# Reverse Regret Query

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

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

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

## ABSTRACT

Reverse operators have lately gained much attention in the realm of multi-criteria decision-making. While traditional (forward) operators, such as skyline, seek to identify products that may interest a customer, reverse (backward) 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 it. However, paying attention purely to rankings might cause misleading results, as rankings emphasize the relative positions of products without accounting for the score differences between them. 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 employ product scores to evaluate products, enabling a more accurate identification of prospective customers for a product.

We refer to our problem as the reverse regret query and make several contributions. Firstly, for the special case in which each product is described by two attributes, we propose an algorithm *Sweeping* that only takes linear time. Secondly, for the general case in which each product can be described by an arbitrary number of 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.

**PVLDB Reference Format:**
Weicheng Wang, Raymond Chi-Wing Wong, H. V. Jagadish, and Min Xie. Reverse Regret Query. PVLDB, 14(1): XXX-XXX, 2020.
doi:XX.XX/XXX.XX

## 1 INTRODUCTION

The *multi-criteria decision-making* problem [1, 20, 26, 41] has recently became increasingly significant, given the growing accessibility of extensive collections of products with multiple attributes. The goal is to pinpoint products that are align with customers' preferences. The *reverse* version of this problem [33, 34] has also garnered attention. Instead of exploring the products from the customer's standpoint, it identifies prospective customers from the product manufacturer's perspective. By mining the characteristics of products, it reveals insights into what types of customers are likely to be interested in a particular product. Many *reverse* operators [30, 33–35] have been proposed for this topic, which is applicable in multiple scenarios, particularly for product design, e.g., evaluating a new car model for a market, improving the design of a cell phone to broaden appeal, and identifying new software opportunities.

Take the scenario of a car market to motivate the problem. Suppose that there is a set $\mathcal{D}$ of cars in the market as shown in Table 1. Each car is described by several attributes, e.g., horsepower and safety rating. A manufacturer wants to make a production plan for a particular car $\boldsymbol{q}$. For an optimal production plan, it is crucial to identify the potential market for car $\boldsymbol{q}$. If there are only a handful of prospective customers, a small production plan is warranted. Otherwise, the production plan should be extensive.

In the literature, customer preference is commonly modeled by a linear score function [26, 38, 42]. It quantifies the importance of attributes to a customer as a vector $\boldsymbol{u}$, called *utility vector*. Each numeric weight $u[i]$ in the utility vector corresponds to an attribute, where a higher $u[i]$ indicates that the attribute is more important to the customer. In the car market example, there are two attributes. We can have a two-dimensional vector $\boldsymbol{u} = (u[1], u[2])$ for each customer that reflects the importance of the two attributes to the customer. Based on the utility vector $\boldsymbol{u}$, each car has a score that is the weighted sum of its attribute values. If the score of car $\boldsymbol{q}$ is high enough, car $\boldsymbol{q}$ is under strong consideration for purchase by the customer. Thus, to find prospective customers, we only need to search for the utility vectors based on which the score of car $\boldsymbol{q}$ is high enough. If the number of such utility vectors is large, there will be many prospective customers of car $\boldsymbol{q}$.

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* [20, 39, 43]. 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 minimal threshold. Intuitively, this requirement posits that the score of car $\boldsymbol{q}$ should be the highest or nearly 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 linear score functions (i.e., utility vectors $\boldsymbol{u}$) based on which the *regret ratio* of $\boldsymbol{q}$ is below a given threshold. Take Table 1 as an example. There are a query car $\boldsymbol{q}$ and three other

Proceedings of the VLDB Endowment, Vol. 14, No. 1 ISSN 2150-8097.
doi:XX.XX/XXX.XX

**Table 1: Dataset Car**

| Car | Horsepower ($\times 10^2$ hp) | Safety Rating | $f_{\boldsymbol{u}_1}(\cdot)$ |
|---|---|---|---|
| $\boldsymbol{p}_1$ | 4.3 | 5 | 4.37 |
| $\boldsymbol{p}_2$ | 4.5 | 4 | 4.45 |
| $\boldsymbol{p}_3$ | 5.0 | 1 | 4.60 |
| $\boldsymbol{q}$ | 4.5 | 2 | 4.25 |

cars $\boldsymbol{p}_1$, $\boldsymbol{p}_2$, and $\boldsymbol{p}_3$ in the market. Assume that the threshold is set to 0.1. Based on utility vector $\boldsymbol{u}_1$, car $\boldsymbol{p}_3$ has the highest score. The regret ratio of car $\boldsymbol{q}$ is calculated as (4.60-4.25) /4.60 = 0.076, which is below threshold 0.1. Thus, utility vector $\boldsymbol{u}_1$ is returned as an output.

To the best of our knowledge, we are the first to study problem RRQ. There are some closely related studies [7, 9, 18, 30, 33–35] focusing on reverse operators, but they are distinct to ours. One representative operator is the *reverse top-$k$ query* [30, 33–35]. It ranks the products in descending order based on their scores w.r.t. each utility vector, and then returns the utility vectors 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 relative positions of products without accounting for the score differences between them, potentially causing a loss of details about the products' attractiveness to prospective customers.

For instance, consider Table 1. The fourth column displays the score of each car based on utility vector $\boldsymbol{u}_1$. 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 be considered interesting to customers. However, this considerable gap in the ranking is caused by a minor score difference. In fact, the score of $\boldsymbol{q}$ is close to those of the others. Car $\boldsymbol{q}$ could attract the customer's attention. It would be imprudent to dismiss the competitiveness of car $\boldsymbol{q}$.

In this paper, we directly utilize product scores to evaluate products, which are more fundamental than product rankings. This approach offers a better assessment of product competitiveness, making it suited for identifying prospective customers. It is worth noting that the concept of the regret ratio compares the score of each product with the highest score in the market. However, "the highest score" might sometimes be overly stringent [2, 5]. To broaden our problem and accommodate various scenarios, following [2, 5], 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 *E-PT* and *A-PC* for the general case of RRQ, in which each product can be described by an arbitrary number of attributes. Algorithm *E-PT* is an exact algorithm that returns complete results. Algorithm *A-PC* is an approximate algorithm that may miss some results but runs faster than *E-PT*.
- We conducted experiments on synthetic and real datasets, verifying that our studied problem evaluates the prospective customers soundly and insightfully, and showing that 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 2. The formal problem definition and characteristics are shown in Section 3. Section 4 describes algorithm *Sweeping* for the special case of RRQ. Section 5 presents two algorithms *E-PT* and *A-PC* for the general case of RRQ. The experiments are shown in Section 6 and finally, Section 7 concludes our paper.

## 2 RELATED WORK

Various operators [1, 3–5, 13, 20], known as *multi-criteria decision-making operators*, have been proposed to return products for customers based on their preferences. Two widely studied operators among them are the skyline query and the top-$k$ query.

The skyline query [1, 4, 12, 15, 17, 21, 27, 29, 31] 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. Many algorithms [1, 16, 22] have been proposed for the skyline query. [1] proposes a divide-and-conquer algorithm. It divides the dataset into several subsets. For each subset, it computes the partial skyline products, and the final skyline is obtained by merging the partial ones. To accelerate, [16] utilizes the R-tree index [12, 27]. It initializes all the products as the candidate skyline products and constructs an R-tree index based on them. Then, it processes the candidates in rounds. In each round, it picks a candidate from the R-tree as a pivot, and refines the candidates by identifying whether the products are skyline or non-skyline products concerning the pivot. [22] further improves the usage of R-trees, by mapping the skyline definitions to the R-tree structure, and then, refining the candidates directly based on the R-tree. The deficiency of a skyline query is that its output size is uncontrollable [36, 39]. It is possible that none of the products are dominated by others, and thus, the whole product dataset is returned to customers.

The top-$k$ query [3, 11, 13, 24, 28, 32, 37] avoids the uncontrollable output size. It returns the best $k$ products based on customer preference. At a high level, it models customer preference by a score function. Based on the score function, each product is associated with a score. A higher score indicates that the product is more preferred by the customer. The top-$k$ query returns the $k$ products with the highest score to the customer. There are many algorithms [3, 11, 24, 37] designed for the top-$k$ query. [3, 11] employ sophisticated techniques, such as *Onion* and *kSkyband*, to first process the dataset into a small set. Then, they quickly obtain the exact output from the processed set. [24, 37] solve the top-$k$ query from the geometric perspective. Unfortunately, there is a practical barrier to use of the top-$k$ query. It requires customers to specify their score functions. Usually, customers have difficulty in explicitly specifying the exact parameters in the score function. If the parameters given to the system differ slightly, the output can vary a lot [19, 37].

Observing the weaknesses of the skyline query and the top-$k$ query, the regret minimization query [2, 5, 19, 20, 23, 40, 41] has been proposed. It combines the benefits of both the skyline query and the top-$k$ query: (1) following the skyline query, it does not require customers to specify their score functions; (1) following the top-$k$ query, it has a controllable output size. 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, 20, 23, 41] that are proposed for the regret minimization query. [5, 23] adopt a greedy strategy that iteratively identifies the product minimizing the difference and add it into $S$. [20, 41] apply sampling strategies. [20] samples the product dataset by partitioning the products into different *cubes*. Then, it selects one product from each cube as the output. [41] samples the customer preference and identifies products as output based on the sampled customer preference.

Among the operators discussed so far, a common characteristic is that they all stand on the customer's perspective. Their target is to find products for customers. Recently, many operators have attempted to reverse the direction and stand on the products' perspective. Their goal is to identify prospective customers for products. One representative operator is the reverse top-$k$ query.

Given a product $\boldsymbol{q}$, the reverse top-$k$ query [30, 33–35] returns all customers such that $\boldsymbol{q}$ is one of the best $k$ products based on customers' preferences. Specifically, [33–35] assume that there is a set $U$ of linear score functions modeling customer preferences. For each score function in $U$, the score of each product can be calculated. [33–35] aim to identify the score functions in set $U$ for which the score of the given product $\boldsymbol{q}$ is among the top-$k$. However, [33–35] only permits set $U$ to comprise a finite number of discrete linear score functions. They cannot deal with the entire linear score function space, i.e., the entire customer preference space.

Motivated by this insufficiency, [30] studies a more general case in which the function set $U$ is set to be the entire linear score function space. It proposes an algorithm via partitioning. At a high level, it partitions the linear score function space into *cells* and build a *cell-tree* to index the cells. By processing the cell-tree, it checks the cells and returns score functions from qualified cells. However, the weakness of [30] is that it measures products merely based on the product rankings. This may result in an inappropriate evaluation of products, as discussed in Section 1. In contrast, our work takes product scores into account. This will provide us with a more comprehensive view of products and, consequently, yield a sounder and more insightful identification of prospective customers.

## 3 PROBLEM DEFINITION

### 3.1 Problem Reverse Regret Query

**Product/Point.** The input consists of a set $\mathcal{D}$ of $n$ products and a query product. Each product is described by $d$ attributes. We represent each product as a $d$-dimensional point $\boldsymbol{p} = (p[1], p[2], ..., p[d])$. 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. In the following, we use the words product/point and attribute/dimension interchangeably.

**Utility Function.** Following [20, 26, 30, 33, 37, 41], we model the customer preference as a linear score function $f_{\boldsymbol{u}}(\boldsymbol{p}) = \boldsymbol{u} \cdot \boldsymbol{p} = \sum_{i=1}^{d} u[i]p[i]$, namely *utility function*. The linear score function is widely used for modeling customer preference [8, 14]. The customer studies conducted by [26, 39] also show that linear score functions can effectively capture how customers evaluate products.

**Table 2: Frequently Used Notations**

| Notation | Definition |
|---|---|
| $\mathcal{D}$ and $\boldsymbol{p}$ | A dataset and a point in the dataset. |
| $\boldsymbol{q}$ | The query point. |
| $n$ and $d$ | The dataset size and the dimension of points. |
| $k$ and $\epsilon$ | Parameters for evaluating points. |
| $\mathcal{U}/\mathcal{L}$ and $\boldsymbol{u}/f_{\boldsymbol{u}}$ | The (2D) utility space and a utility vector/function. |
| $kmax_{\boldsymbol{p}\in\mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p})$ | The utility of the $k$-th ranked point in $\mathcal{D}$ w.r.t $\boldsymbol{u}$. |
| $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u})$ | The $k$-regret ratio of point $\boldsymbol{q}$ w.r.t. $\boldsymbol{u}$. |
| $c$ and $V_c/\mathbb{V}$ | A partition in $\mathcal{U}$ and the volume of $c/\mathcal{U}$. |
| $h_{\boldsymbol{q},\boldsymbol{p}}$ | The hyper-plane of $\boldsymbol{q}$ and $\boldsymbol{p}$. |
| $h^+_{\boldsymbol{q},\boldsymbol{p}}/h^-_{\boldsymbol{q},\boldsymbol{p}}$ | The positive/negative half-space of $h_{\boldsymbol{q},\boldsymbol{p}}$. |
| $lh_k/uh_k$ | The $k$-th ranked inclusive/exclusive hyper-plane. |
| $\wedge(h, \mathcal{L})$ | The intersection between a hyper-plane $h$ and $\mathcal{L}$. |
| $N/Q(N)$ | A node in the P-tree and a counter of $N$. |
| $\mathcal{H}(N)$ | The hyper-planes intersecting the partition in $N$. |
| $\mathcal{N}$ | The total number of sampled utility vectors. |

Each utility function is associated with 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 point $\boldsymbol{p}$ to point $\boldsymbol{q}$, the utility of $\boldsymbol{p}$ is higher than that of $\boldsymbol{q}$, i.e., $f_{\boldsymbol{u}}(\boldsymbol{p}) > f_{\boldsymbol{u}}(\boldsymbol{q})$. Note that, as discussed in [20, 38, 39], 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 3.1.* Consider Table 3 where each point has two dimensions. Let $f_{\boldsymbol{u}}(\boldsymbol{p}) = 0.5p[1] + 0.5p[2]$ (i.e., $\boldsymbol{u} = (0.5, 0.5)$). Then, the utility of $\boldsymbol{p}_1$ w.r.t. $\boldsymbol{u}$ is $f(\boldsymbol{p}_1) = 0.5 \times 0.20 + 0.5 \times 0.92 = 0.56$. The utilities of other points w.r.t. $\boldsymbol{u}$ can be computed similarly.

**Regret Point.** Given a dataset $\mathcal{D}$ and a utility function $f_{\boldsymbol{u}}$, the points in $\mathcal{D}$ can be ranked based on their utilities w.r.t. $\boldsymbol{u}$ in descending order. Let us denote the utility of the $k$-th ranked point by $kmax_{\boldsymbol{p}\in\mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p})$. Returning to Table 3, since point $p_1$ ranks the second w.r.t. $\boldsymbol{u}$, 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 $\boldsymbol{q}$, we utilize $kmax_{\boldsymbol{p}\in\mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p})$ following [2, 5]. Specifically, the $k$-regret ratio of $\boldsymbol{q}$ is defined below.

*Definition 3.2 (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 the difference between the utility of $\boldsymbol{q}$ and that of the $k$-th ranked point in $\mathcal{D}$ w.r.t. $\boldsymbol{u}$. The ratio falls in the range $[0, 1]$. If $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u})$ is below a small threshold $\epsilon$ (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 $(k, \epsilon)$-*regret point* w.r.t. $\boldsymbol{u}$.

*Example 3.3.* Continue Example 3.1, where $\boldsymbol{u} = (0.5, 0.5)$. Suppose that $\boldsymbol{q} = (0.4, 0.7)$ and $\epsilon = 0.1$. Since $2max_{\boldsymbol{p}\in\mathcal{D}} f_{\boldsymbol{u}}(\boldsymbol{p}) = 0.56$,

**Table 3: Dataset**

| $p$ | $p[1]$ | $p[2]$ | $f_{(0.5,0.5)}(p)$ |
|---|---|---|---|
| $p_1$ | 0.2 | 0.92 | 0.56 |
| $p_2$ | 0.70 | 0.54 | 0.62 |
| $p_3$ | 0.60 | 0.30 | 0.45 |

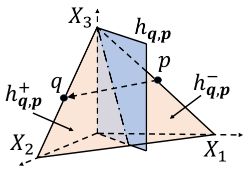


**Figure 1: Hyper-plane**

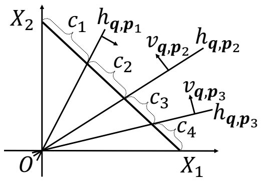


**Figure 2: Two Dim**

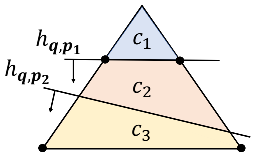


**Figure 3: High Dim**

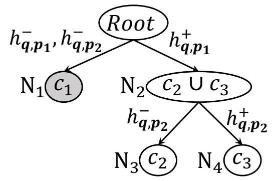


**Figure 4: P-Tree**

the 2-regret ratio of point $q$ is $2\text{-}regratio(q, u) = \max(0, 0.56 - 0.55)/0.56 = 0.018 < \epsilon$. Thus, $q$ is a $(2, 0.1)$-regret point w.r.t. $u$.

We now formally define problem reverse regret query (RRQ). The frequently used notations can be found in Table 2.

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

Back to Example 3.3, utility vector $u = (0.5, 0.5)$ will be returned as an output for problem RRQ since $q$ is a (2, 0.1)-regret point w.r.t. $u$. Here $u$ is said to be 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.*

## 3.2 Problem Characteristics

In this section, we formalize our problem RRQ from a geometric perspective. In a $d$-dimensional geometric space $\mathbb{R}^d$, each utility vector $u \in \mathcal{U}$ can be seen as a point. Recall that we assume that (1) $u[i] \geq 0$ for every dimension, and (2) $\sum_{i=1}^{d} u[i] = 1$. The utility space $\mathcal{U}$ is a *polyhedron* [6] in $\mathbb{R}^d$, e.g., a line segment when $d = 2$ as shown in Figure 2 and a triangle when $d = 3$ as shown in Figure 1.

For each $p \in \mathcal{D}$, we can build a *hyper-plane* $h_{q,p}$ with query point $q$ in $\mathbb{R}^d$: $\{r \in \mathbb{R}^d | r \cdot (q - (1 - \epsilon)p) = 0\}$, which passes through the origin with its unit norm in the same direction as $q - (1 - \epsilon)p$ [6]. Figure 1 shows an example of a hyper-plane. For each utility vector $u \in \mathcal{U} \cap h_{q,p}$, we have $u \cdot (q - (1 - \epsilon)p) = 0$, i.e., $f_u(q) = (1 - \epsilon)f_u(p)$. The hyper-plane $h_{q,p}$ divides the utility space into two half-spaces. The half-space $h^+_{q,p}$ above $h_{q,p}$, called *positive half-space*, contains all utility vectors $u$ such that $u \cdot (q - (1-\epsilon)p) > 0$, i.e., $f_u(q) > (1-\epsilon)f_u(p)$. The half-space $h^-_{q,p}$ below $h_{q,p}$, called *negative half-space*, contains all the utility vectors $u$ such that $u \cdot (q - (1 - \epsilon)p) < 0$, i.e., $f_u(q) < (1 - \epsilon)f_u(p)$. Note that in this paper, the half-space, $h^+_{q,p}$ or $h^-_{q,p}$, is bounded by utility space $\mathcal{U}$ instead of being defined on the entire geometric space.

*Example 3.4.* Consider the points in Table 3. Assume that $q = (0.4, 0.7)$ and threshold $\epsilon = 0.1$. For point $p_1$, we can build a hyper-plane $h_{q,p_1}$ in space $\mathbb{R}^2$: $\{r \in \mathbb{R}^2 | r \cdot (0.22, -0.13) = 0\}$ shown as a line in Figure 2. The black arrow is in the same direction as vector $(0.22, -0.13)$. Similarly for hyper-planes $h_{q,p_2}$ and $h_{q,p_3}$.

We then apply the concepts of hyper-plane/half-space to our problem RRQ. For each point $p \in \mathcal{D}$, we build a hyper-plane $h_{q,p}$ based on $q$ and it will divide the utility space into two half-spaces. Since $|\mathcal{D}| = n$, there are $n$ hyper-planes and $2n$ half-spaces. For any two half-spaces, e.g., $h^+_{q,p}$ and $h^+_{q,p'}$, we say $h^+_{q,p}$ is *covered* by $h^+_{q,p'}$, denoted by $h^+_{q,p} \subseteq h^+_{q,p'}$ if any utility vector in $h^+_{q,p}$ is also contained in $h^+_{q,p'}$. For example, in Figure 2, half-space $h^+_{q,p_2}$ is covered by $h^+_{q,p_3}$. The $n$ hyper-planes divide the utility space into $O(n^{d-1})$ regions, called *partitions* [6]. Each partition is *covered* by a total of $n$ (positive or negative) half-spaces. Formally, we say a partition $c$ is covered by a half-space, say $h^+_{q,p}$, denoted by $c \subseteq h^+_{q,p}$ if any utility vector in partition $c$ is also in half-space $h^+_{q,p}$.

The following lemma shows how to determine if query point $q$ is a $(k, \epsilon)$-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 appendix.

Lemma 3.5. *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 $q$ is a $(k, \epsilon)$-regret point w.r.t. any utility vectors in partition $c$.*

*Example 3.6.* Continue Example 3.4 as shown in Figure 2, where $\epsilon = 0.1$. There are three hyper-planes $h_{q,p_1}$, $h_{q,p_2}$, and $h_{q,p_3}$, which divide the utility space into four partitions $c_1$, $c_2$, $c_3$, and $c_4$. Partition $c_1 = h^-_{q,p_1} \cap h^+_{q,p_2} \cap h^+_{q,p_3}$ is covered by *one* negative half-space $h^-_{q,p_1}$, and two positive half-spaces $h^+_{q,p_2}$ and $h^+_{q,p_3}$. Thus, query point $q$ is a (2, 0.1)-regret point w.r.t. any utility vectors in partition $c_1$.

According to Lemma 3.5, we can solve problem RRQ in roughly three steps: (1) construct a hyper-plane $h_{q,p}$ for each point $p \in \mathcal{D}$; (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 $k = 2$ in Example 3.6. Hyper-planes $h_{q,p_1}$, $h_{q,p_2}$, and $h_{q,p_3}$ divide the utility space into four partitions. Partitions $c_1$, $c_2$, and $c_3$ will be returned since they are covered by fewer than two negative half-spaces.

## 4 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 the hyper-planes, the resulting partitions are ordered sub-segments of $\mathcal{L}$. 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 $k - n$ positive half-spaces), it is a qualified partition. The algorithm returns all qualified partitions as its output.

However, two issues affect the efficiency of *Sweeping*. Firstly, there are many partitions that need to be checked. Since we build a hyper-plane for each point $p \in \mathcal{D}$, there are $n$ hyper-planes that divide the utility space into $O(n)$ partitions. Secondly, it is costly to check whether a partition is qualified to be returned. Since each partition is covered by a total of $n$ (positive or negative) half-spaces, we need $O(n)$ time to count the number of negative half-spaces. In the following, we address these two issues, respectively.

**Partition Reduction.** Let us consider $\boldsymbol{r} = (1, 0)$ in $\mathcal{L}$ as a reference utility vector in space $\mathbb{R}^2$. The negative half-space of a hyper-plane may contain $\boldsymbol{r}$. For example, in Figure 2, negative half-spaces $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$ and $h^-_{\boldsymbol{q},\boldsymbol{p}_3}$ contain $\boldsymbol{r}$, while $h^-_{\boldsymbol{q},\boldsymbol{p}_1}$ does not. We call a hyper-plane an *inclusive hyper-plane* (resp. an *exclusive hyper-plane*) if its negative half-space contains $r$ (resp. does not contain $\boldsymbol{r}$).

Consider all the inclusive hyper-planes. For ease of illustration, 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 4.1. *Query point $\boldsymbol{q}$ is not a $(k, \epsilon)$-regret point w.r.t. any utility vectors $\boldsymbol{u}$ in the negative half-space of $lh_k$ (i.e., $\boldsymbol{u} \in lh^-_k$).*

Lemma 4.1 indicates that the partitions in $lh^-_k$ are not qualified to be returned. Therefore, we can directly omit those partitions. To illustrate, consider Figure 2 and assume $k = 1$. There are two inclusive hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}_2}$ and $h_{\boldsymbol{q},\boldsymbol{p}_3}$. Since $h_{\boldsymbol{q},\boldsymbol{p}_2}$ 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 $uh_k$. Then, the partitions in $uh^-_k$ can also be omitted.

Lemma 4.2. *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 its positive half-space $h^+$ (resp. negative half-space $h^-$) first, then the intersection $\wedge(h, \mathcal{L})$, and finally its negative half-space $h^-$ (resp. positive half-space $h^+$). 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 the intersection $\wedge(h_{\boldsymbol{q},\boldsymbol{p}_2}, \mathcal{L})$, we learn that 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 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 4.3. *The checking cost for each partition is reduced to $O(1)$.*

We now summarize our algorithm *Sweeping*. The pseudocode is shown in Algorithm 1. In the beginning, to exclude the partitions in negative half-spaces $lh^-_k$ and $uh^-_k$ from consideration, we find the $k$-th ranked inclusive hyper-plane $lh_k$ and the $k$-th ranked exclusive hyper-plane $uh_k$ (line 1), and filter out the hyper-planes that intersect $\mathcal{L}$ in $lh^-_k$ or $uh^-_k$ (line 2). We rank the remaining hyper-planes based on their intersections with $\mathcal{L}$ and start the sweep process (line 3). We begin with the partition next to $\wedge(uh_k, \mathcal{L})$ and set $Q$ to be the number of negative half-spaces covering it. Then, 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 1 (resp. by subtracting 1) and we check whether partition $c$ is qualified to be returned (lines 5-7). The process stops when all partitions in $uh^+_k \cap lh^+_k$ are swept (line 4).

**Algorithm 1:** Algorithm *Sweeping*

```
   Input: A point set $\mathcal{D}$, a query point $\boldsymbol{q}$, parameters $k$ and $\epsilon$.
   Output: The set $C$ of qualified partitions.
1  Find the $k$-th ranked inclusive hyper-plane $lh_k$ and the $k$-th
     ranked exclusive hyper-plane $uh_k$.
2  Filter out the hyper-planes intersecting $\mathcal{L}$ in $lh^-_k$ or $uh^-_k$.
3  Rank the remaining hyper-planes $h$ according to $\wedge(h, \mathcal{L})$.
4  for each partition $c$ in $uh^+_k \cap lh^+_k$ do
5      Update integer $Q$ for partition $c$;
6      if $Q < k$ then
7          $C \leftarrow C \cup \{c\}$;
8  return The set $C$ of qualified partitions.
```

To illustrate, consider Figure 2 and assume $k = 1$. There are three hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}_1}$, $h_{\boldsymbol{q},\boldsymbol{p}_2}$, and $h_{\boldsymbol{q},\boldsymbol{p}_3}$. The top-ranked exclusive (resp. inclusive) hyper-plane is $h_{\boldsymbol{q},\boldsymbol{p}_1}$ (resp. $h_{\boldsymbol{q},\boldsymbol{p}_2}$), i.e., $uh_k = h_{\boldsymbol{q},\boldsymbol{p}_1}$ and $lh_k = h_{\boldsymbol{q},\boldsymbol{p}_2}$. We filter out hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}_3}$ since it intersects $\mathcal{L}$ in $lh^-_k$. Then, we rank the remaining hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}_1}$ and $h_{\boldsymbol{q},\boldsymbol{p}_2}$, and conduct the sweep from $\wedge(h_{\boldsymbol{q},\boldsymbol{p}_1}, \mathcal{L})$ to $\wedge(h_{\boldsymbol{q},\boldsymbol{p}_2}, \mathcal{L})$. In $h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_2}$, there is only one partition $c_2$ left with $Q = 0$. Since $Q < k$, partition $c_2$ is returned as the final output, i.e., $C = \{c_2\}$.

Theorem 4.4. *The time complexity of* Sweeping *is $O(n)$.*

## 5 HIGH DIMENSIONAL ALGORITHMS

In this section, we consider the general case of problem RRQ in which each point has multiple dimensions (i.e., $d \geq 2$). We present two algorithms in Section 5.1 and Section 5.2, respectively. The first algorithm, termed *Partition Tree (E-PT)*, is an exact algorithm. It returns all qualified partitions. The second algorithm, named *Progressive Construction (A-PC)*, is an approximate algorithm. It sacrifices some qualified partitions to achieve a better execution time.

### 5.1 Exact Algorithm

In a high dimensional space, it is hard to check all partitions formed by $n$ hyper-planes. The main idea of our algorithm is to construct a tree-structured index for the partitions, called *Partition Tree*, or *P-Tree* in short. For each point $\boldsymbol{p} \in \mathcal{D}$, we build a hyper-plane $h_{\boldsymbol{p},\boldsymbol{q}}$ with query point $\boldsymbol{q}$ and insert $h_{\boldsymbol{p},\boldsymbol{q}}$ into the P-Tree. The P-Tree incrementally maintains the partitions in the utility space based on the hyper-planes inserted so far. When all the hyper-planes are inserted, the partitions in the P-Tree that are covered by fewer than $k$ negative half-spaces are returned as the output.

*5.1.1 Partition Tree.* The P-Tree is a tree-based data structure. It indexes the partitions in the utility space formed by the hyper-planes inserted so far. Each leaf node contains one partition and each internal node contains the union of partitions from its reachable leaves. For ease of illustration, 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 whole triangle represents the utility space, which 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 P-Tree with three leaves, which 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 partitions $c_2$ and $c_3$, respectively. Since partition $c_2 \cup c_3$ is covered by neither $h^-_{\boldsymbol{q},\boldsymbol{p}_1}$ nor $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$, $Q(N_2) = 0$.

We build the P-Tree incrementally by inserting hyper-planes one at a time. Initially, the root node is constructed, whose partition is the entire utility space. When a hyper-plane is inserted, the partitions stored in some leaves are further divided into smaller ones, and thus, the corresponding leaves are split accordingly. Specifically, for each hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ inserted, we carry out a top-down insertion starting from the root node. Suppose that the insertion comes to a node $N$ in the P-Tree. We process it 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 its children 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 $c_1 = c \cap h^-_{\boldsymbol{q},\boldsymbol{p}}$ and $Q(N') = Q(N)+1$; the second child contains partition $c_2 = c \cap h^+_{\boldsymbol{q},\boldsymbol{p}}$ and $Q(N'') = Q(N)$. Note that for the first child node $N'$, if $Q(N') \geq k$, it will be marked as invalid, since partition $c_1$ is already covered by $k$ negative half-spaces, and thus, $c_1$ is not qualified to be returned.

To illustrate, suppose $k = 2$. We build a P-Tree based on the partitions and hyper-planes as shown in Figure 3. In the beginning, the partition in the root is the entire utility space $\mathcal{U}$. Since no negative half-space covers the utility space, $Q(Root) = 0$. 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. We utilize the *extreme points* (i.e. corner points [6]) of a partition. Intuitively, a partition $c$ is the intersection of multiple half-spaces, i.e., a *convex polyhedron* [6]. In a convex polyhedron, any points can be represented as the non-negative weighted sum of the extreme points of the polyhedron. Thus, we can learn the relationship between a partition and a hyper-plane based on the extreme points of the partition. Specifically, assume that partition $c$ has $x$ extreme points $\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_x$.

Lemma 5.1. *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 half-space $h^+_{\boldsymbol{q},\boldsymbol{p}_2}$ and another two in half-space $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$. 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. In Section 5.1.2, we will provide strategies to reduce the checking cost to $O(1)$ time.

*5.1.2 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.** The P-Tree construction requires building a hyper-plane for each point in $\mathcal{D}$. Since $|\mathcal{D}| = n$, there are $n$ hyper-planes built and inserted into the P-Tree. To save the cost, we reduce the number of hyper-planes that need to be inserted.

Consider a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$. Suppose 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., $h^-_{\boldsymbol{q},\boldsymbol{p}} \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}'}$). If hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ is not inserted into the P-Tree, (1) for any partitions covered by $h^-_{\boldsymbol{q},\boldsymbol{p}}$, they remain unqualified since they are already covered by $k$ half-spaces $h_{\boldsymbol{q},\boldsymbol{p}'}$; (2) for any partitions covered by $h^+_{\boldsymbol{q},\boldsymbol{p}}$, the numbers of negative half-spaces covering them are not affected. Therefore, there is no need to insert hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ into the P-Tree. Based on this idea, we only insert hyper-planes $h_{\boldsymbol{q},\boldsymbol{p}}$ into the P-Tree if its negative half-space is covered by fewer than $k$ negative half-spaces. Lemma 5.2 shows how to check if a negative half-space $h^-_{\boldsymbol{q},\boldsymbol{p}}$ is covered by another $h^-_{\boldsymbol{q},\boldsymbol{p}'}$. 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 5.2. *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 $h^-_{\boldsymbol{q},\boldsymbol{p}_2}$, i.e., $h^-_{\boldsymbol{q},\boldsymbol{p}_3} \subseteq h^-_{\boldsymbol{q},\boldsymbol{p}_2}$.

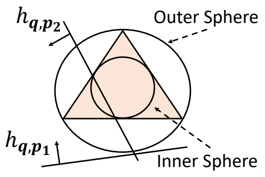


**Figure 5: Relationship**

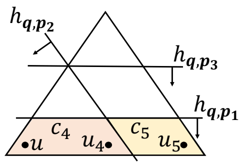


**Figure 6: Sampled $u$**

**Insertion Order.** During the P-Tree construction, some nodes may be marked as invalid and will not be processed by future hyper-plane insertions. We want to develop an insertion order for hyper-planes, so that if a node is indeed invalid, it can be marked as early as possible. In this way, we can reduce the number of nodes processed for each hyper-plane insertion, thereby saving computation cost.

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

**Relationship Checking.** We present strategies to speed up the relationship checking between a partition and a hyper-plane. Firstly, we 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. $c'$).

Lemma 5.3. *If partition $c$ is covered by a half-space $h^+_{q,p}$ (resp. $h^-_{q,p}$), partition $c'$ must be covered by the half-space $h^+_{q,p}$ (resp. $h^-_{q,p}$).*

Consider Figure 4. Suppose that there is a half-space covering the partition in node $N_2$. Since nodes $N_3$ and $N_4$ are the two children of $N_2$, this half-space must cover the partitions in $N_3$ and $N_4$.

Our second strategy is to approximate the partition with a sphere, which allows for faster relationship checking, as 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 Euclidean distance from the center $\mathbb{B}_c$ of a sphere to a hyper-plane $h$. If $dist(h, \mathbb{B}_c)$ is smaller than the radius of the sphere, the sphere must intersect the hyper-plane. Otherwise, we can easily determine which half-space the sphere is in based on the center of the sphere. For example, in Figure 5, the larger sphere is in half-space $h^+_{q,p_1}$, since (1) $dist(h, \mathbb{B}_c)$ is larger than the radius of the sphere and (2) the center of the sphere is in half-space $h^+_{q,p_1}$.

We introduce two concepts: the *outer sphere* and *inner sphere* of a partition $c$. Assume that partition $c$ has $x$ extreme points $\boldsymbol{e}_1, \boldsymbol{e}_2, ..., \boldsymbol{e}_x$. 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., $\mathbb{O}_r = \max\{\text{dist}(\boldsymbol{e}_1, \mathbb{O}_c), \text{dist}(\boldsymbol{e}_2, \mathbb{O}_c), ..., \text{dist}(\boldsymbol{e}_x, \mathbb{O}_c)\}$, where $\text{dist}(\boldsymbol{e}_i, \mathbb{O}_c)$ denotes the Euclidean distance between $\boldsymbol{e}_i$ and $\mathbb{O}_c$). Intuitively, the outer sphere of a partition $c$ is a sphere that covers $c$.

Lemma 5.4. *If a half-space (i.e., $h^+_{q,p}$ or $h^-_{q,p}$) covers the outer sphere of a partition, the half-space covers the partition.*

Recall that a partition $c$ is the intersection of multiple half-spaces. We call the hyper-planes corresponding to these half-spaces the *component hyper-planes* of partition $c$. Assume that partition $c$ has $y$ component hyper-planes $h_1, h_2, ..., h_y$. For the inner sphere of partition $c$, we define its 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 component hyper-plane (i.e., $\mathbb{I}_r = \min\{\text{dist}(h_1, \mathbb{I}_c), \text{dist}(h_2, \mathbb{I}_c), ..., \text{dist}(h_y, \mathbb{I}_c)\}$, where $\text{dist}(h_i, \mathbb{I}_c)$ denotes the smallest Euclidean distance from $\mathbb{I}_c$ to any point in $h_i$). Intuitively, the inner sphere of a partition $c$ is a sphere covered by $c$.

Lemma 5.5. *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 $h^+_{q,p_1}$, the partition is also in half-space $h^+_{q,p_1}$. Since hyper-plane $h_{q,p_2}$ intersects the inner sphere, hyper-plane $h_{q,p_2}$ 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 5.1.1

**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 $h_{q,p}$ comes to a leaf node $N$ and the partition in $N$ intersects $h_{q,p}$, we simply store $h_{q,p}$ 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 $h_{q,p}$ in $\mathcal{H}(N)$ and create two children for node $N$. One child $N'$ contains partition $c_1 = c \cap h^-_{q,p}$ and $Q(N') = Q(N)+1$; the other child $N''$ contains partition $c_2 = c \cap h^+_{q,p}$ 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 $h_{q,p} \in \mathcal{H}(N')$, if its half-space (either $h^-_{q,p}$ or $h^+_{q,p}$) covers partition $c_1$, we remove $h_{q,p}$ from $\mathcal{H}(N')$. Besides, $Q(N')$ is updated to be $Q(N') = Q(N') + 1$ if $c_1 \subseteq h^-_{q,p}$. After the refinement, if $Q(N') + |\mathcal{H}(N')| < k$, we stop. Otherwise, $N'$ is recursively split. Similarly for node $N''$.

*5.1.3 Summary and Analysis.* We are ready to present our algorithm *E-PT*. The pseudocode is shown in Algorithm 2. In the beginning, we reduce the number of hyper-planes inserted, by filtering

**Algorithm 2:** Algorithm *E-PT*

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

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

   Lazy_Split(node $N$)
22 $h_{\boldsymbol{q},\boldsymbol{p}} \leftarrow$ the oldest hyper-plane in $\mathcal{H}(N)$;
23 $\mathcal{H}(N) = \mathcal{H}(N) \setminus \{h_{\boldsymbol{q},\boldsymbol{p}}\}$;
24 Create two children $N'$ and $N''$ for $N$;
25 $c_1 = c \cap h^-_{\boldsymbol{q},\boldsymbol{p}}$; $Q(N') = Q(N) + 1$; $\mathcal{H}(N') = \mathcal{H}(N)$;
26 $c_2 = c \cap h^+_{\boldsymbol{q},\boldsymbol{p}}$; $Q(N'') = Q(N)$; $\mathcal{H}(N'') = \mathcal{H}(N)$;
27 Refine($N'$); Refine($N''$);

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

out the hyper-planes whose negative half-space is covered by at least $k$ negative half-spaces (line 1). Then, the remaining hyper-planes are inserted into the P-Tree based on their $W(h_{\boldsymbol{q},\boldsymbol{p}})$ in descending order (lines 1-3). When all the 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 4).

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 5-6). If $Q(N) \geq k$, node $N$ is marked as invalid (lines 7-8). If node $N$ is a valid internal node, the insertion proceeds to its children recursively (lines 9-11). 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 12-13). (2) Suppose $c \cap h^-_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$ and $c \cap h^+_{\boldsymbol{q},\boldsymbol{p}} \neq \emptyset$ (line 14). If node $N$ is an internal node, the insertion proceeds to its children recursively (lines 15-17). If node $N$ is a valid leaf, we add the hyper-plane to set $\mathcal{H}(Q)$ (lines 18-19). 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 20-21).

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 22-24). Each child stores its own partition and counter, and inherits hyper-plane set $\mathcal{H}(N)$ (lines 25 - 26). 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 27-35). 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 36-37).

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

THEOREM 5.6. *The time complexity of* 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 $c \cap h^+_{\boldsymbol{q},\boldsymbol{p}}$), we only add one hyper-plane to an existing partition $c$. Thus, the time cost for each partition construction is low [6, 39], resulting in a small $O(\alpha)$.

## 5.2 Approximate Algorithm

In algorithm *E-PT*, we need to construct partitions for internal nodes. Although those partitions play an important role in hierarchical indexing, 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.

*5.2.1 Progressive Construction.* Our algorithm operates by randomly sampling a set of utility vectors in the utility space, which serve as the basis for progressive partition construction. 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 $k\text{-}regratio(\boldsymbol{q}, \boldsymbol{u}) < \epsilon$, there must exist a qualified partition containing $\boldsymbol{u}$ in the utility space such that query point $\boldsymbol{q}$ is a $(k, \epsilon)$-regret point. In this case, we construct a partition $c_{\boldsymbol{u}}$ based on utility vector $\boldsymbol{u}$ and add it to the final output. Specifically, partition $c_{\boldsymbol{u}}$ is the intersection of $n$ (positive or negative) half-spaces. For each $\boldsymbol{p} \in \mathcal{D}$ such that $(1 - \epsilon)f_{\boldsymbol{u}}(\boldsymbol{p}) < f_{\boldsymbol{u}}(\boldsymbol{q})$, we build a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ and $h^+_{\boldsymbol{q},\boldsymbol{p}}$ is used for partition construction. For each $\boldsymbol{p'} \in \mathcal{D}$ such that $(1 - \epsilon)f_{\boldsymbol{u}}(\boldsymbol{p'}) > f_{\boldsymbol{u}}(\boldsymbol{q})$, we build a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p'}}$ and $h^-_{\boldsymbol{q},\boldsymbol{p'}}$ is used for partition construction.

To illustrate, consider the points in Table 3. 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 $c_{\boldsymbol{u}} = h^+_{\boldsymbol{q},\boldsymbol{p}_1} \cap h^-_{\boldsymbol{q},\boldsymbol{p}_2} \cap h^+_{\boldsymbol{q},\boldsymbol{p}_3}$.

Lemma 5.7. *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 $(k, \epsilon)$-regret point w.r.t. $\boldsymbol{u}'$.*

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

Lemma 5.8. *If the sampled utility vector $\boldsymbol{u}$ is in 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 $c_4$, i.e., $\boldsymbol{u} \in c_4$.

#### 5.2.2 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. Given a utility vector $\boldsymbol{u}$, 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. 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 $\mathcal{D}^-_{\boldsymbol{u}_2}$. Specifically, for each $\boldsymbol{p} \in \mathcal{D}^+_{\boldsymbol{u}_1}$, we build a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ and use $h^+_{\boldsymbol{q},\boldsymbol{p}}$ to construct the partition. For each $\boldsymbol{p}' \in \mathcal{D}^-_{\boldsymbol{u}_2}$, we build a hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}'}$ and use $h^-_{\boldsymbol{q},\boldsymbol{p}'}$ to construct the partition. The final partition is $c_{\boldsymbol{u}_1,\boldsymbol{u}_2} = (\cap_{\boldsymbol{p} \in \mathcal{D}^+_{\boldsymbol{u}_1}} h^+_{\boldsymbol{q},\boldsymbol{p}}) \cap (\cap_{\boldsymbol{p}' \in \mathcal{D}^-_{\boldsymbol{u}_2}} h^-_{\boldsymbol{q},\boldsymbol{p}'})$. Similarly for $\mathcal{D}^+_{\boldsymbol{u}_2} \subseteq \mathcal{D}^+_{\boldsymbol{u}_1}$.

Lemma 5.9. *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 $\boldsymbol{u}' \in c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$, query point $\boldsymbol{q}$ is a $(k, \epsilon)$-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 $c_5$.

#### 5.2.3 Summary and Analysis.

The pseudocode of algorithm *A-PC* is shown in Algorithm 3. We randomly sample a set $U$ of utility vectors (line 1). For each sampled utility 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 3-5). 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 6-9). Similarly for $\mathcal{D}^+_{\boldsymbol{u}_2} \subseteq \mathcal{D}^+_{\boldsymbol{u}_1}$ (lines 10-12). 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 13-14).

**Algorithm 3:** Algorithm *A-PC*

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

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 that we can find. However, if $\mathcal{N}$ is large, the excessive number of 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 $(k, \epsilon)$-regret point w.r.t. many utility vectors. We regard such a qualified partition as an influential partition since if it is not included in the final output, we miss a large number of qualified utility vectors. In contrast, a qualified partition with a small volume contributes little to the final output. Thus, we primarily target at finding qualified partitions whose volumes are larger than a predefined threshold.

Specifically, we use $V_c$ (resp. $\mathbb{V}$) to denote the volume of a qualified partition $c$ (resp. the utility space $\mathcal{U}$) and denote by $V_c/\mathbb{V}$ the volume ratio of partition $c$ to the utility space $\mathcal{U}$. Given a real number $\rho$, our goal is to find the qualified partitions such that $V_c/\mathbb{V} > \rho$.

Lemma 5.10. *Given a confidence parameter $\delta$ and a sampling size $\mathcal{N} = O((1/\rho^2)(d + \ln(1/\delta)))$, for each qualified partition $c$ with $V_c/\mathbb{V} > \rho$, our sampling strategy can find it, with confidence $1 - \delta$.*

## 6 EXPERIMENT

In this section, we present our experimental evaluation. We begin by describing the experimental setting in Section 6.1. Then in Section 6.2, we present a user study that showcases 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 6.3 and 6.4, respectively. Finally, our findings are summarized in Section 6.5.

## 6.1 Experimental Setting

The experiments were run on a machine with a 3.10GHz CPU and a 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 [1, 22, 37, 39]. The synthetic datasets are *anti-correlated* (Anti), *correlated* (Cor), and *independent* (Indep) [1, 22]. They represent typical data distributions in multi-criteria decision-making. The real datasets are *Island*, *Weather*, *Car*, and *NBA* [37, 39]. 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 [30, 44] 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 the performances of algorithms by varying the following parameters: (1) parameter $k$; (2) threshold $\epsilon$; (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 [30, 44], we set parameters $k = 10$ 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* [30] and *PBA+* [44] by their execution times. Each algorithm was run 30 times with different query points that were randomly generated, 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 its execution time, excluding its preprocessing time (which can be more than $10^4$ seconds).

## 6.2 User Study

To motivate our problem, in this section, we explored the difference between (1) the product rankings and (2) the product utilities for evaluating products. Following [37–39], we conducted a user study on dataset *Car* to gain valuable insights into how customers evaluate cars in a used car market. Each car in this dataset is described by four attributes (i.e., price, year of manufacture, horsepower and used kilometer). We recruited thirty participants and used an interactive algorithm called *Adaptive* [26] to learn their exact utility functions (i.e., their preferences). With the learned utility function, the cars' utilities and rankings were determined. Then, we provided each participant with their top-$x$ cars, claiming that these results would be of interest to them, where $x$ had three settings: $x = 1$, $x = 5$, and $x = 10$. Next, we found the cars whose $x$-regratios were smaller than 0.1 and uniformly selected five of them. The participants were presented with these five selected cars and were asked to indicate whether they were interested in them.

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 cars presented; and (2) the average rank, i.e., the average ranks of the cars that are of interest to the participants among the cars presented. Figure 7 shows these two results. For different settings of $x$, 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 not sufficient for determining customers' interest and may lead to the loss of details about the product's attractiveness to prospective customers. The regret ratio, which focuses on utilities directly, works better than ranking in accurately modeling customer preferences for products.

## 6.3 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 8(a) and (b), respectively. For the output quality, we quantify it following the *accuracy* measurement in [19]. Specifically, we (1) randomly select 10000 utility vectors in $\mathcal{U}$, (2) for each utility vector selected, we check if it is a qualified utility vector by verifying if it is in the partitions returned by the exact algorithm *E-PT*, and (3) we report the *accuracy* of algorithm *A-PC* to be the percentage of qualified utility vectors that are also in the partitions returned by *A-PC*. As shown in Figure 8(a), 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. However, 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 larger execution time (Figure 8(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 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 $\epsilon$, where other parameters were set by default. In Figure 9(a), we varied parameter $k$ from 1 to 40. Our algorithms achieve significant improvements. They reduce execution time by up to 1-2 orders of magnitude compared to the existing ones. For instance, when $k = 10$, our algorithms take at most $10^{-3}$ seconds, while the best existing algorithm takes $10^{-2}$ seconds. When parameter $k$ increases, the execution times of all algorithms become larger. This is because the increasing $k$ relaxes the returned condition, leading to a larger number of

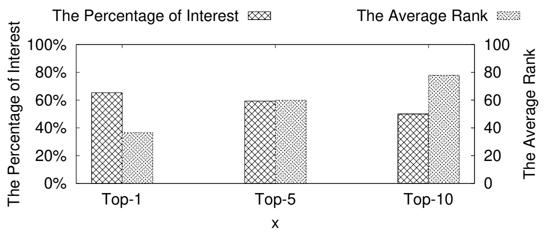
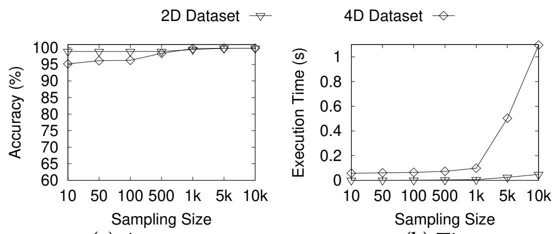
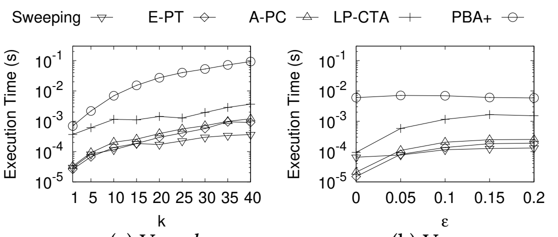


**Figure 7: User Study**



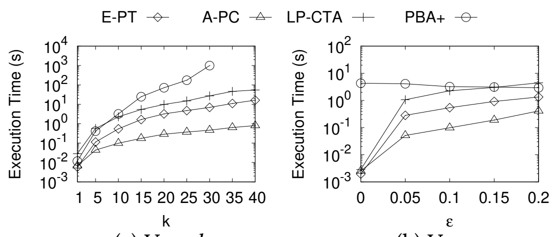
(a) Accuracy (b) Time

**Figure 8: Sampling Size**



(a) Vary $k$ (b) Vary $\epsilon$

**Figure 9: 2D**



(a) Vary $k$ (b) Vary $\epsilon$

**Figure 10: 4D**

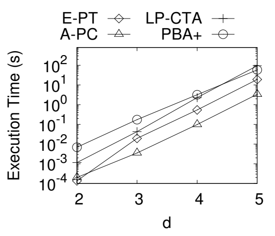


**Figure 11: Vary $d$**

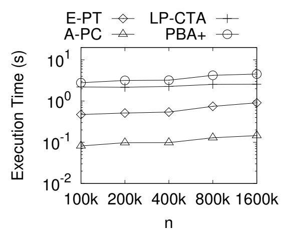


**Figure 12: Vary $n$**

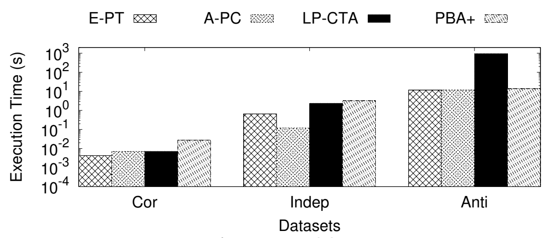


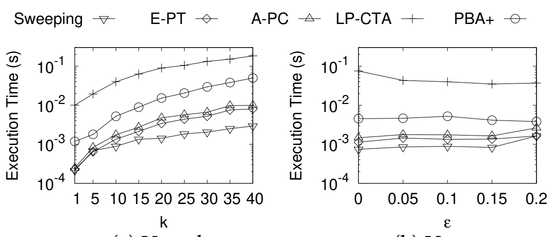
**Figure 13: Type**

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 9(b), we varied parameter $\epsilon$ 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 $\epsilon$ 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 are set by default. Since algorithm *Sweeping* is only designed for the two-dimensional special case, we exclude it in the experiment. Figure 10(a) shows the execution time of each algorithm when we increased $k$ from 1 to 40. As shown there, algorithm *PBA+* performs the worst. For example, when $k = 30$, our algorithms *E-PT* and *A-PC* take 6.9 seconds and 0.46 seconds, respectively, while algorithm *PBA+* spends 996.7 seconds. Note that we do not show the results of algorithm *PBA+* in Figure 10(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 hyper-planes 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 5.1.2), 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 10(b), we varied parameter $\epsilon$ 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 in the internal nodes of *E-PT*). Furthermore, all algorithms except *PBA+* experience a slowdown when $\epsilon$ increases since more partitions are qualified to be returned, leading to longer execution times. Note that algorithm *PBA+* becomes faster with larger $\epsilon$. 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 $\epsilon$ increases, more utility vectors become qualified, allowing some partitions to be directly returned without refinement, which reduces the overall cost.

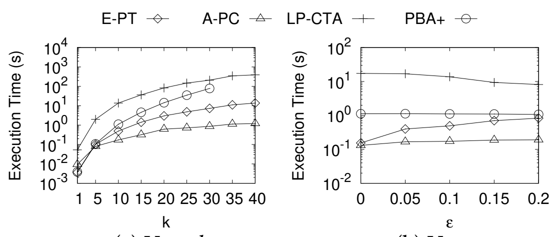
**Scalability.** We studied the scalability of algorithms by varying the dimension $d$, the dataset size $n$, and the type of datasets.

*Varying d.* In Figure 11, 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.



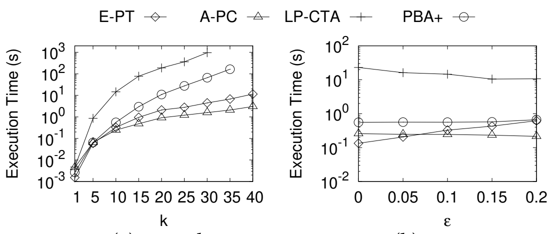
(a) Vary $k$ (b) Vary $\epsilon$

**Figure 14: Island**



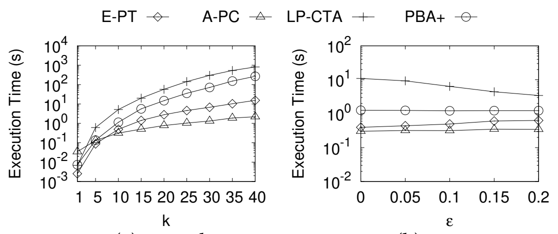
(a) Vary $k$ (b) Vary $\epsilon$

**Figure 15: Weather**



(a) Vary $k$ (b) Vary $\epsilon$

**Figure 16: Car**



(a) Vary $k$ (b) Vary $\epsilon$

**Figure 17: NBA**

*Varying n.* In Figure 12, 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 $n = 800,000$, while 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 13, 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. Algorithm *LP-CTA* is the slowest one and it takes 974.8 seconds to run. 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.

## 6.4 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 $\epsilon$. The results on datasets *Island*, *Weather*, *Car*, and *NBA* are shown in Figures 14, 15, 16, and 17, 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.

## 6.5 Summary

The experiments demonstrate that our formulation of problem RRQ, considering the product utilities, provides a better assessment of products than rankings. In our user study, the percentages of interest are at least 50%. Furthermore, our algorithms demonstrated the superiority over the best-known existing ones. (1) Our algorithms are efficient. Compared with the existing algorithms, we achieve significant improvements in the execution time. 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 than algorithm *E-PT* by providing an approximate solution, while algorithm *E-PT* can return an exact solution. (3) Our algorithms 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*, which return exact solutions and approximate solutions, respectively, solve the general case of RRQ most efficiently.

## 7 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 $(k, \epsilon)$-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 to find the solution. 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 are interested in applying the reverse regret query to dynamic datasets.

## REFERENCES

[1] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proceedings of the International Conference on Data Engineering*. 421–430.

[2] Wei Cao, Jian Li, Haitao Wang, Kangning Wang, Ruosong Wang, Raymond Chi-Wing Wong, and Wei Zhan. 2017. k-Regret Minimizing Set: Efficient Algorithms and Hardness. In *20th International Conference on Database Theory*. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 11:1–11:19.

[3] Yuan-Chi Chang, Lawrence Bergman, Vittorio Castelli, Chung-Sheng Li, Ming-Ling Lo, and John R Smith. 2000. The Onion Technique: Indexing for Linear Optimization Queries. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of data*. 391–402.

[4] Lei Chen and Xiang Lian. 2008. Efficient Processing of Metric Skyline Queries. *IEEE Transactions on Knowledge and Data Engineering* 21, 3 (2008), 351–365.

[5] Sean Chester, Alex Thomo, S. Venkatesh, and Sue Whitesides. 2014. Computing k-Regret Minimizing Sets. In *Proceedings of the VLDB Endowment*, Vol. 7. VLDB Endowment, 389–400.

[6] Mark De Berg, Otfried Cheong, Marc Van Kreveld, and Mark Overmars. 2008. *Computational geometry: Algorithms and applications*. Springer Berlin Heidelberg.

[7] Evangelos Dellis and Bernhard Seeger. 2007. Efficient Computation of Reverse Skyline Queries. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 291–302.

[8] James Dyer and Rakesh Sarin. 1979. Measurable Multiattribute Value Functions. *Operations Research* 27 (08 1979), 810–822.

[9] Yunjun Gao, Qing Liu, Baihua Zheng, Li Mou, Gang Chen, and Qing Li. 2015. On Processing Reverse k-Skyband and Ranked Reverse Skyline Queries. *Information Sciences* 293 (2015), 11–34.

[10] Parke Godfrey. 2004. Skyline Cardinality for Relational Processing: How Many Vectors are Maximal?. In *Foundations of Information and Knowledge Systems: Third International Symposium, FoIKS 2004 Wilheminenburg Castle, Austria, February 17-20, 2004 Proceedings 3*. Springer, 78–97.

[11] Zhenqiang Gong, Guang-Zhong Sun, Jing Yuan, and Yanjing Zhong. 2009. 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, 288–305.

[12] Antonin Guttman. 1984. R-Trees: A Dynamic Index Structure for Spatial Searching. In *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data* (Boston, Massachusetts) *(SIGMOD '84)*. Association for Computing Machinery, New York, NY, USA, 47–57.

[13] Ihab F Ilyas, George Beskales, and Mohamed A Soliman. 2008. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Computing Surveys (CSUR)* 40, 4 (2008), 1–58.

[14] Ralph Keeney, Howard Raiffa, and David Rajala. 1979. Decisions with Multiple Objectives: Preferences and Value Trade-Offs. *Systems, Man and Cybernetics, IEEE Transactions on* 9 (08 1979), 403 – 403.

[15] Mohamed E Khalefa, Mohamed F Mokbel, and Justin J Levandoski. 2008. Skyline Query Processing for Incomplete Data. In *2008 IEEE 24th International Conference on Data Engineering*. IEEE, 556–565.

[16] Donald Kossmann, Frank Ramsak, and Steffen Rost. 2002. Shooting Stars in the Sky: An Online Algorithm for Skyline Queries. In *Very Large Data Bases Conference*.

[17] Ken CK Lee, Wang-Chien Lee, Baihua Zheng, Huajing Li, and Yuan Tian. 2010. Z-SKY: An Efficient Skyline Query Processing Framework based on Z-order. *The VLDB Journal* 19 (2010), 333–362.

[18] Xiang Lian and Lei Chen. 2008. Monochromatic and Bichromatic Reverse Skyline Search over Uncertain Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. 213–226.

[19] Danupon Nanongkai, Ashwin Lall, Atish Das Sarma, and Kazuhisa Makino. 2012. Interactive Regret Minimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 109–120.

[20] Danupon Nanongkai, Atish Das Sarma, Ashwin Lall, Richard J. Lipton, and Jun Xu. 2010. Regret-Minimizing Representative Databases. In *Proceedings of the VLDB Endowment*, Vol. 3. VLDB Endowment, 1114–1124.

[21] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2003. An Optimal and Progressive Algorithm for Skyline Queries. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*. 467–478.

[22] Dimitris Papadias, Yufei Tao, Greg Fu, and Bernhard Seeger. 2005. Progressive Skyline Computation in Database Systems. *ACM Transactions on Database Systems* 30, 1 (2005), 41–82.

[23] Peng Peng and Raymond Chi-Wing Wong. 2014. Geometry Approach for k-Regret Query. In *Proceedings of the International Conference on Data Engineering*. 772–783.

[24] Peng Peng and Raymong Chi-Wing Wong. 2015. K-Hit Query: Top-k Query with Probabilistic Utility Function. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 577–592.

[25] Jeff M Phillips. 2012. Chernoff-hoeffding Inequality and Applications. *arXiv preprint arXiv:1209.6396* (2012).

[26] Li Qian, Jinyang Gao, and H. V. Jagadish. 2015. Learning User Preferences by Adaptive Pairwise Comparison. In *Proceedings of the VLDB Endowment*, Vol. 8. VLDB Endowment, 1322–1333.

[27] Timos K. Sellis, Nick Roussopoulos, and Christos Faloutsos. 1987. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *Proceedings of the 13th International Conference on Very Large Data Bases (VLDB '87)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 507–518.

[28] Mohamed A Soliman, Ihab F Ilyas, and Kevin Chen-Chuan Chang. 2006. Top-k Query Processing in Uncertain Databases. In *2007 IEEE 23rd International Conference on Data Engineering*. IEEE, 896–905.

[29] Kian-Lee Tan, Pin-Kwang Eng, Beng Chin Ooi, et al. 2001. Efficient Progressive Skyline Computation. In *VLDB*, Vol. 1. 301–310.

[30] Bo Tang, Kyriakos Mouratidis, and Man Lung Yiu. 2017. Determining the Impact Regions of Competing Options in Preference Space. In *Proceedings of the 2017 ACM International Conference on Management of Data*. 805–820.

[31] Yufei Tao, Xiaokui Xiao, and Jian Pei. 2006. Subsky: Efficient Computation of Skylines in Subspaces. In *22nd International Conference on Data Engineering (ICDE'06)*. IEEE, 65–65.

[32] Yufei Tao, Xiaokui Xiao, and Jian Pei. 2007. Efficient Skyline and Top-k Retrieval in Subspaces. *IEEE Transactions on Knowledge and Data Engineering* 19, 8 (2007), 1072–1088.

[33] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Nørvåg. 2010. Reverse Top-k Queries. In *2010 IEEE 26th International Conference on Data Engineering (ICDE 2010)*. IEEE, 365–376.

[34] Akrivi Vlachou, Christos Doulkeridis, Yannis Kotidis, and Kjetil Norvag. 2011. Monochromatic and Bichromatic Reverse Top-k Queries. *IEEE Transactions on Knowledge and Data Engineering* 23, 8 (2011), 1215–1229.

[35] Akrivi Vlachou, Christos Doulkeridis, Kjetil Nørvåg, and Yannis Kotidis. 2013. Branch-and-bound Algorithm for Reverse Top-k Queries. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 481–492.

[36] Weicheng Wang and Raymond Chi-Wing Wong. 2022. Interactive Mining with Ordered and Unordered Attributes. *Proceedings of the VLDB Endowment* 15, 11 (2022), 2504–2516.

[37] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. 2021. Interactive Search for One of the Top-k. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 13 pages.

[38] Weicheng Wang, Raymond Chi-Wing Wong, and Min Xie. 2023. Interactive Search with Mixed Attributes. In *2023 IEEE 39th International Conference on Data Engineering (ICDE)*. 2276–2288.

[39] Min Xie, Raymond Chi-Wing Wong, and Ashwin Lall. 2019. Strongly Truthful Interactive Regret Minimization. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 281–298.

[40] Min Xie, Raymond Chi-Wing Wong, and Ashwin Lall. 2020. An Experimental Survey of Regret Minimization Query and Variants: Bridging the Best Worlds between Top-k Query and Skyline Query. *VLDB Journal* 29, 1 (2020), 147–175.

[41] Min Xie, Raymond Chi-Wing Wong, Jian Li, Cheng Long, and Ashwin Lall. 2018. Efficient K-Regret Query Algorithm with Restriction-Free Bound for Any Dimensionality. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, USA, 959–974.

[42] Min Xie, Raymond Chi-Wing Wong, Peng Peng, and Vassilis J. Tsotras. 2020. Being Happy with the Least: Achieving $\alpha$-Happiness with Minimum Number of Tuples. In *Proceedings of the International Conference on Data Engineering*. 1009–1020.

[43] Guangyi Zhang, Nikolaj Tatti, and Aristides Gionis. 2023. Finding Favourite Tuples on Data Streams with Provably Few Comparisons. *arXiv preprint arXiv:2307.02946* (2023).

[44] Jiahao Zhang, Bo Tang, Man Lung Yiu, Xiao Yan, and Keming Li. 2022. T-LevelIndex: Towards Efficient Query Processing in Continuous Preference Space. In *Proceedings of the 2022 International Conference on Management of Data* (Philadelphia, PA, USA). Association for Computing Machinery, New York, NY, USA, 2149–2162.

## Appendix A PROOF

**Proof of Lemma 3.5.** **If.** Suppose that partition $c$ is covered by $t$ negative half-spaces and $n - t$ positive half-spaces, where $t < k$. Consider each utility vector $\boldsymbol{u} \in c$. Based on the definition of positive half-spaces, there are $n - t$ points $\boldsymbol{p} \in \mathcal{D}$ such that $\boldsymbol{u} \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}) > 0$, i.e., $(f_{\boldsymbol{u}}(\boldsymbol{p}) - f_{\boldsymbol{u}}(\boldsymbol{q}))/f_{\boldsymbol{u}}(\boldsymbol{p}) < \epsilon$. In this case, we have

$$\frac{(t+1)max_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p}) - f_{\boldsymbol{u}}(\boldsymbol{q})}{(t+1)max_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p})} < \epsilon$$

. Since $t < k$, $(t+1)max_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p}) \geq kmax_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p})$. Thus,

$$\frac{kmax_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p}) - f_{\boldsymbol{u}}(\boldsymbol{q})}{kmax_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p})} < \epsilon$$

. This means that point $\boldsymbol{q}$ is a $(k,\epsilon)$-regret point w.r.t. $\boldsymbol{u}$.

**Only if.** Suppose that point $\boldsymbol{q}$ is a $(k,\epsilon)$-regret point w.r.t. any utility vectors $\boldsymbol{u}$ in partition $c$. Consider each utility vector in $\boldsymbol{u} \in c$. Based on the definition of the $(k,\epsilon)$-regret point, we have

$$\frac{kmax_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p}) - f_{\boldsymbol{u}}(\boldsymbol{q})}{kmax_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p})} < \epsilon$$

. In this case, $\forall t' \geq k$, since $kmax_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p}) \geq t'max_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p})$, we have

$$\frac{t'max_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p}) - f_{\boldsymbol{u}}(\boldsymbol{q})}{t'max_{\boldsymbol{p}\in\mathcal{D}}f_{\boldsymbol{u}}(\boldsymbol{p})} < \epsilon$$

. This means that there are at least $n - k + 1$ points $\boldsymbol{p} \in \mathcal{D}$ such that $\boldsymbol{u} \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}) > 0$. Thus, partition $c$ must be covered by more than $n - k$ positive half-spaces. □

**Proof of Lemma 4.1.** Assume that there are $n_l$ inclusive hyper-planes. Denote the $i$-th ranked inclusive hyper-plane by $lh_i$, where $i \in [1, n_l]$. For any $j$-th ranked hyper-plane $lh_j$, where $j \in [1, k-1]$, we have $lh_k^- \subseteq lh_j^-$. This means any utility vectors in $lh_k^-$ must be in at least $k$ negative half-spaces ($lh_1^-$, $lh_2^-$, ..., $lh_k^-$). Thus, query point $\boldsymbol{q}$ cannot be a $(k,\epsilon)$-regret point w.r.t. any utility vectors in $lh_k^-$. □

**Proof of Lemma 4.2.** Denote the $i$-th ranked inclusive (resp. exclusive) hyper-plane by $lh_i$ (resp. $uh_i$), where $i \in [1,k]$. Since the partitions in halfspaces $lh_k^-$ and $uh_k^-$ are ignored, we only need to consider the hyper-planes that intersect $\mathcal{L}$ in $lh_k^+ \cap uh_k^+$. For all the inclusive hyper-planes, there are at most $k$ inclusive hyper-planes that intersect $\mathcal{L}$ in $lh_k^+$ (i.e., $lh_1, lh_2, ..., lh_k$). Similarly, for all the exclusive hyper-planes, there are at most $k$ exclusive hyper-planes that intersect $\mathcal{L}$ in $uh_k^+$ (i.e., $uh_1, uh_2, ..., uh_k$). Thus, there are at most $2k$ hyper-planes that divide $lh_k^+ \cap uh_k^+$ into $O(k)$ partitions. □

**Proof of Lemma 4.3.** For each partition, we add or subtract $Q$ by 1 to obtain the number of negative half-spaces that cover it. The time cost is $O(1)$. □

**Proof of Theorem 4.4.** We need $O(n)$ time to find the $k$-th ranked inclusive hyper-plane $lh_k$ and the $k$-th ranked exclusive hyper-plane $uh_k$. Then, we need $O(n)$ time to filter out the hyper-planes that intersect with $\mathcal{L}$ in $lh_k^-$ or $uh_k^-$. Based on the proof of Lemma 4.2, there are $O(k)$ hyper-planes left after the filtering. To rank the remaining hyper-planes, we need $O(k \log k)$ time. Finally, based on Lemma 4.3, the checking cost for each partition is $O(1)$. We need $O(k)$ time to check if $O(k)$ partitions are qualified to be returned. Therefore, our algorithm needs $O(n + k\log k)$ time in total. Since $k << n$, the time complexity is $O(n)$. □

**Proof of Lemma 5.1.** Since a partition $c$ is an intersection of several half-spaces, it is a convex polyhedron. One property of the convex polyhedron is that any point in the partition can be represented by the extreme points of the partition [6]. Specifically, for each point $\boldsymbol{r}$ in the partition, we have $\boldsymbol{r} = \sum_{i=1}^{x} t_i\boldsymbol{e}_i$, where $t_i$ is a positive real number such that $\sum_{i=1}^{x} t_i = 1$.

If $\forall i \in [1,x]$, $e_i \in h_{\boldsymbol{q},\boldsymbol{p}}^+$, i.e., $\boldsymbol{e}_i \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}) > 0$, for any point $\boldsymbol{r}$ in the partition, we have

$$\begin{aligned} \boldsymbol{r} \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}) &= (\sum_{i=1}^{x} t_i\boldsymbol{e}_i) \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p}) \\ &= \sum_{i=1}^{m} (t_i\boldsymbol{e}_i \cdot (\boldsymbol{q} - (1-\epsilon)\boldsymbol{p})) \\ &> 0 \end{aligned}$$

Thus, any point $\boldsymbol{r}$ in the partition must be in half-space $h_{\boldsymbol{q},\boldsymbol{p}}^+$. Similarly for the case where $\forall i \in [1,x]$, $e_i \in h_{\boldsymbol{q},\boldsymbol{p}}^-$.

If both $h_{\boldsymbol{q},\boldsymbol{p}}^+$ and $h_{\boldsymbol{q},\boldsymbol{p}}^-$ contain some extreme points, the partition must intersect the hyper-plane. □

**Proof of Lemma 5.2.** Suppose that $\forall i \in [1,d], v_{\boldsymbol{q},\boldsymbol{p}}[i] \geq v_{\boldsymbol{q},\boldsymbol{p}'}[i]$. For any utility vectors $\boldsymbol{u} \in h_{\boldsymbol{q},\boldsymbol{p}}^-$, i.e., $\boldsymbol{u} \cdot \boldsymbol{v}_{\boldsymbol{q},\boldsymbol{p}} < 0$, we have

$$\begin{aligned} \boldsymbol{u} \cdot \boldsymbol{v}_{\boldsymbol{q},\boldsymbol{p}'} &= \sum_{i=1}^{d} u[i] \cdot v_{\boldsymbol{q},\boldsymbol{p}'}[i] \\ &\leq \sum_{i=1}^{d} u[i] \cdot v_{\boldsymbol{q},\boldsymbol{p}}[i] \ (since\ u[i] \geq 0) \\ &= \boldsymbol{u} \cdot \boldsymbol{v}_{\boldsymbol{q},\boldsymbol{p}} \\ &< 0 \end{aligned}$$

Based on the derivation, for any utility vectors $\boldsymbol{u} \in h_{\boldsymbol{q},\boldsymbol{p}}^-$, we have $\boldsymbol{u} \in h_{\boldsymbol{q},\boldsymbol{p}'}^-$ and thus $h_{\boldsymbol{q},\boldsymbol{p}}^- \subseteq h_{\boldsymbol{q},\boldsymbol{p}'}^-$. □

**Proof of Lemma 5.3.** Since $N'$ is a child node of $N$, we have $c' \subseteq c$. If $c \subseteq h_{\boldsymbol{q},\boldsymbol{p}}^+$ (resp. $c \subseteq h_{\boldsymbol{q},\boldsymbol{p}}^-$), then $c' \subseteq c \subseteq h_{\boldsymbol{q},\boldsymbol{p}}^+$ (resp. $c' \subseteq c \subseteq h_{\boldsymbol{q},\boldsymbol{p}}^-$). □

**Proof of Lemma 5.4.** Suppose that a half-space (i.e., $h_{\boldsymbol{q},\boldsymbol{p}}^+$ or $h_{\boldsymbol{q},\boldsymbol{p}}^-$) covers the outer sphere of a partition. Since the outer sphere covers the partition, the half-space must cover the partition. □

**Proof of Lemma 5.5.** Suppose that hyper-plane $h_{\boldsymbol{q},\boldsymbol{p}}$ intersects the inner sphere of a partition. There exists two utility vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ in the inner sphere such that $\boldsymbol{u}_1 \in h_{\boldsymbol{q},\boldsymbol{p}}^+$ and $\boldsymbol{u}_2 \in h_{\boldsymbol{q},\boldsymbol{p}}^-$. Since the inner sphere is covered by the partition, $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ must be in the partition. Thus, there exist two utility vectors $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ in the partition such that $\boldsymbol{u}_1 \in h_{\boldsymbol{q},\boldsymbol{p}}^+$ and $\boldsymbol{u}_2 \in h_{\boldsymbol{q},\boldsymbol{p}}^-$. This indicates that the hyper-plane must intersect the partition. □

**Proof of Theorem 5.6.** Assuming independent and uniformly distributed data points, following [10], our hyper-planes reduction strategy can reduce the number of hyper-planes to $n' = O(k\frac{log^{d-1}n}{d!})$. Consider the insertion of the $i$-th hyper-plane. The majority computational cost of algorithm *E-PT* is the children creation for the leaves. By the zone theorem [6], the number of leaves that need to be split is $O(i^{d-2})$. Suppose that the cost of a single child creation is $O(\alpha)$. The total cost required by the insertion of the $i$-th hyper-plane is $O(\alpha \cdot i^{d-2})$. Thus, the overall time complexity is $\sum_{i=1}^{n'} O(\alpha \cdot i^{d-2}) = O(\alpha \cdot (n')^{d-1}) = O(\alpha \cdot (k\frac{log^{d-1}n}{d!})^{d-1})$. □

**Proof of Lemma 5.7.** Partition $c_{\boldsymbol{u}}$ is the intersection of a total of $n$ half-spaces. For each positive half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$, we have $(1 - \epsilon)f_{\boldsymbol{u}}(\boldsymbol{p}) < f_{\boldsymbol{u}}(\boldsymbol{q})$ for point $\boldsymbol{p}$, meaning that $\boldsymbol{u}$ must be in $h^+_{\boldsymbol{q},\boldsymbol{p}}$. For each negative half-space $h^-_{\boldsymbol{q},\boldsymbol{p}'}$, we have $(1 - \epsilon)f_{\boldsymbol{u}}(\boldsymbol{p}') > f_{\boldsymbol{u}}(\boldsymbol{q})$ for point $\boldsymbol{p}'$, meaning that $\boldsymbol{u}$ must be in $h^-_{\boldsymbol{q},\boldsymbol{p}'}$. Therefore, $\boldsymbol{u}$ must be in partition $c_{\boldsymbol{u}}$.

Since query point $\boldsymbol{q}$ is a $(k, \epsilon)$-regret point w.r.t. $\boldsymbol{u}$, when constructing partition $c_{\boldsymbol{u}}$, we only select fewer than $k$ points to build the negative half-spaces. As the number of negative half-spaces covering partition $c_{\boldsymbol{u}}$ is below $k$, query point $\boldsymbol{q}$ must be a $(k, \epsilon)$-regret point w.r.t. any utility vectors $\boldsymbol{u}' \in c_{\boldsymbol{u}}$. □

**Proof of Lemma 5.8.** Partitions $c_{\boldsymbol{u}}$ and $c$ are the intersections of $n$ half-spaces, i.e., they are convex polyhedrons [6]. One property is that if they are not the same, they must be constructed by at least one different half-space, and thus, they must be distinct partitions. Assume that partition $c_{\boldsymbol{u}}$ and partition $c$ are not the same. Based on Lemma 5.7, the sampled utility vector $\boldsymbol{u}$ must be in partition $c_{\boldsymbol{u}}$. Then, there is a contradiction since $\boldsymbol{u}$ cannot be in two distinct partitions simultaneously. □

**Proof of Lemma 5.9.** Assume that $\mathcal{D}^+_{\boldsymbol{u}_1} \subseteq \mathcal{D}^+_{\boldsymbol{u}_2}$. For each positive half-space $h^+_{\boldsymbol{q},\boldsymbol{p}}$, point $\boldsymbol{p}$ is from set $\mathcal{D}^+_{\boldsymbol{u}_1}$. We have $(1-\epsilon)f_{\boldsymbol{u}_1}(\boldsymbol{p}) < f_{\boldsymbol{u}_1}(\boldsymbol{q})$ and $(1 - \epsilon)f_{\boldsymbol{u}_2}(\boldsymbol{p}) < f_{\boldsymbol{u}_2}(\boldsymbol{q})$, meaning that both $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ must be in $h^+_{\boldsymbol{q},\boldsymbol{p}}$. For each negative half-space $h^-_{\boldsymbol{q},\boldsymbol{p}'}$, point $\boldsymbol{p}'$ is from set $\mathcal{D}^-_{\boldsymbol{u}_2}$. We have $(1 - \epsilon)f_{\boldsymbol{u}_1}(\boldsymbol{p}') > f_{\boldsymbol{u}_1}(\boldsymbol{q})$ and $(1 - \epsilon)f_{\boldsymbol{u}_2}(\boldsymbol{p}') > f_{\boldsymbol{u}_2}(\boldsymbol{q})$, meaning that $\boldsymbol{u}_1$ and $\boldsymbol{u}_2$ must be in $h^-_{\boldsymbol{q},\boldsymbol{p}'}$. Therefore, $\boldsymbol{u}$ must be in partition $c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$.

When constructing partition $c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$, we build positive half-spaces based on the point set $\mathcal{D}^+_{\boldsymbol{u}_1}$. Since query point $\boldsymbol{q}$ is a $(k, \epsilon)$-regret point w.r.t. $\boldsymbol{u}_1$, set $\mathcal{D}^+_{\boldsymbol{u}_1}$ contains more than $n - k$ points. Thus, there will be more than $n - k$ positive half-spaces used to construct partition $c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$. In other words, any utility vector in $c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$ must be in fewer than $k$ negative half-spaces. In this way, query point $\boldsymbol{q}$ must be a $(k, \epsilon)$-regret point w.r.t. any utility vectors in $c_{\boldsymbol{u}_1,\boldsymbol{u}_2}$. □

**Proof of Theorem 5.10.** Denote the set that contains all the sampled utility vectors in the $i$-th qualified partition by $S_i$. To guarantee that the $i$-th qualified partition $c_i$ is found, there should be at least one sampled utility vector in $c_i$, i.e., $|S_i| > 0$. Then, if we want to find all the qualified partitions, we require $|S_i| > 0$ for each qualified partition. Consider each qualified partition such that $V_i/\mathbb{V} > \rho$. If $|\frac{|S_i|}{N} - \frac{V_i}{\mathbb{V}}| \leq \rho$, then $|S_i| > 0$.

According to the well-known Chernoff-Hoerffding Inequality [25, 42], if sampling size $N = O((1/\tau^2)(d+\ln(1/\delta))$, we can achieve $|\frac{|S_i|}{N} - \frac{V_i}{\mathbb{V}}| \leq \tau$, with probability at least $1 - \delta$. Thus, when the sampling size $N = O((1/\rho^2)(d + \ln(1/\delta)))$, we can achieve $|\frac{|S_i|}{N} - \frac{V_i}{\mathbb{V}}| \leq \rho$ (which leads to $|S_i| > 0$) with probability at least $1 - \delta$. □