom}}, q_j\| \text{ then} \)
9. \( \forall j \in \{1, \ldots, n\}, \text{ remove non-skyline points in } L_j \)
10. \( \text{FS}(Q) \leftarrow \bigcup_{j=1}^{n} L_j \)
11. return \( \text{FS}(Q) \)

4.1.1. Candidate Selection Step (lines 4-9)

While the round-robin scheduling switches to another query point after each single access, TFSS continues sorted access on \( H_i \) without switching, until the point it accesses is a “promising” terminating point (denoted \( p_{\text{prom}} \)). Intuitively, \( p_{\text{prom}} \) is far from all query points. Thus, \( p_{\text{prom}} \) is more likely to locate in front of each ordered set \( L_j \) than the other points. In other words, it has the smallest \( \sum_{j=1}^{n} v_j - \|p_{\text{prom}}, q_j\| \), where \( v_j \) is the distance from \( q_j \) to the latest accessed point \( p_j \) on \( H_j \).

However, accessing all data points for finding the most promising terminating points defeats the purpose of a sub-linear algorithm. Instead, we initialize \( p_{\text{prom}} \) with a randomly selected point and start sorted access on \( H_1 \), until accessing a more promising point object \( p_{\text{new}} \) with \( \sum_{j=1}^{n} v_j - \|p_{\text{new}}, q_j\| < \sum_{j=1}^{n} v_j - \|p_{\text{prom}}, q_j\| \). At this point, we update \( p_{\text{prom}} \) with \( p_{\text{new}} \) (lines 8-9) and continue the next sorted access on \( H_2 \). Similarly, during sorted accesses on \( H_2 \), if it accesses another \( p_{\text{new}} \), it updates current \( p_{\text{prom}} \) into \( p_{\text{new}} \), and continues the sorted access after switching to another query point. Through
these iterations, $p_{prom}$ eventually converges to the overall promising terminating point, and Algorithm TFSS terminates when all sorted accesses reach such a terminating point, as depicted by the dotted line in Figure 7(b).

4.1.2. Filtering Step (lines 10-11)

Once the candidates, i.e., $\bigcup_{i=1}^{n} L_i$, are identified, non-skyline points among these candidates are pruned out. While a naïve filtering algorithm would test the dominance between every candidate pair, we focus such tests only to point pairs within the same $L_i$ without loss of accuracy, because, if $p \in L_i$ and $p' \notin L_i$, then $\|p, q_i\| > \|p', q_i\|$ and $p'$ cannot dominate $p$, as proved in IDS [17].

While TFSS reduces the number of dominance tests from naïve exhaustive pairwise tests, there is still huge room for optimization. In contrast to limiting dominance tests to point pairs, we can exploit the spatial locality of points to prune out a group of points, i.e., batch pruning. Considering that the cost of dominance tests in the filtering step dominates the overall cost, as Figure 8 illustrates (with default setting in Table 2), intelligent batch pruning rules will significantly reduce the overall cost.

![Graph](image)

(a) Elapsed time

(b) Ratio

Figure 8: Bottleneck of TFSS

In the next section, we propose another algorithm, which enhances TFSS with respect to the following criteria.

- **Batch pruning**: Unlike TFSS pruning out one point at a time, our proposed algorithm enables batch pruning of multiple points with spatial locality.

- **Progressiveness**: Unlike TFSS that can return the skyline points only at the termination, our proposed algorithm produces the skyline points in a progressive manner.

- **Scalability**: Unlike TFSS requiring to maintain $|CH_v(Q)|$ priority queues, our proposed algorithm, by maintaining a single heap, incurs low storage and maintenance overhead.
4.2 Branch-and-Bound Farthest Spatial Skyline: BBFS

This section develops Algorithm BBFS, which enables batch pruning, using spatial locality of multiple points within a Minimum Bounding Rectangle (MBR) of an R-tree. We first sketch the overall structure of BBFS, and then introduce technical details for our batch pruning rules.

4.2.1 Overall Structure

Algorithm BBFS is essentially a top-down branch-and-bound search on an R-tree accessing nodes in decreasing order of the sum of distances from query points. More specifically, we maintain a max-heap $H$ for visited MBRs initialized as the root of an R-tree. We keep $H$ ordered in descending order of $\text{SumDist}(e, Q)$ of MBR $e$, and then iteratively pop $e$ with the highest score and push its child nodes in the R-tree. Formally,

$$\text{SumDist}(e, Q) = \sum_{q \in Q} \text{MaxDist}(q, e),$$

where

$$\text{MaxDist}(q, e) = \max_{p \in e} \|q, p\|.$$

Note that we can replace this function with any other monotone aggregation functions, such as $\sum \ln(\text{MaxDist}(q, e) + 1)$ used in [4], without loss of accuracy. Due to the monotonic nature, function scores for the MBRs yet to be accessed are bounded by the ones in the queue, such that unaccessed MBRs are guaranteed not to affect the results, i.e., the heap can be incrementally materialized without loss of accuracy. Theorem 3 formally proves this property.

**Theorem 3.** Given query points $Q'$, no point can dominate another point popped earlier from $H$ in Algorithm BBFS.

**Proof.** We prove this by contradiction. $H$ enables to access MBRs and points in descending order of $\text{SumDist}(e, Q')$. Assume that a point $p$ is popped from $H$ earlier than another point $p'$ but $p'$ dominates $p$. Since $p'$ dominates $p$, by Definition 1, $\forall q \in Q', \|p', q\| \geq \|p, q\|$ and $\exists q' \in Q', \|p', q'\| > \|p, q'\|$. Thus, $\sum_{q \in Q'} \|p', q\| > \sum_{q \in Q'} \|p, q\|$ which contradicts our assumption that $p$ is popped earlier than $p'$.

We now present how this algorithm addresses the three design goals we identified above.

- **Batch pruning:** By accessing points in the unit of MBR, we can prune out multiple objects sharing spatial locality to further reduce the number of dominance tests.
- **Progressiveness:** Accessing MBRs in the order of a monotone function ensures that any skyline point identified in the middle of execution cannot be dominated by other objects yet to be accessed (with larger SumDist scores).
- **Scalability:** BBFS maintains only one max-heap, in contrast to TFSS maintaining $|CH_v(Q)|$ queues.
4.2.2. Batch Processing Rules

We discuss how to reduce the number of dominance tests using batch pruning and bypassing rules as described below:

- **Batch pruning rule**: By leveraging “dominance” concept of Lemma 3, we may safely prune out all points in an MBR with only a few dominance tests.

- **Batch bypassing rule**: Some skyline points are inherently “incomparable” with all the others, so we may bypass dominance tests with those skyline points (We will formally state this bypassing rule in Theorem 6).

First, the batch pruning rule eliminating the unnecessary MBRs, or batches of dominated points, early on from the heap, significantly reduces not only the unnecessary I/O-cost of accessing leaf nodes (data points) but also the CPU-cost of performing dominance tests. Second, the batch bypassing rule, though incurring extra I/O-cost to access additional information required to find incomparable points, significantly reduces the CPU-cost and also the overall cost.

(1) **Batch pruning rule**: We now formally discuss our (1) batch pruning rule and (2) its efficient computation.

First, we will prove that if a point \( p \) dominates all four corner points of an MBR \( e \), \( p \) dominates all points in the MBR \( e \) (Theorem 4) using the following two lemmas:

**Lemma 5.** Given a parabola \( b \) with focus \( f \) and directrix \( \ell \), for any point \( p \) lying on \( \ell \), the bisecting line \( \ell_\perp(f, p) \) of \( f \) and \( p \) is the tangent of the parabola at a point \( p' \) on \( b \) whose \( x \)-coordinate is same as \( p \).

**Proof.** A parabola is a locus of points equidistant to a point (focus \( f \)) and a line (directrix \( \ell \)). By the definition of parabola, the bisecting line \( \ell_\perp(f, p) \) and the tangent at \( p' \) are equal. \( \square \)

![Figure 9: On Lemma 6](image)

**Lemma 6.** If a point \( p \) dominates two points \( p_1 \) and \( p_2 \), \( p \) dominates every point on the line segment \( \overline{p_1p_2} \).
Proof. Let \( b \) be a parabola of a point \( p \) and a line passing through two points \( p_1 \) and \( p_2 \). Since \( p \) dominates \( p_1 \) and \( p_2 \), the query points lie in the intersecting area (shaded area in Figure 9) of two half planes \( h(p_1, p) \) and \( h(p_2, p) \). For any point \( p' \) lying on \( p_1p_2 \), the bisecting line \( b' \) of \( pp' \) is a tangent of the parabola by Lemma 5. Since the half plane \( h(p', p) \) always includes \( h(p_1, p) \cap h(p_2, p) \), \( p \) dominates \( p' \).

**Theorem 4 (Batch pruning rule).** A point \( p \) dominates every point in an MBR \( e \) if \( p \) dominates \( e \)'s four corner points: \( p_1, p_2, p_3 \), and \( p_4 \).

**Proof.** By Lemma 6, \( p \) dominates every point on the line segment \( p_i p_j \) where \( i = 1, 2, 3 \) and \( j = 2, 3, 4 \). Then, for any point \( p' \) lying inside the MBR \( e \), \( p \) dominates \( p' \) because we can draw a line segment which passes through \( p' \), and whose two endpoints lie on the line segments \( p_i p_j \)'s where \( i = 1, 2, 3 \) and \( j = 2, 3, 4 \). Thus, the two endpoints are dominated by \( p \), and \( p' \) is also dominated by \( p \) by Lemma 6.

Second, we discuss how the computation of the above rule can be expedited in some scenarios. Specifically, instead of testing dominance with all four corner points, one can reduce such tests to three farther points, i.e., \( p_1, p_2, \) and \( p_3 \) as shown in Figure 10, in the scenario where point \( p \) falls within the shaded area in the figure. More formally,

**Theorem 5 (Expedited pruning rule).** Let \( e \) be an MBR whose four corner points are \( p_1(l_x, u_y), p_2(l_x, l_y), p_3(u_x, l_y), \) and \( p_4(u_x, u_y) \) where \( l_x < u_x \) and \( l_y < u_y \). Suppose that \( p \) is not a point such that \( l_x \leq p_x \leq u_x \) or \( l_y \leq p_y \leq u_y \). If \( p_i \) is the closest point to \( p \) and \( p \) dominates \( p_j (\forall j \neq i) \), then \( p \) dominates \( e \).

**Proof.** Without loss of generality, assume that \( p_4 \) is the closest point to \( p \). \( p_4 \) is on either the triangle \( \triangle pp_1 p_2 \) or \( \triangle pp_2 p_3 \). By Lemma 8 below, \( p \) also dominates \( p_4 \). Thus, \( p \) dominates any point \( p' \) in \( e \) by Theorem 4. See Figure 10.

![Figure 10: On Theorem 5](image1)

![Figure 11: On Lemma 7](image2)

The following two lemmas support the proof of the above theorem:

**Lemma 7.** If a point \( p_1 \) dominates a point \( p_2 \), \( p_1 \) dominates any other point \( p \) on the line segment \( p_1 p_2 \).
Proof. Since \( p_1 \) dominates \( p_2 \), \( Q \) is in the half plane \( h(p_2, p_1) \). \( h(p_2, p_1) \subset h(p, p_1) \) and \( Q \) is also in the half plane \( h(p, p_1) \) because \( p \) is on the line segment \( pp_1 \) as Figure 11 depicts. Thus, \( p_1 \) dominates \( p \) by Lemma 3. \( \square \)

Lemma 8. If a point \( p \) dominates two points \( p_1 \) and \( p_2 \), \( p \) dominates any other points \( p' \) on the triangle \( \triangle pp_1p_2 \).

Proof. We can always draw a line segment that starts from \( p \), passing through \( p' \) and ending at a point \( p'' \) on the line segment \( p_1p_2 \). Thus, \( p \) dominates \( p'' \) by Lemma 6 and then \( p \) dominates \( p' \) by Lemma 7. \( \square \)

(2) Batch bypassing rule: We now discuss how we can reduce the number of dominance tests between incomparable skyline points in Theorem 6.

Theorem 6. If a Voronoi cell \( VC(p) \) intersects \( CH_I(Q) \), \( p \) does not dominate any other data points, i.e., \( p' \in P \setminus \{p\} \).

Proof. We prove this by contradiction. Assume that \( p \) dominates another point \( p' \). \( CH_I(Q) \subset h(p', p) \) by Lemma 3, and \( VC(p) \cap CH_I(Q) \subset h(p', p) \). Therefore, every point in \( VC(p) \cap CH_I(Q) \) is closer to \( p' \) than \( p \), which contradicts our assumption by the definition of Voronoi cell \( VC(p) \). \( \square \)

By Theorem 6, the skyline points whose Voronoi cells intersect with \( CH_I(Q) \) are “incomparable” with all the other points. Algorithm BBFS maintains the skyline points of those “intersecting” cells and others separately into respective sets \( FS_I(Q) \) and \( FS_N(Q) \), and ignores dominance tests on incomparable skyline points in \( FS_I(Q) \).

For an efficient intersection test between \( VC(p) \) and \( CH_I(Q) \), as Figure 12 illustrates, we categorize a point \( p \) into three cases: (1) \( p \) is “contained” in \( CH(Q) \), e.g., \( p_0 \), (2) \( VC(p) \) and \( CH_I(Q) \) are “disjoint,” e.g., \( p_1 \), and (3) \( VC(p) \) “overlaps” with \( CH_I(Q) \), e.g., \( p_2 \). Specifically, we implement each case as follows:
• **Contained:** If \( p \) lies in \( CH(Q) \), \( VC(p) \) certainly intersects with \( CH_I(Q) \). Checking such containment can be done in \( \log(|CH_v(Q)|) \) using a binary search on the edges of convex hull. A binary search described in [20] can be used to find the closest edge and check the containment with respect to the edge found.

• **Disjoint:** If two circles containing \( CH(Q) \) and \( VC(p) \) do not intersect, e.g., two dashed circles in Figure 12, \( VC(p) \) cannot intersect \( CH_I(Q) \). An efficient way to identify one of such circles for \( CH(Q) \) is using the centroid of \( CH_v(Q) \) as the center point \( c \) and setting its radius as the distance to the farthest vertex. Similarly, the circle for \( VC(p) \) can be generated by selecting \( p \) itself as the center point and setting its radius as the distance to the farthest vertex.

• **Overlapped:** If at least one edge of \( VC(p) \) intersects with \( CH(Q) \), \( VC(p) \) intersects with \( CH_I(Q) \). To check whether each edge intersects, we use a parametric line-clipping algorithm [21].

By sequentially testing these three cases, the skyline points are separated into \( FS_I(Q) \) or \( FS_N(Q) \). More specifically, “contained” \( p \) goes to \( FS_I(Q) \), “disjoint” \( p \) goes to \( FS_N(Q) \), and “overlapped” \( p \) is checked for the possible intersection and goes to \( FS_I(Q) \) if it intersects, and \( FS_N(Q) \) otherwise.

This intersection test reduces unnecessary I/Os by bypassing dominance tests between incomparable skylines, and incurs none or very limited I/Os. Contained case requires only in-memory computations on \( CH_v(Q) \) and disjoint case does not require the loading of Voronoi cells when the radii are known from traversing R-tree (radii can be retrieved with zero cost as the byproduct of a tree traversal, by simply augmenting each R-tree’s leaf node with the radius). Only the points in the overlapped case require BBFS to read the Voronoi cells from the disk.

With the above three cases, BBFS performs dominance tests only on skyline points in \( FS_N(Q) \), as described in line 6 of the pseudo-code formally describing Algorithm BBFS building upon the above theorems and lemmas.

As a running example, Figure 13 illustrates Algorithm BBFS with 20 data points and four convex query points. To illustrate, Figure 13(a) shows an R-tree index, Figure 13(b) shows the corresponding Voronoi diagram, and Figure 13(c) describes the search procedure for skyline points.

At the first iteration, BBFS initializes the max-heap \( H \) with three child nodes of the root node, i.e., \( N_1, N_2, \) and \( N_3 \). Since \( N_3 \) among these has the maximum SumDist, this node at the top of \( H \) is popped, and its child nodes, i.e., \( N_{10}, N_{11}, \) and \( N_{12} \), are inserted. After nine iterations, \( p_9 \) at the top of \( H \) is popped and identified as a skyline point because \( FS_N(Q) \) is empty. Next, \( p_1 \) becomes the second skyline since \( \ell_1(p_9,p_1) \) intersects the convex hull \( \square q_1 q_2 q_3 q_4 \) (Lemma 4). Similarly, \( p_2 \) and \( p_8 \) also become skyline points since no point among the current skyline points dominates them. These four points \{\( p_9, p_1, p_2, p_8 \)\} are inserted into \( FS_N(Q) \), as they can potentially dominate other points according to Theorem 6. In contrast, at the 15-th iteration, \( p_{17} \) is inserted into \( FS_I(Q) \) because
Algorithm 2: BBFS($Q, T$)

**Input**: Set of query points $Q$; R-tree $T$

**Output**: Set of skyline points $FS(Q)$

1. $Q' \leftarrow CH_v(Q)$; /* compute $CH(Q)$ */
2. $FS_I(Q) \leftarrow \{}$; $FS_N(Q) \leftarrow \{}$
3. Initialize a priority queue $H$ with $T$’s root node
4. **while** $H$ is not empty **do**
5. \[ e \leftarrow H.pop() \]
6. \[ \text{if } \forall s \in FS_N(Q), \ s \notin Q \ e \text{ then} \]
7. \[ \text{if } e \text{ is a leaf node then} \]
8. \[ FS_I(Q).insert(e) \]
9. \[ \text{else} \]
10. \[ FS_N(Q).insert(e) \]
11. \[ \text{else} \]
12. \[ \text{foreach child } c \text{ of } e \text{ do} \]
13. \[ \text{Compute SumDist}(c, Q') \]
14. \[ H.push(c) \]
15. \[ FS(Q) \leftarrow FS_I(Q) \cup FS_N(Q) \]
16. **return** $FS(Q)$

$VC(p_{17})$ intersects the convex hull in Figure 13(b). Thus $p_{17}$ cannot dominate any other points according to Theorem 6. The 24-th iteration illustrates how MBR $N_8$ can be pruned out by $p_8$, by testing dominance with four corner points (Theorem 4). In the next step, MBR $N_6$ is also dominated by $p_1$ with only three dominance tests (Theorem 5). Finally, at the 29-th iteration, $H$ becomes empty and we retrieve the 14 skyline points, i.e., nine points in $FS_N(Q)$ and five points in $FS_I(Q)$. The shaded cells in Figure 13(b) are Voronoi cells of points in $FS_I(Q)$.

### 4.3. Approximate FSSQ (AFSSQ)

In this section, we discuss how we can reduce the processing cost by approximation without significantly affecting the quality of the results. There are two approaches to achieve this goal.

- **Approximating computation**: A widely adopted approach is to approximate the process of computing $FS(Q)$ to identify $FS_a(Q) \subset FS(Q)$ that “best represents” $FS(Q)$.

- **Approximating queries**: Alternatively, we can approximate query $Q$ into its representative subset $Q_a$ and output $FS(Q_a)$ as approximate answers.

In this paper, we pursue the query approximation approach, since adopting an existing work taking another approach requires a linear scan. In clear contrast, the query approximation approach not only avoids such a scan but also
reduces the costs by reducing \(|CH_a(Q)|\) which directly affects the performances of both TFSS and BBFS.

For effective query approximation, we first identify the quality metric. In particular, we adopt the classic quality metrics of precision and recall. More specifically, we require all approximate results \(FS(Q_a)\) to be skyline points, i.e., \(FS(Q_a) \subseteq FS(Q)\), or perfect precision, and aim at maximizing recall:

- **Perfect precision**: \(FS(Q_a)\) must not include false positive skyline points, i.e., \(FS(Q_a) \subseteq FS(Q)\).

- **Maximum recall**: \(|FS(Q_a)|\) should be “close” to \(|FS(Q)|\).

We can easily ensure the first requirement by using Lemma 2, which states that if \(Q_a \subseteq CH_a(Q)\) then \(FS(Q_a) \subseteq FS(Q)\).
The second requirement is abstracted as an expensive optimization problem of finding $Q_a \subset CH_v(Q)$ of size $k$, i.e., $|Q_a| = k$, such that

$$Q_a = \arg \min_{\forall Q' \subset CH_v(Q), |Q'|=k} |FS(Q) - FS(Q')|$$

(5)

To avoid exhaustive enumeration of query subsets, we approximate the problem to find $Q_a$ such that $CH(Q_a)$ is similar to $CH(Q)$. Recall that the dominance test is determined by the intersection with a query convex hull, which indicates if the shape of the query convex hull $CH(Q_a)$ is similar to that of $CH(Q)$, $FS(Q_a)$ and $FS(Q)$ would also be similar.

To find approximate query points $Q_a \subset Q$ such that $CH(Q_a)$ and $CH(Q)$ are similar, we use the Hausdorff distance [22]. The Hausdorff distance, which is a similarity measure between two sets of points $A$ and $B$, has been extensively studied for applications in computational geometry and graphics. This distance metric is formally defined as follows:

$$H(A, B) = \max \{d(A, B), d(B, A)\}$$

(6)

where

$$d(A, B) = \max_{a \in A} \min_{b \in B} \|a, b\|$$

(7)

Lower Hausdorff distance indicates higher similarity between $A$ and $B$.

Using the Hausdorff distance, our problem can be reformulated as follows:

$$Q_a = \arg \min_{\forall Q' \subset CH_v(Q), |Q'|=k} H(CH(Q), CH(Q'))$$

(8)

where $CH_v(Q)$ denotes the given convex query points and $k < |CH_v(Q)|$ denotes the number of approximate convex points. Note that two arguments $CH(Q)$ and $CH(Q')$ of $H(\cdot, \cdot)$ are convex hulls different from their convex points $CH_v(Q)$ and $CH_v(Q')$, respectively.

However, this problem is equivalent to the $k$-center problem, known to be NP-hard [23]. Although a dynamic-programming based exact solution satisfying Equation (8) is a possibility, we do not pursue such a solution due to the prohibitive costs for the large query points set $CH_v(Q)$. Therefore, we develop a greedy strategy, which iteratively eliminates a point from $CH_v(Q)$ until we identify $k$ representative query points. At each step, such strategy analyzes query points and identifies the one that “least impacts” the Hausdorff distance $H$. More formally,

$$q = \arg \min_{q' \in Q} H(CH(Q' - \{q\}), CH(Q'))$$

(9)

Suppose that two points $q_l$ and $q_r$ are consecutive points of a point $p$ on both sides in the convex hull. Then, $H(CH(Q' - \{q\}), CH(Q'))$ is equal to the distance from $q$ to $q_lq_r$, or the “height” of the triangle $\triangle qq_lq_r$ (see Figure 14).

As a heuristic function to quantify such an impact for point $q$, we use the “height” of the triangle $\triangle qq_lq_r$. Figure 14 illustrates such triangle for $q$. 

20
We now argue why such height suggests the degree of impact. By Lemma 4, given query points $Q$, two points $p_1$ and $p_2$ are incomparable if the bisecting line $\ell_1(p_1, p_2)$ intersects $CH(Q)$. Otherwise, either one dominates another. See Figure 14. Such bisecting line $l$ can be categorized into the following four cases: (1) $l$ is like $\ell_1$ in the figure, passing through only $\triangle qqq$, (2) $l$ is like $\ell_2$ passing through both $\triangle qqq$ and $CH(Q' - \{q\})$, (3) $l$ is like $\ell_3$ not passing through both $\triangle qqq$ and $CH(Q' - \{q\})$, and (4) $l$ is like $\ell_4$ passing through only $CH(Q' - \{q\})$. Among these cases, eliminating $q$ only affects the skyline results in the first case, as the two end points defining the bisecting line $\ell_1$ are originally incomparable with respect to $Q'$ but either one becomes dominated with respect to $Q' - \{q\}$ after eliminating $q$. Meanwhile, in all the other cases, the skyline points stay the same, before and after eliminating $q$. Figure 14 shows that, $l$ is more likely to fall into the first case, if the “height” of triangle, or $H(CH(Q' - \{q\}), CH(Q'))$, is higher.

![Figure 14: On Equation 9](image)

5. Experiments

This section reports our experimental results to evaluate our proposed algorithms. First, we report our experimental settings in Section 5.1. Second, we validate the efficiency and effectiveness of our algorithms for processing FSSQ over various synthetic datasets and a real-life dataset in Section 5.2. Our experiments were carried out on a machine with a Xeon 3.20 GHz CPU and 1GB RAM running LINUX. All implementations were written in C++.

5.1. Experimental Settings

This section describes how we generate synthetic datasets and query points to evaluate the scalability over extensive evaluation settings, and introduces a real-life dataset of points of interest (POI) widely adopted in spatial database literature.

5.1.1. Synthetic Data and Query

To extensively evaluate the efficiency over synthetic datasets, we vary the following three parameters: (1) data cardinality $|P|$, (2) query cardinality $|Q|$, and (3) the size $|R(Q)|$ of rectangle $R(Q)$ bounding $Q$. More specifically, we
generate uniformly distributed datasets from 0.2M to 1M in two-dimensional unit space. With these datasets, we investigate the effect of the number of query points from 5 to 100. Simultaneously, we generate the query points by selecting a rectangle $R(Q)$ of varying sizes (30% to 70% of the data space) and randomly distributing the query points in that $R(Q)$. Table 2 summarizes the parameter setup with the default values used in our experiments.

**Table 2: Parameter setup for synthetic datasets**

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting : default</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dimensionality</td>
<td>2</td>
</tr>
<tr>
<td>Cardinality $</td>
<td>P</td>
</tr>
<tr>
<td>Distribution of $P$</td>
<td>Uniform</td>
</tr>
<tr>
<td>Number of query points $</td>
<td>Q</td>
</tr>
<tr>
<td>Size of bounding rectangle $</td>
<td>R(Q)</td>
</tr>
</tbody>
</table>

5.1.2. Real-life Data and Query

We evaluate our proposed algorithms with a real-life POI dataset corresponding to scenarios of Examples 2 in California. Note that, we normalize all data and query points into Cartesian coordinates $(x, y)$ where $0 < x, y < 1$. We use 104,770 potential locations\(^1\) as the data points, and 2,341 Starbucks locations\(^2\) as the query points. More specifically, we randomly select $|Q|$ query points among these Starbucks locations. Figure 15 depicts such POI dataset: Figure 15(a) shows 10,000 randomly sampled POIs and Figure 15(b) shows 2,341 Starbucks locations.

\(^1\)http://www.cs.fsu.edu/~lifeifei/SpatialDataset.htm
\(^2\)http://poidirectory.com/
5.2. Experimental Results

This section reports the experimental results over several synthetic datasets, and a real-life dataset described in Section 5.1. We evaluate our exact and approximation algorithms respectively. Note that each result is measured as the average of 100 evaluations.

5.2.1. Efficiency of FSSQ

We evaluate the efficiency of two non-index algorithms, SFS and SaLSa, and our proposed algorithms, TFSS (baseline), BBFS-N, and BBFS over varying $|P|$, $|Q|$, and $|R(Q)|$, where BBFS-N is a naive version of BBFS excluding Theorems 5 and 6. Note that we compare SFS and SaLSa, non-index state-of-the-art algorithms due to the dynamic nature of transformed data as described in Section 4.1.

In particular, we adopt four different measures: (1) elapsed time up to returning the whole skyline points, (2) I/O cost, or the number of accessing R-tree nodes (whose size is fixed to 1 kilobyte) and Voronoi cells, (3) the number of dominance tests, and (4) the maximum heap size, or the number of entries (MBRs and points) inserted into the heap, during the execution.

Note that we implemented SFS and SaLSa as main-memory algorithms loading the entire dataset into the main memory. We thus cannot report I/Os for these algorithms and instead report only (1) elapsed time and (3) the number of dominance tests for them.

(1) Synthetic Datasets: Table 3 summarizes the average number of skyline points over various parameter settings (the shaded row shows the result under the default setting). The reason why the number of skylines increases, especially as $|Q|$ and $|R(Q)|$ increase, will be explained with experiment results.

Table 3: Average $|FS(Q)|$ over $|P| (\times 10^6)$, $|Q|$, and $|R(Q)|$

| $|P|$  | $|FS(Q)|$ | $|Q|$  | $|CH(Q)|$ | $|FS(Q)|$ | $|R(Q)|$ | $|CH(Q)|$ | $|FS(Q)|$ |
|-------|-----------|-------|----------|-----------|--------|----------|-----------|
| 0.2   | 1,062.52  | 5     | 4.19     | 148.83    | 0.3    | 7.81     | 202.60    |
| 0.4   | 1,817.04  | 20    | 7.65     | 1,817.04  | 0.5    | 7.65     | 1,817.04  |
| 0.6   | 2,484.97  | 40    | 9.43     | 6,231.07  | 0.6    | 7.76     | 7,240.39  |
| 0.8   | 3,133.31  | 60    | 10.41    | 13,060.11 | 0.7    | 7.99     | 21,530.64 |
| 1.0   | 3,769.01  | 80    | 11.18    | 19,033.80 |        |          |           |
|       |           | 100   | 12.19    | 21,399.16 |        |          |           |

Figure 16 shows the effect of the number of data points $|P|$. Observe from Figure 16(a) and (c) that TFSS performs comparably to SFS and SaLSa. However, TFSS has an advantage of selectively loading only necessary data into the memory. This enables scalability to handle a large dataset that cannot fit memory with low I/O cost, unlike SFS and SaLSa requiring I/O-expensive external sorting in that case. For example, when $|P| = 1M$, TFSS loads less than 25K data points, i.e., only 2.5% of entire data, into heaps. Moreover, BBFS-N and BBFS reduce memory usage further by using only a single heap, compared to TFSS requiring multiple heaps.

23
Observe also that, BBFS-N and BBFS considerably outperform others in all measures. By applying the batch pruning rule, BBFS-N and BBFS perform much less I/Os and dominance tests than TFSS as Figure 16(b) and (c) depict, respectively. In particular, the number of I/Os proves the “sub-linearity” of our algorithms at which we aimed. For instance, under the default setting, BBFS accesses only 5% of the dataset. Since the number of accessed MBRs is about 400 and the number of entries in an MBR is about 50, the total number of accessed points is about 20K, which is 5% of the dataset with |P| = 0.4M. Moreover, the performance gap between BBFS-N and BBFS shows the effectiveness of the batch bypassing rule, which overcomes additional I/Os to read Voronoi cells, by significantly saving CPU cost to reduce the overall elapsed time.

The performance graphs in Figure 16(a) and (c) show similar tendencies, which suggest that the number of dominance tests well represents the overall elapsed time. Note that Figure 16(a) and (c) are log-scaled.

Figure 16: Over cardinality |P| for synthetic datasets

Figure 17 shows the effect of query cardinality |Q|. Overall, we can make similar observations as for data cardinality in Figure 16, except that the elapsed time scales more gracefully over |Q| in Figure 17(a), compared to Figure 16(a). This difference can be explained by the fact that the performance of our algo-
Algorithm depends on $|CH_v(Q)|$ and $|CH_v(Q)| \ll |Q|$. Table 3 reports the number of convex query points $|CH_v(Q)|$.

![Graph](image1.png)  ![Graph](image2.png)

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

![Graph](image3.png)  ![Graph](image4.png)

(c) Dominance tests (d) Heap size

Figure 17: Over query cardinality $|Q|$ for synthetic datasets

Figure 18 describes the effect of the size of $R(Q)$. We can observe a similarity to previous experiments over $|P|$ and $|Q|$ as reported in Figure 16 and 17 respectively. However, we note that the elapsed time is more sensitive to $|R(Q)|$, which can be explained by the high sensitivity of the number of skyline points over $|R(Q)|$ as reported in Table 3. In the transformed $|CH_v(Q)|$-dimensional space, attribute values, distances from query points, tend to be “anti-correlated,” as query sparsity increases, where far points from one query are highly likely to be close to another query. In the presence of anti-correlation, the size of skyline results increases considerably, which negatively affects the overall performances. However, our proposed algorithm BBFS shows high scalability over $|R(Q)|$. Observe from Figure 18(a) that its performance gap from BBFS-N increases as $|R(Q)|$ increases, because the batch bypassing rule of BBFS gets more effective as $|FS_I(Q)|$ increases.

(2) California’s POI Dataset: Figure 15(c) shows the skyline points of 10,000 sampled POIs in Figure 15(a) with respect to 10 Starbucks as query points selected in Figure 15(b). For this query, $CH_v(Q)$ consists of 7 query
points, and the skyline points plotted are scattered over the whole data space. Figure 19 shows performance results over varying $|Q|$. We randomly select $|Q|$ Starbucks locations as query points using 104,770 all POIs as data points. Note that we show only the results of BBFS-N and BBFS over the elapsed time and dominance tests, because they outperform all the others consistently with results obtained from the synthetic datasets. As the performance results over synthetic datasets demonstrated in Figure 17, the BBFS still outperforms BBFS-N, which suggests the effectiveness of the bypassing rule.

5.2.2. Efficiency and Effectiveness of AFSSQ

We now validate our approximation algorithm developed in Section 4.3 with respect to efficiency and result quality. First, to evaluate efficiency, we report the elapsed time. Second, to evaluate result quality of approximate skyline points $FS(Q_a)$, we use recall metrics, i.e., $|FS(Q_a)|/|FS(Q)|$. Using these metrics, we compare three different algorithms Exact, ShapeApp, and NaïveApp, where Exact is BBFS itself, ShapeApp is our proposed approximation algorithm, and NaïveApp is a baseline approximation algorithm randomly selecting $k$ query points among convex query points, i.e., satisfying only perfect precision require-
Figure 19: Over query cardinality $|Q|$ for the POI dataset

Figure 20: Over the number $k$ of reduced query points for synthetic datasets

6. Conclusion

We introduced the pareto-optimization on facility or business locations as the FSSQ problem for spatial databases and proved that the duality between...
SSQ and FSSQ does not hold. We studied a baseline algorithm TFSS and then proposed an efficient algorithm BBFS, which significantly outperforms TFSS by exploiting spatial locality of spatial data points. In addition, our proposed algorithm decreases memory overhead, and also supports progressive retrieval. We also studied its approximation in order to trade accuracy for efficiency. We evaluated our proposed algorithms over synthetic and real-life datasets.

References


