Reverse Regret Query

Weicheng Wang, Raymond Chi-Wing Wong✉
Hong Kong University of Science and Technology
wwangby@connect.ust.hk, raywong@cse.ust.hk

H. V. Jagadish
University of Michigan
jag@umich.edu

Min Xie✉
Shenzhen Institute of Computing Sciences
xiemin@sics.ac.cn

***Abstract*—Reverse operators have lately gained much attention within the realm of multi-criteria decision-making. While forward operators, such as skyline, seek to identify products that may interest a customer, reverse operators identify prospective customers who are likely to be attracted to a particular product. Specifically, for each customer, they assign scores to all products w.r.t. the customer's preference and then rank the products based on these scores. If the particular product ranks high, the customer is considered a prospective customer for that product. However, relying purely on rankings might cause misleading results, as rankings emphasize the products' relative positions without accounting for their score differences. In a competitive market, a comparatively low-ranked product may have a score that is nearly indistinguishable from that of the top-tier product(s), and thus, may still be interesting to the customer. In this paper, we directly utilize scores to evaluate products, enabling more accurate identification of prospective customers.**

We refer to our problem as the reverse regret query (RRQ) and make several contributions. First, for the special case in which each product is described by two attributes, we propose an algorithm *Sweeping* that only takes linear time. Second, for the general case in which each product can be described by multiple attributes, we present two algorithms: an exact algorithm *E-PT* and a faster approximate algorithm *A-PC*. We conducted experiments on synthetic and real datasets. The results confirm that evaluating products via scores provides a sound and insightful way of identifying prospective customers. Under typical settings, our proposed algorithms execute faster than existing ones by 1-3 orders of magnitude.

***Index Terms*—reverse regret, query optimization**

I. Introduction

The *reverse* version of the *multi-criteria decision-making* problem [1]–[4] has recently become increasingly significant. Unlike the *forward* version [5]–[7], which aims to pinpoint products that align with customers' preferences, the *reverse* version seeks to identify prospective customers who are likely to be interested in a particular product. This problem arises in multiple scenarios, particularly for product design, where a manufacturer seeks to create products with broad appeal in a heterogeneous marketplace.

To motivate the problem, consider a simplified situation of a car manufacturer. Suppose that there is a set $\mathcal{D}$ of cars $\boldsymbol{p_1}$, $\boldsymbol{p_2}$, etc., in the market, each of which is described by two attributes as shown in Table I. The manufacturer wants to estimate the demand for the design of a new car $\boldsymbol{q}$.

In the literature [7]–[9], the customer's preference is commonly modeled by a *utility function* $f_{\boldsymbol{u}}$. This utility function is different for each prospective customer. Consider a customer with utility function $f_{\boldsymbol{u}_1}$. Each car has a function score, as shown in the rightmost column of the table. If the score of car $\boldsymbol{q}$ is high enough, car $\boldsymbol{q}$ will be considered for purchase by the customer. Thus, to find prospective customers, we need to search for the utility functions based on which the score of car $\boldsymbol{q}$ is high enough. If the number of such utility functions is large, there will be many prospective customers for car $\boldsymbol{q}$.

Table I: Dataset Car

Car	Horsepower ($\times 10^2$ hp)	Safety Rating	$f_{\boldsymbol{u}_1}(\cdot)$
$\boldsymbol{p}_1$	4.5	5	4.55
$\boldsymbol{p}_2$	4.6	4	4.54
$\boldsymbol{p}_3$	5.0	1	4.60
$\boldsymbol{q}$	4.8	2	4.52

When mentioning that "the score of car $\boldsymbol{q}$ is high enough" for a prospective customer, we refer to a widely applied measurement called *regret ratio* [5], [10], [11]. Let $\boldsymbol{p}^*$ represent the car with the highest score in the market. The *regret ratio* of $\boldsymbol{q}$ is defined as the proportion of the score difference between $\boldsymbol{q}$ and $\boldsymbol{p}^*$ to the score of $\boldsymbol{p}^*$. We require the regret ratio of $\boldsymbol{q}$ to be below a small threshold, which posits that the score of car $\boldsymbol{q}$ is close to the highest in the market. Formally, we define problem reverse regret query (RRQ). Given a query product $\boldsymbol{q}$, the goal is to identify all prospective customers of $\boldsymbol{q}$. In essence, it seeks all the utility functions based on which the *regret ratio* of $\boldsymbol{q}$ is below a given threshold.

To the best of our knowledge, we are the first to study problem RRQ. There are some closely related studies [1]–[4], [12]–[14] focusing on reverse operators, but they are distinct to ours. One representative operator is the *reverse top-k query* [1]–[4]. It ranks the products in descending order based on their scores w.r.t. each utility function, and then returns the utility functions where car $\boldsymbol{q}$ ranks within the top-k positions. The limitation of this work is that it concentrates on product rankings, which are secondary sources of information derived from product scores. This information emphasizes the products' relative positions without accounting for their score differences, potentially obscuring details about the product's attractiveness to prospective customers.

For instance, consider Table I. The fourth column displays the score of each car based on utility function $f_{\boldsymbol{u}_1}$. It is easy to see that car $\boldsymbol{q}$ ranks last among the cars listed. Suppose that $k = 3$. Following the reverse top-k query, car $\boldsymbol{q}$ would not interest customers. However, this considerable gap in the ranking is caused by a minor score difference, i.e., only about 1.7% (calculated as $\frac{4.6-4.52}{4.6}$). The score of $\boldsymbol{q}$ is close to those of the others. Car $\boldsymbol{q}$ could attract customers' attention. It would be imprudent to dismiss the competitiveness of car $\boldsymbol{q}$.

In this paper, we directly utilize product scores to evaluate

products. It is worth noting that the concept of the regret ratio uses the highest score as a pivot. However, "the highest score" might sometimes be overly stringent [15], [16]. To broaden our problem and accommodate various scenarios, following [15], [16], we relax the comparison from the highest score to the k-th highest score, where $k \geq 1$. This relaxation allows for more flexibility in evaluating products, ultimately enhancing the versatility and effectiveness of our problem.

Contributions. Our contributions are described as follows.

- To the best of our knowledge, we are the first to propose problem *reverse regret query* (problem RRQ). This problem employs a comprehensive evaluation of products (i.e., product scores) to identify prospective customers effectively.
- We propose algorithm *Sweeping* for the special case of RRQ in which each product is described by two attributes. We prove that algorithm *Sweeping* only takes linear time cost.
- We propose two algorithms for the general case of RRQ, in which each product is described by multiple attributes. Algorithm *E-PT* is an exact algorithm that returns complete results (i.e., all prospective customers). Algorithm *A-PC* is an approximate algorithm that may miss some results (i.e., miss some prospective customers) but runs faster than *E-PT*.
- We conducted experiments on synthetic and real datasets to show that our problem setting evaluates prospective customers soundly and insightfully, and our algorithms execute several orders of magnitude faster than existing ones.

The rest of the paper is organized as follows. The related work is reviewed in Section II. The formal problem definition and characteristics are shown in Section III. Section IV describes algorithm *Sweeping* for the special case of RRQ. Section V presents two algorithms *E-PT* and *A-PC* for the general case of RRQ. The experiments are shown in Section VI and finally, Section VII concludes our paper.

II. Related Work

The *multi-criteria decision-making operators* [5], [15], [17]–[19] can be classified into *forward* and *reverse* versions.

There are two widely studied forward operators: the skyline query and the top-k query. The skyline query [17], [20]–[27] is designed to return the skyline products, i.e., the products that are not *dominated* by other products. A product $\boldsymbol{p}$ is said to dominate another product $\boldsymbol{q}$ if $\boldsymbol{p}$ is not worse than $\boldsymbol{q}$ in each attribute and is strictly better in at least one attribute. The deficiency of a skyline query is that its output size is uncontrollable [10], [28]. It is possible that none of the products are dominated by others, and thus, the whole product dataset is returned to customers. In contrast, the top-k query [18], [19], [29]–[33] returns a controllable output size. It models customer preference by a utility function. Based on the utility function, each product is associated with a score and the k products with the highest scores are returned to the customer. Several algorithms [19], [29], [30], [33] are designed for the top-k query. [19], [29] employ sophisticated techniques, such as *Onion* and *kSkyband*, to process the dataset into a small set. Then, they quickly obtain the exact output from the small set. [30], [33] solve the top-k query from the geometric perspective. Unfortunately, there is a practical barrier to the use of the top-k query. It requires customers to specify their utility functions, but customers usually have difficulty in explicitly specifying the exact parameters of the utility function.

Another forward operator is the regret minimization query [5], [6], [15], [16], [34]. The regret minimization query aims to find a representative set S such that the difference between the customer's favorite product in set S and that in the whole product dataset is minimized. There are many algorithms [5], [6], [15], [34] that are proposed for the regret minimization query. [15], [34] adopt a greedy strategy that iteratively identifies the product minimizing the difference and add it into S. [5] samples the product dataset by partitioning the products into different *cubes*. Then, it selects one product from each cube as the output. [6] samples the customer preference and identifies products as output based on the sampled customer preference.

Among the forward operators discussed, they commonly take the customer's perspective to identify products. Instead, the reverse operators seek out potential customers for products. One representative operator is the reverse top-k query.

Given a product $\boldsymbol{q}$, the reverse top-k query [1]–[4] returns all customers such that $\boldsymbol{q}$ is one of the best k products based on customers' preferences. Specifically, [2]–[4] assume that there is a set U of utility functions modeling customer preferences. The objective is to find those utility functions based on which the score of product $\boldsymbol{q}$ is among the top-k. However, [2]–[4] only allow set U to comprise a finite number of *discrete* utility functions. They cannot deal with the entire continuous utility function space, i.e., the entire customer preference space.

Motivated by this insufficiency, [1] studies a more general case in which set U is set to be the entire continuous utility function space. It proposes an algorithm via partitioning. Intuitively, it partitions the utility function space into *cells* and builds a *cell-tree* to index the cells. By processing the cell-tree, it checks the cells and returns utility functions from qualified cells. The weakness of [1] is that it measures products based on the product rankings. This may result in an inappropriate evaluation of products, as discussed in Section I. In contrast, our work considers product scores, which provides a more comprehensive view of products, and consequently, yields a sounder identification of prospective customers. Besides, the cell-tree in [1] is not efficient. In Section VI, we adapted it to compare with our algorithms. The results show that our algorithms execute several orders of magnitude faster than it.

III. Problem Definition

A. Problem Reverse Regret Query

Product/Point. The input consists of a set $\mathcal{D}$ of n products and a query product, each described by d attributes. We represent each product as a d-dimensional point $\boldsymbol{p} = (p[1], p[2], ..., p[d])$. In the following, we use the words product/point and attribute/dimension interchangeably. Without loss of generality, we assume that each dimension is normalized to $(0, 1]$ and a large value in each dimension is preferred by customers. Note that humans typically consider a limited number of

attributes in their decision-making process. Following previous studies [1], [28], we specifically focus on scenarios where the number of attributes remains relatively modest (e.g., $d \leq 7$). Nevertheless, it is important to mention that our methodologies proposed herein are applicable to any setting of d.

Utility Function. Following [1], [2], [5]–[7], [33], we model the customer preference as a linear utility function

$$f_{\boldsymbol{u}}(\boldsymbol{p}) = \boldsymbol{u} \cdot \boldsymbol{p} = \sum_{i=1}^{d} u[i]p[i],$$

which is widely used for modeling customer preference [35], [36]. The experimental studies conducted by [7], [10] also show that linear utility functions can effectively capture how customers evaluate products. Each utility function corresponds to a *utility vector* $\boldsymbol{u} = (u[1], u[2], ..., u[d])$ that captures the importance of each attribute to a customer. A larger $u[i]$ indicates that the i-th attribute is more important to the customer, where $i \in [1, d]$. The domain $\mathcal{U}$ of all utility vectors is called the *utility space*. Moreover, the function score $f_{\boldsymbol{u}}(\boldsymbol{p})$, also known as the *utility* of $\boldsymbol{p}$ w.r.t. $\boldsymbol{u}$, represents how much a customer favors point $\boldsymbol{p}$. If a customer prefers $\boldsymbol{p}$ to $\boldsymbol{q}$, then $f_{\boldsymbol{u}}(\boldsymbol{p}) > f_{\boldsymbol{u}}(\boldsymbol{q})$. Note that, as discussed in [5], [8], [10], the norm of $\boldsymbol{u}$ does not affect the semantics of a utility function. Therefore, without loss of generality, we assume that (1) $u[i] \geq 0$ for each dimension $i \in [1, d]$, and (2) $\sum_{i=1}^{d} u[i] = 1$.

Example 1. Consider Table II where each point has two dimensions. Let $f_{\boldsymbol{u}}(\boldsymbol{p}) = 0.5p[1]+0.5p[2]$ (i.e., $\boldsymbol{u} = (0.5, 0.5)$). The utility of $\boldsymbol{p}_1$ w.r.t. $\boldsymbol{u}$ is $f(\boldsymbol{p}_1) = 0.5 \times 0.20 + 0.5 \times 0.92 = 0.56$. We can obtain the utilities of other points in a similar way.

Regret Point. Given a utility function $f_{\boldsymbol{u}}$, the points in a dataset $\mathcal{D}$ can be ranked by their utilities w.r.t. $\boldsymbol{u}$ in a descending order. Let us denote the utility of the k-th ranked point by $kmax_{\boldsymbol{p} \in \mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p})$. For example, in Table II, since point p_1 ranks the second, we have $2max_{\boldsymbol{p} \in \mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p}) = f_{\boldsymbol{u}}(\boldsymbol{p}_1) = 0.56$. To evaluate the query point $\boldsymbol{q}$, following [15], [16], we employ the concept of k-regret ratio by utilizing $kmax_{\boldsymbol{p} \in \mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p})$.

Definition 1 (k-Regret Ratio). Given a dataset $\mathcal{D}$ and a utility function $f_{\boldsymbol{u}}$, the k-regret ratio of the query point $\boldsymbol{q}$ is:

$$k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) = \frac{\max(0, kmax_{\boldsymbol{p} \in \mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p}) - f_{\boldsymbol{u}}(\boldsymbol{q}))}{kmax_{\boldsymbol{p} \in \mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p})}$$

Intuitively, the k-regret ratio measures how close the utility of $\boldsymbol{q}$ is to that of the k-th ranked point in $\mathcal{D}$ w.r.t. $\boldsymbol{u}$. Here, we consider the ratio instead of the direct score difference to achieve the property of *scale invariance* [5]. Besides, the numerator incorporates a maximum operation to ensure that the k-regret ratio remains non-negative. The k-regret ratio falls in the range $[0, 1]$. If $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u})$ is below a small threshold ϵ (i.e., $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) < \epsilon$), the utility of $\boldsymbol{q}$ is only slightly smaller or even larger than that of the k-th ranked point. In this case, we say point $\boldsymbol{q}$ is a (k, ϵ)-*regret point* w.r.t. $\boldsymbol{u}$.

Example 2. Continue Example 1, where $\boldsymbol{u} = (0.5, 0.5)$. Suppose that $\boldsymbol{q} = (0.4, 0.7)$ and $\epsilon = 0.1$. The 2-regret ratio of point $\boldsymbol{q}$ is $2\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) = \max(0, 0.56 - 0.55)/0.56 = 0.018 < \epsilon$. Thus, $\boldsymbol{q}$ is a $(2, 0.1)$-regret point w.r.t. $\boldsymbol{u}$.

We now formally define the Reverse Regret Query problem (RRQ). The frequently used notations are summarized in a table that can be found in the technical report [37].

Problem 1 (Reverse Regret Query (RRQ)). Given a dataset $\mathcal{D}$, a query point $\boldsymbol{q}$, an integer k, and a threshold ϵ, we want to find all $\boldsymbol{u} \in \mathcal{U}$ such that $\boldsymbol{q}$ is a (k, ϵ)-regret point w.r.t. $\boldsymbol{u}$.

There are two parameters k and ϵ in Problem 1. Parameter k controls the selection of the *pivot* point. If k is small, we focus on the very best point as the pivot. Parameter ϵ decides the level of closeness required between the query point and the selected pivot. A smaller ϵ implies a stricter criterion for considering a point as sufficiently close. Back to Example 2, utility vector $\boldsymbol{u} = (0.5, 0.5)$ can be returned as an output for problem RRQ since $\boldsymbol{q}$ is a (2, 0.1)-regret point w.r.t. $\boldsymbol{u}$. We say $\boldsymbol{u}$ is a *qualified* utility vector. Note that the utility space $\mathcal{U}$ is a continuous space containing infinite utility vectors. The complete output of problem RRQ are regions in $\mathcal{U}$ that contain qualified utility vectors. Such regions are called *qualified regions*.

B. Problem Characteristics

We formalize our problem RRQ from a geometric perspective. In a d-dimensional geometric space $\mathbb{R}^d$, each utility vector $\boldsymbol{u} \in \mathcal{U}$ can be seen as a point. Recall that we assume (1) $u[i] \geq 0$ for every dimension, and (2) $\sum_{i=1}^{d} u[i] = 1$. The utility space $\mathcal{U} = \{\boldsymbol{u} \in \mathbb{R}_+^d | \sum_{i=1}^{d} u[i] = 1\}$ is a *polyhedron* [38] in $\mathbb{R}^d$, e.g., a line segment when $d = 2$ as shown in Figure 2 or a triangle when $d = 3$ as shown in Figure 1.

For any point $\boldsymbol{p} \in \mathcal{D}$, we can build a *hyper-plane* $h_{\boldsymbol{q},\boldsymbol{p}}$ with query point $\boldsymbol{q}$ in the geometric space $\mathbb{R}^d$ as follows.

$$h_{\boldsymbol{q},\boldsymbol{p}} : \{\boldsymbol{r} \in \mathbb{R}^d | \boldsymbol{r} \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}) = 0\}$$

The hyper-plane passes through the origin with its unit norm in the same direction as $\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}$ [38]. Figure 1 shows an example. For each utility vector $\boldsymbol{u} \in \mathcal{U} \cap h_{\boldsymbol{q},\boldsymbol{p}}$, we have

$$\boldsymbol{u} \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}) = 0, \text{ i.e., } f_{\boldsymbol{u}}(\boldsymbol{q}) = (1-\epsilon) f_{\boldsymbol{u}}(\boldsymbol{p})$$

Hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ divides the utility space into two half-spaces. The half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$ above $h_{\boldsymbol{q},\boldsymbol{p}}$ (yellow part), called *positive half-space*, contains all utility vectors $\boldsymbol{u}$ such that $f_{\boldsymbol{u}}(\boldsymbol{q}) > (1-\epsilon) f_{\boldsymbol{u}}(\boldsymbol{p})$. The half-space $h^-_{\boldsymbol{q},\boldsymbol{p}}$ below $h_{\boldsymbol{q},\boldsymbol{p}}$ (red part), called *negative half-space*, contains all the utility vectors $\boldsymbol{u}$ such that $f_{\boldsymbol{u}}(\boldsymbol{q}) < (1-\epsilon) f_{\boldsymbol{u}}(\boldsymbol{p})$. The unit norm $\frac{\boldsymbol{q}-(1-\epsilon)\boldsymbol{p}}{||\boldsymbol{q}-(1-\epsilon)\boldsymbol{p}||}$ (black arrow) of hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ points to positive half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$. Note that the half-space is bounded by utility space $\mathcal{U}$ instead of being defined on the entire geometric space.

Example 3. Consider Table II. Suppose $\boldsymbol{q} = (0.4, 0.7)$ and threshold $\epsilon = 0.1$. For point $\boldsymbol{p}_1$, we build a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_1}$ in space $\mathbb{R}^2$: $\{\boldsymbol{r} \in \mathbb{R}^2 | \boldsymbol{r} \cdot (0.22, -0.13) = 0\}$ shown as a line in Figure 2. Its unit norm (black arrow) is in the same direction as vector $(0.22, -0.13)$. Similarly for $h_{\boldsymbol{q},\boldsymbol{p}_2}$ and $h_{\boldsymbol{q},\boldsymbol{p}_3}$.

For any two half-spaces, e.g., $h^+_{\boldsymbol{q},\boldsymbol{p}}$ and $h^+_{\boldsymbol{q},\boldsymbol{p}'}$, we say half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$ is *covered* by half-space $h^+_{\boldsymbol{q},\boldsymbol{p}'}$, denoted by $h^+_{\boldsymbol{q},\boldsymbol{p}} \subseteq h^+_{\boldsymbol{q},\boldsymbol{p}'}$, if any utility vector in $h^+_{\boldsymbol{q},\boldsymbol{p}}$ is also contained in $h^+_{\boldsymbol{q},\boldsymbol{p}'}$. For example, in Figure 2, half-space $h^+_{\boldsymbol{q},\boldsymbol{p}_2}$ is covered by $h^+_{\boldsymbol{q},\boldsymbol{p}_3}$.

Table II: Dataset

$\boldsymbol{p}$	$p[1]$	$p[2]$	$f_{(0.5,0.5)}(\boldsymbol{p})$
$\boldsymbol{p}_1$	0.2	0.92	0.56
$\boldsymbol{p}_2$	0.70	0.54	0.62
$\boldsymbol{p}_3$	0.60	0.30	0.45

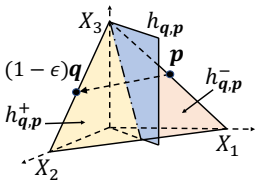


Figure 1: Hyper-plane

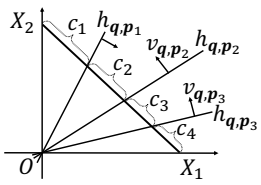


Figure 2: Two Dim

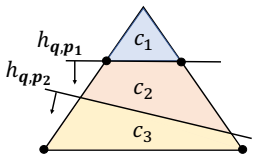


Figure 3: High Dim

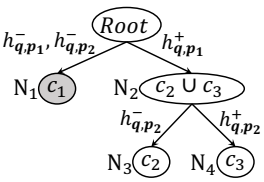


Figure 4: P-Tree

We now apply the hyper-plane/half-space concepts to problem RRQ. For each point $\boldsymbol{p} \in \mathcal{D}$, we build a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ with $\boldsymbol{q}$, dividing the utility space into two half-spaces. Since $|\mathcal{D}| = n$, there are n hyper-planes and $2n$ half-spaces. The n hyper-planes divide the utility space into $O(n^{d-1})$ regions, called *partitions* [38]. Each partition is *covered* by n (positive or negative) half-spaces. Here, we say a partition c is covered by a half-space, e.g., $h^+_{\boldsymbol{q},\boldsymbol{p}}$, denoted by $c \subseteq h^+_{\boldsymbol{q},\boldsymbol{p}}$, if any utility vector in partition c is also in half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$.

Example 4. Continue Example 3 as shown in Figure 2, where $\epsilon = 0.1$. The three hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}_1}$, $h_{\boldsymbol{q},\boldsymbol{p}_2}$, and $h_{\boldsymbol{q},\boldsymbol{p}_3}$ divides the utility space into four partitions c_1, c_2, c_3, and c_4. Partition $c_1 = h^-_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_2} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_3}$ is covered by *one* negative half-space $h^-_{\boldsymbol{q},\boldsymbol{p}_1}$, and two positive half-spaces $h^+_{\boldsymbol{q},\boldsymbol{p}_2}$ and $h^+_{\boldsymbol{q},\boldsymbol{p}_3}$.

The following lemma shows how to determine if query point $\boldsymbol{q}$ is a (k,ϵ)-regret point w.r.t. the utility vectors in a partition. For lack of space, the proofs of some theorems/lemmas in this paper can be found in the technical report [37].

Lemma 1. If and only if a partition c is covered by fewer than k negative half-spaces (i.e., more than $n-k$ positive half-spaces), query point $\boldsymbol{q}$ is a (k,ϵ)-regret point w.r.t. any utility vectors in partition c.

Based on Lemma 1, we can solve problem RRQ in roughly three steps: (1) construct a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ for each point $\boldsymbol{p} \in \mathcal{D}$; (2) compute the partitions based on the hyper-planes constructed; and (3) obtain the partitions that are covered by fewer than k negative half-spaces. To illustrate, assume that $k = 2$ in Example 4. Partitions c_1, c_2, and c_3 will be returned since they are covered by fewer than two negative half-spaces.

IV. Two Dimensional Algorithm

We begin with a special case of problem RRQ in which each point has two dimensions (i.e., $d = 2$). We propose algorithm *Sweeping*, which performs well theoretically and empirically. Intuitively, in a 2-dimensional geometric space $\mathbb{R}^2$, as depicted in Figure 2, the utility space $\mathcal{U}$ is a line segment $\mathcal{L}$ from (0,1) to (1,0). Once the utility space is divided by hyper-planes, the partitions are ordered sub-segments of $\mathcal{L}$ (e.g., in Figure 2, we have $c_1 \to c_2 \to c_3 \to c_4$). Algorithm *Sweeping* conducts a sweep along $\mathcal{L}$ (from (0,1) to (1,0)) and checks each partition sequentially. If a partition is covered by fewer than k negative half-spaces (i.e., more than $n-k$ positive half-spaces), it is a qualified partition to be returned. However, two issues affect the efficiency of *Sweeping*. Firstly, there are many partitions that need to be checked since for every $\boldsymbol{p} \in \mathcal{D}$, a hyper-plane will be created, resulting in $O(n)$ partitions. Secondly, it is costly to check whether a partition is qualified to be returned. Since each partition is covered by n (positive or negative) half-spaces, we need $O(n)$ time to count the negative half-spaces. In the following, we address these two issues, respectively.

Partition Reduction. Let $\boldsymbol{r} = (1,0)$ in $\mathcal{L}$ be a reference utility vector in space $\mathbb{R}^2$. We call a hyper-plane an *inclusive hyper-plane* if its negative half-space contains r, and an *exclusive hyper-plane* otherwise. In Figure 2, $h_{\boldsymbol{q},\boldsymbol{p}_2}$ and $h_{\boldsymbol{q},\boldsymbol{p}_3}$ are two inclusive hyper-planes, and $h_{\boldsymbol{q},\boldsymbol{p}_1}$ is an exclusive hyper-plane.

For all the inclusive hyper-planes, let us rank them based on their intersections with line segment $\mathcal{L}$. Denote the intersection between a hyper-plane h and $\mathcal{L}$ by $\wedge(h,\mathcal{L})$. If $\wedge(h,\mathcal{L})$ is farther away from the X_1-axis, the hyper-plane h ranks higher. For instance, in Figure 2, hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_2}$ ranks higher than hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_3}$, since its intersection with $\mathcal{L}$ is farther away from the X_1-axis than that of $h_{\boldsymbol{q},\boldsymbol{p}_3}$. With a slight abuse of notation, denote the k-th ranked inclusive hyper-plane by lh_k.

Lemma 2. Query point $\boldsymbol{q}$ is not a (k,ϵ)-regret point w.r.t. any utility vectors in the negative half-space of lh_k (i.e., $\boldsymbol{u} \in lh^-_k$).

Lemma 2 indicates that the partitions in lh^-_k are not qualified to be returned. Thus, we can directly omit those partitions. To illustrate, consider Figure 2 with $k = 1$. Since inclusive hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_2}$ ranks the first, partitions $c_3 \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}_2}$ and $c_4 \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}_2}$ can be omitted. Similarly, we can rank all the exclusive hyper-planes based on their intersections with $\mathcal{L}$. If the intersection is farther away from the X_2-axis, the hyper-plane ranks higher. Denote the k-th ranked exclusive hyper-plane by uh_k. The partitions in uh^-_k can also be omitted.

Lemma 3. Based on the partition reduction strategy, we reduce the number of partitions that we need to consider to $O(k)$.

Checking Cost Reduction. Recall that the sweep direction of our algorithm is from (0,1) to (1,0). For each inclusive (resp. exclusive) hyper-plane h, the sweep will pass through $h^+ \to \wedge(h,\mathcal{L}) \to h^-$ (resp. $h^- \to \wedge(h,\mathcal{L}) \to h^+$). In other words, when the sweep reaches $\wedge(h,\mathcal{L})$ of an inclusive (resp. exclusive) hyper-plane h, we know that the remaining partitions to be swept are covered by (resp. are not covered by) h^-. For example, consider an inclusive hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_2}$ in Figure 2. When the sweep reaches intersection $\wedge(h_{\boldsymbol{q},\boldsymbol{p}_2},\mathcal{L})$, the partitions to be swept, say c_3 and c_4, must be covered by $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$.

Following this idea, we maintain an integer Q to track the number of negative half-spaces during the sweep. Assume that partition c is currently being swept and Q holds the number of negative half-spaces covering c. As the sweep advances, the sweep reaches an intersection $\wedge(h,\mathcal{L})$ and the next partition is c'. If hyper-plane h is an inclusive hyper-plane, its negative half-space h^- must cover the remaining partitions (including c') to be swept. Thus, we can obtain the number of negative

Algorithm 1: Algorithm *Sweeping*

1 **Input:** point set $\mathcal{D}$, query point $\boldsymbol{q}$, parameters k and ϵ.
2 **Output:** the set $\mathcal{C}$ of qualified partitions.
3 Find hyper-planes lh_k and uh_k;
4 Exclude the partitions in lh_k^- or uh_k^-;
5 **for** *each partition* c *in* $uh_k^+ \cap lh_k^+$ **do**
6 Update integer Q for partition c;
7 **if** $Q < k$ **then**
8 $\mathcal{C} \leftarrow \mathcal{C} \cup \{c\}$;
9 **return** The set $\mathcal{C}$ of qualified partitions.

half-spaces that cover partition c' by simply adding 1 to Q. Similarly, if hyper-plane h is an exclusive hyper-plane, we can obtain the number by subtracting 1 from Q.

Lemma 4. The checking cost for each partition is $O(1)$.

The pseudocode of algorithm *Sweeping* is shown in Algorithm 1. In the beginning, we find hyper-planes lh_k and uh_k, and exclude the partitions in negative half-spaces lh_k^- or uh_k^- (lines 3-4). Then, we start the sweep process from the partition next to $\wedge(uh_k, \mathcal{L})$ and set Q to be the number of negative half-spaces covering it. The sweep moves towards $\wedge(lh_k, \mathcal{L})$. When we reach the intersection $\wedge(h, \mathcal{L})$ of an inclusive (resp. exclusive) hyper-plane h and enter a new partition c, Q is updated by adding (resp. subtracting) 1 and we check whether partition c is qualified to be returned (lines 6-8). The process stops when all partitions in $uh_k^+ \cap lh_k^+$ are swept (line 5).

To illustrate, consider Figure 2 with $k = 1$. There are three hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}_1}$, $h_{\boldsymbol{q},\boldsymbol{p}_2}$, and $h_{\boldsymbol{q},\boldsymbol{p}_3}$, where $uh_k = h_{\boldsymbol{q},\boldsymbol{p}_1}$ and $lh_k = h_{\boldsymbol{q},\boldsymbol{p}_2}$. We exclude partitions c_1, c_3, and c_4 since they are in either $h^-_{\boldsymbol{q},\boldsymbol{p}_1}$ or $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$. The sweep starts from $\wedge(h_{\boldsymbol{q},\boldsymbol{p}_1}, \mathcal{L})$ to $\wedge(h_{\boldsymbol{q},\boldsymbol{p}_2}, \mathcal{L})$. For partition c_2, since $Q = 0 < k$, it is returned as the final output, i.e., $\mathcal{C} = \{c_2\}$.

Theorem 1. The time complexity of *Sweeping* is $O(n)$.

V. High Dimensional Algorithms

We consider the general case of problem RRQ in which each point has multiple dimensions (i.e., $d \geq 2$). The first algorithm, termed *Partition Tree (E-PT)*, is an exact algorithm. It returns all qualified partitions (i.e., covered by fewer than k negative half-spaces). The second algorithm, named *Progressive Construction (A-PC)*, is an approximate algorithm. It sacrifices some qualified partitions to achieve a better execution time.

A. Exact Algorithm

We construct a tree-structured index, called *Partition Tree* (or *P-Tree* for short), to manage the partitions in the utility space divided by the hyper-planes. For each point $\boldsymbol{p} \in \mathcal{D}$, a hyper-plane $h_{\boldsymbol{p},\boldsymbol{q}}$ is built with query point $\boldsymbol{q}$ and is inserted into the P-Tree. The P-Tree incrementally indexes the partitions divided by the hyper-planes inserted so far. Upon inserting all hyper-planes, the partitions in the P-Tree that are covered by fewer than k negative half-spaces are returned.

1) Partition Tree: In the P-Tree, each leaf node contains one partition and each internal node contains the union of partitions from its reachable leaves. When the context is clear, we also call a union of partitions as *a partition*. For each internal/leaf node N, we use a counter $Q(N)$ to record the number of negative half-spaces that cover the partition stored in N. Consider Figure 3 as an example. The utility space is divided into three partitions by two hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}_1}$ and $h_{\boldsymbol{q},\boldsymbol{p}_2}$. Figure 4 shows the corresponding P-Tree. It has three leaves that contain partitions c_1, c_2, and c_3, respectively. The internal node N_2 contains partition $c_2 \cup c_3$ since it has two reachable leaves, containing c_2 and c_3, respectively. Since partition $c_2 \cup c_3$ is covered by neither $h^-_{\boldsymbol{q},\boldsymbol{p}_1}$ nor $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$, $Q(N_2) = 0$.

The P-Tree is incrementally built by inserting one hyper-plane at a time. Initially, the root is constructed, whose partition is the entire utility space. When a hyper-plane is inserted, the partitions stored in some leaves are divided into smaller ones, causing those leaves to split. Specifically, each hyper-plane insertion begins at the root and proceeds downward. At each node N, the insertion is conducted based on the relationship between $h_{\boldsymbol{q},\boldsymbol{p}}$ and the partition c stored in N as follows.

Case 1: Half-space $h^-_{\boldsymbol{q},\boldsymbol{p}}$ covers partition c, i.e., $c \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}}$. We keep partition c unchanged and increase $Q(N)$ by 1. If $Q(N) \geq k$, node N is marked as invalid and will not be processed by any future hyper-plane insertions. This is because partition c is already covered by k negative half-spaces, and thus, none of the utility vectors in c is qualified to be returned. If $Q(N) < k$ and node N is an internal node, we process the children of N recursively.

Case 2: Half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$ covers partition c, i.e., $c \subseteq h^+_{\boldsymbol{q},\boldsymbol{p}}$. We keep partition c and $Q(N)$ unchanged. The children of N (if any) will not be recursively processed.

Case 3: Hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ intersects partition c, i.e., $c \cap h^-_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$ and $c \cap h^+_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$. Hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ divides partition c into two sub-partitions. If node N is an internal node, the insertion proceeds directly to its children. Otherwise, we build two children N' and N'' for N, containing the two sub-partitions of c, respectively. Precisely, the first child contains partition $c' = c \cap h^-_{\boldsymbol{q},\boldsymbol{p}}$ and $Q(N') = Q(N) + 1$; the second child contains partition $c'' = c \cap h^+_{\boldsymbol{q},\boldsymbol{p}}$ and $Q(N'') = Q(N)$. Note that for node N', if $Q(N') \geq k$, it will be marked as invalid, since partition c' is already covered by k negative half-spaces, and thus, c' is not qualified to be returned.

To illustrate, we build a P-Tree based on Figure 3 with $k = 2$. In the beginning, the partition in the root is the entire utility space $\mathcal{U}$ and $Q(Root) = 0$. Assume that the first hyper-plane inserted is $h_{\boldsymbol{q},\boldsymbol{p}_1}$. It divides the utility space into two partitions $h^-_{\boldsymbol{q},\boldsymbol{p}_1}$ and $h^+_{\boldsymbol{q},\boldsymbol{p}_1}$ (Case 3). We build two children N_1 and N_2 for the root. Node N_1 contains partition $c_1 = h^-_{\boldsymbol{q},\boldsymbol{p}_1}$ and $Q(N_1) = 1$; node N_2 contains partition $h^+_{\boldsymbol{q},\boldsymbol{p}_1}$ and $Q(N_2) = 0$. The second hyper-plane inserted is $h_{\boldsymbol{q},\boldsymbol{p}_2}$. The insertion first comes to the root. Since $h_{\boldsymbol{q},\boldsymbol{p}_2}$ intersects the utility space, the insertion directly proceeds to the children N_1 and N_2 of the root (Case 3). For node N_1, half-space $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$ covers partition c_1 (Case 1). We increase $Q(N_1)$ by

1. Since $Q(N_1) \geq k$, node N_1 is marked as invalid. For node N_2, hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_2}$ divides partition $h^+_{\boldsymbol{q},\boldsymbol{p}_1}$ into two sub-partitions $c_2 = h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^-_{\boldsymbol{q},\boldsymbol{p}_2}$ and $c_3 = h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_2}$ (Case 3). We build two children N_3 and N_4 for N_2, containing partitions c_2 and c_3, respectively, and set $Q(N_3) = 1$ and $Q(N_4) = 0$. The final P-Tree is shown in Figure 4.

During the insertion, a critical step is to check the relationship between a hyper-plane and a partition c. We decide such relationship based on the *extreme points* (i.e. corner points [38]) of the partition. Specifically, assume that partition c has x extreme points $\boldsymbol{e}_1$, $\boldsymbol{e}_2$, ..., $\boldsymbol{e}_x$.

Lemma 5. Given a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$, if $\forall i \in [1, x]$, $e_i \in h^+_{\boldsymbol{q},\boldsymbol{p}}$ (resp. $e_i \in h^-_{\boldsymbol{q},\boldsymbol{p}}$), partition c is covered by $h^+_{\boldsymbol{q},\boldsymbol{p}}$ (resp. $h^-_{\boldsymbol{q},\boldsymbol{p}}$). Otherwise, hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ intersects partition c.

Consider partition $c_2 \cup c_3$ in Figure 3. It is a trapezoid with four extreme points (shown as black dots). Hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_2}$ intersects the partition, since there are two extreme points in $h^+_{\boldsymbol{q},\boldsymbol{p}_2}$ and another two in $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$. Suppose that a partition has x extreme points. It takes $O(x)$ time to check the relationship between all extreme points and the hyper-plane.

2) Acceleration: We develop several strategies to accelerate the P-Tree construction: (1) reduce the number of hyper-planes inserted; (2) establish an efficient order for hyper-plane insertion; (3) speed up the relationship checking between a hyper-plane and a partition; and (4) reduce the number of children creation for leaf nodes.

Hyper-plane Reduction. We reduce the number of hyper-planes inserted into the P-Tree. Consider a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$. Assume that there exist k hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}'}$ such that their negative half-spaces $h^-_{\boldsymbol{q},\boldsymbol{p}'}$ cover half-space $h^-_{\boldsymbol{q},\boldsymbol{p}}$ (i.e., $h^-_{\boldsymbol{q},\boldsymbol{p}} \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}'}$). Then, for any partitions covered by $h^-_{\boldsymbol{q},\boldsymbol{p}}$, they are unqualified since they are already covered by k half-spaces $h_{\boldsymbol{q},\boldsymbol{p}'}$; for any partitions covered by $h^+_{\boldsymbol{q},\boldsymbol{p}}$, the numbers of negative half-spaces covering them are not affected by $h_{\boldsymbol{q},\boldsymbol{p}}$. Therefore, the insertion of hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ does not affect the qualification of a partition. Based on this idea, we only insert the hyper-planes into the P-Tree if their negative half-spaces are covered by fewer than k negative half-spaces. Lemma 6 shows how to check if a negative half-space $h^-_{\boldsymbol{q},\boldsymbol{p}}$ is covered by another negative half-space $h^-_{\boldsymbol{q},\boldsymbol{p}'}$. Let $\boldsymbol{v}_{\boldsymbol{q},\boldsymbol{p}}$ and $\boldsymbol{v}_{\boldsymbol{q},\boldsymbol{p}'}$ be the unit norms of hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}}$ and $h_{\boldsymbol{q},\boldsymbol{p}'}$, respectively.

Lemma 6. If $\forall i \in [1, d]$, $v_{\boldsymbol{q},\boldsymbol{p}}[i] \geq v_{\boldsymbol{q},\boldsymbol{p}'}[i]$, then $h^-_{\boldsymbol{q},\boldsymbol{p}} \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}'}$.

Consider Figure 2 as an example. Vectors $\boldsymbol{v}_{\boldsymbol{q},\boldsymbol{p}_2}$ and $\boldsymbol{v}_{\boldsymbol{q},\boldsymbol{p}_3}$ represent the unit norms of hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}_2}$ and $h_{\boldsymbol{q},\boldsymbol{p}_3}$, respectively. Since $v_{\boldsymbol{q},\boldsymbol{p}_3}[1] \geq v_{\boldsymbol{q},\boldsymbol{p}_2}[1]$ and $v_{\boldsymbol{q},\boldsymbol{p}_3}[2] \geq v_{\boldsymbol{q},\boldsymbol{p}_2}[2]$, half-space $h^-_{\boldsymbol{q},\boldsymbol{p}_3}$ is covered by half-space $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$.

Insertion Order. During the P-Tree construction, a node N will be marked as invalid and will not be processed by future hyper-plane insertions if $Q(N) \geq k$. We develop an insertion order for hyper-planes, so that if a node is indeed invalid, it can be marked as early as possible, reducing the number of nodes to be processed for hyper-plane insertions.

We utilize the half-spaces covering relationships to determine the insertion order. Consider two hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}}$ and $h_{\boldsymbol{q},\boldsymbol{p}'}$, where $h^-_{\boldsymbol{q},\boldsymbol{p}} \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}'}$. It is easy to see that $h^-_{\boldsymbol{q},\boldsymbol{p}'}$ covers more partitions than $h^-_{\boldsymbol{q},\boldsymbol{p}}$. Suppose that we insert hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}'}$ before $h_{\boldsymbol{q},\boldsymbol{p}}$. More nodes in the P-Tree will have their counters $Q(N)$ incremented by 1, and thus, the possibility of nodes in the P-Tree being marked as invalid increases. Following this intuition, for each hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$, we compute the number of negative half-spaces that are covered by $h^-_{\boldsymbol{q},\boldsymbol{p}}$, denote by $W(h_{\boldsymbol{q},\boldsymbol{p}})$. Then, we insert hyper-planes into the P-Tree based on their $W(h_{\boldsymbol{q},\boldsymbol{p}})$ in the descending order.

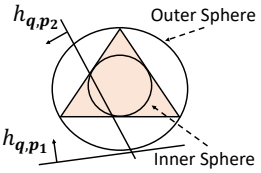


Figure 5: Relationship

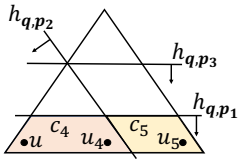


Figure 6: Sampled $\boldsymbol{u}$

Relationship Checking. We present strategies to speed up the relationship checking between a partition and a hyper-plane. Our first strategy is to utilize the hierarchy of the P-Tree. Assume that node N' is a child of node N, where N (resp. N') contains partition c (resp. partition c').

Lemma 7. If partition c is covered by a half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$ (resp. $h^-_{\boldsymbol{q},\boldsymbol{p}}$), partition c' must be covered by $h^+_{\boldsymbol{q},\boldsymbol{p}}$ (resp. $h^-_{\boldsymbol{q},\boldsymbol{p}}$).

Consider Figure 4. Nodes N_3 and N_4 are two children of N_2. If there is a half-space covering the partition in node N_2, it must cover the partitions in N_3 and N_4.

Our second strategy is to approximate partitions as spheres, since it only takes $O(1)$ time to check the relationship between a sphere and a hyper-plane. Let $dist(h, \mathbb{B}_c)$ denote the smallest Euclidean distance from a sphere's center $\mathbb{B}_c$ to any point in a hyper-plane h. If $dist(h, \mathbb{B}_c)$ is smaller than the sphere's radius, the sphere must intersect the hyper-plane. Otherwise, we can easily determine which half-space the sphere is in, by checking the location of the sphere's center. For example, in Figure 5, the larger sphere is in half-space $h^+_{\boldsymbol{q},\boldsymbol{p}_1}$, since (1) $dist(h, \mathbb{B}_c)$ is larger than the sphere's radius and (2) the sphere's center is in half-space $h^+_{\boldsymbol{q},\boldsymbol{p}_1}$.

Consider a partition c that has x extreme points $\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_x$. We define the *outer sphere* and *inner sphere* of partition c as follows. For the outer sphere, we define its center $\mathbb{O}_c$ to be the average of all extreme points (i.e., $\mathbb{O}_c = \sum_{i=1}^{x} \boldsymbol{e}_i/x$), and its radius $\mathbb{O}_r$ to be the largest Euclidean distance from the center to any extreme point (i.e., $\mathbb{O}_r = \max\{\mathsf{dist}(\boldsymbol{e}_1, \mathbb{O}_c), \mathsf{dist}(\boldsymbol{e}_2, \mathbb{O}_c), ..., \mathsf{dist}(\boldsymbol{e}_x, \mathbb{O}_c)\}$, where $\mathsf{dist}(\boldsymbol{e}_i, \mathbb{O}_c)$ denotes the Euclidean distance between $\boldsymbol{e}_i$ and $\mathbb{O}_c$). Intuitively, the outer sphere of partition c is a sphere that covers c.

Lemma 8. If a half-space (i.e., $h^+_{\boldsymbol{q},\boldsymbol{p}}$ or $h^-_{\boldsymbol{q},\boldsymbol{p}}$) covers the outer sphere of a partition, the half-space covers the partition.

Consider the inner sphere of partition c. Assume that there are y *boundary hyper-planes* $h_1, h_2, ..., h_y$ of partition c (i.e., the hyper-planes that bound partition c). We define the inner sphere's center $\mathbb{I}_c$ to be the average of all extreme points (i.e., $\mathbb{I}_c = \sum_{i=1}^{x} \boldsymbol{e}_i/x$), and its radius $\mathbb{I}_r$ to be the smallest Euclidean distance from the center to any boundary hyper-plane (i.e.,

$\mathbb{I}_r = \min\{\mathsf{dist}(h_1, \mathbb{I}_c), \mathsf{dist}(h_2, \mathbb{I}_c), ..., \mathsf{dist}(h_y, \mathbb{I}_c)\}$, where $\mathsf{dist}(h_i, \mathbb{I}_c)$ denotes the smallest Euclidean distance from $\mathbb{I}_c$ to any point in h_i). Intuitively, the inner sphere of partition c is a sphere that is covered by c.

Lemma 9. If a hyper-plane intersects the inner sphere of a partition, the hyper-plane intersects the partition.

Figure 5 shows a partition (represented as a triangle), along with its outer and inner spheres. Since the outer sphere is in half-space $h^+_{\boldsymbol{q},\boldsymbol{p}_1}$, the partition is also in $h^+_{\boldsymbol{q},\boldsymbol{p}_1}$. Since hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_2}$ intersects the inner sphere, hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_2}$ also intersects the partition. Note that if the relationship cannot be determined using the above two strategies, we can still rely on the extreme points method as discussed in Section V-A1

Lazy Split. For the P-Tree construction, one time-consuming step is to create children for the leaves. To accelerate the construction, our idea is to reduce the number of children creation. Consider a leaf node N. Suppose the partition c in N intersects a set $\mathcal{H}(N)$ of hyper-planes. Since $|\mathcal{H}(N)|$ hyper-planes correspond to $|\mathcal{H}(N)|$ negative half-spaces, for any utility vector in partition c, it must be covered by at most $Q(N) + |\mathcal{H}(N)|$ negative half-spaces. If $Q(N)+|\mathcal{H}(N)| < k$, any utility vector in partition c is covered by fewer than k negative half-spaces. Thus, partition c could potentially be returned in its entirety, making the split and children creation for N unnecessary.

Following this idea, we maintain a hyper-plane set $\mathcal{H}(N)$ for each leaf node. When the insertion of a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ comes to a leaf node N and the partition in N intersects $h_{\boldsymbol{q},\boldsymbol{p}}$, we simply store $h_{\boldsymbol{q},\boldsymbol{p}}$ in set $\mathcal{H}(N)$ rather than creating two children immediately. The children creation is only triggered when a leaf N has $Q(N) + |\mathcal{H}(N)| \geq k$. This condition can be met in two cases. (1) The hyper-plane inserted intersects the partition c in N (which makes $|\mathcal{H}(N)|$ increase by 1). (2) The negative half-space of the hyper-plane inserted covers partition c (which makes $Q(N)$ increase by 1).

When $Q(N) + |\mathcal{H}(N)| \geq k$, we attempt to reduce the hyper-planes in $\mathcal{H}(N)$ by splitting node N. Firstly, we pop out the oldest hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ in $\mathcal{H}(N)$ and create two children for node N. One child N' contains partition $c_1 = c \cap h^-_{\boldsymbol{q},\boldsymbol{p}}$ and $Q(N') = Q(N) + 1$; the other child N'' contains partition $c_2 = c \cap h^+_{\boldsymbol{q},\boldsymbol{p}}$ and $Q(N'') = Q(N)$. Both children inherit $\mathcal{H}(N)$ from node N. However, since partitions c_1 and c_2 are sub-partitions of c, the hyper-planes in $\mathcal{H}(N)$ may not intersect partitions c_1 or c_2. Thus secondly, we refine $\mathcal{H}(N')$ and $\mathcal{H}(N'')$ in nodes N' and N'', respectively. Consider node N' as an example. For any hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}} \in \mathcal{H}(N')$, if its half-space (either $h^-_{\boldsymbol{q},\boldsymbol{p}}$ or $h^+_{\boldsymbol{q},\boldsymbol{p}}$) covers partition c_1, we remove $h_{\boldsymbol{q},\boldsymbol{p}}$ from $\mathcal{H}(N')$. Besides, $Q(N')$ is updated to be $Q(N') = Q(N') + 1$ if $c_1 \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}}$. After the refinement, if $Q(N') + |\mathcal{H}(N')| < k$, we stop. Otherwise, N' is recursively split. Similarly for node N''.

3) Summary and Analysis: The pseudocode of algorithm *E-PT* is shown in Algorithm 2. Initially, we reduce the number of hyper-planes inserted, by filtering out the hyper-planes whose negative half-spaces are covered by at least k negative half-spaces (line 3). Then, the remaining hyper-planes are

Algorithm 2: Algorithm *E-PT*

1 **Input:** point set $\mathcal{D}$, query point $\boldsymbol{q}$, parameters k and ϵ.
2 **Output:** the set $\mathcal{C}$ of qualified partitions.
3 Filter and rank the hyper-planes based on $W(h_{\boldsymbol{q},\boldsymbol{p}})$;
4 **for** *each hyper-plane* $h_{\boldsymbol{q},\boldsymbol{p}}$ **do**
5 **Insert**(Root, $h_{\boldsymbol{q},\boldsymbol{p}}$);
6 **return** All leaves N with $Q(N) + |\mathcal{H}(N)| < k$.

Insert(node N, hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$)
7 **if** $c \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}}$ **then**
8 $Q(N) = Q(N) + 1$;
9 **if** $Q(N) \geq k$ **then**
10 Mark node N invalid;
11 **else if** *node* N *is an internal node* **then**
12 **for** *each child* N' *of* N **do**
13 **Insert**(N', $h_{\boldsymbol{q},\boldsymbol{p}}$);
14 **else if** $Q(N) + |\mathcal{H}(N)| \geq k$ **then**
15 **Lazy_Split**(N);
16 **else if** $c \cap h^-_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$ *and* $c \cap h^+_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$ **then**
17 **if** *node* N *is an internal node* **then**
18 **for** *each child* N' *of* N **do**
19 **Insert**(N', $h_{\boldsymbol{q},\boldsymbol{p}}$);
20 **else**
21 Add $h_{\boldsymbol{q},\boldsymbol{p}}$ into $\mathcal{H}(N)$;
22 **if** $Q(N) + |\mathcal{H}(N)| \geq k$ **then**
23 **Lazy_Split**(N);

Lazy_Split(node N)
24 $h_{\boldsymbol{q},\boldsymbol{p}} \leftarrow$ the oldest hyper-plane in $\mathcal{H}(N)$;
25 $\mathcal{H}(N) = \mathcal{H}(N) \setminus \{h_{\boldsymbol{q},\boldsymbol{p}}\}$;
26 Create two children N' and N'' for N;
27 $c_1 = c \cap h^-_{\boldsymbol{q},\boldsymbol{p}}$; $Q(N') = Q(N) + 1$; $\mathcal{H}(N') = \mathcal{H}(N)$;
28 $c_2 = c \cap h^+_{\boldsymbol{q},\boldsymbol{p}}$; $Q(N'') = Q(N)$; $\mathcal{H}(N'') = \mathcal{H}(N)$;
29 **Refine**(N'); **Refine**(N'');

Refine(node N)
30 **for** *each hyper-plane* $h_{\boldsymbol{q},\boldsymbol{p}}$ *in* $\mathcal{H}(N)$ **do**
31 **if** $c \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}}$ *or* $c \subseteq h^+_{\boldsymbol{q},\boldsymbol{p}}$ **then**
32 Remove hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ from $\mathcal{H}(N)$;
33 **if** $c \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}}$ **then**
34 $Q(N) = Q(N) + 1$;
35 **if** $Q(N) \geq k$ **then**
36 Mark node N invalid;
37 **return**;
38 **if** $Q(N) + |\mathcal{H}(N)| \geq k$ **then**
39 **Lazy_Split**(N);

inserted into the P-Tree in the descending order of their $W(h_{\boldsymbol{q},\boldsymbol{p}})$ (lines 3-5). When all hyper-planes are inserted, the partitions that (1) are stored in the leaves and (2) are covered by fewer than k negative half-spaces are returned (line 6).

The insertion of each hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ is conducted in a top-down manner. Suppose that the insertion comes to a node N that stores a partition c. (1) If $c \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}}$, $Q(N)$ is updated to be $Q(N) = Q(N) + 1$ (lines 7-8). If $Q(N) \geq k$, node N is marked as invalid (lines 9-10). If node N is a valid internal node, the insertion proceeds to its children recursively (lines 11-13). If node N is a valid leaf, we check whether $Q(N) + |\mathcal{H}(N)| \geq k$ and split N if such condition holds (lines 14-15). (2) Suppose $c \cap h^-_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$ and $c \cap h^+_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$ (line 16). If node N is an internal node, the insertion proceeds to its children recursively (lines 17-19). If node N is a valid leaf, we add the hyper-plane to set $\mathcal{H}(N)$ (lines 20-21). The update of $\mathcal{H}(N)$ may lead to $Q(N) + |\mathcal{H}(N)| \geq k$. If this is the case, we split node N (lines 22-23). To split node N, we pop out the oldest hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ in $\mathcal{H}(N)$ and create two children N' and N'' for N based on $h_{\boldsymbol{q},\boldsymbol{p}}$ (lines 24-26). Each child stores its own partition and counter, and inherits set $\mathcal{H}(N)$ (lines 27-28). Note that $\mathcal{H}(N')$ and $\mathcal{H}(N'')$ will be refined, by only maintaining the hyper-planes that intersect the new partition in the child (lines 29-37). After the refinement, if $Q(N') + |\mathcal{H}(N')| \geq k$ (resp. $Q(N'') + |\mathcal{H}(N'')| \geq k$), we recursively split node N' (node N'') (lines 38-39).

Denote by $O(\alpha)$ the creation cost of a single node. The following theorem shows the time complexity of our algorithm.

Theorem 2. The time complexity of our algorithm *E-PT* is $O(\alpha \cdot (k\frac{log^{d-1}n}{d!})^{d-1})$.

Note that $O(\alpha)$ mainly depends on the cost of partition construction. Our algorithm *E-PT* constructs partitions incrementally. When building a new partition (either $c \cap h^-_{\boldsymbol{q},\boldsymbol{p}}$ or $c \cap h^+_{\boldsymbol{q},\boldsymbol{p}}$), there is only one hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ that intersects the existing partition c. Thus, the time cost for each partition construction is low [10], [38], resulting in a small $O(\alpha)$.

B. Approximate Algorithm

Algorithm *E-PT* constructs partitions for internal nodes, which are important for hierarchical indexing. However, none of them will be included in the output. This motivates us to design a more efficient algorithm *A-PC*, to ensure that each partition constructed is indeed a qualified partition.

1) Progressive Construction: We randomly sample a set of utility vectors in the utility space. For each sampled utility vector $\boldsymbol{u}$, we compute the k-regret ratio of the query point $\boldsymbol{q}$, i.e., $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u})$. If $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) < \epsilon$, we construct a partition $c_{\boldsymbol{u}}$ based on $\boldsymbol{u}$ and add $c_{\boldsymbol{u}}$ to the final output. Specifically, partition $c_{\boldsymbol{u}}$ is the intersection of n half-spaces. Denote by $D^+_{\boldsymbol{u}}$ (resp. $D^-_{\boldsymbol{u}}$) the set of points $\boldsymbol{p}$ such that $(1-\epsilon)f_{\boldsymbol{u}}(\boldsymbol{p}) < f_{\boldsymbol{u}}(\boldsymbol{q})$ (resp. $(1-\epsilon)f_{\boldsymbol{u}}(\boldsymbol{p}) > f_{\boldsymbol{u}}(\boldsymbol{q})$). These two sets can be easily computed based on the utilities. We have

$$c_{\boldsymbol{u}} = \left(\bigcap_{\boldsymbol{p}\in\mathcal{D}^+_{\boldsymbol{u}}} h^+_{\boldsymbol{q},\boldsymbol{p}}\right) \cap \left(\bigcap_{\boldsymbol{p'}\in\mathcal{D}^-_{\boldsymbol{u}}} h^-_{\boldsymbol{q},\boldsymbol{p'}}\right).$$

Lemma 10. Given partition $c_{\boldsymbol{u}}$ that is constructed based on a sampled utility vector $\boldsymbol{u}$, we have $\boldsymbol{u} \in c_{\boldsymbol{u}}$ and for any utility vector $\boldsymbol{u}' \in c_{\boldsymbol{u}}$, query point $\boldsymbol{q}$ is a (k,ϵ)-regret point w.r.t. $\boldsymbol{u}'$.

To illustrate, consider the points in Table II. Suppose that query point $\boldsymbol{q} = (0.4, 0.7)$ and threshold $\epsilon = 0.1$. Consider a sampled utility vector $\boldsymbol{u} = (0.5, 0.5)$. If $k = 2$, then $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) = 0.018 < \epsilon$, and thus, query point $\boldsymbol{q}$ is a $(2, 0.1)$-regret point w.r.t. $\boldsymbol{u}$. We build three hyper-planes and construct a partition $c_{\boldsymbol{u}} = h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^-_{\boldsymbol{q},\boldsymbol{p}_2} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_3}$.

To avoid constructing the same partition based on different sampled utility vectors, we use the following lemma to check if a partition $c_{\boldsymbol{u}}$ based on a sampled $\boldsymbol{u}$ is already constructed.

Lemma 11. If the sampled utility vector $\boldsymbol{u}$ is in a partition c (i.e., $\boldsymbol{u} \in c$), then partition $c_{\boldsymbol{u}}$ is the same as partition c.

Consider Figure 6 as an example. Suppose that partition c_4 shown in pink is already constructed based on a sampled utility vector $\boldsymbol{u}_4$. For another sampled utility vector $\boldsymbol{u}$, we will not construct the partition repeatedly since $\boldsymbol{u}$ is in partition c_4.

2) Acceleration: We develop an effective strategy to reduce the number of partitions to be constructed. Consider two partitions $c_4 = h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_2} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_3}$ and $c_5 = h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^-_{\boldsymbol{q},\boldsymbol{p}_2} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_3}$ as shown in Figure 6. Since they are the intersections of different half-spaces, they are constructed independently based on different sampled utility vectors if no optimization strategies are used. However, if we can directly construct the union of partition c_4 and c_5 (i.e., partition $c_4 \cup c_5$) based on a sampled utility vector $\boldsymbol{u}$, we only need to call the partition construction once instead of twice. Following this idea, we improve our method of constructing partitions as follows. Consider any two sampled utility vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, where $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}_1) < \epsilon$ and $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}_2) < \epsilon$. If $\mathcal{D}^+_{\boldsymbol{u}_1} \subseteq \mathcal{D}^+_{\boldsymbol{u}_2}$, we create a partition $c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$ based on the points in $\mathcal{D}^+_{\boldsymbol{u}_1}$ and $\mathcal{D}^-_{\boldsymbol{u}_2}$, i.e.,

$$c_{\boldsymbol{u}_1,\boldsymbol{u}_2} = \left(\bigcap_{p\in\mathcal{D}^+_{\boldsymbol{u}_1}} h^+_{\boldsymbol{q},\boldsymbol{p}}\right) \cap \left(\bigcap_{p'\in\mathcal{D}^-_{\boldsymbol{u}_2}} h^-_{\boldsymbol{q},\boldsymbol{p'}}\right)$$

Similarly for the case where $\mathcal{D}^+_{\boldsymbol{u}_2} \subseteq \mathcal{D}^+_{\boldsymbol{u}_1}$.

Lemma 12. Given partition $c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$ that is constructed based on sampled utility vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, we have $\boldsymbol{u}_1, \boldsymbol{u}_2 \in c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$ and for any utility vector $\boldsymbol{u}' \in c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$, query point $\boldsymbol{q}$ is a (k,ϵ)-regret point w.r.t. $\boldsymbol{u}'$.

Back to the example discussed in Figure 6. For the sampled utility vector $\boldsymbol{u}_4$, $\mathcal{D}^+_{\boldsymbol{u}_4} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3\}$ and $\mathcal{D}^-_{\boldsymbol{u}_4} = \emptyset$. For the sampled utility vector $\boldsymbol{u}_5$, $\mathcal{D}^+_{\boldsymbol{u}_5} = \{\boldsymbol{p}_1, \boldsymbol{p}_3\}$ and $\mathcal{D}^-_{\boldsymbol{u}_5} = \{\boldsymbol{p}_2\}$. Since $\mathcal{D}^+_{\boldsymbol{u}_5} \subseteq \mathcal{D}^+_{\boldsymbol{u}_4}$, we construct a partition based on the points in $\mathcal{D}^+_{\boldsymbol{u}_5}$ and $\mathcal{D}^-_{\boldsymbol{u}_4}$. The partition is $c = h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_3}$, which is an union of partitions c_4 and c_5.

3) Summary and Analysis: The pseudocode of algorithm *A-PC* is shown in Algorithm 3. Initially, we randomly sample a set U of utility vectors (line 3). For each vector $\boldsymbol{u}$, we compute $\mathcal{D}^+_{\boldsymbol{u}}$ and $\mathcal{D}^-_{\boldsymbol{u}}$ and only keep $\boldsymbol{u}$ in U if $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) < \epsilon$ (lines 4-7). Then, we refine the utility vectors in U as follows. For each pair $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$, if $\mathcal{D}^+_{\boldsymbol{u}_1} \subseteq \mathcal{D}^+_{\boldsymbol{u}_2}$, we only keep one utility vector, say $\boldsymbol{u}_1$, in U and set $\mathcal{D}^-_{\boldsymbol{u}_1} = \mathcal{D}^-_{\boldsymbol{u}_2}$ (lines 8-11).

Algorithm 3: Algorithm *A-PC*

1 **Input:** point set $\mathcal{D}$, query point $\boldsymbol{q}$, parameters k and ϵ.
2 **Output:** the set $\mathcal{C}$ of qualified partitions.
3 Randomly sample a set U of utility vectors;
4 **for** *each sampled* $\boldsymbol{u} \in U$ **do**
5 Obtain set $\mathcal{D}^+_{\boldsymbol{u}}$ and $\mathcal{D}^-_{\boldsymbol{u}}$;
6 **if** $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) > \epsilon$ **then**
7 Delete $\boldsymbol{u}$ from U;
8 **for** *each any pair* $\boldsymbol{u}_1, \boldsymbol{u}_2 \in U$ **do**
9 **if** $\mathcal{D}^+_{\boldsymbol{u}_1} \subseteq \mathcal{D}^+_{\boldsymbol{u}_2}$ **then**
10 Update $\mathcal{D}^-_{\boldsymbol{u}_1} = \mathcal{D}^-_{\boldsymbol{u}_2}$;
11 Delete $\boldsymbol{u}_2$ from U;
12 **else if** $\mathcal{D}^+_{\boldsymbol{u}_2} \subseteq \mathcal{D}^+_{\boldsymbol{u}_1}$ **then**
13 Update $\mathcal{D}^-_{\boldsymbol{u}_2} = \mathcal{D}^-_{\boldsymbol{u}_1}$;
14 Delete $\boldsymbol{u}_1$ from U;
15 **for** *each* $\boldsymbol{u} \in U$ **do**
16 Build a partition based on $\boldsymbol{u}$ and insert it into set $\mathcal{C}$;
17 **return** The set $\mathcal{C}$ of qualified partitions.

Similarly for $\mathcal{D}^+_{\boldsymbol{u}_2} \subseteq \mathcal{D}^+_{\boldsymbol{u}_1}$ (lines 12-14). After the refinement, we create a partition for each remaining utility vector $\boldsymbol{u}$ in U based on its $\mathcal{D}^+_{\boldsymbol{u}}$ and $\mathcal{D}^-_{\boldsymbol{u}}$ (lines 15-16).

Denote by $\mathcal{N}$ the total number of sampled utility vectors. The only remaining issue is to set a proper value for $\mathcal{N}$. If $\mathcal{N}$ is small, it may limit the number of qualified partitions found. If $\mathcal{N}$ is large, excessive sampled vectors may lead to a high computational cost. To strike a balance, we focus our sampling strategy on finding "influential" partitions, which are quantified by their volumes. Intuitively, if a qualified partition has a large volume, it means that query point $\boldsymbol{q}$ is a (k, ϵ)-regret point w.r.t. many utility vectors. We regard such a partition as an influential partition since if it is not included in the output, we miss a large number of qualified utility vectors. Thus, we primarily target at finding qualified partitions whose volumes are larger than a predefined threshold. Specifically, let us use V_c (resp. $\mathbb{V}$) to denote the volume of a qualified partition c (resp. the utility space $\mathcal{U}$). Given a real number ρ, our goal is to find the qualified partitions such that $V_c/\mathbb{V} > \rho$.

Lemma 13. Given a confidence parameter δ and a sampling size $\mathcal{N} = O((1/\rho^2)(d + \ln(1/\delta))$, for each qualified partition c with $V_c/\mathbb{V} > \rho$, we can find it with confidence $1 - \delta$.

VI. Experiment

In this section, we present our experimental evaluation. We begin by describing the experimental setting in Section VI-A. Then, in Section VI-B, we showcase the benefit of the reverse regret query, compared to traditional reverse queries that focus on rankings instead of utilities. Next, we compare the performance of our algorithms against existing ones on synthetic and real datasets in Sections VI-C and VI-D, respectively. Finally, our findings are summarized in Section VI-E.

A. Experimental Setting

The experiments were run on a machine with 3.10GHz CPU and 16GB RAM. All programs were implemented in C/C++.

Datasets. We conducted experiments on synthetic and real datasets that were commonly used in existing studies [17], [33], [39]. The synthetic datasets are *anti-correlated* (Anti), *correlated* (Cor), and *independent* (Indep) [17], [39]. They represent typical data distributions in multi-criteria decision-making. The real datasets are *Island*, *Weather*, *Car*, and *NBA* [10], [33]. Dataset *Island* contains 63,383 2-dimensional geographic locations. Dataset *Weather* includes 178,080 records described by four attributes. Dataset *Car* comprises 69,052 used cars described by four attributes. Dataset *NBA* has 16,916 players and five attributes are used to describe the performance of each player. For all datasets, each dimension was normalized to (0, 1]. Note that existing studies [1], [40] preprocessed datasets to include k-*skyband* points only. To maintain consistency and enable a fair comparison of our algorithms with existing ones, we also preprocessed the datasets in the same manner by retaining only k-skyband points.

Parameter Setting. We evaluated algorithms by varying the following parameters: (1) parameter k; (2) threshold ϵ; (3) the number of dimensions d; (4) the dataset size n; and (5) the dataset type (e.g., Anti, Cor, and Indep). Unless stated explicitly, following the default setting of [1], [40], we set $k = 10$ and $\epsilon = 0.1$ by default, and the synthetic datasets were set by default as follows: $d = 4$, $n = 400,000$, and type: Indep.

Algorithms & Measurement. We evaluated our algorithms *Sweeping*, *E-PT*, and *A-PC* against existing methods *LP-CTA* [1] and *PBA+* [40] by their execution times. We generated 30 query points by assigning random values within the range $(0, 1]$ to each dimension. Each algorithm was then tested with these query points, running 30 times, and the average result was reported. Since existing algorithms were not designed to solve our problem originally, we adapted them as follows.

- Algorithm *LP-CTA* is designed to find customers who are interested in a given product merely based on the product rankings. It divides the utility space into partitions using its designed hyper-planes. We replaced its designed hyper-planes with ours, and followed its strategy to construct and return qualified partitions.
- Algorithm *PBA+* builds a hierarchical tree-based structure to store partitions in the utility space. Each partition in the i-th level corresponds to a point that has the i-th highest utility w.r.t. any utility vector in the partition. We performed a top-down search to check the partitions in the tree. For each partition, we compared its corresponding point with the query point, and determined whether the partition (or part of the partition) was qualified to be returned. Note that algorithm *PBA+* builds the hierarchical tree-based structure in a preprocessing step and uses it in later queries. We reported its querying time as the execution time, excluding its preprocessing time (which can be more than 10^4 seconds).

B. User Study

To motivate our problem, we explored the difference between the reverse top-k query and the reverse regret query.

Firstly, following [8], [10], [33], we conducted a user study on dataset *Car* to compare (1) the product rankings (used by the reverse top-k query) and (2) the product utilities (used by the reverse regret query) for evaluating products. We recruited thirty participants and used an interactive algorithm *Adaptive* [7] to learn their exact utility functions. Based on the learned utility function, the cars' utilities were known. We provided each participant with their top-k cars (i.e., k cars with the highest utilities), claiming that these results would be of interest to them, where k had three settings: $k = 1$, $k = 5$, and $k = 10$. Then, we found the cars whose k-$regratio$ were smaller than 0.1 and uniformly selected five of them. The participants were asked to indicate if they were interested in these five selected cars. We collected two types of results: (1) the percentage of interest, i.e., the percentages of cars that are of interest to the participants among the five selected cars; and (2) the average rank, i.e., the average ranks of the cars that are of interest to the participants among the five cars. Figure 7 shows the results. For different k, the percentages of interest are at least 50% and the average ranks are up to 75.1. These findings suggest that as secondary sources of information derived from product utilities, rankings are insufficient for determining customers' interests. The regret ratio, which focuses on utilities directly, works better in modeling customer preferences for products.

Secondly, we compared the results of the reverse regret query with those of the reverse top-k query. Let S denote the set of utility functions learned in the user study. With different k and ϵ, we identified subsets $S_1 \subseteq S$ and $S_2 \subseteq S$, which contained the utility vectors in the result of the reverse top-k query and the reverse regret query, respectively. Considering S_2 as a hypothetical "ground truth" for comparison purposes (not the real ground truth), we obtained the precision, recall, and F1 of S_1. Figure 8 shows the results. In many cases, results S_1 are different from S_2. These findings, in conjunction with our earlier observations (the product utilities evaluate products better than product rankings), suggest that the reverse regret query is more effective in finding prospective customers.

C. Results on Synthetic Datasets

Accuracy. We conducted a study to explore the effect of the number $\mathcal{N}$ of sampling utility vectors on the output quality and the execution time in Figure 9(a) and 9(b), respectively. For the output quality, we quantified it following the *accuracy* measurement in [41]. Specifically, we (1) randomly selected 10000 utility vectors in $\mathcal{U}$, (2) for each utility vector selected, we checked if it was a qualified utility vector by verifying if it was in the partitions returned by the exact algorithm *E-PT*, and (3) we reported the *accuracy* of algorithm *A-PC* to be the percentage of qualified utility vectors that were also in the partitions returned by *A-PC*. Figure 9(a) shows the accuracy of *A-PC* is high with different sampling sizes. In particular, when more utility vectors are sampled (i.e., $\mathcal{N}$ is larger), the accuracy increases, as expected. Besides, on the four-dimensional dataset, to achieve the same accuracy, we need to sample more utility vectors than on the two-dimensional dataset. This is because the utility space in the four-dimensional dataset is larger, and thus, there will be more qualified partitions. Moreover, when the sampling size is larger, it takes more time to process the samples, leading to a longer execution time (Figure 9(b)). To strike a balance, we set the sampling size $\mathcal{N}$ for algorithm *A-PC* to be $10 \times (d-1)$ by default in the rest of the experiments.

Two-dimensional Dataset. We compared our algorithms *Sweeping*, *E-PT*, and *A-PC* against existing ones on a two-dimensional dataset (i.e., d = 2) by varying parameters k and ϵ, where other parameters were set by default. In Figure 10(a), we varied parameter k from 1 to 40. Our algorithms achieve significant improvements. They reduce the execution time by up to 1-2 orders of magnitude compared to the existing ones. When k increases, the execution times of all algorithms become longer. This is because the increasing k relaxes the returned condition, leading to more qualified partitions to be processed. Nevertheless, our algorithm *Sweeping* only increases slightly in execution time since it processes partitions in linear time. In contrast, the existing algorithm *PBA+* is heavily affected by the increasing k. For example, it runs 2 orders of magnitude slower than the others when $k = 40$. This indicates that its hierarchical tree-based index is not efficient in handling a large number of partitions. In Figure 10(b), we varied parameter ϵ from 0 to 0.2. Our algorithms consistently take the shortest time in all cases. For instance, when $\epsilon = 0.2$, algorithm *Sweeping* is 20 times and 60 times faster than existing algorithms *LP-CTA* and *PBA+*, respectively. Moreover, when ϵ increases, the execution time of our algorithm *Sweeping* is almost indifferent. This again verifies the stability of algorithm *Sweeping* under different parameter settings.

Four-dimensional Dataset. We also evaluated our algorithms on a four-dimensional dataset (i.e., d = 4), where the other parameters were set by default. Since algorithm *Sweeping* is only designed for the two-dimensional special case, we excluded it from the experiment. Figure 11(a) shows the execution time of each algorithm when we increased k from 1 to 40. As shown there, algorithm *PBA+* performs the worst. Note that we do not show the results of algorithm *PBA+* in Figure 11(a) when $k \geq 30$ since its pre-processing step (for computing the tree) costs more than 10^4 seconds. Algorithm *LP-CTA* runs 3-5 times slower than our algorithms on average. This is because it spends much time checking the relationship between hyperplanes and partitions via solving the costly Linear Programming (LP) problems. In contrast, our algorithm *E-PT* adopts effective strategies to speed up the relationship checking (see Section V-A2), and our algorithm *A-PC* even avoids such costly relationship checking. Although all algorithms need more time to execute given a larger parameter k, as expected, our algorithms consistently run the fastest in all cases. In Figure 11(b), we varied parameter ϵ from 0 to 0.2. Our algorithms work the best. They are at least four times faster than the existing ones. Our algorithm *A-PC* is 3-10 times faster than *E-PT* since it avoids some partition constructions (e.g., the partitions

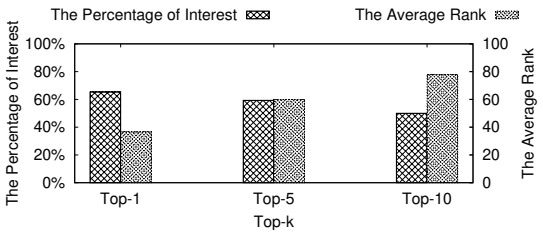
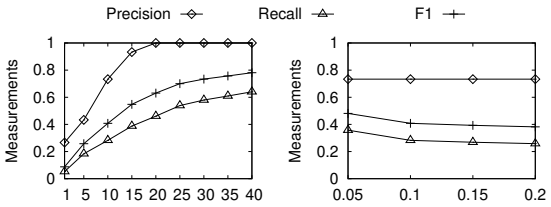
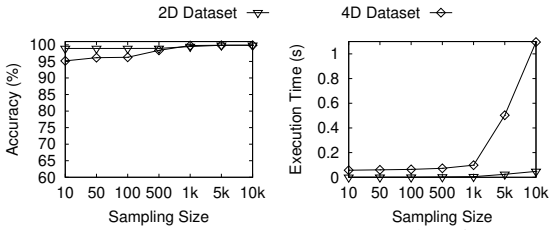


Figure 7: User Study



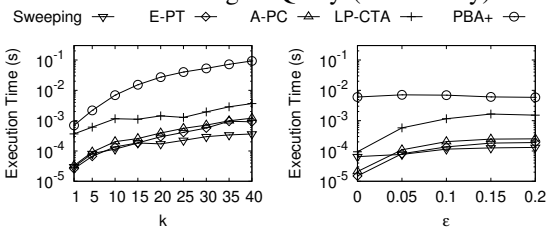
(a) Vary k (b) Vary ϵ

Figure 8: The Results of Reverse Top-k Query Compared with the Results of Reverse Regret Query (User Study)



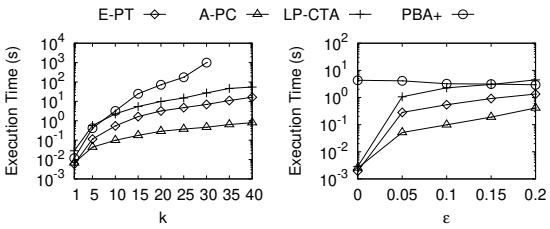
(a) Accuracy (b) Time

Figure 9: Accuracy



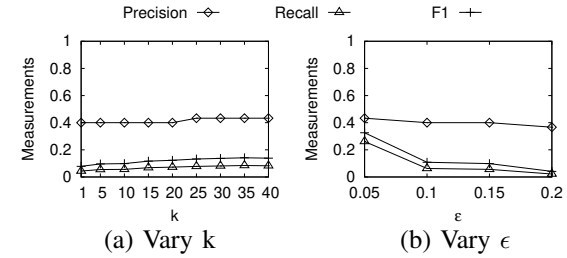
(a) Vary k (b) Vary ϵ

Figure 10: 2D



(a) Vary k (b) Vary ϵ

Figure 11: 4D



(a) Vary k (b) Vary ϵ

Figure 12: The Results of Reverse Top-k Query Compared with the Results of Reverse Regret Query (4D)

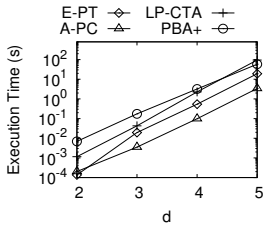


Figure 13: Vary d

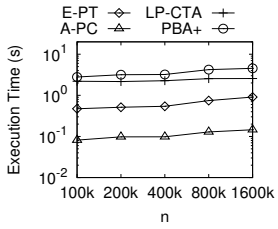


Figure 14: Vary n

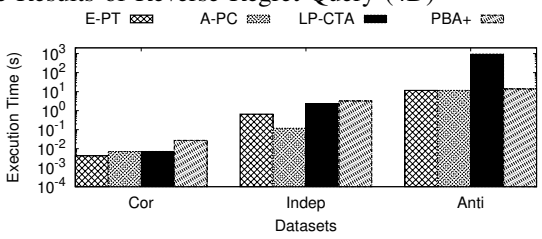
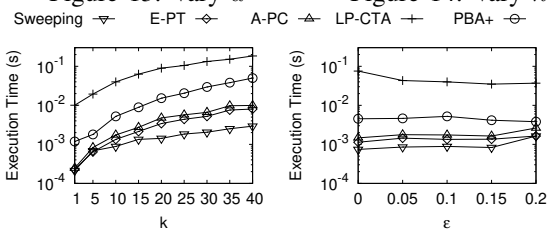
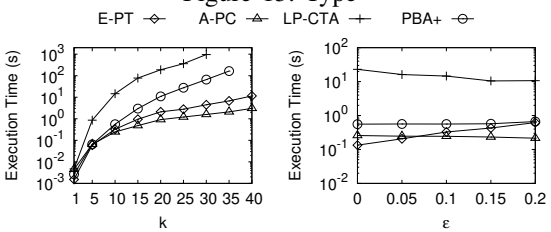


Figure 15: Type



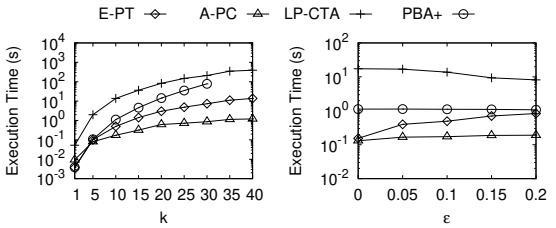
(a) Vary k (b) Vary ϵ

Figure 16: Island



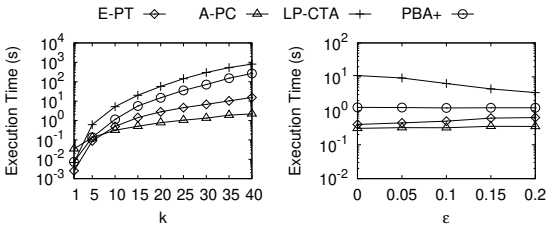
(a) Vary k (b) Vary ϵ

Figure 17: Car



(a) Vary k (b) Vary ϵ

Figure 18: Weather



(a) Vary k (b) Vary ϵ

Figure 19: NBA

in the internal nodes of *E-PT*). Furthermore, all algorithms except *PBA+* experience a slowdown when ϵ increases since more partitions are qualified to be returned, leading to longer execution times. Note that algorithm *PBA+* becomes faster with larger ϵ. It pre-constructs partitions without knowing the regret ratio criterion, resulting in some partitions that need to be shrunk or refined (e.g., filtering out unqualified utility vectors) before being returned. As ϵ increases, more utility vectors become qualified, allowing some partitions to be returned directly without refinement, which reduces the overall cost.

We also compared the results of the reverse top-k query with ours (i.e., the results of the reverse regret query) on the four-dimensional dataset. We assumed that our results were the ground truth (just for comparison, not the real ground truth), and obtained the precision, recall, and F1 of the results of the reverse top-k query. The results are shown in Figure 12. In all cases, the measurements are low, indicating that the results of the reverse top-k query differ largely from ours.

Scalability. We studied the scalability of algorithms by varying the dimension d, the dataset size n, and the type of datasets.

Varying d. In Figure 13, we evaluated the scalability of algorithms w.r.t. the dimension d. Compared with the existing algorithms, our algorithms *E-PT* and *A-PC* consistently take the shortest execution time for all values of d. For instance, when $d = 4$, algorithms *LP-CTA* and *PBA+* run about 2.3 and 3.2 seconds, respectively, while algorithms *E-PT* and *A-PC* finish in 0.5 and 0.1 seconds, respectively.

Varying n. In Figure 14, we studied the scalability of all algorithms w.r.t. the dataset size n. Our algorithms *E-PT* and *A-PC* scale well. For example, their execution times are less than 0.75 seconds even if $n = 800,000$, and the others run up to 2.55 seconds. Note that the execution times of all algorithms become larger with the increasing dataset size, since more hyper-planes have to be constructed and there are more qualified partitions to be returned.

Varying type. In Figure 15, we ran all algorithms on three types of synthetic datasets: anti-correlated (Anti), correlated (Cor), and independent (Indep). Our algorithms are the best on all datasets. On the correlated dataset, all the algorithms run within 10^{-2} seconds. This is because the attributes in the dataset are correlated. We only need to build hyper-planes based on a few points to form partitions in the utility space. In contrast, all the algorithms run slower on the anti-correlated dataset. This is because the attributes in the dataset are anti-correlated with each other, and thus, we need to consider a lot of points in the dataset in order to decide the qualified partitions in the utility space.

D. Results on Real Datasets

We studied the performance of our algorithms *Sweeping*, *E-PT* and *A-PC*, on 4 real datasets by varying parameters k and ϵ. The results on datasets *Island*, *Weather*, *Car*, and *NBA* are shown in Figures 16, 18, 17, and 19, respectively. We only present the results of *Sweeping* on the *Island* dataset, as it is only applicable to the two-dimensional special case. Our algorithms *E-PT* and *A-PC* outperform competitors substantially in execution times. For instance, when $k = 40$, both *E-PT* and *A-PC* spend at most 15.4 seconds on dataset *NBA*, while the existing algorithms *LP-CTA* and *PBA+* take 810.1 seconds and 266.2 seconds, respectively. When $k = 35$, our algorithms take within 13.8 seconds on dataset *Weather*, while the existing algorithm *LP-CTA* spends 347.7 seconds. Note that we do not show the complete results of algorithm *PBA+* on some datasets due to its costly preprocessing step (more than 10^4 seconds). Similarly, we omit the results of *LP-CTA* when its execution time exceeds 10^4 seconds.

E. Summary

The experiments demonstrate that our formulation of problem RRQ provides a better assessment of prospective customers. In our user study, the percentages of interest are at least 50% and the average ranks are up to 75.1. Furthermore, our algorithms exhibit superior performance. (1) Our algorithms are efficient. For example, our algorithm *Sweeping* runs 180 times faster than the existing algorithm *PBA+* on a two-dimensional dataset when $k = 30$; our algorithms *E-PT* and *A-PC* spend at most 6.94 seconds on a four-dimensional dataset when $k = 30$, while the existing algorithm *PBA+* takes 995.7 seconds. (2) Algorithm *A-PC* achieves a faster speed (up to 20 times) than algorithm *E-PT* by providing an approximate solution, while algorithm *E-PT* can return an exact solution. (3) Our algorithms also scale well w.r.t. the type of dataset, the number of dimensions, and the dataset size. For example, our algorithm *A-PC* spends 0.14 seconds on the dataset with the size of 1600k, while the existing algorithm *PBA+* takes 4.5 seconds. In summary, our algorithm *Sweeping* runs in the shortest time for the special case of RRQ. Our algorithms *E-PT* and *A-PC* solve the general case of RRQ the most efficiently.

VII. Conclusion

In this paper, we aim to identify the prospective customers for a given product, by finding all utility vectors such that the given product is a (k, ϵ)-regret product. Firstly, we focus on a special case where each product is described by two attributes (i.e., $d = 2$). We propose algorithm *Sweeping* that only takes linear time. Secondly, we consider the general case where each product can be described by multiple attributes (i.e., $d \geq 2$). We present an exact algorithm *E-PT* and an approximate algorithm *A-PC*, which perform well theoretically and empirically. Extensive experiments verify that our algorithms are efficient. As for future work, we want to apply the reverse regret query under the streaming and dynamic setting (e.g., considering the time of attribute) and extend the regret condition from a point to the top as the pivot.

Acknowledgment

We are grateful to the anonymous reviewers for their constructive comments. The research is supported in part by China NSFC 62202313, Guangdong Basic and Applied Basic Research Foundation 2022A1515010120, and NSF grants 2106176 and 2312931.

REFERENCES

[1] B. Tang, K. Mouratidis, and M. L. Yiu, "Determining the impact regions of competing options in preference space," in *Proceedings of the 2017 ACM International Conference on Management of Data*, 2017, pp. 805–820.

[2] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg, "Reverse top-k queries," in *2010 IEEE 26th International Conference on Data Engineering*. IEEE, 2010, pp. 365–376.

[3] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag, "Monochromatic and bichromatic reverse top-k queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 23, no. 8, pp. 1215–1229, 2011.

[4] A. Vlachou, C. Doulkeridis, K. Nørvåg, and Y. Kotidis, "Branch-and-bound algorithm for reverse top-k queries," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 481–492.

[5] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu, "Regret-minimizing representative databases," in *Proceedings of the VLDB Endowment*, vol. 3, no. 1–2. VLDB Endowment, 2010, p. 1114–1124.

[6] M. Xie, R. C.-W. Wong, J. Li, C. Long, and A. Lall, "Efficient k-regret query algorithm with restriction-free bound for any dimensionality," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2018, p. 959–974.

[7] L. Qian, J. Gao, and H. V. Jagadish, "Learning user preferences by adaptive pairwise comparison," in *Proceedings of the VLDB Endowment*, vol. 8, no. 11. VLDB Endowment, 2015, p. 1322–1333.

[8] W. Wang, R. C.-W. Wong, and M. Xie, "Interactive search with mixed attributes," in *IEEE International Conference on Data Engineering (ICDE)*, 2023, pp. 2276–2288.

[9] M. Xie, R. C.-W. Wong, P. Peng, and V. J. Tsotras, "Being happy with the least: Achieving α-happiness with minimum number of tuples," in *Proceedings of the International Conference on Data Engineering*, 2020, pp. 1009–1020.

[10] M. Xie, R. C.-W. Wong, and A. Lall, "Strongly truthful interactive regret minimization," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2019, p. 281–298.

[11] G. Zhang, N. Tatti, and A. Gionis, "Finding favourite tuples on data streams with provably few comparisons," *arXiv preprint arXiv:2307.02946*, 2023.

[12] Y. Gao, Q. Liu, B. Zheng, L. Mou, G. Chen, and Q. Li, "On processing reverse k-skyband and ranked reverse skyline queries," *Information Sciences*, vol. 293, pp. 11–34, 2015.

[13] E. Dellis and B. Seeger, "Efficient computation of reverse skyline queries," in *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 2007, p. 291–302.

[14] X. Lian and L. Chen, "Monochromatic and bichromatic reverse skyline search over uncertain databases," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008, pp. 213–226.

[15] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides, "Computing k-regret minimizing sets," in *Proceedings of the VLDB Endowment*, vol. 7, no. 5. VLDB Endowment, 2014, p. 389–400.

[16] W. Cao, J. Li, H. Wang, K. Wang, R. Wang, R. C.-W. Wong, and W. Zhan, "k-Regret Minimizing Set: Efficient Algorithms and Hardness," in *20th International Conference on Database Theory*. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2017, pp. 11:1–11:19.

[17] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *Proceedings of the International Conference on Data Engineering*, 2001, p. 421–430.

[18] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, pp. 1–58, 2008.

[19] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith, "The onion technique: Indexing for linear optimization queries," in *Proceedings of the 2000 ACM SIGMOD International Conference on Management of data*, 2000, pp. 391–402.

[20] L. Chen and X. Lian, "Efficient processing of metric skyline queries," *IEEE Transactions on Knowledge and Data Engineering*, vol. 21, no. 3, pp. 351–365, 2008.

[21] Y. Tao, X. Xiao, and J. Pei, "Subsky: Efficient computation of skylines in subspaces," in *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 2006, pp. 65–65.

[22] K.-L. Tan, P.-K. Eng, B. C. Ooi *et al.*, "Efficient progressive skyline computation," in *VLDB*, vol. 1, 2001, pp. 301–310.

[23] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, 2003, pp. 467–478.

[24] M. E. Khalefa, M. F. Mokbel, and J. J. Levandoski, "Skyline query processing for incomplete data," in *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 2008, pp. 556–565.

[25] K. C. Lee, W.-C. Lee, B. Zheng, H. Li, and Y. Tian, "Z-sky: An efficient skyline query processing framework based on z-order," *The VLDB Journal*, vol. 19, pp. 333–362, 2010.

[26] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '84. New York, NY, USA: Association for Computing Machinery, 1984, p. 47–57.

[27] T. K. Sellis, N. Roussopoulos, and C. Faloutsos, "The r+-tree: A dynamic index for multi-dimensional objects," in *Proceedings of the 13th International Conference on Very Large Data Bases*, ser. VLDB '87. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1987, p. 507–518.

[28] W. Wang and R. C.-W. Wong, "Interactive mining with ordered and unordered attributes," *Proceedings of the VLDB Endowment*, vol. 15, no. 11, pp. 2504–2516, 2022.

[29] Z. Gong, G.-Z. Sun, J. Yuan, and Y. Zhong, "Efficient top-k query algorithms using k-skyband partition," in *Scalable Information Systems: 4th International ICST Conference, INFOSCALE 2009, Hong Kong, June 10-11, 2009, Revised Selected Papers 4*. Springer, 2009, pp. 288–305.

[30] P. Peng and R. C.-W. Wong, "K-hit query: Top-k query with probabilistic utility function," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '15. New York, NY, USA: Association for Computing Machinery, 2015, p. 577–592.

[31] M. A. Soliman, I. F. Ilyas, and K. C.-C. Chang, "Top-k query processing in uncertain databases," in *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 2006, pp. 896–905.

[32] Y. Tao, X. Xiao, and J. Pei, "Efficient skyline and top-k retrieval in subspaces," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 8, pp. 1072–1088, 2007.

[33] W. Wang, R. C.-W. Wong, and M. Xie, "Interactive search for one of the top-k," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2021.

[34] P. Peng and R. C.-W. Wong, "Geometry approach for k-regret query," in *Proceedings of the International Conference on Data Engineering*, 2014, pp. 772–783.

[35] R. Keeney, H. Raiffa, and D. Rajala, "Decisions with multiple objectives: Preferences and value trade-offs," *Systems, Man and Cybernetics, IEEE Transactions on*, vol. 9, pp. 403 – 403, 08 1979.

[36] J. Dyer and R. Sarin, "Measurable multiattribute value functions," *Operations Research*, vol. 27, pp. 810–822, 08 1979.

[37] W. Wang, R. C.-W. Wong, H. Jagadish, and M. Xie, "Reverse regret query," Tech. Rep., 2023. [Online]. Available: https://www.cse.ust.hk/~raywong/paper/ReverseRegretQuery-TechnicalReport.pdf

[38] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars, *Computational geometry: Algorithms and applications*. Springer Berlin Heidelberg, 2008.

[39] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *ACM Transactions on Database Systems*, vol. 30, no. 1, p. 41–82, 2005.

[40] J. Zhang, B. Tang, M. L. Yiu, X. Yan, and K. Li, "T-levelindex: Towards efficient query processing in continuous preference space," in *Proceedings of the 2022 International Conference on Management of Data*. New York, NY, USA: Association for Computing Machinery, 2022, p. 2149–2162.

[41] D. Nanongkai, A. Lall, A. Das Sarma, and K. Makino, "Interactive regret minimization," in *Proceedings of the ACM SIGMOD International Conference on Management of Data*. New York, NY, USA: ACM, 2012, p. 109–120.