Interactive Top-k Query

Weicheng Wang, Min Xie, Raymond Chi-Wing Wong, and Victor Junqiu Wei

Abstract—Given a large database, it is challenging to assist users in finding tuples that align with their preferences. The traditional top-k query attempts to address this by retrieving the best k tuples w.r.t. the user's preference. However, in practice, it is difficult for a user to specify his/her preference explicitly. Motivated by this, we propose the interactive top-k query, which enhances the top-k query with *user interaction*. In the interactive top-k query, a user is engaged in a series of questions. Each question consists of two tuples and asks the user to indicate which one s/he prefers. Based on the user feedback, the user's preference is learned implicitly, and finally s of the top-k tuples w.r.t. the learned preference is returned, where $s \leq k$ is a user-defined integer. We present two algorithms, namely 2DSEG and 2DPI$^+$, in a 2-dimensional space, which conduct a segment-based and a partition-based binary search, respectively. In a d-dimensional space ($d \geq 2$), we develop a general algorithm UNIFY under a unified framework, which subsumes several existing algorithms as special cases. Experiments were conducted on synthetic and real datasets, demonstrating that our algorithms outperform the existing ones by asking fewer questions within less execution time.

Index Terms—query processing, top-k query, interactive query.

1 INTRODUCTION

ONE major task of database systems is to assist users in finding tuples that align with their preferences. This task appears in a variety of applications [2], [3], e.g., purchasing a car, renting an apartment, and finding a dating partner. To illustrate, consider the application of purchasing a car. Suppose that there is a car database and each car is described by some attributes, e.g., price, horsepower, and fuel efficiency. Alice may want to purchase a cheap car with a high horsepower. Based on her preference, the database system is tasked with searching for candidate cars for Alice.

Various queries were proposed for such scenario, known as *multi-criteria decision-making queries* [4]. They characterize the user preference by a *utility function*. Each tuple is then associated with a *utility* (i.e., a function score), reflecting how well the tuple aligns with the user's preference. Guided by the utilities, these queries return desired tuples as output.

There are two representative queries: the top-k query [5], [6] and the skyline query [7]. The top-k query returns k tuples with the highest utilities, namely the top-k tuples, w.r.t. the user's utility function. However, it relies on the assumption that users know their utility functions precisely, which is often unrealistic in practice [8], [9]. For example, Alice may have difficulty in specifying how much she cares about price or horsepower. If the utility function differs slightly, the output can vary a lot. The skyline query does not require users to explicitly state their utility functions. Instead, it considers all possible utility functions and returns tuples that have the highest utilities (i.e., top-1) w.r.t. at least a utility function. Unfortunately, the output size of the skyline query is uncontrollable, potentially overwhelming users with excessive results [8], [10]. Moreover, while users often have diverse individual preferences, the skyline query is limited to global preferences, treating all users uniformly. For instance, it might return the same cars to both the youth and the elderly, despite their different focuses.

Motivated by these limitations, we exploit *user interaction* to help users find tuples. Specifically, users are engaged in a series of questions. Each question consists of two tuples and asks the user to indicate which one s/he prefers. This kind of interaction naturally appears in daily life. For instance, a car seller may show Alice two cars and ask her which one she prefers. A real estate agent may show Alice two houses and ask her: which one suits your taste better? Based on the user feedback, the user's utility function is learned implicitly. When sufficient information is collected, some of the top-k tuples (i.e., tuples with the top-k highest utilities) w.r.t. the learned utility function can be identified and returned.

Formally, we extend the work in [1] and propose a novel problem called *Interactive Top-k Query (ITQ)*. Given two positive integers k and s, where $s \leq k$, we interact with a user by asking as few questions as possible, and return a set of s tuples, where each tuple is among the user's top-k tuples. Problem ITQ has two advantages. (1) Unlike the top-k query, it does not require a user to specify an explicit utility function. (2) Unlike the skyline query, it ensures a controllable output size (i.e., only s of the top-k tuples are returned).

One characteristic of problem ITQ is its trade-off between the output size $s \in [1, k]$ and the number of questions asked to the user. Intuitively, a large output size necessitates more questions since it requires learning the user utility function more precisely to identify sufficient top-k tuples. Our experiments in Section 6 also verify this: when s varies from 1 to 20, there would be a dozen more questions asked. Leveraging this characteristic, users can flexibly set s according to their specific needs. For example, suppose that Alice plans to rent an apartment for several months. Since this is a short-term need, it is not necessary for her to spend a lot of time cherry-picking. Alice can opt for a small s and answer a few questions. Conversely, consider another sce-

- *Weicheng Wang is with Hong Kong University of Science and Technology. E-mail: wwangby@connect.ust.hk*
- *Min Xie is with the Shenzhen Institute of Computing Sciences. E-mail: xiemin@sics.ac.cn*
- *Raymond Chi-Wing Wong is with Hong Kong University of Science and Technology. E-mail: raywong@cse.ust.hk*
- *Victor Junqiu Wei is with Hong Kong University of Science and Technology. E-mail: victorwei@ust.hk*

Corresponding Author: Min Xie

nario of buying a house. Since it is one of the big purchases in Alice's life, she does not mind answering more questions as long as there are enough candidate houses recommended for her to make the final decision. In this case, Alice can set a large s even if there are more questions to be answered.

To the best of our knowledge, we are the first to study problem ITQ. Some related studies [1], [8], [9], [11], [12] also involve user interaction, but they differ from ours. [1] proposes returning one of the top-k tuples. It limits the output size to one. As discussed above, our work is an extension of [1], which returns s of the top-k tuples ($s \geq 1$). [8], [9] attempt to find one tuple such that a criterion called the *regret ratio*, evaluating how "regretful" a user is when seeing the returned tuple instead of all tuples in the database, is minimized. Similarly, they also focus on returning just one tuple. [11] aims to rank *all* tuples (including the top-k tuples) and [12] focuses on approximating the user's utility function. As could be noticed, both [11] and [12] are not tailored to identifying some of the top-k tuples.

Unfortunately, the existing algorithms cannot be adapted to solve problem ITQ satisfactorily. They ask users a large number of questions, which is quite troublesome. For instance, in our experiments in Section 6, when $k \geq 50$, most adapted algorithms [8], [9], [11], [12] ask more than 40 questions, which is not acceptable in real-world scenarios [13].

Contributions. Our contributions are listed as follows.

- To the best of our knowledge, we are the first to propose the problem of returning s of the top-k tuples by interacting with the user. We prove a lower bound $\Omega(\log_2 \frac{n}{k-s+1})$ on the number of questions asked.
- We propose two algorithms 2DSEG and 2DPI$^+$ in a 2-dimensional space, based on the idea of "binary search". In particular, the latter asks $O(\log_2 \frac{nk}{(k-s+1)^2})$ questions to find the desired tuples, which is asymptotically optimal in terms of the number of questions asked given a fixed k.
- We develop a general algorithm UNIFY under a unified framework in a d-dimensional space, which subsumes existing algorithms as special cases. It not only performs well empirically, but also has provable (asymptotically optimal) guarantees on the number of questions asked given fixed d and k, under certain conditions.
- We conducted experiments to demonstrate the superiority of our algorithms. The results show that our algorithms are able to return s of the top-k tuples using nearly half the number of questions asked by existing algorithms.

This journal extension adds substantial new technical contributions over [1]: (1) We generalize problem IST (interactive search for one of the top-k) in [1] to problem ITQ (interactive top-k query) (Section 3); intuitively, IST [1] can be regarded as a special case of ITQ with $s = 1$. (2) We extend the 2-dimensional algorithm 2DPI in [1] to the generalized problem ITQ, with novel speedup techniques (Lemma 4, Section 4.2). (3) We propose a new segment-based binary search algorithm for 2-dimensional ITQ (Section 4.1). (4) We develop a general algorithm UNIFY under a unified framework in a d-dimensional space (Section 5); it subsumes the algorithms HDPI and RH in [1] as special cases. Under this framework, we design new strategies to decide (a) how to generate the set of candidate questions and (b) how to pick the next question from the candidates. (5) We extend all theoretical lower bounds and performance guarantee from problem IST (special case $s = 1$) to problem ITQ (general case $s \geq 1$); this extension is non-trivial (Theorems 1-5). (6) We discuss potential user errors when users answer questions and develop innovative techniques for them. (7) We conducted experiments to evaluate the newly proposed algorithms for the generalized problem ITQ (Section 6).

In the following, we discuss related work in Section 2 and formally define problem ITQ in Section 3. In Section 4, we propose the 2-dimensional algorithms 2DPI$^+$ and 2DSEQ. In Section 5, we develop the general algorithm UNIFY in a d-dimensional space. Experiments are presented in Section 6, and Section 7 concludes this paper.

2 Related Work

We classify existing queries based on whether interaction is involved, i.e., *non-interactive queries* and *interactive queries*.

In addition to the top-k query and the skyline query described in Section 1, there are two other widely studied non-interactive queries, namely the similarity query [14], [15] and the regret minimizing query [10], [16], [17]. The similarity query finds tuples that are close to a given query tuple w.r.t. a given distance function [18]. However, it relies on the assumption that the query tuple and the distance function are known in advance, which may not always hold in practice. The regret minimizing query [10], [16], [17] avoids this problem. It minimizes a criterion called *regret ratio* that evaluates how regretful a user is when s/he sees the returned tuples instead of the whole database. However, it is hard for the regret minimizing query to achieve a small output size and a small regret ratio simultaneously. When the output size is small, the regret ratio is typically large [9], [17], [19].

To overcome the deficiencies of non-interactive queries, some studies utilize user interaction. [20], [21] propose the interactive skyline query, which reduces the output size by interacting with the user. However, it only learns the user preference on attribute values (e.g., a user prefers value *red* to *yellow* in the color attribute). Even if the preference on all values is obtained, its output size can still be arbitrarily large [1], [10]. The interactive similarity query [18], [22], [23] enhances the similarity query, by learning the query tuple and distance function via user interaction. However, it requires a user to assign *relevance scores* to thousands of tuples during the interaction, which is too demanding in practice.

[8] proposes the interactive regret minimizing query, which returns a tuple such that the regret ratio is smaller than a given threshold. Intuitively, it asks a user questions. Each question consists of several tuples, and the user is asked to tell which one s/he prefers. However, it displays *fake tuples* during the interaction, which are artificially constructed (not selected from the database). This might produce unrealistic tuples (e.g., a car with 10 dollars and 50000 horsepower) and the user may be disappointed if the displayed tuples with which s/he is satisfied do not exist [9]. To overcome this, [9] proposes the strongly truthful interactive regret minimizing query, which utilizes *real tuples* (selected from the database). However, it requires asking many questions within a long execution time. For instance, as shown in Section 6, its proposed algorithms need twice

more questions and two orders of magnitude longer execution time than ours under typical settings.

[12] focuses on approximating the user's preference via user interaction. Since it does not specifically concentrate on identifying the top-k tuples, it may involve unnecessary questions that are less relevant to our problem ITQ. For example, assume that Alice prefers car $\boldsymbol{p}_1$ to both $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$. Her preference between $\boldsymbol{p}_2$ and $\boldsymbol{p}_3$ is less important for our goal of identifying the top-k tuples, but this additional comparison might be useful in [12].

In the field of machine learning, our problem is related to the problem of *learning to rank* [11], [24], [25], [26], which learns the ranking of tuples by interacting with the user. However, most of the existing algorithms [24], [25], [26] only consider the relations between tuples (where a relation means that a tuple is preferable to another tuple) and do not utilize their interrelation (e.g., attribute "price" has an interrelation showing that a tuple with \$200 may be better than a tuple with \$500 since \$200 is cheaper), and thus, require asking users many questions. [11] considers the interrelation between tuples to learn the ranking. However, it focuses on deriving the ranking of all tuples, which requires asking many questions that are less relevant to our problem ITQ due to a similar reason stated for [12].

Our work extends [1] by returning s of the top-k tuples via user interaction with real tuples. [1] studies the problem of finding *one* of the top-k tuples. However, users may want multiple tuples returned in some scenarios. Thus, we extend [1] from returning one tuple to returning s tuples, expecting the proposed algorithms can be applied in more scenarios. For example, when a tourist travels to a new city, he would expect to have several recommended attractions. In a job search scenario, a job seeker often prefers viewing multiple job opportunities that closely match their profile. Besides, our extension also avoids the weaknesses of existing studies: (1) We do not require a user to provide an exact utility function (required by the top-k query) or a distance function (required by the similarity query). (2) We return s of the top-k tuples (but the skyline query has an uncontrollable output size and [1] only returns one tuple) (3) We guarantee that the returned tuples must be among the top-k tuples (but the regret minimizing query does not have such an intuitive interpretation). (4) We use real tuples during the interaction (but [8] utilizes fake tuples). (5) We ask a few easy questions during the interaction, unlike those existing studies that require heavy user effort. Firstly, [11], [24], [25], [26] ask a lot of questions since they require to learn a full ranking. Secondly, [24], [25], [26] do not utilize the interrelation between tuples, and thus, involve unnecessary questions. Thirdly, [18], [22] ask users to assign relevance scores to tuples, which is hard for users to handle compared to selecting a tuple among two candidates in each question.

3 Problem Definition & Characters

The input of our problem is a set $\mathcal{D}$ of n tuples described by d attributes. We regard each tuple as a d-dimensional point $\boldsymbol{p} = (p[1], p[2], ..., p[d])$. The i-th dimension $p[i]$ of the point is the i-th attribute of the tuple, where $i \in [1, d]$. In the rest of the paper, we use the words "tuple/point" and "attribute/dimension" interchangeably. Without loss of generality, we assume that each dimension is normalized to $(0, 1]$ and a larger value is preferred [9], [27]. Table 1 is an example. It contains five points in a 2-dimensional space, where each dimension is normalized.

Following [1], [9], [28], we model the user's preference by a linear function called *utility function*. The linear function is one of the most proliferate and effective representations since the inception of utility modeling [29], [30]. As verified in [12], it can effectively capture how real users assess the suitability of multi-attribute tuples. Formally, a linear function is defined as follows, which is a mapping $\mathbb{R}_+^d \rightarrow \mathbb{R}_+$.

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

- $\boldsymbol{u} = (u[1], u[2], ..., u[d])$ can be seen as a d-dimensional non-negative vector, called *utility vector*. Each $u[i]$ represents the importance of the i-th attribute to the user, where $i \in [1, d]$. A larger $u[i]$ means that the attribute is more important. The domain of $\boldsymbol{u}$ is called the *utility space* and denoted by $\mathcal{U}$. Without loss of generality, following [1], [9], we assume that $\sum_{i=1}^{d} u[i] = 1$ for the ease of illustration.
- $f_{\boldsymbol{u}}(\boldsymbol{p})$ is called the *utility* of $\boldsymbol{p}$ w.r.t. $\boldsymbol{u}$. It represents to what extent a user prefers $\boldsymbol{p}$, where a higher utility means that $\boldsymbol{p}$ is more preferred. Given the user's utility vector $\boldsymbol{u}$, the utility of each point in $\mathcal{D}$ can be computed, and the k points that have the largest utilities are the top-k points.

Consider Table 1. Let $\boldsymbol{u} = (0.4, 0.6)$. The utility function is expressed as $f_{\boldsymbol{u}}(\boldsymbol{p}) = 0.4p[1] + 0.6p[2]$. The utility of $\boldsymbol{p}_3$ w.r.t. $\boldsymbol{u}$ is $f_{\boldsymbol{u}}(\boldsymbol{p}_3) = 0.4 \times 0.5 + 0.6 \times 0.8 = 0.68$. The utilities of the other points can be computed similarly. Points $\boldsymbol{p}_1$ and $\boldsymbol{p}_3$ with the highest utilities are the top-2 points.

3.1 Problem ITQ

Given a point set $\mathcal{D}$, our goal is to return s of the top-k points with the help of user interaction, where $s \leq k$. Intuitively, we follow the interactive framework in [1], [9] to interact with a user for rounds. In each round, there are three steps.

- *Point Selection.* We adaptively select two points as a question and ask the user to pick the one s/he prefers.
- *Information Maintenance.* Based on the user feedback, we learn the user's utility function implicitly and update the information maintained for finding the top-k points.
- *Stopping Condition.* We check if the stopping condition is satisfied. If yes, we stop the interaction and return the result. Otherwise, we start another interactive round.

Formally, we are interested in the following problem.

Problem 1. **(Interactive Top-k Query (ITQ))** Given a point set $\mathcal{D}$, integers k and s ($s \leq k$), we ask a user as few questions as possible to find s of the user's top-k points.

As a generalization of problem IST presented in [1] that returns one of the top-k points, our problem ITQ proposed in this journal extension returns s of the top-k points. The problem in [1] can be regarded as a special case of problem ITQ when $s = 1$. The following theorem gives a non-trivial extension on the lower bound from its counterpart in [1]. Due to the lack of space, the proofs of lemmas/theorems can be found in the appendix.

Theorem 1. There exists a point set $\mathcal{D}$ of size n such that any algorithm needs to ask a user $\Omega(\log_2 \frac{n}{k-s+1})$ questions to find s of the user's top-k points.

p	$p[1]$	$p[2]$	$f_{\boldsymbol{u}}(p)$
$\boldsymbol{p}_1$	0	1.0	0.60
$\boldsymbol{p}_2$	0.3	0.7	0.54
$\boldsymbol{p}_3$	0.5	0.8	0.68
$\boldsymbol{p}_4$	0.7	0.4	0.52
$\boldsymbol{p}_5$	1.0	0	0.40

TABLE 1: Database ($\boldsymbol{u} = (0.4, 0.6)$)

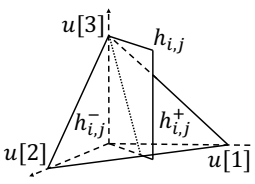


Fig. 1: Hyper-plane

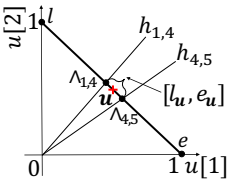


Fig. 2: 2DSeg

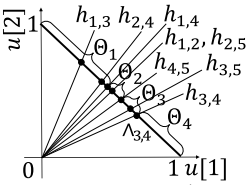


Fig. 3: 2DPI$^+$

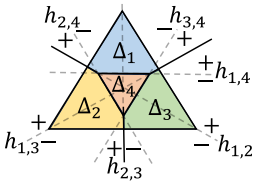


Fig. 4: 3D Cells

3.2 Geometric Foundation

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

For each pair of points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$, we can build a *hyper-plane* [31] in a d-dimensional $\mathbb{R}^d$:

$$h_{i,j} = \{\boldsymbol{r} \in \mathbb{R}^d \mid \boldsymbol{r} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) = 0\}$$

It passes through the origin with its unit normal in the same direction as vector $\boldsymbol{p}_i - \boldsymbol{p}_j$. Hyper-plane $h_{i,j}$ divides space $\mathbb{R}^d$ into two halves, called *half-spaces* [31]. The half-space above (resp. below) hyper-plane $h_{i,j}$, denoted by $h^+_{i,j}$ (resp. $h^-_{i,j}$), contains all the utility vectors $\boldsymbol{u} \in \mathbb{R}^d$ such that $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$ (resp. $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) < 0$). This said, if the user's utility vector $\boldsymbol{u}$ is in $h^+_{i,j}$ (resp. $h^-_{i,j}$), the utility of $\boldsymbol{p}_i$ must be higher (resp. lower) than that of $\boldsymbol{p}_j$ w.r.t. the user's utility vector [9]. Figure 1 shows an example of a hyper-plane.

Combining these definitions, the lemma below shows our foundation for learning the user's utility vector.

Lemma 1. Given $\mathcal{U}$ and two points $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ presented as a question to a user, if and only if the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, the user's utility vector is in $h^+_{i,j} \cap \mathcal{U}$.

Based on Lemma 1, our idea is to interact with a user to progressively narrow down the utility space. Let us refer to the narrowed utility space as the *utility range* and denote it by $\mathcal{R}$. The utility range is the intersection of a set of half-spaces and utility space $\mathcal{U}$, i.e., it is a polyhedron [31] that contains the user's utility vector. As the interaction proceeds, $\mathcal{R}$ gradually becomes smaller, ultimately becoming sufficiently small to determine the user's top-k points.

Throughout this process, we often need to handle a polyhedron $\mathcal{P}$ (since $\mathcal{R}$ is a polyhedron) and a hyper-plane $h_{i,j}$. There are three kinds of relationships between $\mathcal{P}$ and $h_{i,j}$.

(1) $\mathcal{P}$ is in $h^+_{i,j}$ (i.e., $\mathcal{P} \subseteq h^+_{i,j}$).
(2) $\mathcal{P}$ is in $h^-_{i,j}$ (i.e., $\mathcal{P} \subseteq h^-_{i,j}$).
(3) $\mathcal{P}$ intersects $h_{i,j}$ (i.e., $\mathcal{P} \cap h^+_{i,j} \neq \emptyset$ and $\mathcal{P} \cap h^-_{i,j} \neq \emptyset$).

If $h_{i,j}$ intersects $\mathcal{U}$, we denote the intersection by $\wedge_{i,j}$. To determine the relationship between $h_{i,j}$ and $\mathcal{P}$, it suffices to examine the set $\mathcal{V}$ of *vertices* of $\mathcal{P}$ [31]. Here, a point is said to be a *vertex* of $\mathcal{P}$ if it is a corner point of $\mathcal{P}$. For example, in Figure 1, points $(0, 0, 1)$, $(0, 0, 1)$, and $(0, 0, 1)$ are the vertices of the utility space.

Lemma 2. If $\forall \boldsymbol{v} \in \mathcal{V}$, $\boldsymbol{v} \in h^+_{i,j}$ (resp. $\boldsymbol{v} \in h^-_{i,j}$), then $\mathcal{P}$ must lie in $h^+_{i,j}$ (resp. $h^-_{i,j}$). Otherwise, $\mathcal{P}$ intersects with $h_{i,j}$.

One can also speed up the relationship checking using *bounding balls* and *bounding rectangles* (see details in [1]).

4 Two Dimensional Algorithm

In this section, we propose two algorithms 2DSEG and 2DPI$^+$ for 2-dimensional ITQ, which conduct a segment-based and a partition-based binary search, respectively.

Recall that in a 2-dimensional space $\mathbb{R}^2$, we assume that $u[1] + u[2] = 1$ for the utility vector. Thus, the utility space $\mathcal{U}$ is a line segment. With a slight abuse of notation, let us denote a line segment by $\mathcal{L} = [l, e]$, where l and e are two endpoints of the segment, e.g., $\mathcal{U} = [(0, 1), (1, 0)]$ in Figure 2. The length of $\mathcal{L}$ is the Euclidean distance between l and e.

Consider any two points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$ and suppose that hyper-plane $h_{i,j}$ (which is a line in space $\mathbb{R}^2$) intersects $\mathcal{U}$ at intersection $\wedge_{i,j}$. For any $\boldsymbol{u} \in h^+_{i,j} \cap \mathcal{U}$, $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$; for any $\boldsymbol{u} \in h^-_{i,j} \cap \mathcal{U}$, $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) < 0$. For example, $h_{1,4}$ intersects $\mathcal{U}$ at intersection $\wedge_{1,4}$ in Figure 2. We have $\forall \boldsymbol{u} \in [(0, 1), \wedge_{1,4}]$, $f_{\boldsymbol{u}}(\boldsymbol{p}_1) > f_{\boldsymbol{u}}(\boldsymbol{p}_4)$ and $\forall \boldsymbol{u} \in [\wedge_{1,4}, (1, 0)]$, $f_{\boldsymbol{u}}(\boldsymbol{p}_1) < f_{\boldsymbol{u}}(\boldsymbol{p}_4)$.

Given a point set $\mathcal{D}$, we can build a hyper-plane based on each pair of points. Some of these hyper-planes can be used to divide utility space $\mathcal{U}$ into a number of small sub-segments, called *cells*, such that (1) each cell is bounded by two intersections and (2) the top-k points w.r.t. any utility vectors in the same cell are identical. Our idea is to identify the cell containing the user's utility vector so that the output can be directly derived from the corresponding top-k points. Below, we first propose algorithm 2DSEG that identifies the target cell without computing all cells in $\mathcal{U}$. Then, we present algorithm 2DPI$^+$ that processes multiple cells together as a partition. Both algorithms conduct a binary search and have theoretical bounds on the number of questions asked.

4.1 Segment-based Binary Search

Algorithm 2DSEG maintains a utility range that contains the user's utility vector (as discussed in Section 3.2). In space $\mathbb{R}^2$, the utility range is a line segment $\mathcal{L} = [l, e]$. Initially, $\mathcal{L}$ is the entire utility space, i.e., $\mathcal{L} = [(0, 1), (1, 0)]$. Then, the algorithm iteratively picks the middle utility vector $\boldsymbol{u}$ in $\mathcal{L}$ (i.e., $\boldsymbol{u} = \frac{l+e}{2}$) and computes the cell containing $\boldsymbol{u}$, denoted by $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ (the cell construction will be shown later). Based on $l_{\boldsymbol{u}}$ and $e_{\boldsymbol{u}}$, $\mathcal{L}$ is divided into three parts: (a) $[l, l_{\boldsymbol{u}}]$, (b) $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$, and (c) $[e_{\boldsymbol{u}}, e]$. The algorithm then follows the interactive framework (Section 3.1) to interact with the user, learning which part contains the user's utility vector (details of the user interaction will be shown later).

(1) Assume that part (b), i.e., cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$, contains the user's utility vector. Then, any s of the cell's corresponding top-k points can be selected and returned.
(2) Assume that part (a) (resp. part (c)) contains the user's utility vector. $\mathcal{L}$ is set to $\mathcal{L} = [l, l_{\boldsymbol{u}}]$ (resp. $\mathcal{L} = [e_{\boldsymbol{u}}, e]$) and the process continues. In this case, the length of $\mathcal{L}$ is at least reduced by half, since the algorithm picks the middle utility vector to construct cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$.

Cell Construction. Given a utility vector $\boldsymbol{u}$, our goal is to construct a cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ in $\mathcal{L}$ such that (a) $\boldsymbol{u}$ is in $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ and (b) the top-k points w.r.t. any utility vector in the cell are the same. Denote the set of top-k points w.r.t. $\boldsymbol{u}$ by $\mathcal{S}$. Then given a point $\boldsymbol{p}_i \in \mathcal{S}$ and a point $\boldsymbol{p}_j \in \mathcal{D} \setminus \mathcal{S}$, the utility of $\boldsymbol{p}_i$ must be higher than that of $\boldsymbol{p}_j$ w.r.t. *any* utility vector $\boldsymbol{u}'$ in the cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$. In other words, any utility vector in the cell must be in $h^+_{i,j}$, i.e., $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}] \subseteq h^+_{i,j}$. In light of this, the cell can be computed by $\mathcal{L} \cap \bigcap_{\boldsymbol{p}_i \in \mathcal{S} \text{ and } \boldsymbol{p}_j \in \mathcal{D} \setminus \mathcal{S}} h^+_{i,j}$, whose endpoints are the intersections of $\mathcal{L}$ and some hyper-planes.

Lemma 3. Given $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ above, we can conclude that $\boldsymbol{u}$ must lie in $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ and for any utility vector $\boldsymbol{u}'$ in $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$, the top-$k$ points w.r.t. $\boldsymbol{u}'$ are the same as those w.r.t. $\boldsymbol{u}$.

Consider Table 1 and let $k = 2$. Since $\mathcal{L} = [(0,1),(1,0)]$, we have $\boldsymbol{u} = (\frac{1}{2}, \frac{1}{2})$ and the top-2 points w.r.t. $\boldsymbol{u}$ are $\boldsymbol{p}_3$ and $\boldsymbol{p}_4$. The cell containing $\boldsymbol{u}$ is $\mathcal{L} \cap \bigcap_{\boldsymbol{p}_i, \boldsymbol{p}_j} h^+_{i,j}$, where $\boldsymbol{p}_i \in \{\boldsymbol{p}_3, \boldsymbol{p}_4\}$ and $\boldsymbol{p}_j \in \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_5\}$, i.e., $[\wedge_{1,4}, \wedge_{4,5}]$ (Figure 2).

User Interaction. Assume that the endpoint $l_{\boldsymbol{u}}$ (resp. $e_{\boldsymbol{u}}$) of cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ is the intersection $\wedge_{i,j}$ (resp. $\wedge_{i',j'}$) of $\mathcal{L} = [l, e]$ and hyper-plane $h_{i,j}$ (resp. $h_{i',j'}$). We first present the user with points $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ (point selection). If the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, the user's utility vector must be in $[l, \wedge_{i,j}]$, and thus, we set $\mathcal{L} = [l, \wedge_{i,j}]$ (information maintenance). Suppose that the user prefers $\boldsymbol{p}_j$ to $\boldsymbol{p}_i$. We continue to present the user with points $\boldsymbol{p}_{i'}$ and $\boldsymbol{p}_{j'}$ (point selection). If the user prefers $\boldsymbol{p}_{j'}$ to $\boldsymbol{p}_{i'}$, the user's utility vector must be in $[\wedge_{i',j'}, e]$, and thus, we set $\mathcal{L} = [\wedge_{i',j'}, e]$ (information maintenance). Otherwise, the user's utility vector must be in $[\wedge_{i,j}, \wedge_{i',j'}]$. Since $[\wedge_{i,j}, \wedge_{i',j'}]$ is a cell, the top-k points w.r.t. any utility vector in it are the same. We return any s of the cell's corresponding top-k points (stopping condition).

Consider the cell $[\wedge_{1,4}, \wedge_{4,5}]$ in Figure 2. We present the user with $\boldsymbol{p}_1$ and $\boldsymbol{p}_4$, followed by the presentation of $\boldsymbol{p}_4$ and $\boldsymbol{p}_5$. Assume that the user prefers $\boldsymbol{p}_4$ to both $\boldsymbol{p}_1$ and $\boldsymbol{p}_5$. the user's utility vector must be in $[\wedge_{1,4}, \wedge_{4,5}]$. We can return any s of its top-2 points (i.e., $\{\boldsymbol{p}_3, \boldsymbol{p}_4\}$) to the user.

Analysis. The pseudocode is shown in Algorithm 1. Initially, $\mathcal{L}$ is the entire utility space $\mathcal{U}$. Every time we construct a cell based on the middle vector in $\mathcal{L}$, and update $\mathcal{L}$ by asking at most two questions. In this case, either the length of $\mathcal{L}$ will be reduced at least by half, or the target cell will be found. Denote by L_{min} the minimum length of the cell containing the user's utility vector. We have the following theorem.

Theorem 2. Algorithm 2DSEG solves the 2-dimensional ITQ by interacting with a user for $O(\log_2 \frac{\sqrt{2}}{L_{min}})$ rounds.

4.2 Partition-based Binary Search

Algorithm 2DSEG regards each *individual* cell as a basic unit. However, we observe that it is possible to regard *multiple* cells together as a basic unit to reduce computation, if they share at least s common top-k points. This observation leads us to develop algorithm 2DPI$^+$ that combines multiple cells into a *partition* (a longer line segment) for processing.

Intuitively, the algorithm consists of two parts: partition construction and user interaction. It first divides the utility space $\mathcal{U}$ into a number of disjoint partitions $\boldsymbol{\Theta} = \{\Theta_1, \Theta_2, ..., \Theta_m\}$. Each partition Θ corresponds to a set S of s points that are among the top-k points w.r.t. any utility vector $\boldsymbol{u} \in \Theta$. Then, it follows the interactive framework (Section 3.1) to interact with the user, adopts binary search on partitions to learn which partition contains the user's utility vector and finally, returns the corresponding points from that partition.

Algorithm 1: Algorithm 2DSEG

```
Input: A point set 𝒟, integers s and k
Output: A set of s of the user's top-k points
ℒ ← [l, e] where l ← (0,1) and e ← (1,0);
while ℒ is not a cell do
    u ← (l+e)/2;
    [l_u, e_u] ← the cell [∧_{i,j}, ∧_{i',j'}] containing u;
    Display p_i and p_j to a user;
    if the user prefers p_i to p_j then
        ℒ ← [l, ∧_{i,j}];
    else
        Display p_{i'} and p_{j'} to a user;
        if the user prefers p_{i'} to p_{j'} then
            ℒ ← [∧_{i',j'}, e];
        else
            ℒ ← [∧_{i,j}, ∧_{i',j'}];
return Any s of the top-k points corresponding to ℒ
```

Partition Construction. To construct partitions, we sweep through the utility space $\mathcal{U}$ from $(0,1)$ to $(1,0)$ with two data structures. Let $\boldsymbol{u}_c$ denote the utility vector currently being swept in $\mathcal{U}$. The first data structure is a queue $\mathcal{Q}$ that ranks the points in $\mathcal{D}$ in descending order according to their utilities w.r.t. $\boldsymbol{u}_c$. The second data structure is a min-heap $\mathcal{H}$ that records the *valid* intersections $\wedge_{i,j}$ formed by all pairs of neighbouring points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{Q}$, where an intersection is valid if $\wedge_{i,j} \in [\boldsymbol{u}_c, (1,0)]$. Note that the ranks of $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ only change once when $\boldsymbol{u}_c$ reaches $\wedge_{i,j}$, i.e., $\forall \boldsymbol{u} \in [(0,1), \wedge_{i,j}]$, $f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j)$ and $\forall \boldsymbol{u} \in [\wedge_{i,j}, (1,0)]$, $f_{\boldsymbol{u}}(\boldsymbol{p}_i) < f_{\boldsymbol{u}}(\boldsymbol{p}_j)$. The key of $\mathcal{H}$ is the distance from $\wedge_{i,j}$ to $\boldsymbol{u}_c$.

At the beginning of sweeping, $\boldsymbol{u}_c = (0,1)$. $\mathcal{Q}$ and $\mathcal{H}$ are initialized according to $\boldsymbol{u}_c$. The first partition Θ_1 starts at $(0,1)$ and its corresponding set S_1 is initialized with the first k points in $\mathcal{Q}$. During the sweeping, we continually pop out intersections from $\mathcal{H}$ until $\mathcal{H}$ is empty. Based on each intersection popped out, we update $\mathcal{Q}$ and $\mathcal{H}$, and build the partitions and their corresponding sets accordingly.

Specifically, suppose that an intersection $\wedge_{i,j}$ is popped out from $\mathcal{H}$ and partition Θ_x is being built. We set $\boldsymbol{u}_c = \wedge_{i,j}$ and swap $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ in $\mathcal{Q}$. Then, we insert into $\mathcal{H}$ two new intersections (if valid): (1) the intersection formed by $\boldsymbol{p}_i$ and its new neighbour in $\mathcal{Q}$ and (2) the one formed by $\boldsymbol{p}_j$ and its new neighbour in $\mathcal{Q}$. If $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ are the $(k+1)$-th and k-th points in $\mathcal{Q}$ after swapping, we perform two more steps. Firstly, we update $\mathcal{Q}$ to speed up subsequent processing.

Lemma 4. If it is the k-th time for $\boldsymbol{p}_i$ to lower its rank in $\mathcal{Q}$, we can remove $\boldsymbol{p}_i$ from $\mathcal{Q}$ in the subsequent sweeping.

Note that if $\boldsymbol{p}_i$ is removed from $\mathcal{Q}$, the intersection formed by $\boldsymbol{p}_i$ in $\mathcal{Q}$ is also deleted. Secondly, we delete $\boldsymbol{p}_i$ from S_x, since $\boldsymbol{p}_i$ is no longer a top-k point w.r.t. the currently swept vector $\boldsymbol{u}_c$. If $|S_x| = s - 1$, we end partition Θ_x at $\boldsymbol{u}_c$ and set $S_x = S_x \cup \{\boldsymbol{p}_i\}$. The construction of next partition Θ_{x+1} starts at $\boldsymbol{u}_c$ and the first k points in $\mathcal{Q}$ are added to S_{x+1}.

Consider the points in Table 1. Let $k = 2$ and $s = 2$. Initially, $\boldsymbol{u}_c = (0,1)$. We set $\mathcal{Q} =< \boldsymbol{p}_1, \boldsymbol{p}_3, \boldsymbol{p}_2, \boldsymbol{p}_4, \boldsymbol{p}_5 >$ based

Algorithm 2: Algorithm 2DPI$^+$

Input: A point set $\mathcal{D}$, integers s and k
Output: A set of s of the user's top-k points
$\boldsymbol{u}_c = (0, 1)$ and $x \leftarrow 1$;
Initialize $\mathcal{Q}$ and $\mathcal{H}$ based on $\boldsymbol{u}_c$, $\boldsymbol{\Theta} \leftarrow \emptyset$;
Start Θ_1 at $(0, 1)$, $S_1 \leftarrow$ the first k points in $\mathcal{Q}$;
while $|\mathcal{H}| > 0$ **do**
  Pop out $\wedge_{i,j}$ from $\mathcal{H}$ and set $\boldsymbol{u}_c = \wedge_{i,j}$;
  Swap the positions of $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ in $\mathcal{Q}$;
  if $\boldsymbol{p}_i$ *is the* $(k+1)$*-th point in* $\mathcal{Q}$ *after swapping* **then**
    Delete $\boldsymbol{p}_i$ from S_x;
    if $|S_x| = s - 1$ **then**
      End Θ_x at $\boldsymbol{u}_c$, $S_x \leftarrow S_x \cup \{\boldsymbol{p}_i\}$;
      $\boldsymbol{\Theta} \leftarrow \boldsymbol{\Theta} \cup \{\Theta_x\}$, $x \leftarrow x + 1$;
      Start Θ_x at $\boldsymbol{u}_c$, $S_x \leftarrow$ the first k points in $\mathcal{Q}$;
  if *it is the* k*-th time for* $\boldsymbol{p}_i$ *to lower its rank in* $\mathcal{Q}$ **then**
    Remove $\boldsymbol{p}_i$ from $\mathcal{Q}$; // Speedup by Lemma 4
  Insert new valid intersections into $\mathcal{H}$;
$left \leftarrow 1, right \leftarrow |\boldsymbol{\Theta}|$;
while $|\boldsymbol{\Theta}| > 1$ **do**
  $x \leftarrow left - 1 + \lfloor \frac{right - left + 1}{2} \rfloor$;
  Find $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$, where intersection $\wedge_{i,j}$ is at e_x;
  Display $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ to a user;
  if *the user prefers* $\boldsymbol{p}_i$ *to* $\boldsymbol{p}_j$ **then**
    $right \leftarrow x$;
  else
    $left \leftarrow x + 1$;
  $\boldsymbol{\Theta} \leftarrow < \Theta_{left}, ..., \Theta_{right} >$
return The point set S of the only partition Θ in $\boldsymbol{\Theta}$

on the utilities of points w.r.t. $\boldsymbol{u}_c$, and $\mathcal{H} = \{\wedge_{1,3}, \wedge_{2,4}, \wedge_{4,5}\}$ where $\wedge_{3,2}$ is not included since it is not valid. As shown in Figure 3, the first partition Θ_1 begins with $(0, 1)$ and $S_1 = \{\boldsymbol{p}_1, \boldsymbol{p}_3\}$. We pop out intersection $\wedge_{1,3}$ from $\mathcal{H}$ and update $\mathcal{Q}$ by swapping the positions of $\boldsymbol{p}_1$ and $\boldsymbol{p}_3$. Then, we insert a new intersection $\wedge_{1,2}$ (formed by $\boldsymbol{p}_1$ with its new neighbor $\boldsymbol{p}_2$) into $\mathcal{H}$. Since $\boldsymbol{p}_1$ and $\boldsymbol{p}_3$ are not the third and second points in $\mathcal{Q}$ after swapping, we pop out another intersection from $\mathcal{H}$ to proceed. The final result (i.e., four partitions $\Theta_1, \Theta_2, \Theta_3, \Theta_4$) is shown in Figure 3.

Lemma 5. Algorithm 2DPI$^+$ divides the utility space $\mathcal{U}$ into the fewest partitions in $O(nk \log n + \frac{nk^2}{(k-s+1)^2})$ time.

User Interaction. Suppose that the partitions $\boldsymbol{\Theta} = \{\Theta_1, \Theta_2, ..., \Theta_m\}$ and their corresponding set $\mathbf{S} = \{S_1, S_2, ..., S_m\}$ are obtained. Our algorithm locates the partition that contains the user's utility vector through a *binary search* in $\boldsymbol{\Theta}$. Specifically, in each interactive round, it finds the median partition $\Theta_x = [l_x, e_x]$ in $\boldsymbol{\Theta}$, where l_x and e_x are the two endpoints of Θ_x. We present a user with points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$, where intersection $\wedge_{i,j}$ is at e_x (point selection). If the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, based on Lemma 1, the user's utility vector must be in $h^+_{i,j}$, and thus, we can delete all the partitions from $\boldsymbol{\Theta}$ that are in $h^-_{i,j}$. Since Θ_x is a median partition, half of the partitions in $\boldsymbol{\Theta}$ are deleted. Similarly, if the user prefers $\boldsymbol{p}_j$ to $\boldsymbol{p}_i$, the other half of partitions are deleted from $\boldsymbol{\Theta}$ (information maintenance). The interaction stops when there is only one partition left in $\boldsymbol{\Theta}$, and its corresponding point set is returned as the output (stopping condition).

Continue the example with Figure 3, where $\boldsymbol{\Theta} = \{\Theta_1, \Theta_2, \Theta_3, \Theta_4\}$ and $\Theta_2 = [l_2, e_2]$ is the median partition. We present a user with $\boldsymbol{p}_4$ and $\boldsymbol{p}_5$ since intersection $\wedge_{4,5}$ is at e_2. If the user prefers $\boldsymbol{p}_4$ to $\boldsymbol{p}_5$, $\boldsymbol{\Theta}$ is updated to be $\{\Theta_1, \Theta_2\}$. Then, we conduct another interactive round.

Analysis. The pseudocode is shown in Algorithm 2, which extends its counterpart (2DPI) in [1] for problem ITQ.

Theorem 3. Algorithm *2DPI*$^+$ solves the 2-dimensional ITQ by interacting with a user for $O(\log_2 \frac{nk}{(k-s+1)^2})$ rounds.

Corollary 1. For fixed k, *2DPI*$^+$ is asymptotically optimal in terms of the number of questions for 2-dimensional ITQ.

5 High Dimensional Algorithms

In this section, we propose a general algorithm under a unified framework, called UNIFY, for high-dimensional ITQ. It also follows the interactive framework (Section 3.1).

Below, we first describe the information maintenance and the stopping conditions. Then, we show generalized point selection, followed by the algorithm and its extension.

5.1 Information Maintenance & Stopping Condition

Based on the idea in Section 3.2, algorithm UNIFY maintains a polyhedron $\mathcal{R} \subseteq \mathcal{U}$, called *utility range*, which contains the user's utility vector. Initially, $\mathcal{R}$ is the entire utility space, i.e., $\mathcal{R} = \{\boldsymbol{r} \in \mathbb{R}^d_+ \mid \sum_{i=1}^d u[i] = 1\}$. In each interactive round, a user is presented with two points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$ and is asked to indicate which one s/he prefers. Based on the user's feedback, $\mathcal{R}$ is updated to be $\mathcal{R} \cap h^+_{i,j}$ or $\mathcal{R} \cap h^-_{i,j}$ (Lemma 1). We define two stopping conditions based on $\mathcal{R}$.

Stopping Condition 1 (SC1): Criterion on Hyper-planes. The algorithm can stop if, for any pair of points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$, $h_{i,j}$ does not intersect $\mathcal{R}$ (i.e., $\mathcal{R} \subseteq h^+_{i,j}$ or $\mathcal{R} \subseteq h^-_{i,j}$). The rationale is as follows. Since the user's utility vector $\boldsymbol{u}$ is in $\mathcal{R}$, $\boldsymbol{u}$ must be in either $h^+_{i,j}$ or $h^-_{i,j}$, and thus, the relative rank between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ w.r.t. $\boldsymbol{u}$ is known. Because this holds for all pairs in $\mathcal{D}$, the ranks of all points in $\mathcal{D}$ w.r.t. $\boldsymbol{u}$ are known. In this case, we can identify any s of the top-k points.

Stopping Condition 2 (SC2): Criterion on Points. The algorithm can stop if there exists a set S of s points in $\mathcal{D}$, where each point in S is a top-k point w.r.t. any utility vector in $\mathcal{R}$. Formally, Lemma 6 gives a sufficient condition to check whether a point $\boldsymbol{p}_i \in \mathcal{D}$ is a top-k point w.r.t. any $\boldsymbol{u} \in \mathcal{R}$.

Lemma 6. Given $\mathcal{R}$ and point $\boldsymbol{p}_i \in \mathcal{D}$, $\boldsymbol{p}_i$ is a top-k point w.r.t. any $\boldsymbol{u} \in \mathcal{R}$ if $|\{\boldsymbol{p}_j \in \mathcal{D} \setminus \{\boldsymbol{p}_i\} \mid \mathcal{R} \not\subseteq h^+_{i,j}\}| < k$.

Intuitively, if $\mathcal{R} \not\subseteq h^+_{i,j}$, there exists $\boldsymbol{u} \in \mathcal{R}$ such that $\boldsymbol{u} \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i) > 0$ (i.e., the utility of $\boldsymbol{p}_j$ is higher than that of $\boldsymbol{p}_i$ w.r.t. $\boldsymbol{u}$). If the number of such points $\boldsymbol{p}_j$ is smaller than k, $\boldsymbol{p}_i$ is guaranteed to be a top-k point w.r.t. any $\boldsymbol{u} \in \mathcal{R}$.

To determine whether there exists s points that satisfy Lemma 6, a naive idea is to check each $\boldsymbol{p} \in \mathcal{D}$. However, it could be time-consuming if $\mathcal{D}$ is large. To reduce the burden, we randomly sample a utility vector $\boldsymbol{u}$ in $\mathcal{R}$ and only check each of the top-k points w.r.t. $\boldsymbol{u}$ using Lemma 6. In this way, we could reduce the number of points checked from n to k. If there are s points satisfying Lemma 6, we stop the interaction process and return the s points as the output to the user.

5.2 Generalized Point Selection

During the interaction, each question consists of two points, which corresponds to a hyper-plane to narrow $\mathcal{R}$. To decide which pair of points should be selected as the next question, it is equivalent to selecting a good hyper-plane. This naturally leads to two questions: (Q1) Can we restrict/prioritize a set $\mathcal{C}$ of candidate hyper-planes to reduce the $O(|\mathcal{D}|^2)$ search space of hyper-planes, without affecting the effectiveness of results? (Q2) Given such a set $\mathcal{C}$, which hyper-plane should we pick next, so as to narrow $\mathcal{R}$ the most *effectively*? Below we answer these two critical questions, respectively.

Candidate Hyper-plane Construction (CH). We develop four strategies to construct the candidate hyper-plane set $\mathcal{C}$.

(CH1: Skyband) We preprocess $\mathcal{D}$ to a set $\mathcal{D}_{sky}$ of k-skyband points [32]. Intuitively, $\mathcal{D}_{sky}$ contains the top-k points for any utility vector in $\mathcal{R}$. Then $\mathcal{C} = \{h_{i,j} \mid \forall \boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}_{sky}\}$.

(CH2: Randomized Grouping) We initialize a random order of all points in $\mathcal{D}$. Denote the i-th point in the random order by $\boldsymbol{p}_i$ and define a set $\mathcal{C}_i = \{h_{i,j} \mid j \in [1, i-1]\}$ (i.e., hyperplanes formed by $\boldsymbol{p}_i$ and its previous points). Given a positive integer X, we first combine the first X sets as the initial candidate hyper-plane set, i.e., $\mathcal{C} = \bigcup_{i=1}^{X} \mathcal{C}_i$ (since the first several sets $\mathcal{C}_i$ contain only a few candidate hyper-planes). If the current set $\mathcal{C}$ does not contain any hyper-planes that intersect $\mathcal{R}$ (since only those hyper-planes can be used to narrow $\mathcal{R}$), we proceed with the next set $\mathcal{C}_{i+1}$ as $\mathcal{C}$.

(CH3: Sampling) We sample some utility vectors from the current $\mathcal{R}$ (including the vertices of $\mathcal{R}$) and find the set of top-k points w.r.t. each sampled utility vector. For each unique set $\mathcal{S}$ of top-k points, we build a polyhedron in $\mathcal{R}$, called *cell*, such that the points in $\mathcal{S}$ are the top-k points w.r.t. any utility vector in it. The hyper-planes that constitute the boundaries of the cell are included in the candidate hyper-plane set $\mathcal{C}$. During the interaction, utility range $\mathcal{R}$ will be narrowed. If there is only one cell left in $\mathcal{R}$, we sample the utility vectors and construct $\mathcal{C}$ again. Note that since all the vertices of $\mathcal{R}$ are involved in the sampled utility vectors, there will be more than one unique set of top-k points found (i.e., more than one cell built). Otherwise, if all the vertices of $\mathcal{R}$ correspond to the same set $\mathcal{S}$ of top-k points, each point in $\mathcal{S}$ satisfies Lemma 6 and thus, SC2 is achieved (see below).

Lemma 7. If the top-k points w.r.t. any vertices of $\mathcal{R}$ are the same, each of these top-k points in $\mathcal{S}$ satisfies Lemma 6.

(CH4: Top-1 Points) This strategy is designated for $s = 1$. We define a set C to be the set of all *convex points* in $\mathcal{D}$ [31], which has the highest utilities (i.e., top-1) w.r.t. at least one $\boldsymbol{u} \in \mathcal{U}$. Then $\mathcal{C} = \{h_{i,j} \mid \boldsymbol{p}_i, \boldsymbol{p}_j \in C\}$. Different from the set $\mathcal{S}$ in CH3, C is a deterministic subset of points in $\mathcal{D}$, rather than being retrieved based on sampled utility vectors.

Hyper-plane Selection (HS). Given a set $\mathcal{C}$ of hyper-planes constructed using CH1-CH4, the next issue is how to select a good hyper-plane as a question so that $\mathcal{R}$ is narrowed effectively, making it easier to meet the stopping conditions. Note that every time $\mathcal{R}$ is narrowed, we refine the hyper-planes in $\mathcal{C}$ by only keeping those intersecting $\mathcal{R}$. We develop three hyper-plane selection (HS) strategies in the following.

(HS1: Covering Score) Given a hyper-plane $h_{i,j}$, assume that utility range $\mathcal{R}$ is updated to $\mathcal{R} \cap h_{i,j}^{+}$ (resp. $\mathcal{R} \cap h_{i,j}^{-}$). Denote by M_+ (resp. M_-) the number of hyper-planes in $\mathcal{C}$ that no longer intersect $\mathcal{R} \cap h_{i,j}^{+}$ (resp. $\mathcal{R} \cap h_{i,j}^{-}$). The *covering score* of $h_{i,j}$ is defined as $\min\{M_+, M_-\}$. Intuitively, a hyper-plane $h_{i,j}$ with a higher covering score means that, if $h_{i,j}$ is used to ask a question, more hyper-planes in $\mathcal{C}$ will no longer intersect the updated utility range. This can help to achieve SC1 quickly. Following this idea, we select the hyper-plane in $\mathcal{C}$ with the highest covering score as the question.

Algorithm 3: Algorithm UNIFY

```
Input: A point set D, integers s and k
Output: A set of s of the user's top-k points
R ← {u ∈ R^d_+ | Σ_{i=1}^{d} u[i] = 1};
C ← the first set of hyper-planes by CH1-CH4;
while Neither SC1 nor SC2 is satisfied do
    if C = ∅ (e.g., in CH2 and CH3) then
        C ← the next set of hyper-planes;
    h_{i,j} ← a hyper-plane in C by HS1-HS3;
    Display p_i and p_j to a user;
    Update R and C based on the user feedback;
return s of the top-k points according to SC1-SC2;
```

(HS2: Even Score) We divide the current $\mathcal{R}$ into a number of disjoint polyhedrons $\boldsymbol{\Delta} = \{\Delta_1, \Delta_2, ..., \Delta_m\}$, called *cells*, such that the top-k points w.r.t. any utility vector in the same cell are identical [33]. Given a hyper-plane $h_{i,j}$, assume w.l.o.g. that the user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$. A cell Δ in $\boldsymbol{\Delta}$ can be (1) removed from $\boldsymbol{\Delta}$ if $\Delta \in h_{i,j}^{-}$ or (2) narrowed if Δ intersects $h_{i,j}$. Our idea is to reduce the size of $\boldsymbol{\Delta}$ quickly to achieve SC2. In the ideal case where both $h_{i,j}^{+}$ and $h_{i,j}^{-}$ contain exactly half of the cells in $\boldsymbol{\Delta}$, no matter whether $\boldsymbol{p}_i$ or $\boldsymbol{p}_j$ is preferred, half of the cells in $\boldsymbol{\Delta}$ can be removed. Following this idea, we select the hyper-plane that divides the cells in $\boldsymbol{\Delta}$ the most *evenly*. To evaluate the "evenness", we define *an even score* for each hyper-plane.

Definition 1. The *even score* of $h_{i,j}$ is defined to be $\min\{N_+, N_-\} - \beta N$, where $\beta > 0$ is a balancing parameter, and N_+, N_- and N are the number of cells in $\boldsymbol{\Theta}$ that are in $h_{i,j}^{+}$, are in $h_{i,j}^{-}$ and intersect with $h_{i,j}$, respectively.

Intuitively, a higher even score means that the hyper-plane divides the cells in $\boldsymbol{\Delta}$ more evenly. The first term $\min\{N_+, N_-\}$ shows that we want more cells in $h_{i,j}^{+}$ or $h_{i,j}^{-}$ (and as even as possible); those cells may be removed from $\boldsymbol{\Delta}$, depending on the user preference. The second term $-\beta N$ gives a penalty for those cells intersecting $h_{i,j}$ since they will not contribute to the reduction of $\boldsymbol{\Delta}$, regardless of the user's answer. We select the hyper-plane in $\mathcal{C}$ with the highest even score as the next question.

(HS3: Geometric Center) The idea is to narrow $\mathcal{R}$ quickly to achieve SC2. We select hyperplane $h_{i,j} \in \mathcal{C}$ which divides $\mathcal{R}$ the most *"evenly"*, hoping to reduce the size of $\mathcal{R}$ by half after the question. Since it is costly to compute the exact size of $\mathcal{R}$, we adopt the following heuristic. Denote the center of $\mathcal{R}$ by $\mathcal{R}_c = \sum_{v \in \mathcal{V}_\mathcal{R}} v / |\mathcal{V}_\mathcal{R}|$, where $\mathcal{V}_\mathcal{R}$ is the set of vertices of $\mathcal{R}$. We select the hyperplane in $\mathcal{C}$ which is the closest one to $\mathcal{R}_c$.

5.3 Algorithm

We are ready to describe algorithm UNIFY. The pseudocode is shown in Algorithm 3. Initially, $\mathcal{R}$ is set to be the entire

utility space and the candidate hyper-plane set $\mathcal{C}$ is constructed based on a CH strategy (i.e., CH1-CH4). During the interaction, the algorithm iteratively selects a hyper-plane from $\mathcal{C}$ as the next question. According to the user feedback, $\mathcal{R}$ and $\mathcal{C}$ are updated. The interaction continues until a stopping condition is met. Note that if $\mathcal{C}$ becomes empty before a stopping condition is met (which may occur when using the CH2/CH3 strategy), the algorithm will generate a new $\mathcal{C}$ based on the current $\mathcal{R}$ by the corresponding CH strategy and repeat the process.

We illustrate UNIFY using CH1+HS2 in Figure 4, where $\mathcal{D}_{sky} = \{\boldsymbol{p}_1, \boldsymbol{p}_2, \boldsymbol{p}_3, \boldsymbol{p}_4\}$, leading to 6 hyper-planes in $\mathcal{C}$ and 4 cells in $\boldsymbol{\Delta}$. The half-space above (resp. below) each hyper-plane is labeled "+" (resp. "-"). Let $\beta = 0.1$. For $h_{1,2}$, its even score is $\min\{1, 1\} - 0.1 \times 2 = 0.8$. The even scores of other hyper-planes can be computed similarly and the one with the highest even score is selected as the next question.

Remark *(Special Cases).* UNIFY is a general algorithm. It subsumes existing algorithms in [1] as special cases. More specifically, algorithm HDPI in [1] can be regarded as UNIFY with CH4 and HS2, and algorithm RH in [1] can be regarded as UNIFY with CH2 (with $X = 2$) and HS3.

Analysis. Below we present the theoretical guarantees.

Theorem 4. Algorithm UNIFY solves the d-dimensional ITQ by interacting with a user for $O(n^2)$ rounds.

Theorem 4 gives a general bound on the number of questions for problem ITQ. Nonetheless, the bound can be tightened when considering the special cases of UNIFY. Consider UNIFY using CH2, denoted by UNIFY(CH2),

Theorem 5. Algorithm UNIFY(CH2) solves the d-dimensional ITQ by interacting with a user for $O(kd \log n)$ rounds in expectation with $X = 8\sqrt{kd}$.

Corollary 2. Algorithm UNIFY(CH2) is asymptotically optimal in terms of the number of questions in expectation for d-dimensional ITQ if k and d are fixed.

5.4 Extension: User Error

Algorithm UNIFY follows the widely adopted assumption that users always provide *correct* feedback [9]. Here, consider two points $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$, and assume that $f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j)$ w.r.t. the user's utility vector $\boldsymbol{u}$. We say that a user's feedback is *correct* if the user claims that s/he prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$. However, in real-world scenarios, users may provide *incorrect* feedback (i.e., the user tells that s/he prefers $\boldsymbol{p}_j$ to $\boldsymbol{p}_i$) due to some reasons, e.g., a mis-click. Such incorrect feedback could generate wrong half-spaces, which in turn leads to a wrong update of utility range $\mathcal{R}$, affecting the output of the algorithm. For example, although $f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j)$, utility range $\mathcal{R}$ is updated to be $\mathcal{R} \cap h^-_{i,j}$. In the following, we extend UNIFY to UNIFY-ERR, which is capable of tolerating errors users make when providing feedback.

We distinguish two types of questions in UNIFY-ERR: (1) *Normal question*. The hyper-plane $h_{i,j}$ intersects $\mathcal{R}$, and thus, can narrow $\mathcal{R}$ based on the user feedback. Note that all questions in UNIFY are normal questions. (2) *Testing question*. The hyper-plane $h_{i,j}$ does not intersect $\mathcal{R}$, i.e., $\mathcal{R} \subseteq h^+_{i,j}$ or $\mathcal{R} \subseteq h^+_{j,i}$. The preference between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ should have been known based on $\mathcal{R}$. Assume w.l.o.g. that $\mathcal{R} \subseteq h^+_{i,j}$ but the user prefers $\boldsymbol{p}_j$ to $\boldsymbol{p}_i$ (an inconsistency since $h^+_{j,i} \cap \mathcal{R} = \emptyset$). Then we know that the current $\mathcal{R}$ may need to be adjusted.

UNIFY-ERR differs from UNIFY by asking Y testing questions after every Y normal questions (the normal questions are asked using the same strategies as in UNIFY), where Y is a user-defined positive integer. If there is no inconsistency after the $2Y$ questions (i.e., the intersection of the $2Y$ resulting half-spaces is not empty), we mark that the Y normal and Y testing questions are correctly answered. We proceed with the next Y normal questions, followed by another Y testing questions. If there is an inconsistency (i.e., the intersection of the $2Y$ resulting half-spaces is empty), there exists incorrect feedback within those $2Y$ questions.

To tackle this, we move the $2Y$ hyper-planes (each corresponds to one of the $2Y$ questions) so that the intersection of their half-spaces is non-empty, which gives a new utility range $\mathcal{R}$. Below we explain (a) how to move the $2Y$ hyper-planes to get an non-empty intersection as a new $\mathcal{R}$ and (b) how to construct the testing questions in UNIFY-ERR.

Hyper-plane Movement. To decide the hyper-plane movement, we adopt the widely used Bradley–Terry model [25], [34], [35] to simulate user feedback. Specifically, given two points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$, the probability that a user tells s/he prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, denoted by $P_{\boldsymbol{u}}(\boldsymbol{p}_i \succ \boldsymbol{p}_j)$, is defined to be

$$P_{\boldsymbol{u}}(\boldsymbol{p}_i \succ \boldsymbol{p}_j) = \begin{cases} \frac{1}{1+exp^{-|f_{\boldsymbol{u}}(\boldsymbol{p}_i) - f_{\boldsymbol{u}}(\boldsymbol{p}_j)|}} & f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j) \\ 1 - \frac{1}{1+exp^{-|f_{\boldsymbol{u}}(\boldsymbol{p}_j) - f_{\boldsymbol{u}}(\boldsymbol{p}_i)|}} & f_{\boldsymbol{u}}(\boldsymbol{p}_i) < f_{\boldsymbol{u}}(\boldsymbol{p}_j) \end{cases}$$

Intuitively, the probability of providing correct feedback depends on the gap between the utilities of $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$. If the gap is larger, the user is more likely to provide correct feedback. Thus, if there exists an inconsistency, we move each hyper-plane by a certain offset to make the resulting half-space include more utility vectors, and the utility vectors to be included are those leading to a small utility gap between $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$, since users with those utility vectors are more likely to give incorrect feedback, leading to inconsistency. The lemma below formally gives the movement offset.

Lemma 8. Given a probability threshold $\epsilon \leq 0.5$ and a half-space $h^+_{i,j} = \{\boldsymbol{u} \in \mathbb{R}^d \mid r \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0\}$, where the user tells s/he prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, if we move $h_{i,j}$ by an offset $\ln \frac{1-\epsilon}{\epsilon}$ to get $h^{\epsilon+}_{i,j} = \{\boldsymbol{u} \in \mathbb{R}^d \mid \boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) - \ln \frac{\epsilon}{1-\epsilon} > 0\}$, we have $P_{\boldsymbol{u}}(\boldsymbol{p}_i \succ \boldsymbol{p}_j) \geq 1 - \epsilon$ for each $\boldsymbol{u} \in h^{\epsilon+}_{i,j}$.

Taken a user-defined probability threshold ϵ and the set $\mathcal{C}_{2Y}$ of $2Y$ hyper-planes as input, we assume w.l.o.g. that for each $h_{i,j}$ in $\mathcal{C}_{2Y}$, the user (correctly or incorrectly) answers that s/he prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$. Based on Lemma 8, we compute a new utility region $\mathcal{R} = \bigcap_{h^+_{i,j} \in \mathcal{C}_{2Y}} h^{\epsilon+}_{i,j} \cap \mathcal{U}$ so that the probability that the user provides correct feedback is above the threshold. Note that if the above intersection is still empty, we can invite the user to set a larger ϵ, and repeat the process until the intersection is non-empty.

Consider two normal questions (the black dashed hyper-planes $h_{1,2}$ and $h_{1,3}$) in Figure 5, where $\mathcal{R}$ is shown in the shaded region (i.e., $\mathcal{R} = h^+_{1,2} \cap h^+_{1,3} \cap \mathcal{U}$). Let $h_{4,5}$ and $h_{4,6}$ be two testing questions (the red dashed hyper-planes), where the user prefers $\boldsymbol{p}_4$ to both $\boldsymbol{p}_5$ and $\boldsymbol{p}_6$. Since $h^+_{4,5} \cap h^+_{4,6} \cap \mathcal{R}$ is empty, we move each of the four hyper-plane by a certain offset (shown in arrows in Figure 6). The new utility range $\mathcal{R} = h^{\epsilon+}_{1,2} \cap h^{\epsilon+}_{1,3} \cap h^{\epsilon+}_{4,5} \cap h^{\epsilon+}_{4,6} \cap \mathcal{U}$ is shown shaded in Figure 6.

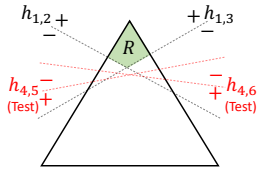


Fig. 5: Error

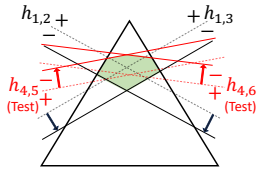


Fig. 6: Error (moved)

Testing Question Construction. Recall that $\mathcal{R}_c$ is the geometric center of $\mathcal{R}$ (see the discussion of HS3 in Section 5.2). We randomly select a few pairs of points in $\mathcal{D}$ to build candidate hyper-planes and pick $h_{i,j}$ as a testing question such that (a) $h_{i,j}$ does not intersect $\mathcal{R}$ and (b) $h_{i,j}$ is the closest to $\mathcal{R}_c$. Intuitively, if a user has provided incorrect feedback in the Y normal questions, his/her utility vector $\boldsymbol{u}$ is not the resulting $\mathcal{R}$. To detect this, $\mathcal{R}$ and $\boldsymbol{u}$ should lie in different half-spaces of $h_{i,j}$, since otherwise, the user will answer the testing question correctly. Therefore, we adopt condition (b) so that utility range $\mathcal{R}$ and the user's utility vector $\boldsymbol{u}$ are more likely to lie in different half-spaces of $h_{i,j}$.

6 Experiments

We conducted experiments on a machine with M3 chip and 96GB RAM. All programs were implemented in C/C++.

Datasets. We adopt commonly used synthetic/real datasets in existing studies [9], [19], [32], [36]. The synthetic datasets are *anti-correlated*, generated by the generator for *skyline* queries [7], [36]. The real datasets are *Island*, *Car*, and *NBA*. Dataset *Island* contains 63,383 2-dimensional geographic locations. Dataset *Car* consists of 68,010 used cars, described by four attributes (price, year of production, horsepower, and used kilometers), where we only keep the cars whose attribute values are in a certain range following [1]. Dataset *NBA* has 16,916 players after incomplete records are excluded. Six attributes are used to describe the performance of players on the court (e.g., points, rebound, and so on).

For all datasets, each dimension is normalized to $(0, 1]$. Note that existing studies [9], [37] preprocessed datasets to contain skyline points only (which are all possible top-1 points for some utility function) since they look for (close to) top-1 point. Consistent with their setting, we preprocessed all the datasets to include k-skyband points (which are the top-k points w.r.t. at least a utility function) [32] since we are interested in some of the top-k points.

Algorithms. We compared our algorithms 2DSeg, 2DPI$^+$, and Unify with the following baselines Median [9], Hull [9], ActiveRanking [11], UH-Random [9], UH-Simplex [9], and PrefLearning [12]. The baselines were adapted to suit our problem as follows.

- Algorithms Median and Hull only work on the 2-dimensional datasets. They are designed to return the user's top-1 point. We extended them to returning s of the user's top-k points with two modifications. Firstly, their *point deletion condition* is modified. A point is deleted if it cannot be one of the user's top-k points (originally top-1 point) according to the learned information. Secondly, their *stopping condition* is altered. The algorithms stop if there are fewer than k points left (originally 1 point left).
- Algorithm ActiveRanking focuses on learning the full ranking of points by interacting with the user. We arbitrarily return s of top-k points when the ranking is obtained.
- Algorithms UH-Simplex and UH-Random are proposed to return the user's (close to) top-1 point by interacting with the user. We also modified their *point deletion condition* and their *stopping condition* as stated before.
- Algorithm PrefLearning approximates the user's utility vector by interacting with the user. We set the error threshold ϵ to 10^{-9} (since the learnt utility vector could be close to the theoretical optimum) and arbitrarily return s of user's top-k points w.r.t. the learnt utility vector.

Note that our work is an extension of [1]. In [1], there are three algorithms proposed: 2DPI, HDPI, and RH. Our newly proposed algorithms are the general versions of them (from $s = 1$ to $s \geq 1$). Concretely, 2DPI$^+$ is a general version of 2DPI, with novel speedup techniques. Algorithm Unify equipped with CH4 and HS2 (resp. equipped with CH2 and HS3) is a general version of HDPI (resp. RH).

Parameter Setting. We evaluated the performance of each algorithm by varying (1) the parameters s and k, (2) the dimensionality d, and (3) the dataset size n. Unless stated explicitly, the default setting on synthetic datasets is $n = 100,000$, $d = 4$, $k = 20$, and $s = 5$, following [1], [9], [38].

Performance Measurement. We evaluated the performance of each algorithm by two measurements: (1) *the execution time* which is the processing time; (2) *the number of questions asked* which is the number of rounds interacting with the user. Each algorithm was conducted 10 times with different randomly generated utility vectors (i.e., utility functions) and the average performance was reported.

6.1 Performance Study of Algorithm Unify

We first evaluated the strategies of candidate hyper-plane construction (CH) and hyper-plane selection (HS) in Unify.

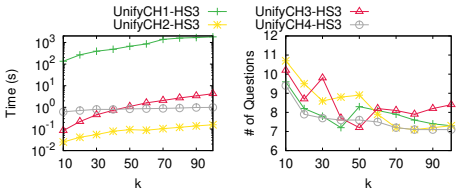
In Figures 7-9, we fixed the HS strategy to HS3 and tested the effect of CH strategies, by varying parameters s and k. CH4 only appears in Figure 7 since it is designed for $s = 1$ only. The result shows that Unify with CH1 was significantly slower than the others since it constructs more candidate hyper-planes. Nonetheless, Unify with different CH strategies are comparable in the number of questions.

Similarly in Figures 10-12, we fixed the CH strategy to CH3 and tested the effect of different HS strategies. We also included CH4 in Figure 10 since it is designed for $s = 1$. The result shows that HS3 took the shortest execution time in most cases. This efficiency stems from its simplified necessity for computing the distances between points and hyper-planes. As for the number of questions asked, HS1 asked the fewest questions in most cases, which demonstrates its effectiveness in narrowing utility range $\mathcal{R}$.

For the ease of presentation, we only report two versions of Unify in the rest experiments: one with CH2 and HS3 (UnifyCH2-HS3) and with CH3 and HS1 (UnifyCH3-HS1).

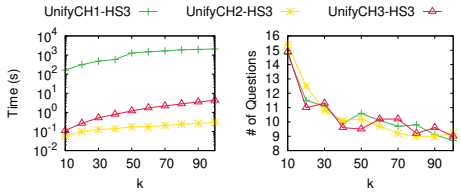
6.2 Performance on Synthetic Datasets

Varying k and s. We compared our algorithms against existing ones on 2-dimensional and 4-dimensional synthetic



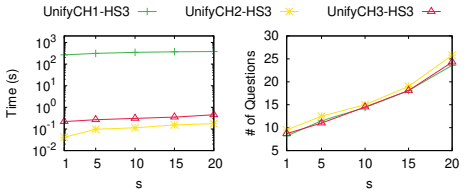
(a) (b)

Fig. 7: Vary CH - vary k ($s = 1$)



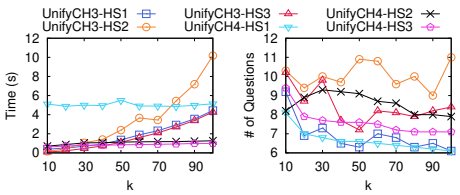
(a) (b)

Fig. 8: Vary CH - vary k ($s = 5$)



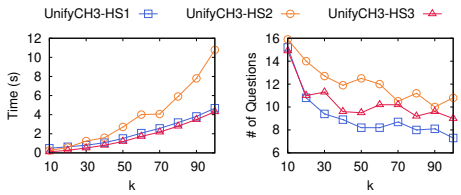
(a) (b)

Fig. 9: Vary CH - vary s



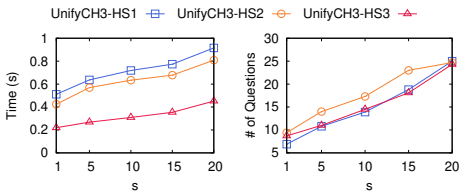
(a) (b)

Fig. 10: Vary HS - vary k ($s = 1$)



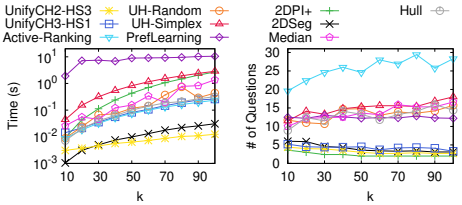
(a) (b)

Fig. 11: Vary HS - vary k ($s = 5$)



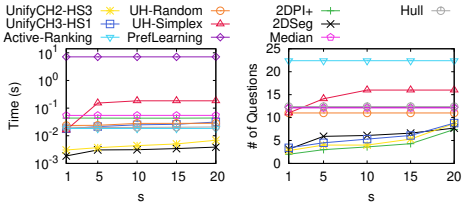
(a) (b)

Fig. 12: Vary HS - vary s



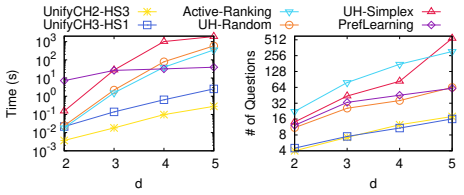
(a) (b)

Fig. 13: Vary k (2D)



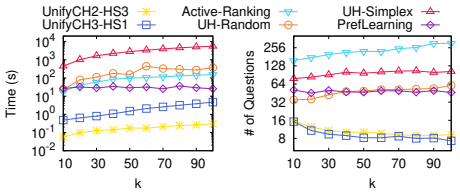
(a) (b)

Fig. 14: Vary s (2D)



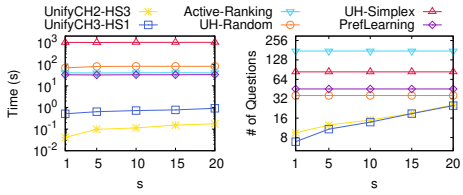
(a) (b)

Fig. 15: Vary dimension d



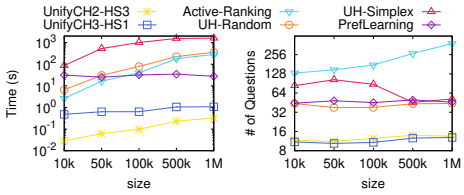
(a) (b)

Fig. 16: Vary k (4D)



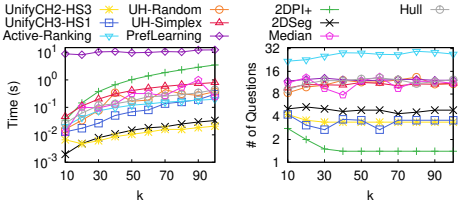
(a) (b)

Fig. 17: Vary s (4D)



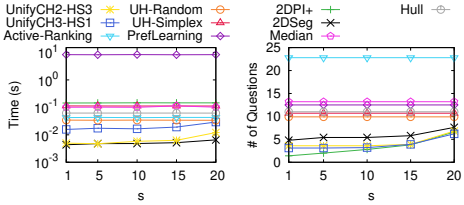
(a) (b)

Fig. 18: Vary size n



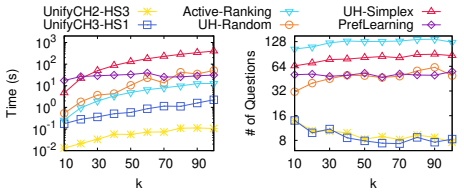
(a) (b)

Fig. 19: Vary k (Island)



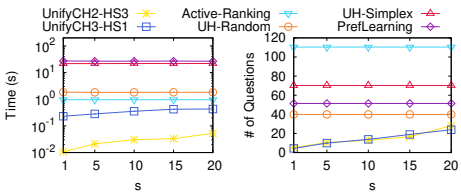
(a) (b)

Fig. 20: Vary s (Island)



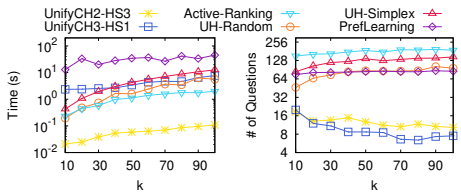
(a) (b)

Fig. 21: Vary k (Car)



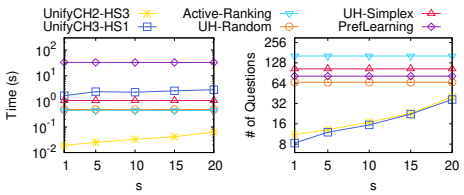
(a) (b)

Fig. 22: Vary s (Car)



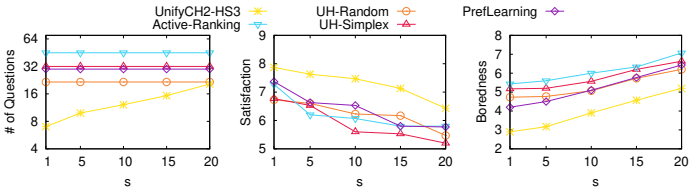
(a) (b)

Fig. 23: Vary k (NBA)



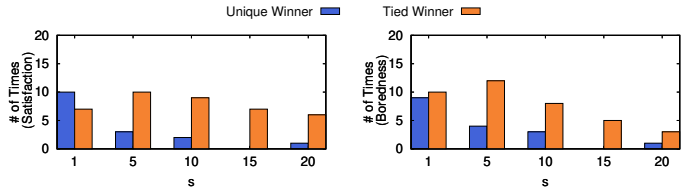
(a) (b)

Fig. 24: Vary s (NBA)



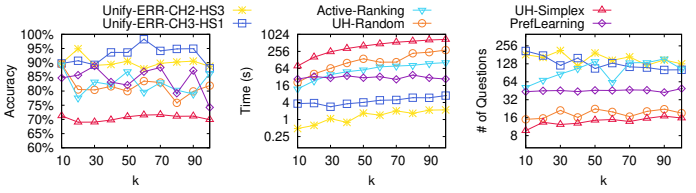
(a) (b) (c)

Fig. 25: User Study



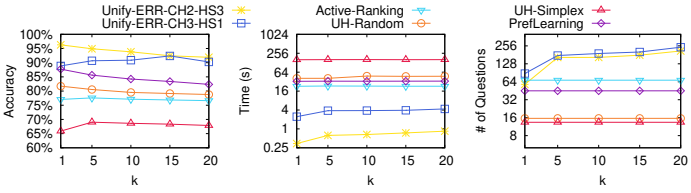
(a) (b)

Fig. 26: Case Study



(a) (b) (c)

Fig. 27: Vary k (User Error)



(a) (b) (c)

Fig. 28: Vary s (User Error)

datasets, by varying k and s. Each algorithm was evaluated by the execution time and the number of questions asked.

2D dataset (varying k). Fixing $s = 5$, we varied k from 10 to 100 on a 2-dimensional synthetic dataset. As shown in Figure 13(a), when k increased, all algorithms took longer execution time as expected, since more points need to be processed (the input size increases with k due to the k-skyband preprocessing). Nevertheless, all algorithms (except PREFLEARNING) could finish within 3 seconds. PREFLEARNING ran slow due to its costly data structure *spherical tree* [12]. Figure 13(b) shows different trends between our algorithms and the existing ones in terms of the number of questions asked. Our algorithms tended to ask fewer questions given a larger k. This is because with a larger k, the constraint for a point to be returned is more relaxed, and thus, there are more "qualified" points that can be returned. Therefore, it suffices to learn a less precise utility vector (with fewer questions) to find the desired output. In contrast, existing algorithms did not have this trend since they do not fully investigate the relation between the learned preference and the top-k points. Even if the output constraint is relaxed, they still need to learn a precise utility vector. Our algorithms consistently performed the best, e.g., *2DPI*$^+$ asked 6-26 fewer questions than existing ones.

2D dataset (varying s). Fixing $k = 20$, we varied s from 1 to 20 on a 2-dimensional synthetic dataset. Figure 14 shows that when s increased, our algorithms got slightly slower and asked more questions. This is expected since the user's utility vector had to be learned more precisely to identify more top-k points. However, most existing algorithms were not sensitive to s. For example, the execution time and the number of questions of ACTIVERANKING were almost indifferent to s. This is because ACTIVERANKING focused on learning the full ranking of points, which was not affected by parameter s. Nevertheless, our algorithms asked almost half of the questions compared to existing ones in most cases. For example, when $s = 10$, our algorithm 2DPI$^+$ only asked 3.6 questions, as opposed to 11 questions asked by the best existing algorithm UH-RANDOM.

4D dataset (varying k). Figure 16 shows the performance of algorithms on a 4-dimensional dataset by varying k from 10 to 100 and fixing $s = 5$. ACTIVERANKING was ineffective and inefficient since it learned the full ranking. It asked the largest number of questions, e.g., 223 questions when $k = 50$, and took 2-3 orders of magnitude longer time than our algorithms. UH-RANDOM, UH-SIMPLEX and PREFLEARNING were also slow. The first two spent hundreds of seconds comparing each pair of points for the costly *point deletion* step and PREFLEARNING needed to build and maintain costly data structures. In contrast, UNIFYCH2-HS3 and UNIFYCH3-HS1 took less than 0.3 seconds and 4.7 seconds for any k, respectively. For the number of questions asked, PREFLEARNING asked 3-6 times more questions than our algorithms. UH-RANDOM and UH-SIMPLEX did not work well either. For example, when $k = 100$, they asked 60.8 and 103.6 questions, respectively, while our algorithms asked at most 9.2 questions. Better still, our algorithms reduced the number of questions asked by 40% when k increased from 10 to 100, due to the relaxed output constraint.

4D dataset (varying s). Figure 17 showed the results on a 4-dimensional synthetic dataset by varying s from 1 to 20 and fixing $k = 20$. Our algorithms still beat existing ones by asking fewer questions in a shorter time. Besides, consistent with 2-dimensional results, when s increased from 1 to 20, the number of questions we asked rose from 6.9 to 25.8, while all existing algorithms performed almost the same w.r.t. different s, due to the same reason as in 2D datasets.

Scalability. We studied the scalability of algorithms by varying the dimensionality d and the dataset size n, respectively.

Varying d. In Figure 15, we tested our scalability by varying d from 2 to 5. Figure 15(a) shows the execution time. Our algorithms scaled the best. For instance, when $d = 5$, they took a maximum of 2.6 seconds, while the fastest existing algorithm PREFLEARNING took 39.2 seconds. Figure 15(b) shows the number of questions asked. Our algorithms asked at least 63% fewer questions than the existing ones for all d. When $d = 5$, our algorithms asked at most 17.9 questions, as opposed to 61.4, 64, and 544.7 questions by PREFLEARNING, UH-RANDOM and UH-SIMPLEX, respectively.

Varying n. In Figure 18, we studied the scalability w.r.t. the dataset size n by varying it from 10k to 1M on 4-dimensional synthetic datasets. Our algorithms ran 1-4 orders of mag-

nitude faster and asked at least 68% fewer questions than existing algorithms. For example, when n = 500k, our algorithms took at most 1.05 seconds and asked 13.9 questions, while the most efficient PREFLEARNING took 34.7 seconds and the most effective UH-RANDOM asked 43.5 questions.

6.3 Performance on Real Datasets

We first evaluated the performance of algorithms on real datasets by varying parameters k and s. Then, we show a user study to verify our effectiveness on *real users* and a case study to justify the necessity of returning multiple tuples.

Varying k and s. The results on *Island*, *Car*, and *NBA* are shown in Figures 19-24. As shown there, our algorithms performed well in terms of both the execution time and the number of questions asked. Consider the results on *Car* (Figures 21-22). Our algorithms spent the shortest time and asked the fewest questions. For instance, when $k = 100$ and $s = 5$, our algorithms asked 8.3 questions within 2.3 seconds, while the best existing algorithm UH-RANDOM asked 49.2 questions in 50.8 seconds. This again verifies the efficiency and effectiveness of our algorithms.

User study. We conducted a user study on *Car* to demonstrate our effectiveness on real users. Following [1], [9], [38], we randomly selected 1000 candidate cars from the dataset. Each car was described by four attributes, namely price, year of purchase, horsepower, and used kilometers. We recruited 30 participants and reported their average results.

For the ease of presentation, we only compared UNIFY with CH2 and HS3 against the four existing algorithms, since it ran the fastest as shown in Figures 21-22. Fixing $k = 20$, each algorithm was tested under 5 settings of s, namely s=1/5/10/15/20. Note that different from the previous experiments, where the utility vector was randomly generated, the user's utility vector was unknown in the user study. Thus, we re-adapted algorithm *PrefLearning* (instead of the way described previously) following [38].

Each algorithm was evaluated by three measurements. (1) *The number of questions asked*. (2) *Satisfaction*. It is a score from 1 to 10 given by each participant (the higher the better), which indicates how satisfied the participant is when s/he sees the set of returned cars. (3) *Boredness*. It is a score from 1 to 10 given by each participant (the smaller the better), which indicates how bored the participant feels when s/he sees the returned cars after being asked several questions.

Figure 25 summarizes the results. UNIFYCH2-HS3 performed the best under all measurements. When $s = 10$, the number of questions asked by UNIFYCH2-HS3 was 12.17, while existing algorithms asked more than 21 questions. In particular, ACTIVERANKING asked 45.13 questions. Better still, the satisfactions and the degrees of boredness of our algorithm were 7.47 and 3.9, respectively, while the satisfactions of existing algorithms were less than 6.5, and their degrees of boredness were more than 5.07.

Motivation study. We conducted a motivation study on *Car* by utilizing our algorithm UNIFYCH2-HS3 to show the necessity of returning $s \geq 1$ tuples (instead of $s = 1$). We also recruited 30 participants and reported their total results.

Fixing $k = 20$, we compared the scores of *satisfaction* and *boredness* (as introduced in the user study) across 5 cases: $s = 1/5/10/15/20$. We say that a case is the winner case of a participant in terms of *satisfaction* (resp. *boredness*) if it receives the highest score (resp. the lowest score) among the five cases from that participant. Note that multiple cases may receive the same highest (resp. lowest) score. Figures 26(a) and 26(b) show the number of times each case won in terms of the *satisfaction* and *boredness*, respectively, where we classified winners into two categories: *unique winner* (i.e., only one case receives the highest score) and *tied winner* (i.e., multiple cases receive the same highest score).

The results indicate that each case had its own share of wins, verifying the diverse user preferences w.r.t. different s. Since our algorithms allow users to set s according to their specific needs, this flexibility makes them highly versatile and capable of handling different scenarios.

6.4 Extended Study: User Error

Recall that in Section 5.4, we extend algorithm UNIFY to UNIFY-ERR, to tolerate the mistakes the users make when answering questions. In the following, we compared algorithm UNIFY-ERR with existing ones, focusing on their robustness in tolerating user errors. To simulate user feedback, we again applied the Bradley–Terry model (see Section 5.4).

Let S be the set of s points returned by an algorithm and $\mathcal{S}^*$ be the set of true top-k points. Denote by $U(S)$ the utility sum of all points in set S, i.e., $U(S) = \sum_{\boldsymbol{p} \in S} f_{\boldsymbol{u}}(\boldsymbol{p})$. To measure the quality of S, we followed [1] to include an additional measurement, called *accuracy*. Specifically, the *accuracy* of S is defined to be $\frac{U(S)}{U_{\min}}$, where $U_{\min} = \min_{S' \subseteq \mathcal{S}^* \text{ and } |S'|=s} U(S')$. Intuitively, $U_{\min}$ is the minimum utility sum for any s of true top-k points and any valid solution of problem ITQ has its utility sum as least $U_{\min}$. In other words, a higher accuracy is more desirable since it means that S is closer to a valid solution for problem ITQ.

Figures 27-28 show the results. Our algorithms consistently outperformed the existing ones in terms of accuracy. Their accuracy surpasses 88% in all cases. This verified the robustness of our algorithms, i.e., even if the user made mistakes during the interaction, our algorithm could still return high-utility points. Besides, our algorithms required the shortest execution time. Although our algorithms asked more questions, this is reasonable since they asked additional testing questions to ensure the accuracy of the results.

6.5 Summary

The experiments justified the superiority of our algorithms over the best-known existing ones: (1) We were effective and efficient. Our algorithms asked fewer questions within less time (e.g., when $k = 100$, our algorithms asked 80% fewer questions than existing ones on a 4-dimensional dataset). (2) Our algorithms scaled well on the dimensionality and the dataset size (e.g., our algorithms asked at most 13.9 questions when n = 500k on a 4-dimensional dataset, as opposed to at least 43.5 questions by existing algorithms). (3) Our algorithms achieved the best performance on real users. (4) The error-tolerant variant of our algorithm is robust (e.g., it consistently achieved at least 88% accuracy in all cases).

7 CONCLUSION

In this paper, we study the interactive top-k query (problem ITQ) and develop interactive algorithms for searching s of

the user's top-k tuples with as little user effort as possible. In a 2-dimensional space, we propose algorithms 2DPI$^+$ and 2DSEG. In a d-dimensional space, we present algorithm UNIFY. All algorithms perform well both theoretically and empirically. We also extend our study to the case where the user may make mistakes during interaction and develop a robust variant of UNIFY, called UNIFY-ERR. Extensive experiments were conducted to show that our algorithms are efficient, effective and robust for problem ITQ. As for future work, we consider dynamic datasets and data stream.

REFERENCES

[1] W. Wang, R. C.-W. Wong, and M. Xie, "Interactive search for one of the top-k," in *SIGMOD*, 2021.
[2] M. Xie, T. Chen, and R. C.-W. Wong, "Findyourfavorite: An interactive system for finding the user's favorite tuple in the database," in *SIGMOD*, 2019.
[3] M. Xie, R. C.-W. Wong, P. Peng, and V. J. Tsotras, "Being happy with the least: Achieving α-happiness with minimum number of tuples," in *ICDE*, 2020.
[4] W. Wang, R. C.-W. Wong, H. V. Jagadish, and M. Xie, "Reverse regret query," in *ICDE*, 2024.
[5] J. Lee, G.-w. You, and S.-w. Hwang, "Personalized top-k skyline queries in high-dimensional space," *Information Systems*, 2009.
[6] P. Peng and R. C.-W. Wong, "K-hit query: Top-k query with probabilistic utility function," in *SIGMOD*, 2015.
[7] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001.
[8] D. Nanongkai, A. Lall, A. Das Sarma, and K. Makino, "Interactive regret minimization," in *SIGMOD*, 2012.
[9] M. Xie, R. C.-W. Wong, and A. Lall, "Strongly truthful interactive regret minimization," in *SIGMOD*, 2019.
[10] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu, "Regret-minimizing representative databases," in *VLDB*, 2010.
[11] K. G. Jamieson and R. D. Nowak, "Active ranking using pairwise comparisons," in *NIPS*, 2011.
[12] L. Qian, J. Gao, and H. V. Jagadish, "Learning user preferences by adaptive pairwise comparison," in *VLDB*, 2015.
[13] M. Revilla and C. Ochoa, "Ideal and maximum length for a web survey," *International Journal of Market Research*, 2017.
[14] T. Seidl and H.-P. Kriegel, "Efficient user-adaptable similarity search in large multimedia databases," in *VLDB*, 1997.
[15] I. Bartolini, P. Ciaccia, V. Oria, and M. T. Özsu, "Flexible integration of multimedia sub-queries with qualitative preferences," *Multimedia Tools and Applications*, 2007.
[16] P. Peng and R. C.-W. Wong, "Geometry approach for k-regret query," in *ICDE*, 2014.
[17] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides, "Computing k-regret minimizing sets," in *VLDB*, 2014.
[18] I. Bartolini, P. Ciaccia, and M. Patella, "Domination in the probabilistic world: Computing skylines for arbitrary correlations and ranking semantics," *TODS*, 2014.
[19] G. Koutrika, E. Pitoura, and K. Stefanidis, "Preference-based query personalization," *Advanced Query Processing*, 2013.
[20] J. Lee, G.-W. You, S.-W. Hwang, J. Selke, and W.-T. Balke, "Interactive skyline queries," *Information Sciences*, 2012.
[21] W.-T. Balke, U. Güntzer, and C. Lofi, "Eliciting matters – controlling skyline sizes by incremental integration of user preferences," in *Advances in Databases: Concepts, Systems and Applications*, 2007.
[22] G. Salton, *Automatic Text Processing: The Transformation, Analysis, and Retrieval of Information by Computer*, 1989.
[23] I. Bartolini, P. Ciaccia, and F. Waas, "Feedbackbypass: A new approach to interactive similarity query processing," in *VLDB*, 2001.
[24] T.-Y. Liu, "Learning to rank for information retrieval," in *SIGIR*, 2010.
[25] L. Maystre and M. Grossglauser, "Just sort it! a simple and effective approach to active preference learning," in *ICML*, 2017.
[26] B. Eriksson, "Learning to top-k search using pairwise comparisons," in *AISTATS*, 2013.
[27] M. Xie, R. C.-W. Wong, J. Li, C. Long, and A. Lall, "Efficient k-regret query algorithm with restriction-free bound for any dimensionality," in *SIGMOD*, 2018.
[28] A. Asudeh, A. Nazi, N. Zhang, G. Das, and H. V. Jagadish, "Rrr: Rank-regret representative," in *SIGMOD*, 2019.
[29] J. Dyer and R. Sarin, "Measurable multiattribute value functions," *Operations Research*, 1979.
[30] R. Keeney, H. Raiffa, and D. Rajala, "Decisions with multiple objectives: Preferences and value trade-offs," *Systems, Man and Cybernetics, IEEE Transactions on*, 1979.
[31] M. De Berg, O. Cheong, M. Van Kreveld, and M. Overmars, *Computational geometry: Algorithms and applications*, 2008.
[32] Y. Gao, Q. Liu, B. Zheng, L. Mou, G. Chen, and Q. Li, "On processing reverse k-skyband and ranked reverse skyline queries," *Information Sciences*, 2015.
[33] B. Tang, K. Mouratidis, and M. L. Yiu, "Determining the impact regions of competing options in preference space," in *SIGMOD*, 2017.
[34] D. R. Hunter, "Mm algorithms for generalized bradley-terry models," *The annals of statistics*, 2004.
[35] N. B. Shah and M. J. Wainwright, "Simple, robust and optimal ranking from pairwise comparisons," *JMLR*, 2017.
[36] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "Progressive skyline computation in database systems," *TODS*, 2005.
[37] W. Cao, J. Li, H. Wang, K. Wang, R. Wang, R. C.-W. Wong, and W. Zhan, "k-Regret Minimizing Set: Efficient Algorithms and Hardness," in *ICDT*, 2017.
[38] W. Wang, R. C.-W. Wong, and M. Xie, "Interactive search with mixed attributes," in *ICDE*, 2023.

Weicheng Wang is currently a Post-Doctoral Fellow in the Department of Computer Science, the Hong Kong University of Science and Technology (HKUST). He received his Ph.D. degree from the Hong Kong University of Science and Technology (HKUST), in 2023, and BEng degree from the Beijing Normal University (BNU), in 2018. His research interests include database, data mining, and machine learning.

Min Xie is a researcher in Shenzhen Institute of Computing Sciences (SICS). She received her Ph.D. degree from the Hong Kong University of Science and Technology (HKUST) in 2019, and BEng degree from a joint program of Hong Kong University of Science and Technology (HKUST) and Sun Yat-sen University (SYSU) in 2014. Her research interests include database and data mining. She has won several awards, including the best paper award of ICDE 2024 and the best demo award of VLDB 2024.

Raymond Chi-Wing Wong Raymond Chi-Wing Wong received the BSc, MPhil and PhD degrees in computer science and engineering from the Chinese University of Hong Kong (CUHK) in 2002, 2004, and 2008, respectively. He is a professor of the Department of Computer Science and Engineering, the Hong Kong University of Science and Technology. His research interests include database and data mining.

Victor Junqiu Wei is currently working as a research assistant professor in the Department of Computer Science and Engineering (CSE), the Hong Kong University of Science and Technology (HKUST). He obtained his bachelor degree from Nanjing University and PhD degree from Department of Computer Science and Engineering, the Hong Kong University of Science and Technology. He also has several years' working experience in the world-famous research labs in the AI industry including Baidu natural language processing (NLP) group, AI group of WeBank and Noah's Ark Lab of Huawei. He served as a program committee member of many top conferences such as ICDE, KDD, CIKM, SIGSPATIAL, ACL, NAACL, and EMNLP, etc. He has won a lot of honors and award including the best paper candidate of SIGMOD 2020.

APPENDIX A: PROOFS

Proof of Theorem 1: Consider a point set $\mathcal{D}$ which satisfies the following two properties. (1) For each point $\boldsymbol{p} \in \mathcal{D}$, there are $k-s$ points $\boldsymbol{q} \in \mathcal{D} \setminus \{p\}$ that are the same with $\boldsymbol{p}$, i.e., $\forall i \in [1,d]$, $p[i] = q[i]$. (2) Each point $\boldsymbol{p} \in \mathcal{D}$ can be the top-k points w.r.t. some utility vectors in the utility space $\mathcal{U}$. Figure 29 shows an example. All the points in $\mathcal{D}$ have two dimensions and form a quarter circle. Each black dot in the figure represents $k-s+1$ points $\boldsymbol{p}$, which are the top-k points w.r.t. the utility point $\boldsymbol{u} \in \mathcal{U}$ such that $\frac{p[1]}{p[2]} = \frac{u[1]}{u[2]}$.

Define a *top-k set* to be a collection of k points that are the top-k points w.r.t. some utility points in the utility space $\mathcal{U}$. Based on two properties of the point set discussed above, there are $\Omega(\frac{n}{k-s+1})$ different top-k sets. For any two of these top-k sets G_1 and G_2, $|G_1 \cap G_2|$ must be smaller than s, since each point in $\mathcal{D}$ has $k-s$ same points. Thus, in order to determine a set of s of the user's top-k points, it is necessary to learn which top-k set is preferred by the user.

Based on the analysis of [1], [9], any algorithm that aims to identify the top-k sets must proceed in the form of a binary tree. Each internal node corresponds to a question asked to a user and each leaf maps to a top-k set. Since there are $\Omega(\frac{n}{k-s+1})$ top-k sets, there would be $\Omega(\frac{n}{k-s+1})$ leaves. The height of the tree should be $\Omega(\log_2 \frac{n}{k-s+1})$. Thus, any algorithm needs to ask $\Omega(\log_2 \frac{n}{k-s+1})$ questions to identify s of the top-k points. □

Proof of Lemma 1: We prove two implications.

(IF) If a user prefers $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$, we have $f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j)$, i.e., $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$, where $\boldsymbol{u}$ is the user's utility vector. This implies that $\boldsymbol{u} \in h^+_{i,j}$ (or $\boldsymbol{u} \in h^-_{j,i}$), and thus, the user's utility vector $\boldsymbol{u}$ must be in $h^+_{i,j} \cap \mathcal{U}$.

(ONLY IF) If a user's utility vector $\boldsymbol{u}$ is in $h^+_{i,j} \cap \mathcal{U}$, we have $\boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$. This implies that $f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j)$, and thus, the user must prefer $\boldsymbol{p}_i$ to $\boldsymbol{p}_j$. □

Proof of Lemma 2: Let $\mathcal{V} = \{\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_m\}$ be the set of the vertices of $\mathcal{P}$. Since $\mathcal{P}$ is a polyhedron, it is a convex region. For any $\boldsymbol{r} \in \mathcal{P}$, there exist real numbers $a_1, a_2, ..., a_m$ such that $\boldsymbol{r} = a_1\boldsymbol{v}_1 + a_2\boldsymbol{v}_2 + ... + a_m\boldsymbol{v}_m$ and $a_1 + a_2 + ... + a_m = 1$. Suppose that $\forall \boldsymbol{v} \in V$, $\boldsymbol{v} \in h^+_{i,j}$ (i.e., $\boldsymbol{v} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$), we have the following conclusion for any $\boldsymbol{r} \in \mathcal{P}$.

$$\begin{aligned} \boldsymbol{r} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) &= (\sum_{t=1}^{m} a_t \boldsymbol{v}_t) \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) \\ &= \sum_{t=1}^{m} a_t (\boldsymbol{v}_t \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j)) \\ &> 0 \end{aligned}$$

This means $\forall \boldsymbol{r} \in \mathcal{P}$, $\boldsymbol{r} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$. i.e., $\boldsymbol{r} \in h^+_{i,j}$. Similarly, if $\forall \boldsymbol{v} \in V$, $\boldsymbol{v} \in h^-_{i,j}$, then $\forall \boldsymbol{r} \in \mathcal{P}$, $\boldsymbol{r} \in h^-_{i,j}$.

If $\exists \boldsymbol{v} \in \mathcal{V}$, $\boldsymbol{v} \in h^+_{i,j}$ and $\exists \boldsymbol{v}' \in \mathcal{V}$, $\boldsymbol{v}' \in h^-_{i,j}$ (i.e., $\mathcal{P} \cap h^+_{i,j} \neq \emptyset$ and $\mathcal{P} \cap h^-_{i,j} \neq \emptyset$), $\mathcal{P}$ must intersect hyper-plane $h_{i,j}$. □

Proof of Lemma 3: Denote by $\mathcal{S}$ is the set of top-k points w.r.t. $\boldsymbol{u}$. Consider any $\boldsymbol{p}_i \in \mathcal{S}$ and $\boldsymbol{p}_j \in \mathcal{D} \setminus \mathcal{S}$. The utility of $\boldsymbol{p}_i$ must be higher than that of $\boldsymbol{p}_j$ w.r.t. $\boldsymbol{u}$, and thus, $\boldsymbol{u} \in h^+_{i,j}$. Because cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ is the intersection of half-spaces, i.e., $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}] \subseteq h^+_{i,j}$, we can conclude that $\boldsymbol{u} \in [l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$.

Since cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}] \subseteq h^+_{i,j}$, for any $\boldsymbol{p}_i \in \mathcal{S}$ and $\boldsymbol{p}_j \in \mathcal{D} \setminus \mathcal{S}$, the utility of $\boldsymbol{p}_i$ must be higher than that of $\boldsymbol{p}_j$ w.r.t. any $\boldsymbol{u}' \in [l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$. We can conclude that $\mathcal{S}$ is the set of top-k points w.r.t. any $\boldsymbol{u}' \in [l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$. □

Proof of Theorem 2: The length of the utility space is $\sqrt{2}$. After processing hyper-planes $h_{i,j}$ and $h_{i',j'}$, we can either (1) find the target cell with a certain set of top-k points or (2) reduce the length of $\mathcal{L}$ to no more than its half. Note that based on our schema, the selected hyper-planes are the boundaries of cells (i.e., endpoint $l_{\boldsymbol{u}}$ (resp. $e_{\boldsymbol{u}}$) of cell $[l_{\boldsymbol{u}}, e_{\boldsymbol{u}}]$ is the intersection $\wedge_{i,j}$ (resp. $\wedge_{i',j'}$) of $\mathcal{L}$ and hyper-plane $h_{i,j}$ (resp. $h_{i',j'}$)). Thus, when the length of $\mathcal{L}$ is reduced, cells are pruned as a whole. Suppose the *minimum* length of cell containing the user's utility vector is L_{min}. To reduce the length of the utility space to L_{min}, we need $x = 2\log_2 \frac{\sqrt{2}}{L_{min}}$ rounds since $\frac{\sqrt{2}}{2^{x/2}} = L_{min}$. Thus, we need to interact with a user for $O(\log_2 \frac{\sqrt{2}}{L_{min}})$ rounds. □

Proof of Lemma 4: Consider a point $\boldsymbol{p}_j$ that makes $\boldsymbol{p}_i$ lower its rank in $\mathcal{Q}$. The unscanned utility vectors must be in $h^-_{i,j}$, and thus, $\boldsymbol{p}_j$ must rank higher than $\boldsymbol{p}_i$ w.r.t. any unscanned utility vectors. If there are k points that make $\boldsymbol{p}_i$ lower its rank in $\mathcal{Q}$, there are k points that rank higher than $\boldsymbol{p}_i$ w.r.t. any unswept utility vectors. This means point $\boldsymbol{p}_i$ cannot be one of the top-k points w.r.t. any unswept utility vectors and will not affect the partition construction. □

Proof of Lemma 5: Assume that there are m partitions in the optimal case (the fewest partition case). Denote by $\Theta'_x = [l'_x, e'_x]$ and $\Theta_x = [l_x, r_x]$ the x-th partition in the optimal case and the x-th partition obtained by our algorithm, respectively. With a slight abuse of notations, let $\boldsymbol{u}' \prec \boldsymbol{u}$ represents utility vector $\boldsymbol{u}$ is not further from (0,1) than $\boldsymbol{u}$ is. Suppose that for any $x \in [1,m]$, $l'_x \prec l_x$ and $r'_x \prec r_x$. Since $r'_m = (1,0)$, we have $(1,0) = r'_m \prec r_m$. The number of partitions obtained by our algorithm will not be more than that of the optimal case. In the following, we show that for any $x \in [1,m]$, $l'_x \prec l_x$ and $r'_x \prec r_x$ with the help of mathematical induction.

Consider $x = 1$. Since $l'_1 = (0,1)$ and $l_1 = (0,1)$, we have $l'_1 = l_1$. For partition Θ_1, our algorithm ends it when $|S_1| = s-1$. This means partition Θ_1 is continuously extended as long as there is a set of s points which are among the top-k points w.r.t. any $\boldsymbol{u} \in \Theta_1$. Since there exists a set of s points that are among the top-k points w.r.t. any utility vector in $\Theta'_1 = [(0,1), r'_1]$, we have $r'_1 \prec r_1$.

Consider $x \geq 2$. Suppose $l'_{x-1} \prec l_{x-1}$ and $r'_{x-1} \prec r_{x-1}$ are true for the $(x-1)$-th partition. Since $l'_x = r'_{x-1}$ and $l_x = r_{x-1}$, we have $l'_x \prec l_x$, For partition Θ_x, our algorithm ends it when $|S_x| = s-1$. This means partition Θ_x is continuously extended as long as there exists a set of s points that are among the top-k points w.r.t. any utility vector in Θ_x. Since there exists a set of s points that are among the top-k points w.r.t. any utility vector in $\Theta'_x = [l'_x, r'_x]$, we have $r'_x \prec r_x$.

Now consider the time complexity. Let us first explore the number of partitions obtained by our algorithm. For any two nearby partitions Θ_x and Θ_{x+1}, we have $|S_x \cap S_{x+1}| < s$. Otherwise, there are s points which are the top-k points w.r.t. any utility vector $\boldsymbol{u} \in \Theta_x \cup \Theta_{x+1}$ and thus, Θ_x and Θ_{x+1} can be combined. Since $|S_x \cap S_{x+1}| < s$, there should be $k-s+1$ points in S_x whose ranks change with those of

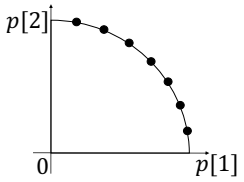


Fig. 29: Lower bound

the other $k-s+1$ points in S_{x+1}. The rank change includes $\frac{(k-s+1)^2}{2}$ intersections. This means the generation of a new partition needs to sweep at least $\frac{(k-s+1)^2}{2}$ intersections. Note that each point $\boldsymbol{p} \in \mathcal{D}$ will be deleted from $\mathcal{Q}$ if its ranking lowers down k times. Since there are n points, there are $O(nk)$ intersections considered. Thus, there should be $O(\frac{nk}{(k-s+1)^2})$ partitions generated by our algorithm.

During the sweep, when an intersection is popped from $\mathcal{H}$, there is a point lowering its rank in $\mathcal{Q}$. Each point $\boldsymbol{p} \in \mathcal{D}$ will be deleted from $\mathcal{Q}$ if its rank lowers k times (Lemma 4). Since there are n points, we need to process $O(nk)$ intersections. Upon visiting an intersection, we update $\mathcal{H}$ and $\mathcal{Q}$ in $O(\log n)$ and $O(1)$ time, respectively, and may need to delete a point from S in $O(1)$ time. The time complexity is $O(nk \log n)$. Besides, when we start a new partition, we need to build a new point set by inserting the first k points in $\mathcal{Q}$ into it. As discussed above, there are $O(\frac{nk}{(k-s+1)^2})$ partitions. The time complexity is $O(\frac{nk^2}{(k-s+1)^2})$. To sum up, the total time complexity of is $O(nk \log n + \frac{nk^2}{(k-s+1)^2})$. □

Proof of Theorem 3: Based on the proof of Lemma 5, there are $O(\frac{nk}{(k-s+1)^2})$ partitions obtained by our algorithm. Since the candidate partitions are reduced by half in each round, $|\boldsymbol{\Theta}|$ is reduced to 1 after $O(\log_2 \frac{nk}{(k-s+1)^2})$ rounds. Thus, algorithm 2DPI$^+$ solves the 2-dimensional ITQ by interacting with a user within $O(\log_2 \frac{nk}{(k-s+1)^2})$ rounds. □

Proof of Corollary 1: Theorem 1 claims a lower bound $\Omega(\log_2 \frac{n}{k-s+1})$ on the number of interactive rounds. Theorem 3 shows algorithm 2DPI$^+$ can find s of the user's top-k points within $O(\log_2 \frac{nk}{(k-s+1)^2})$ interactive rounds. For fixed k, algorithm 2DPI$^+$ is asymptotically optimal in terms of the interactive rounds. □

Proof of Lemma 6: For each point $\boldsymbol{p}_j \in \mathcal{D} \setminus \{\boldsymbol{p}_i\}$, if $\mathcal{R} \subseteq h^+_{i,j}$, we have $\forall \boldsymbol{u} \in \mathcal{R}, \boldsymbol{u} \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i) < 0$. Suppose that $|\{\boldsymbol{p}_j \in \mathcal{D} \setminus \{\boldsymbol{p}_i\} | \ \mathcal{R} \not\subseteq h^+_{i,j}\}| < k$. There must be more than $n-k$ points $\boldsymbol{p}_j \in \mathcal{D} \setminus \{\boldsymbol{p}_i\}$ such that $\forall \boldsymbol{u} \in \mathcal{R}, \boldsymbol{u} \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i) < 0$, i.e., there are more than $n-k$ points in $\mathcal{D} \setminus \{\boldsymbol{p}_i\}$ whose utility is smaller than that of $\boldsymbol{p}_i$ w.r.t. any utility point $\boldsymbol{u} \in \mathcal{R}$. Thus, $\boldsymbol{p}_i$ must be one of the top-k points w.r.t. any $\boldsymbol{u} \in \mathcal{R}$. □

Proof of Lemma 7: Let $\boldsymbol{v}_1, \boldsymbol{v}_2, ..., \boldsymbol{v}_m$ denotes the vertices of $\mathcal{R}$. Since $\mathcal{R}$ is a polyhedron, it is convex. For each utility vector $\boldsymbol{u}$, we have

$$\boldsymbol{u} = a_1 \boldsymbol{v}_1 + a_1 \boldsymbol{v}_1 + ... a_m \boldsymbol{v}_m,$$

where $a_1, a_2, ..., a_m$ are real numbers and $a_1 + a_2 + ... + a_m = 1$. Suppose that all the vertices correspond to the same set S of top-k points. For each $\boldsymbol{p}_i \in S$, we have $\forall l \in [1, m]$, $\boldsymbol{v}_l \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) > 0$, where $\boldsymbol{p}_j \in \mathcal{D} \setminus S$. This means that for any utility vector $\boldsymbol{u} \in \mathcal{R}$, there are $|\mathcal{D} \setminus S|$ points $\boldsymbol{p}_j$ such that

$$\begin{aligned} \boldsymbol{u} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) &= (\sum_{l=1}^{m} a_l \boldsymbol{v}_l) \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) \\ &= \sum_{l=1}^{m} a_l \boldsymbol{v}_l \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) \\ &> 0 \end{aligned}$$

, i.e., $\mathcal{R} \subseteq h^+_{i,j}$. In this case, since $|\mathcal{D} \setminus S| = n - k$, we have $|\{\boldsymbol{p}_j \in S \setminus \{\boldsymbol{p}_i\} | \ \mathcal{R} \not\subseteq h^+_{i,j}\}| < k$. □

Proof of Theorem 4: Since there are n points in $\mathcal{D}$, there are $O(n^2)$ hyper-planes. If each hyper-plane is used in one interactive round, based on Stopping Condition 1 (SC1), algorithm UNIFY solves the d-dimensional ITQ by interacting with a user for $O(n^2)$ rounds. □

Proof of Theorem 5: We consider that UNIFY(CH2) terminates if it fulfills Stopping Condition 1 (SC1), since Stopping Condition 2 (SC2) must be satisfied if SC1 is met.

Since $|\mathcal{D}| = n$ and each pair of points $\boldsymbol{p}_i, \boldsymbol{p}_j \in \mathcal{D}$ can build a hyper-plane, there are $\eta = n(n-1)/2$ hyper-planes $h_{i,j}$. For any x of hyper-planes, they can divide the utility space into disjoint polyhedrons, called regions. As analyzed by [1], [12], the utility space can be divided into at most

$$N_{d-1}(x) = C_x^0 + C_x^1 + ... + C_x^{d-1}$$

regions. In the following, we first assume that η hyper-planes divide the utility space into all $N_{d-1}(\eta)$ regions that are in equal size. Then, we consider the general case.

Suppose that all $N_{d-1}(\eta)$ regions are in equal size. Since the CH2 strategy initializes a random order of points to build hyperplane sets $\mathcal{H}_i$, the hyper-planes in the first $\upsilon - 1$ sets $\bigcup_{i=1}^{\upsilon} \mathcal{H}_i$ divide the utility space into $N_{d-1}(\mu)$ regions that are in equal size, as discussed in [1], [11], where $\mu = (\upsilon - 1)(\upsilon - 2)/2$.

Assume that none of the hyper-planes in sets H_2, ..., $H_{\upsilon-1}$ intersects utility range $\mathcal{R}$. $\mathcal{R}$ becomes one of the regions divided by the first μ hyperplanes. For each hyper-plane $h_{\upsilon,j} \in H_\upsilon$, if it intersects utility range $\mathcal{R}$, we need to ask the user a question and update $\mathcal{R}$ to be $\mathcal{R} \cap h^+_{\upsilon,j}$ or $\mathcal{R} \cap h^-_{\upsilon,j}$. Consider the relationship between $h_{\upsilon,j}$ and the $N_{d-1}(\mu)$ regions divided by the first μ hyper-planes $h_{i,j}$. Hyper-plane $h_{\upsilon,j}$ can be seen as being divided by the first μ hyperplanes $h_{i,j}$. This means there are $N_{d-2}(\mu)$ regions among the $N_{d-1}(\mu)$ regions (divided by the first μ hyperplanes $h_{i,j}$), which intersect $h_{\upsilon,j}$.

Note that the $N_{d-1}(\mu)$ regions divided by the first μ hyperplanes $h_{i,j}$ are in equal size and utility range $\mathcal{R}$ must be one of them before considering H_υ. For each $h_{\upsilon,j} \in H_\upsilon$, since it intersects with $N_{d-2}(\mu)$ regions divided by the first μ hyperplanes, the probability P_υ that $h_{\upsilon,j}$ intersects with

$\mathcal{R}$ (i.e., the probability of asking a question) is as follows.

$$\begin{aligned} P_\upsilon &= \frac{N_{d-2}(\mu)}{N_{d-1}(\mu)} \\ &= \frac{C_\mu^0 + C_\mu^1 + ... + C_\mu^{d-2}}{C_\mu^0 + C_\mu^1 + ... + C_\mu^{d-1}} \\ &\leq \frac{C_\mu^0 + C_\mu^1 + ... + C_\mu^{d-2}}{C_\mu^{d-2} + C_\mu^{d-1}} \\ &= \frac{\sum_{\varphi=0}^{d-2} C_\mu^\varphi}{C_{\mu+1}^{d-1}} \end{aligned}$$

If $2(d-2) \leq \mu$, then $C_\mu^\varphi \leq C_\mu^{d-2}$, where $\varphi = 0, 1, ..., d-3$. Define $1 \leq \alpha \leq d-1$ such that $\alpha C_\mu^{d-2} \geq \sum_{\varphi=0}^{d-2} C_\mu^\varphi$. Then we have

$$P_\upsilon \leq \frac{\alpha C_\mu^{d-2}}{C_{\mu+1}^{d-1}} = \frac{\alpha(d-1)}{\mu+1} \leq \frac{2\alpha(d-1)}{(\upsilon-2)^2}$$

If $2(d-2) > \mu$, the relation between C_μ^φ and C_μ^{d-2} varies, where $\varphi = 0, 1, ..., d-3$. For ease of calculation, we define $P_\upsilon = 1$.

Since $\mu = \frac{(\upsilon-1)(\upsilon-2)}{2}$ and $k \geq 1$, we define

$$P_\upsilon = \begin{cases} 1 & \upsilon \leq 8\sqrt{kd} \\ \frac{2\alpha(d-1)}{(\upsilon-2)^2} & \upsilon > 8\sqrt{kd} \end{cases}$$

Because the probability that a hyperplane in H_υ intersects with the utility range (i.e., the probability of asking a question) is $\boldsymbol{p}_\upsilon$, the expected number of questions asked a user is $\sum_{\upsilon=2}^{n} \sum_{j=1}^{\upsilon-1} P_\upsilon$.

$$\begin{aligned} \sum_{\upsilon=2}^{n} \sum_{j=1}^{\upsilon-1} P_\upsilon &= \sum_{\upsilon=2}^{\lceil 8\sqrt{kd} \rceil} \sum_{j=1}^{\upsilon-1} P_\upsilon + \sum_{\upsilon=\lceil 8\sqrt{kd} \rceil+1}^{n} \sum_{j=1}^{\upsilon-1} P_\upsilon \\ &\leq \frac{(\lceil 8\sqrt{kd} \rceil)(\lceil 8\sqrt{kd} \rceil - 1)}{2} + \\ &\quad \sum_{\upsilon=\lceil 8\sqrt{kd} \rceil+1}^{n} \sum_{j=1}^{\upsilon-1} \frac{2\alpha(d-1)}{(\upsilon-2)^2} \\ &\leq \frac{(8\sqrt{kd})(8\sqrt{kd})}{2} + \\ &\quad \sum_{\upsilon=\lceil 8\sqrt{kd} \rceil+1}^{n} \frac{2\alpha(d-1)}{(\upsilon-2)^2} + \frac{2\alpha(d-1)}{(\upsilon-2)^2-1} \\ &\quad + \frac{2\alpha(d-1)}{(\upsilon-2)^2-2} + ... + \frac{2\alpha(d-1)}{(\upsilon-2)(\upsilon-3)} \\ &\leq 32kd + \sum_{i=(8\sqrt{kd}-2)^2}^{(n-2)^2} \frac{2\alpha(d-1)}{i} \\ &\leq 32kd \log_2((8\sqrt{kd}-2)^2 - 1) + \\ &\quad 2\alpha(d-1) \log_2(\frac{(n-2)^2}{(8\sqrt{kd}-2)^2 - 1}) \\ &\leq 32\alpha kd \log_2(n-2)^2 \end{aligned}$$

Thus, if the cells divided by the η hyperplanes $h_{i,j}$ are in equal size, the expected number of questions asked is $\mathbb{E}_u = O(kd \log_2 n)$.

Consider the general case that all the cells are in random size. If there are fewer than $N_{d-1}(\mu)$ cells, we can consider the size of the missing cells as 0. Let $\mathbf{P}_i$ denote the size ratio of the i-th cell to the whole utility space. Use $\mathcal{N}_i$ and $\mathbb{E}[\mathcal{N}_i]$ to represent the number of questions asked and the expected number of questions asked to locate the i-th cell if the user's utility vector is in the i-th cell. Denote by $\mathbb{E}_g$ the expected number of questions asked in the general case. We have

$$\mathbb{E}_g = \sum_{i=1}^{N_{d-1}(\eta)} \mathbb{E}[\mathcal{N}_i]$$

, where $\eta = n(n-1)/2$. Assume $\mathbf{P}_i$ is bounded by $\mathbf{P}_i \leq \frac{c}{N_{d-1}(\eta)}$, for some constant $c > 1$. In order to obtain the largest $\mathbb{E}_g$, we distribute the probability $\frac{c}{N_{d-1}(\eta)}$ to $j = \frac{N_{d-1}(\eta)}{c}$ cells whose $\mathbb{E}[\mathcal{N}_i]$ is the largest. Without loss of generality, set $\mathbb{E}[\mathcal{N}_i] = \rho$ for these particular cells (with the largest $\mathbb{E}[\mathcal{N}_i]$) and then the largest $\mathbb{E}_g$ should be ρ. For the remaining $N_{d-1}(\eta) - j$ cells, each cell should be bounded by at least d hyperplanes. Since one question (showing points $\boldsymbol{p}_i$ and $\boldsymbol{p}_j$ to the user) is needed for each bounded hyperplane $h_{i,j}$, the number of questions asked for these cells must be larger than d. Therefore, we have

$$\mathbb{E}_u = \frac{1}{N_{d-1}(\eta)} \sum_{i=1}^{N_{d-1}(\eta)} \mathbb{E}[\mathcal{N}_i] \geq \frac{\rho}{N_{d-1}(\eta)} j + d\frac{N_{d-1}(\eta) - j}{N_{d-1}(\eta)}$$

Then we obtain the largest $\mathbb{E}_g$

$$\mathbb{E}_g = \rho \leq c(\mathbb{E}_u - d\frac{N_{d-1}(\eta) - j}{N_{d-1}(\eta)}) \leq c\mathbb{E}_u$$

Therefore, in the general case, the expected number of questions asked is $\mathbb{E}_g = c\mathbb{E}_u = O(ckd \log_2 n)$, where $c > 1$ is a constant. □

Proof of Corollary 2: Theorem 1 claims a lower bound $\Omega(\log_2 \frac{n}{k-s+1})$ on the number of interactive rounds. Theorem 4 shows algorithm UNIFY(CH2) can find s of the user's top-k points within $O(kd \log_2 n)$ interactive rounds in expectation. If k and d are fixed, algorithm UNIFY(CH2) is asymptotically optimal in terms of the interactive rounds in expectation. □

Proof of Lemma 8: Consider any utility vector $\boldsymbol{u}$ such that $f_{\boldsymbol{u}}(\boldsymbol{p}_i) > f_{\boldsymbol{u}}(\boldsymbol{p}_j)$, we have $P_{\boldsymbol{u}}(\boldsymbol{p}_i \succ \boldsymbol{p}_j) > 0.5 > 1 - \epsilon$.

Consider any utility vector $\boldsymbol{u}$ such that $f_{\boldsymbol{u}}(\boldsymbol{p}_i) < f_{\boldsymbol{u}}(\boldsymbol{p}_j)$. Let $h_{i,j}^{\epsilon+} = \{\boldsymbol{r} \in \mathbb{R}^d | \ \boldsymbol{r} \cdot (\boldsymbol{p}_i - \boldsymbol{p}_j) - X > 0\}$.

$$\begin{aligned} & P(\boldsymbol{p}_i \succ \boldsymbol{p}_j) \geq 1 - \epsilon \\ \Leftrightarrow \quad & \frac{1}{1 + exp^{-|f_{\boldsymbol{u}}(\boldsymbol{p}_j) - f_{\boldsymbol{u}}(\boldsymbol{p}_i)|}} \leq \epsilon \\ \Leftrightarrow \quad & \frac{1}{1 + exp^{-|\boldsymbol{u} \cdot (\boldsymbol{p}_j - \boldsymbol{p}_i) - X|}} \leq \epsilon \\ \Leftrightarrow \quad & \frac{1}{1 + exp^{-|-X|}} \leq \epsilon \\ \Leftrightarrow \quad & -X \geq \ln \frac{1-\epsilon}{\epsilon} \\ \Leftrightarrow \quad & X \leq \ln \frac{\epsilon}{1-\epsilon} \end{aligned}$$

□