Answering Why-not Questions on Reverse Top-*k* Queries

Yunjun Gao[†,*] Qing Liu[†] Gang Chen[†] Baihua Zheng[‡] Linlin Zhou[†]

[†]College of Computer Science, Zhejiang University, Hangzhou, China
[*]Innovation Joint Research Center for Cyber-Physical-Society System, Zhejiang University, Hangzhou, China
[‡]School of Information Systems, Singapore Management University, Singapore
[†]{gaoyj, liuq, cg, zlinlin}@zju.edu.cn [‡]bhzheng@smu.edu.sg

ABSTRACT

Why-not questions, which aim to seek clarifications on the missing tuples for query results, have recently received considerable attention from the database community. In this paper, we systematically explore *why-not questions on reverse top-k queries*, owing to its importance in multi-criteria decision making. Given an initial reverse top-*k* query and a missing/why-not weighting vector set W_m that is absent from the query result, why-not questions on reverse top-*k* queries explain why W_m does not appear in the query result and provide suggestions on how to refine the initial query with minimum penalty to include W_m in the refined query result. We first formalize why-not questions on reverse top-*k* queries and reveal their semantics, and then propose a *unified framework* called *WQRTQ* to answer why-not questions on both *monochromatic* and *bichromatic* reverse top-*k* queries. Our framework offers three solutions, namely, (i) modifying a query point *q*, (ii) modifying a why-not weighting vector set W_m and a parameter *k*, and (iii) modifying *q*, W_m, and *k* simultaneously, to cater for different application scenarios. Extensive experimental evaluation using both real and synthetic data sets verifies the effectiveness and efficiency of the presented algorithms.

1. INTRODUCTION

In the past decades, the capability of database has been significantly improved, which enables us to process a variety of complex queries on heterogeneous and humongous datasets. However, the usability of database is far from meeting user needs. As pointed out by Jagadish et al. [22], failing to produce expected results without any explanation is one of the pain points of current database systems that frustrate many users. If a user encounters such cases, intuitively, he/she may pose a *why-not question* to find out *why* his/her expected tuples *do not* appear in the query result. If the database system can provide such clarifications, it helps the users understand initial query better and know how to change the query, hence improving the usability of database.

Since the concept of "*why-not*" was first proposed by Chapman and Jagadish [8], many efforts have been made to answer why-not questions on different queries. Existing work can be classified into three categories. The first category finds the manipulations which are responsible for excluding users' desired tuples. The typical examples include answering users' why-not questions on Select-Project-Join (SPJ) queries [8] and Select-Project-Join-Union-Aggregation (SPJUA) queries [5]. The second category provides a set of data modifications (e.g., insertion, update, etc.) so that the missing tuples can present in the query result. This category also mainly focuses on SPJ queries [20, 38] and SPJUA queries [16, 17]. The third category revises the initial query to generate a refined query whose result contains the user specified missing tuples. Why-not questions on Select-Project-Join-Aggregation (SPJA) queries [30], top-*k* queries [14], reverse skyline queries [21], and image search [4] all belong to this category. Nonetheless, why-not questions are query-dependent, and none of existing work can answer why-not questions on reverse top-*k* queries, which is an important and essential building block for multi-criteria decision making. Therefore, in this paper, we study the problem of *answering why-not questions on reverse top-k queries* by following the third category.

computer	p_1 Dell	p_2 HP	p_3 Sony	p_4 IBM	p_5 Acer	p_6 ASUS	p_7 NEC	q Apple
price	2	6	1	9	7	5	3	4
heat	1	3	9	3	5	8	7	4

(a) The information of computers

costomer	w[*price*]	w[*heat*]
Kevin ($\vec{w}_1$)	0.1	0.9
Anna ($\vec{w}_2$)	0.3	0.7
Tony ($\vec{w}_3$)	0.5	0.5
Julia ($\vec{w}_4$)	0.9	0.1

(b) The customer preferences

computer *id*	p_1	p_2	p_3	p_4	p_5	p_6	p_7	q
score for Kevin	1.1	3.3	8.2	3.6	5.2	7.7	6.6	4
score for Anna	1.3	3.9	6.6	4.8	5.6	7.1	5.8	4
score for Tony	1.5	4.5	5	6	6	6.5	5	4
score for Julia	1.9	5.7	1.8	8.4	6.8	5.3	3.4	4

(c) The scores of computers

Figure 1: Example of reverse top-*k* queries

Before presenting the reverse top-*k* query, we first introduce the top-*k* query. Given a dataset *P*, a positive integer *k*, and a preference function *f*, a top-*k* query retrieves the *k* points in *P* with the *best scores* based on *f*. The points returned by the top-*k* query match users' preferences best and help users to avoid receiving an overwhelming result set. Based on the top-*k* query, Vlachou et al. [31] propose the reverse top-*k* query from the manufacturers' perspective, which has a wide range of applications such as market analysis [24, 31, 33, 34] and location-based services [32]. Given a dataset *P*, a positive integer *k*, a preference function set *W* (in terms of weighting vectors), and a query point *q*, a reverse top-*k* query returns the preference functions in *W* whose top-*k* query results contain *q*. Figure 1 illustrates an example of reverse top-*k* queries. Figure 1(a) records the price and heat production for each computer brand (e.g., Apple, DELL, etc.), and Figure 1(b) lists the customer preferences in terms of weighting vectors by assigning a weight to every attribute. Without loss of generality, we adopt a linear preference function, i.e., $f(\vec{w}, p) = w[heat] \times p.heat + w[price] \times p.price$, to compute the score of a point *p* w.r.t. a weighting vector $\vec{w}$. Figure 1(c) depicts the score of every computer for different customers, and we assume that smaller values are more preferable. Based on Figure 1(c), if Apple issues a reverse top-3 ($k = 3$) query at a

Proceedings of the VLDB Endowment, Vol. 8, No. 7

query point/computer q, Anna and Tony are retrieved as they rank the query computer q as one of their top-3 options. In other words, reverse top-k queries can help Apple to identify the potential customers who are more likely to be interested in its product(s), and thus to assess the impact of product(s) in the market.

Unfortunately, reverse top-k queries only return query results to users *without any explanation*. If the query result does not contain some expected tuples, it may disappoint users. Consider the aforementioned example again. Suppose Kevin and Julia are Apple's existing customers, however, they are not in the result of the reverse top-3 query of q. Apple may feel frustrated and ask "*Why Kevin and Julia do not take Apple as one of their choices? What actions should be taken to win them back?*" If the database system can offer such clarifications, it will help Apple to retain existing customers as well as to attract more new customers, and hence to increase/maintain its market share. In view of this, for the first time, we explore why-not questions on reverse top-k queries, which could be an important and useful tool for market analysis. Given an original reverse top-k query and a why-not weighting vector set W_m that is missing from the query result, why-not questions on reverse top-k queries explain why W_m is not in the query result, and suggest how to refine the original query with minimum penalty to include W_m in the refined query result.

In order to win back missing customers, Apple might (i) change the computer's parameters; (ii) influence and convince the customers to change their preferences; and (iii) change both the computer's parameters and the customers' preferences. Correspondingly, in this paper, we develop a *unified framework* called WQRTQ, which provides three solutions to cater for different application scenarios, to answer why-not questions on reverse top-k queries. Specifically, the first solution is to modify a query point q using the *quadratic programming*. The second solution is a *sampling based method*, which modifies a weighting vector set W_m and a parameter k. The third solution is to modify q, W_m, and k simultaneously, which integrates the *quadratic programming*, *sampling method*, and *reuse technique*. It is worth mentioning that all three solutions can return the refined query with minimum penalty, and can support why-not questions on both *monochromatic* and *bichromatic* reverse top-k queries. Extensive experiments using both real and synthetic datasets show that our proposed algorithms can produce clarifications and suggest changes efficiently. To sum up, the key contributions of this paper are summarized as follows.

- We solve why-not questions on reverse top-k queries. To our knowledge, there is no prior work on this problem.
- We present a unified framework WQRTQ, including three different approaches, to answer why-not questions on both monochromatic and bichromatic reverse top-k queries.
- We conduct extensive experiments with both real and synthetic datasets to demonstrate the effectiveness and efficiency of our proposed algorithms.

The rest of this paper is organized as follows. Section 2 reviews related work. Section 3 presents problem formulation. Section 4 describes our framework and solutions to answer why-not questions on reverse top-k queries. Section 5 reports experimental results and our findings. Finally, Section 6 concludes the paper with some directions for future work.

2. RELATED WORK

In this Section, we review previous work on *top-k queries*, *reverse top-k queries*, *data provenance*, and *why-not questions*.

Top-k queries. Top-k query has received much attention in the database community due to its usefulness. Existing algorithms include convex hull based algorithm *Onion* [7], view based algorithms *LPTA* [11] and *PREFER* [18, 19], layered index based algorithm *AppIR* [36], branch-and-bound algorithm *BRS* [29], dominant graph based top-k query algorithm [39], and top-k query algorithms using cache [35]. It is worth mentioning that, *BRS* is I/O optimal.

Reverse top-k queries. Vlachou et al. [31] firstly introduce the reverse top-k query and consider its two variants, namely, monochromatic and bichromatic versions. To efficiently answer the monochromatic reverse top-k query, Vlachou et al. [31] and Chester et al. [9] present several algorithms in a 2-dimensional (2D) space. The bichromatic top-k query algorithms include *RTA*, *GRTA*, and *BBR* [31, 34]. In addition, Yu et al. [37] develop a dynamic index to support reverse top-k queries, and Ge et al. [12] employ *all top-k queries* to boost the reverse top-k query. More recently, reverse top-k queries are widely studied in market analysis [24, 33], location-based services [32], and uncertain circumstances [23]. It is worth noting that, all the current reverse top-k queries only return the results without any explanation, and thus, the existing techniques designed for reverse top-k queries cannot answer corresponding why-not questions efficiently.

Data provenance. Data provenance explores the derivation of a piece of data that is in a query result [28]. It can help users understand why data tuples exist within a result set. Current approaches for computing data provenance include non-annotation method [10] and annotation approach [3]. Nonetheless, it cannot be applied to clarify the missing tuples in the query result set.

Why-not questions. Chapman and Jagadish [8] first propose the concept of "*why-not*". Since then, lots of efforts have been put into answering why-not questions. The existing approaches can be classified into three categories: (i) *manipulation identification* [5, 8], (ii) *database modification* [16, 17, 20, 38], and (iii) *query refinement* [4, 14, 21, 30]. In addition, Herschel [15] tries to identify hybrid why-not explanations for SQL queries, which combines manipulation identification and query refinement. Meliou et al. [25] aim to find the causality and responsibility for the non-answers of the query. Here, causality is the cause of non-answers to the query, and responsibility captures the notion of degree of causality.

It is noteworthy that our work follows the query refinement model to answer why-not questions on reverse top-k queries, i.e., we modify the parameter(s) and/or a query point and/or why-not point(s) of an original query to include the missing tuples in a refined query result. However, since why-not questions are *query-dependent*, different queries require different query refinement, which explains why existing query refinement techniques cannot be applied directly in our problem and justifies our main contribution, that is to design proper query refinement approaches to support why-not questions on reverse top-k queries.

3. PROBLEM FORMULATION

In this section, we formalize why-not questions on reverse top-k queries, including both monochromatic reverse top-k queries and bichromatic reverse top-k queries. Given a d-dimensional dataset P, a point $p \in P$ is represented in the form of d-tuple vector $\{p[1], \ldots, p[d]\}$, where $p[i]$ is the i-th dimensional value of p. The top-k query ranks/orders the points based on a user specified scoring function f that aggregates the individual score of a point

into an overall scoring value. In this paper, we utilize a *linear scoring function* (or *weighted sum function*) as with [14, 31, 34]. Specifically, in a data space, each dimension i is assigned a weight $w[i]$ indicating the relative importance of the i-th dimension for the query, captured by a weighting vector $\vec{w} = \{w[1], \ldots, w[d]\}$ in which $w[i] \geq 0$ $(1 \leq i \leq d)$ and $\sum_{i=1}^{d} w[i] = 1$. Then, $f(\vec{w}, p) = \sum_{i=1}^{d} w[i] \times p[i]$ captures the aggregated score of any data point p $(\in P)$ with respect to $\vec{w}$. Without loss of generality, we assume that *smaller* scoring values are *preferable* in this paper. Next, we formally define the top-k query below.

Definition 1 (Top-k Query). *Given a d-dimensional dataset P, a positive integer k, and a weighting vector $\vec{w}$, a* top-k query *returns a set of points, denoted as $TOPk(\vec{w})$, such that (i) $TOPk(\vec{w}) \subseteq P$; (ii) $|TOPk(\vec{w})| = k$; and (iii) $\forall p_1 \in TOPk(\vec{w})$, $\forall p_2 \in P - TOPk(\vec{w})$, it holds that $f(\vec{w}, p_1) \leq f(\vec{w}, p_2)$.*

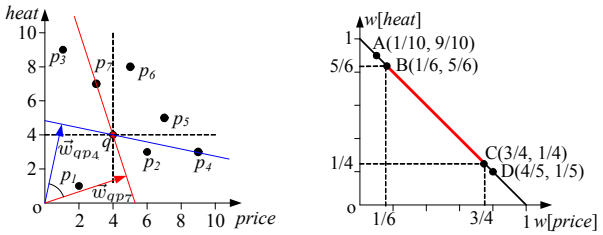
Take the dataset P shown in Figure 1 as an example. We have $TOP3(\vec{w}_1) = \{p_1, p_2, p_4\}$. It is worth mentioning that, if the points share the same score at ranking k-th, only one of them is randomly returned. Based on the definition of the top-k query, we formulate reverse top-k queries, for both monochromatic version and bichromatic version by following [31].

Definition 2 (Monochromatic Reverse Top-k Query) [31]. *Given a d-dimensional dataset P, a positive integer k, and a query point q, a* monochromatic reverse top-k (MRTOPk) query *retrieves a collection of d-dimensional weighting vectors, denoted as $MRTOPk(q)$, such that $\forall \vec{w}_i \in MRTOPk(q)$, it holds that $\exists p \in TOPk(\vec{w}_i), f(\vec{w}_i, q) \leq f(\vec{w}_i, p)$.*

In other words, a MRTOPk query returns all the weighting vectors whose top-k query results include q. For example, Figure 2(a) is the corresponding data distribution of Figure 1(a) without considering the specified customer preferences. As observed, q is inside the top-3 query result for a weighting vector $\vec{w}_{qp4}$(1/6, 5/6) since only p_1 and p_2 have smaller scoring values than q w.r.t. $\vec{w}_{qp4}$. Similarly, q is also within the top-3 query result for a weighting vector $\vec{w}_{qp7}$ (3/4, 1/4). Hence, $\vec{w}_{qp4}$ and $\vec{w}_{qp7}$ are located in $MRTOP3(q)$. Actually, all the weighting vectors with the angles between $\vec{w}_{qp4}$ and $\vec{w}_{qp7}$ (i.e., the segment BC in Figure 2(b)) belong to $MRTOP3(q)$. Different from the MRTOPk query, the bichromatic reverse top-k query takes two datasets into consideration, which is formalized as follows.

Definition 3 (Bichromatic Reverse Top-k Query) [31]. *Given a d-dimensional dataset P, a d-dimensional weighting vector set W, a query point q, and a positive integer k, a* bichromatic reverse top-k (BRTOPk) query *retrieves a set of weighting vectors, denoted as $BRTOPk(q)$, such that (i) $BRTOPk(q) \subseteq W$, and (ii) $\forall \vec{w}_i \in BRTOPk(q)$, it holds that $\exists p \in TOPk(\vec{w}_i), f(\vec{w}_i, q) \leq f(\vec{w}_i, p)$.*

A BRTOPk query finds the weighting vectors in W whose top-k query results contain q. Back to Figure 1 again. As $TOP3(\vec{w}_2) = \{p_1, p_2, q\}$, $\vec{w}_2$ belongs to $BRTOP3(q)$. Finally, we can obtain $BRTOP3(q) = \{\vec{w}_2, \vec{w}_3\}$. It is worth noting that, the only difference between BRTOPk and MRTOPk queries is that the former has the knowledge of user preferences, whereas the latter does not. As mentioned earlier, why-not questions are query-dependent, and thus, the solutions for why-not questions on different queries are usually different. Consequently, based on Definition 2 and Definition 3, we formulate why-not questions on MRTOPk and BRTOPk queries, respectively.



(a) The dataset distribution (b) The weighting space

Figure 2: Example of a monochromatic reverse top-k query

Definition 4 (Why-not Questions on MRTOPk Queries). *Given an original MRTOPk query of a query point q on a dataset P, and a why-not/missing weighting vector set W_m that is excluded from $MRTOPk(q)$,* why-not questions on MRTOPk queries (WQMQ) *answer (i) why $\forall \vec{w}_i \in W_m$, $\vec{w}_i \notin MRTOPk(q)$; and (ii) how to refine the initial MRTOPk query with minimum penalty such that $\forall \vec{w}_i \in W_m$, $\vec{w}_i \in MRTOPk(q)$.*

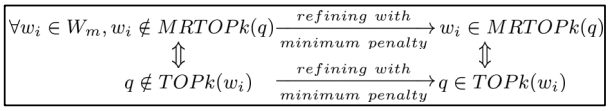
Definition 5 (Why-not Questions on BRTOPk Queries). *Given an original BRTOPk query of a query point q on a dataset P and a weighting vector set W, and a why-not/missing weighting vector set $W_m \subseteq W - BRTOPk(q)$,* why-not questions on BRTOPk queries (WQBQ) *answer (i) why $\forall \vec{w}_i \in W_m$, $\vec{w}_i \notin BRTOPk(q)$; and (ii) how to revise the initial* BRTOPk *query with minimum penalty such that $\forall \vec{w}_i \in W_m$, $\vec{w}_i \in BRTOPk(q)$.*

Note that, why-not weighting vectors for WQMQ might be any weighting vector that is not inside $MRTOPk(q)$, while why-not weighting vectors for WQBQ only come from $W - BRTOPk(q)$. For instance, in Figure 2(b), weighting vectors (1/10, 9/10) and (4/5, 1/5) do not appear in $MRTOP3(q)$, and hence can be used as why-not weighting vectors for why-not questions on *MRTOP3* queries. On the other hand, in Figure 1, $W - BRTOP3(q) = \{\vec{w}_1, \vec{w}_4\}$, thus, we can issue why-not questions on *BRTOP3* queries only using $\vec{w}_1$ and $\vec{w}_4$.

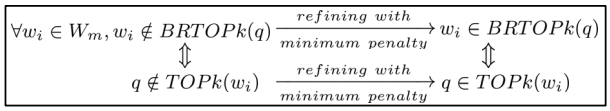
Based on Definition 4 and Definition 5, we need to answer why-not questions on MRTOPk/BRTOPk queries from two aspects, i.e., (i) giving the explanations of why why-not weighting vectors do not appear in the results of MRTOPk/BRTOPk queries, and (ii) providing the suggestions on how to refine the original MRTOPk/BRTOPk queries with minimum penalty for including the why-not weighting vectors.

For the first aspect, if a why-not weighting vector $\vec{w}$ does not present in the result of the MRTOPk/BRTOPk query, there must be more than k points whose scores are smaller than that of q. All those points are responsible for excluding the why-not weighting vector $\vec{w}$ from the query result. Hence, they form the answer for the first aspect. For example, for $\vec{w}_1$ in Figure 1, there are three points, i.e., p_1, p_2, and p_4, with scores smaller than that of q, and thus, $\vec{w}_1$ is not inside the reverse top-3 query result. It is not hard to derive the first aspect of answering why-not questions on MRTOPk/BRTOPk queries as we only need to issue a top-k query for every missing why-not weighting vector. We can use existing progressive top-k query algorithms [19, 29, 39] or all top-k query algorithms [12], which can report incrementally every ranking object one-by-one. The process proceeds until the query point q is contained in the result, and then returns the result to users.

The second aspect of answering why-not questions is to revise the original queries with minimum penalties such that the refined



(a) why-not questions on MRTOP*k* queries



(b) why-not questions on BRTOP*k* queries

Figure 3: Illustrations of why-not questions

query results include the missing why-not weighting vector(s). We find that the essence of the second aspect of answering why-not questions on MRTOP*k* and BRTOP*k* queries is identical. For why-not questions on MRTOP*k* queries, the target is to make every why-not weighting vector appear in $MRTOPk(q)$, i.e., $\forall \vec{w}_i \in W_m$, $\vec{w}_i \in MRTOPk(q)$. Based on Definition 2, $\vec{w} \in MRTOPk(q) \rightarrow q \in TOPk(\vec{w}_i)$ and $\vec{w} \notin MRTOPk(q) \rightarrow q \notin TOPk(\vec{w}_i)$. Hence, why-not questions on MRTOP*k* queries can be re-phrased as: for each why-not weighting vector $\vec{w}_i$ with $q \notin TOPk(\vec{w}_i)$, how to refine the original query with minimum penalty such that $q \in TOPk(\vec{w}_i)$, as shown in Figure 3(a). Similarly, according to Definition 3, we can also re-phrase why-not questions on BRTOP*k* queries, as depicted in Figure 3(b). From Figure 3, it is observed that, these two problems can be transformed to a single problem, i.e., $\forall \vec{w}_i \in W_m$ having $q \notin TOPk(\vec{w}_i)$, how to refine the original query with minimum penalty such that $q \in TOPk(\vec{w}_i)$.

It is worth mentioning that the transformed problem is inherently different from the problem of why-not questions on top-*k* queries [14], i.e., given a why-not point set $P_m \subseteq P$ and a weighting vector $\vec{w}$ having $\forall p_i \in P_m$, $p_i \notin TOPk(\vec{w})$, how to refine the original query with minimum penalty such that $p_i \in TOPk(\vec{w})$. The difference is two-fold. First, these two problems have totally different inputs. The inputs of our problem contain a why-not weighting vector set that captures the preferences of customers and a query point q representing a product of the manufacturer, while why-not questions on top-*k* queries take as inputs a why-not point set that denotes the attributes of products and a weighting vector representing a customer preference. Second, they serve different purposes. Our problem tries to make the product q as one of the top-*k* choices for the set of a given customer preferences, but why-not questions on top-*k* queries try to make all the specified products appear in the top-*k* result of a given weighting vector.

A straightforward way to tackle our problem is to take q as a why-not point and take a why-not weighting vector as a specified weighting vector, and then use the algorithms for why-not questions on top-*k* queries to refine the query. After solving all the why-not weighting vectors, q can be contained in the top-*k* result of every why-not weighting vector. Nevertheless, although the penalty of each refining is minimized, the total penalty of this method might not be the minimum. This is because the refining of the query needs to modify why-not weighting vectors, which has an impact on each other, and thus, it cannot be refined separately. Therefore, the algorithms for why-not questions on top-*k* queries cannot be applied to handle our problem. Based on the above analysis, we develop a *unified framework* to answer why-not questions on both MRTOP*k* and BRTOP*k* queries. Moreover, this paper focuses *only* on the second aspect of answering why-not questions as it is computationally challenging.

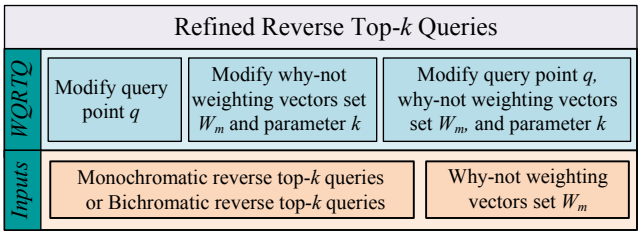


Figure 4: Framework of WQRTQ

4. ANSWERING WHY-NOT QUESTIONS

In this section, we propose a unified framework to answer why-not questions on reverse top-*k* queries, and then detail the framework, which contains three solutions based on the modification of different parameters.

4.1 Framework Overview

Based on the analysis performed in Section 3, we present a unified framework called WQRTQ (i.e., **W**hy-not **Q**uestions on **R**everse **T**op-*k* **Q**ueries), which can answer why-not questions on both monochromatic and bichromatic reverse top-*k* queries. As illustrated in Figure 4, WQRTQ takes as inputs an original monochromatic/bichromatic reverse top-*k* query and the corresponding why-not weighting vector set W_m, and returns the refined reverse top-*k* query with minimum penalty to users. Specifically, it consists of the following three solutions:

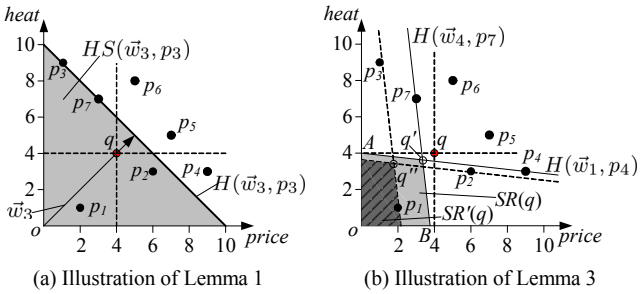
(1) **Modifying *q***. The first solution is to modify a query point q into q', which is to be detailed in Section 4.2. To this end, we introduce the concept of *safe region* (see Definition 7). As long as the query point q' falls into the safe region, the why-not weighting vector set W_m will appear in the reverse top-*k* query result of q'. After getting the safe region, we use the quadratic programming to get q' with the minimum change w.r.t. q.

(2) **Modifying W_m and *k***. The second solution, to be presented in Section 4.3, is to modify a why-not weighting vector set W_m and a parameter k into W_m' and k' respectively, such that the modified W_m' belongs to the result of the reverse top-k' query of q. Towards this, we present a sampling-based method to obtain W_m' and k' having the minimum *penalty*. In particular, we sample a certain number of weighting vectors that may contribute to the final result, and then, the optimal W_m' and k' are returned according to the sample weighting vectors.

(3) **Modifying *q*, W_m, and *k***. Our third solution is to modify a query point q, a why-not weighting vector set W_m, and a parameter k simultaneously, as to be detailed in Section 4.4. After refining, the modified weighting vector set W_m' is contained in the reverse top-k' query result of q'. This solution utilizes the techniques of *quadratic programming*, *sampling method*, and *reuse*. To be more specific, we first fix the range of a query point and sample a certain number of query points. Then, for every sample query point, we employ the second solution to get corresponding optimal (W_m', k'). Finally, the tuple (q', W_m', k') with the smallest penalty is returned.

4.2 Modifying *q*

Intuitively, if Apple finds some customers are not interested in its new computer, it can adjust some computer parameters before putting it into production so that the modified computer becomes one of customers' top-*k* options. In view of this, we propose the



(a) Illustration of Lemma 1 (b) Illustration of Lemma 3

Figure 5: Example of Lemma 1 and Lemma 3

first solution to refine the original reverse top-k query, namely, modifying a query point q, as formally defined below.

DEFINITION 6 (MODIFYING q). *Given a d-dimensional dataset P, a positive integer k, a query point q, and a why-not weighting vector set W_m with $\forall \vec{w}_i \in W_m$, $q \notin TOPk(\vec{w}_i)$, the* modification of a query point *q is to find q' such that* (*i*) $\forall \vec{w}_i \in W_m$, $q' \in TOPk(\vec{w}_i)$; (*ii*) $\forall i \in [1, d]$, $q'[i] \leq q[i]$; *and* (*iii*) *the penalty of q', defined in Equation* (1), *is minimum.*

$$Penalty(q') = \frac{|q - q'|}{|q|} = \frac{\sqrt{\sum_{i=1}^{d} (q[i] - q'[i])^2}}{|q|} \quad (1)$$

It is worth mentioning that, we use Equation (1) to quantify the modification of the product, which is also employed by Padmanabhan et al. [27] to measure quality distortation for the upgraded product. For example, in Figure 1, Kevin and Julia are not in the reverse top-3 result of q. If Apple modifies computer's parameter q(4, 4) to q'(3, 2.5) or q''(2.5, 3.5), the new computer q' or q'' becomes one of the top-3 options for both Kevin and Julia. According to Definition 6, q'' is more preferable as *Penalty*(q') = 0.318 > *Penalty*(q'') = 0.279. Note we only consider decreasing $q[i]$'s value. Since (i) the scoring function is monotonic, (ii) a smaller scoring value is ranked higher, and (iii) a smaller penalty is preferable, there is no need to increase $q[i]$'s value. As an example, assume that q(4, 4) in Figure 1 is modified to q'(5, 1). We can always find another query point (e.g., q''(4, 1) in this case) that has smaller penalty and meanwhile generate smaller scoring value. In other words, the search space for q' can be shrunk to [0, q]. Furthermore, to get a qualified q', we find that it is possible to locate a region within [0, q], namely, q's *safe region*, within which the modified query point q' definitely falls.

DEFINITION 7 (SAFE REGION). *Given a d-dimensional dataset P, a positive integer k, a query point q, and a why-not weighting vector set W_m, a region in the data space is said to be safe for q* (*i.e.*, *q's* safe region), *denoted as $SR(q)$, such that $\forall q' \in SR(q)$ and $\forall \vec{w}_i \in W_m$, $q' \in TOPk(\vec{w}_i)$.*

In other words, if q is modified to q' by moving the query point q anywhere within $SR(q)$, all the why-not weighting vectors will appear in a given reverse top-k query result. It is straightforward that, if we can obtain such $SR(q)$, the answer of our first solution is just the point in $SR(q)$ that is closest to q. In the sequel, we explain how to get $SR(q)$. In a d-dimensional space, given a weighting vector $\vec{w}$ and a point p, we can get a hyperplane, denoted as $H(\vec{w}, p)$, which is perpendicular to $\vec{w}$ and contains the point p. Then, we have the lemma below.

LEMMA 1. *Given a hyperplane $H(\vec{w}, p)$ formed by $\vec{w}$ and p,* (*i*) *if a point p' lies on $H(\vec{w}, p)$, $f(\vec{w}, p') = f(\vec{w}, p)$;* (*ii*) *if a point p'' lies below $H(\vec{w}, p)$, $f(\vec{w}, p'') < f(\vec{w}, p)$; and* (*iii*) *if a point p''' lies above $H(\vec{w}, p)$, $f(\vec{w}, p''') > f(\vec{w}, p)$.*

PROOF. The proof is intuitive, and thus, it is skipped because of space limitation. □

According to Lemma 1, all the points lying on/below/above the hyperplane $H(\vec{w}, p)$ have the same/smaller/bigger scoring value as/than p w.r.t. $\vec{w}$. Figure 5(a) explains Lemma 1 in a 2D space, where the hyperplane $H(\vec{w}_3, p_3)$ is formed by $\vec{w}_3$ and p_3 in Figure 1. Given points p_1 below $H(\vec{w}_3, p_3)$, p_5 above $H(\vec{w}_3, p_3)$ and p_7 on $H(\vec{w}_3, p_3)$, we have $f(\vec{w}_3, p_1) < f(\vec{w}_3, p_3)$, $f(\vec{w}_3, p_5) > f(\vec{w}_3, p_3)$, and $f(\vec{w}_3, p_7) = f(\vec{w}_3, p_3)$. These findings are also consistent with the data listed in Figure 1(c). Based on Lemma 1, the concept of *half space* is stated below.

DEFINITION 8 (HALF SPACE). *Given a hyperplane $H(\vec{w}, p)$, the* half space *formed by $\vec{w}$ and p, denoted as $HS(\vec{w}, p)$, satisfies that $\forall p' \in HS(\vec{w}, p), f(\vec{w}, p') \leq f(\vec{w}, p)$.*

In other words, $HS(\vec{w}, p)$ includes all the points lying on and below the hyperplane $H(\vec{w}, p)$. Figure 5(a) illustrates the half space $HS(\vec{w}_3, p_3)$ formed by $\vec{w}_3$ and p_3, i.e., the shaded area in Figure 5(a). Based on Lemma 1 and Definition 8, we present the following lemmas to explain the construction of q's safe region.

LEMMA 2. *Given a weighting vector $\vec{w}$, and a point p which is the top k-th point of $\vec{w}$, if $q' \in HS(\vec{w}, p)$, $q' \in TOPk(\vec{w})$.*

PROOF. If $q' \in HS(\vec{w}, p)$, $f(\vec{w}, q') \leq f(\vec{w}, p)$ according to Definition 8. Since p is the top k-th point of $\vec{w}$, $q' \in TOPk(\vec{w})$ based on Definition 1. The proof completes. □

LEMMA 3. *Given a why-not weighting vector set $W_m = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n\}$, and a set $\Lambda = \{p_1, p_2, \ldots, p_n\}$ of points ($\forall p_i \in \Lambda$ is the top k-th point w.r.t. its corresponding why-not weighting vector $\vec{w}_i \in W_m$), the safe region of a query point q is the overlapping of all the half spaces formed by $\vec{w}_i$ and p_i, i.e., $SR(q) = \cap_{1 \leq i \leq n} HS(\vec{w}_i, p_i)$.*

PROOF. The proof is straightforward according to Lemma 2 and Definition 8, and hence, it is omitted for space saving. □

Figure 5(b) depicts an example of Lemma 3, which utilizes the dataset shown in Figure 1. Assume that $\vec{w}_1$ and $\vec{w}_4$ are two why-not weighting vectors, the corresponding top 3-rd points for $\vec{w}_1$ and $\vec{w}_4$ are p_4 and p_7, respectively. Therefore, the safe region of q is the overlapping of $HS(\vec{w}_1, p_4)$ and $HS(\vec{w}_4, p_7)$, i.e., the shaded area (i.e., quadrilateral $AoBq'$) in Figure 5(b).

It is worth mentioning that the safe region formed by top (k–1)-th points (denoted as $SR'(q)$) is a subset of $SR(q)$, but $SR'(q)$ does not contain the optimal q'. This is because the hyperplane formed by top (k–1)-th points is always below the hyperplane formed by top k-th points, and hence, $SR'(q)$ is further to q than $SR(q)$. As shown in Figure 5(b), $SR'(q)$ is the corresponding safe region formed by top 2-nd points of $\vec{w}_1$ and $\vec{w}_4$, i.e., p_2 and p_1. Obviously, $SR'(q)$ does not contain q'.

After getting the safe region of q, we can find the optimal query point q' with the minimum cost w.r.t. q. Take Figure 5(b) as an example again, point q' is the desirable refined query point. However, it is found that the safe region is a convex polygon bounded by hyperplanes. Thus, the above safe region computation does not scale well with the dimensionality because computing the intersection of half spaces becomes increasingly complex and prohibitively expensive in high dimensions [2]. Actually, finding the op-

timal query point q' with the minimum cost w.r.t. q is an optimization problem. Moreover, the penalty function of q' (i.e., Equation (1)) can be seen as a quadratic function. In light of this, we employ the quadric programming to find the optimal q' without computing the exact safe region. Specifically, the quadric programming can be represented in the following form:

$$min\ f(x) = \frac{1}{2}x^T Hx + x^T c \quad s.t. \begin{cases} Ax \le b \\ lb \le x \le ub \end{cases} \tag{2}$$

It derives the optimal x that minimizes $f(x)$ under the constraints $Ax \le b$ and $lb \le x \le ub$, in which $f(x)$ is an objective function; H and A are matrixes; x, c, b, lb, and ub are vectors; and superscript T denotes transposition. Our problem is actually an optimization problem, as the goal is to obtain q' with the smallest penalty. Hence, we utilize the quadric programming to obtain the optimal q'. For simplicity, in this paper, we assume that the objective function for our problem is $f(q') = \sum_{i=1}^{d} (q[i] - q'[i])^2 = \frac{1}{2}(q')^T Hq' + (q')^T c$, where H = diag(2, 2, …, 2) is a $d \times d$ diagonal matrix with all eigenvalues being 2, and c = (-2q[1], -2q[2], …, -2q[d]) is a d-dimensional vector.

In addition, given a why-not weighting vector set $W_m = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n\}$ and a point set $\Lambda = \{p_1, p_2, \ldots, p_n\}$ ($p_i \in \Lambda$ is the top k-th point of $\vec{w}_i \in W_m$), the optimal (modified) q' falling into the safe region must satisfy that, $\forall\ \vec{w}_i \in W_m$ and $\forall\ p_i \in \Lambda, f(\vec{w}_i, q') \le f(\vec{w}_i, p_i)$ according to Definition 7, which can be represented by $Aq' \le b$ in Equation (2), where A defined below is a $n \times d$ matrix and $b = (f(\vec{w}_1, p_1), f(\vec{w}_2, p_2), ..., f(\vec{w}_n, p_n))$. As mentioned earlier, the varying range of q is [0, q]. Consequently, $0 \le q' \le q$ corresponds to $lb \le x \le ub$.

$$A = \begin{vmatrix} w_1[1] & w_1[2] & \ldots & w_1[d] \\ w_2[1] & w_2[2] & \ldots & w_2[d] \\ \vdots & \vdots & \ddots & \vdots \\ w_n[1] & w_n[2] & \ldots & w_n[d] \end{vmatrix}$$

Based on the above analysis, we propose the algorithm called MQP to **m**odify the **q**uery **p**oint q, whose pseudo-code is presented in Algorithm 1. First, we adopt the branch-and-bound method to find the top k-th point for every why-not weighting vector (lines 1-12). Then, we use the interior-point quadratic programming algorithm **QuadProg** [26], which is based on a logarithmic barrier function method, to get the optimal refined query point q' (lines 13-14). In particular, **QuadProg** iteratively finds an approximate Newton direction associated with the Karush-Kuhn-Tucker system of equations which characterizes a solution of the logarithmic barrier function problem. Due to space limitation, the details of **QuadProg** are omitted here. The time complexity of MQP is given in Theorem 1 below.

THEOREM 1. The time complexity of MQP algorithm is $O(|RT| \times |W_m| + d^3 \times L)$, in which $|RT|$ is the cardinality of R-tree, d is the dimensionality, and $L = \lceil log(d^3 + 1) \rceil + \lceil log(\theta + 1) \rceil + \lceil log(\omega + 1) \rceil + \lceil log(d + n) \rceil$ with $\omega = max(f(\vec{w}_1, p_1), f(\vec{w}_2, p_2), \ldots, f(\vec{w}_n, p_n))$ and $\theta = max(q[1], q[2], \ldots, q[d])$.

PROOF. MQP algorithm consists of two phases. The first phase is to find the top k-th point for each why-not weighting vector. In the worst case, it needs to traverse the whole R-tree $|W_m|$ times, whose time complexity is $O(|RT| \times |W_m|)$. The second phase is the quadratic programming, whose time complexity is $O(d^3 \times L)$ [26]. Therefore, the total time complexity of MQP is $O(|RT| \times |W_m| + d^3 \times L)$. The proof completes. □

Algorithm 1 Modifying query point q (MQP)

Input: an R-tree RT on a set P of data points, a query point q, a parameter k, a why-not weighting vector set W_m
Output: q'
/* HP is a min-heap; Λ is a set storing the top k-th point for each why-not weighting vector; H and A are matrixes; c, b, lb, and ub are vectors. */
1: **for** each weighting vector $w_i \in W_m$ **do**
2: initialize the min-heap HP with all root entries of RT and $count$ = 0
3: **while** $HP \neq \emptyset$ **do**
4: de-heap the top entry e of HP
5: **if** e is a data point **then**
6: $count$ ++
7: **if** $count = k$ **then**
8: add e to Λ
9: **break**
10: **else** // e is an intermediate (i.e., a non-leaf) node
11: **for** each child entry $e_i \in e$ **do**
12: insert e_i into HP
13: set H, A, c, b, lb, and ub by using W_m, Λ, and q
14: q' = **QuadProg**(H, A, c, b, lb, ub) // interior-point quadratic programming algorithm in [26]
15: **return** q'

4.3 Modifying W_m and k

Imagine that, if the computer q in Figure 1 has been put into production, changing attribute values might not be the best solution. Fortunately, as pointed out by Carpenter and Nakamoto [6], consumer preferences may actually be influenced by proper marketing strategies, such as advertising, which is proved by the example of Wal-Mart [1]. Hence, alternatively, Apple can influence customer to change their preferences, such that the computer q appears in customers' wish list. To this end, we develop the second solution to refine the original reverse top-k query by modifying the customers' preferences. Since customers' preferences are application-dependent and the reverse top-k query studied in this paper involves two types of customers' preferences, i.e., W_m and k, our second solution is to modify a why-not weighting vector set W_m and a parameter k.

Firstly, we give the penalty model to quantify the total changes of W_m and k. We use ΔW_m and Δk to measure the cost of the modification of W_m and k respectively, where

$$\begin{cases} \Delta k = max(0, k' - k) \\ \Delta W_m = \sum_{i=1}^{|W_m|} \sqrt{\sum_{j=0}^{d} (w_i[j] - w_i'[j])^2} \end{cases} \tag{3}$$

It is worth noting that, there is a possibility that the modified k' value may be smaller than the original k value. In this case, we set Δk to 0. For example, assume that (W_m, k = 6) is modified to (W_m', k' = 3). Since q belongs to the top-3 query result of every refined why-not weighting vector, it must also be in the corresponding top-6 query result. Consequently, it is unnecessary to increase the original k. In addition, ΔW_m is just the sum of every why-not weighting vector penalty. In a word, we utilize the sum of ΔW_m and Δk to capture the total change of customer preferences. Given the fact that the customers' tolerances to the changes of W_m and k are different, we utilize two non-negative parameters, namely, α and β with ($\alpha + \beta = 1$), to capture customers' tolerance to the changes of k and W_m, respectively. Then, a normalized penalty model is defined as follows.

$$Penalty(W_m', k') = \alpha \frac{\Delta k}{\Delta k_{max}} + \beta \frac{\Delta W_m}{(\Delta W_m)_{max}}$$

Here, Δk_{max} refers to the maximum value of Δk which can be derived by $(k'_{max} - k)$ with k'_{max} calculated by Lemma 4 below.

LEMMA 4. *Given a set $R = \{r_1, r_2, \ldots, r_n\}$, where $r_i \in R$ is the actual ranking of a query point q under the corresponding why-not weighting vector $\vec{w}_i \in W_m$, then $k'_{max} = max(r_1, r_2, \ldots, r_n)$.*

PROOF. Assume that we have a refined W_m' and k' with $\Delta W_m' = 0$, the corresponding $k' = max(r_1, r_2, \ldots, r_n)$. Any other possible refined W_m'' and k'' with $\Delta W_m'' > 0$ must have its $k'' < max(r_1, r_2, \ldots, r_n)$ or it cannot be the optimal result. Consequently, $k'_{max} = max(r_1, r_2, \ldots, r_n)$, and the proof completes. □

As shown in Figure 1, the actual rankings of q under why-not weighting vectors $\vec{w}_1$ and $\vec{w}_4$ are 4 and 4 respectively, and thus, $k'_{max} = 4$.

Similarly, $(\Delta W_m)_{max}$ refers to the maximum value of ΔW_m, and it has been proven in [13] that $\Delta \vec{w}_i \le \sqrt{1 + \sum_{j=1}^{d} (w_i[j])^2}$. As $\Delta W_m = \sum_{i=1}^{|W_m|} (\Delta \vec{w}_i) \le \sum_{i=1}^{|W_m|} \sqrt{1 + \sum_{j=1}^{d} (w_i[j])^2}$, we have $(\Delta W_m)_{max} = \sum_{i=1}^{|W_m|} \sqrt{1 + \sum_{j=1}^{d} (w_i[j])^2}$. Based on the above analysis, we re-form the normalized penalty model below.

$$Penalty(W_m', k') = \alpha \frac{max(0, k' - k)}{max(r_1, r_2, ..., r_n) - k} + \beta \frac{\sum_{i=1}^{|W_m|} \sqrt{\sum_{j=1}^{d} (w_i[j] - w_i'[j])^2}}{\sum_{i=1}^{|W_m|} \sqrt{1 + \sum_{j=1}^{d} (w_i[j])^2}} \quad (4)$$

Given the fact that customer preferences are application-dependent, Equation (4) provides a reasonable estimation of the differences between customer preferences in terms of the reverse top-k query. Based on Equation (4), we formally define the problem of modifying W_m and k as follows.

DEFINITION 9 (MODIFYING W_M AND K). *Given a d-dimensional dataset P, a positive integer k, a query point q, and a why-not weighting vector set $W_m = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n\}$ ($\forall \vec{w}_i \in W_m$, $q \notin TOPk(\vec{w}_i)$), the modification of W_m and k is to find $W_m' = \{\vec{w}_1', \vec{w}_2', \ldots, \vec{w}_n'\}$ and k', such that (i) $\forall \vec{w}_i' \in W_m'$, $q \in TOPk'(\vec{w}_i')$; and (ii) the $Penalty(W_m', k')$ is minimized.*

Take Figure 1 as an example again and assume that $\alpha = \beta = 0.5$ for simplicity. If we modify Kevin's and Julia's weighting vectors to $\vec{w}_1' = (0.18, 0.82)$ and $\vec{w}_4' = (0.75, 0.25)$ respectively, Kevin and Julia will appear in the reverse top-3 query result of q with *Penalty* = 0.121. Alternatively, we can modify k to $k' = 4$ and remain the weighting vectors unchanged, Kevin and Julia will also appear in the reverse top-4 query result of q with *Penalty* = 0.5. Based on Definition 9, the first modification is better.

Since the function $Penalty(W_m', k')$ is not differentiable when $k' = k$, it is impossible to use a gradient descent based method to compute (W_m', k') with minimal cost. Another straightforward way is to find the optimal (W_m', k') from all the candidates. Although the total number of candidate (W_m', k') is infinite in an infinite weighting vector space, it is certain that only tuples (W_m', k') satisfying Lemma 5 are the candidate tuples for the final result.

LEMMA 5. *Given a why-not weighting vector set $W_m = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n\}$, a refined $W_m' = \{\vec{w}_1', \vec{w}_2', \ldots, \vec{w}_n'\}$ and k', and a set $R' = \{r_1', r_2', \ldots, r_n'\}$ ($r_i' \in R'$ is the actual ranking of q under $\vec{w}_i' \in W_m'$), if a tuple (W_m', k') is a candidate tuple, it holds that: (i) $k' = max(r_1', r_2', \ldots, r_n')$; and (ii) $\forall\ r_i' \in R'$ $(1 \le i \le n)$, there does not exist another weighting vector $\vec{w}_i''$ under which the real ranking of q is r_i' and $|\vec{w}_i'' - \vec{w}_i| < |\vec{w}_i' - \vec{w}_i|$.*

PROOF. First, assume that the statement (i) is not valid, i.e., an answer tuple (W_m', k') has $k' > max(r_1', r_2', \ldots, r_n')$ or $k' < max(r_1', r_2', \ldots, r_n')$. If $k' > max(r_1', r_2', \ldots, r_n') = k''$, $Penalty(W_m', k') > Penalty(W_m', k'')$, and hence, it cannot be the optimal answer. If $k' < max(r_1', r_2', \ldots, r_n')$, then $\exists \vec{w}_i' \in W_m'$, $q \notin TOPk'(\vec{w}_i')$, which contradicts with the statement (i) of Definition 9. Thus, our assumption is invalid. Second, assume that statement (ii) is invalid, i.e., for an answer tuple (W_m', k'), there is a $\vec{w}_i''$ with $|\vec{w}_i'' - \vec{w}_i| < |\vec{w}_i' - \vec{w}_i|$ and meanwhile the actual ranking of q under $\vec{w}_i''$ being r_i'. If $|\vec{w}_i'' - \vec{w}_i| < |\vec{w}_i' - \vec{w}_i|$, then $\Delta W_m'$ is not minimal. Therefore, $Penalty(W_m', k')$ is not minimum, and (W_m', k') cannot be the final result, which contradicts with the condition of Lemma 5. Hence, our assumption is invalid. The proof completes. □

According to Lemma 5, the qualified candidates W_m' and k' interact with each other, which can facilitate their search process. If we fix one parameter, the other one can be computed accordingly. Since the weighting vector space for W_m' is infinite, it is impossible to fix W_m'. Consequently, we try to fix k'. Given a specified dataset and a query point, the range of k' can be determined by the number of the points *incomparable* with q and the number of the points *dominating* q. Specifically, if a point p_1 dominates another point p_2, it holds that, for every $i \in [1, \ldots, d]$, $p_1[i] \le p_2[i]$ and there exists at least one $j \in [1, \ldots, d]$, $p_1[j] < p_2[j]$. If p_1 neither dominates p_2 nor is dominated by p_2, we say that p_1 is incomparable with p_2. For instance, in Figure 2(a), the query point q is dominated by p_1, and it is incomparable with p_3. Given a d-dimensional dataset P and a query point q, we can find all the points (denoted as D) that dominate q and all the points (denoted as I) that are incomparable with q. Thus, a possible ranking of q could be $R_q = \{(|D| + 1), (|D| + 2), \ldots, (|D| + |I| + 1)\}$, which is also the range of k'.

If we fix the query point q's ranking r_i with $r_i \in R_q$, for every why-not weighting vector $\vec{w}_i$, we can find its corresponding $\vec{w}_i'$ with the minimal $|\vec{w}_i' - \vec{w}_i|$ by utilizing the quadratic programming. After finding all these weighting vectors for each $r_i \in R_q$, we can get the optimal W_m' and k'. However, for a single why-not weighting vector, if all rankings of q have to be considered, there are in total $2^{|I|}$ quadratic programming problems in the worst case, as proved in [14]. Totally, for the entire why-not weighting vector set W_m, it needs to solve $|W_m| \times 2^{|I|}$ quadratic programming problems, which is costly. Nonetheless, if we can find $\vec{w}_i''$ that approximates the minimum $|\vec{w}_i' - \vec{w}_i|$, it would save the search significantly even though it is not the exact answer. Hence, in the second solution, we trade the quality of the answer with the running time, and propose a sampling based algorithm, which finds an approximate optimal answer.

The basic idea of the sampling-based algorithm is as follows. We first sample a certain number of weighting vectors from the sample space, and then, we use these sample weighting vectors to find (W_m', k') with minimum penalty. In particular, there are three issues we have to address: (i) how to get the high quality sample weighting vectors; (ii) how to decide a proper sample size; and (iii) how to use the sample weighting vectors to obtain (W_m', k') with minimum penalty. Next, we discuss the three issues in detail.

First, how can we get the high quality sample weighting vectors that have a direct impact on the quality of the final answer? It is worth noting that, the full d-dimensional weighting vector space is the hyperplane $\sum_{i=1}^{d} w[i] = 1$ in which $w[i] \geq 0$ $(1 \leq i \leq d)$. However, if we take the whole weighting vector space, which is very big, as a sample space, the quality of sample weighting vectors may not be desirable. Hence, we have to narrow down the sample space. According to the statement (ii) of Lemma 5, for a fixed k', the modified weighting vector $\vec{w}_i' \in W_m'$ has the minimum $|\vec{w}_i' - \vec{w}_i|$ w.r.t. $\forall \vec{w}_i \in W_m$. Thus, we should sample the weighting vector that can approximate to the minimum $|\vec{w}_i' - \vec{w}_i|$. As proved in [14], for a fixed k', the weighting vector $\vec{w}_i'$, which has the minimum $|\vec{w}_i - \vec{w}_i|$ w.r.t. $\vec{w}_i$, exists in one of the hyperplanes formed by I and q. Specifically, for a point $p \in I$, the hyperplane formed by p and q is: $(\vec{p} - \vec{q}) \cdot \vec{w} = 0$. Consequently, all the hyperplanes intersecting with $\sum_{i=1}^{d} w[i] = 1$ constitute the sample space.

Second, how shall we decide an appropriate sample size? It is well known that, the bigger the sample size, the higher the quality of the result. Nonetheless, it is impossible to sample an infinite number of weighting vectors. A larger sample size increases the cost. Hence, in this paper, we take the sample size $|S|$ as a user specified parameter, which can better meet users' requirements.

Third, how to use the sample weighting vectors to get (W_m', k') with the minimal penalty? There are two possible solutions. The first solution is, for every why-not weighting vector $\vec{w}_i \in W_m$, to find a sample weighting vector $\vec{w}_s$ with minimum $|\vec{w}_s - \vec{w}_i|$, and then replace $\vec{w}_i$ with $\vec{w}_s$. After replacing all why-not weighting vectors, we can obtain a refined W_m'. The corresponding k' can be computed according to Lemma 5(i). The second method is to select randomly $|W_m|$ sample weighting vectors to replace W_m, and we then can get a candidate refined tuple (W_m', k'). The optimal (W_m', k') can be found from the entire candidates.

For the first solution, we can ensure that the refined W_m' is optimal, while the total penalty of W_m' and k' may not be the minimum. For the second solution, if all candidate tuples are considered, there are in total $|S|^{|W_m|}$ instances, whose computation cost could be very expensive. Thus, we present an efficient approach that only examines up to $|S|$ instances, supported by Lemma 6.

LEMMA 6. *Given a candidate tuple* (W_m', k'), *and a weighting vector* $\vec{w}$ (*the ranking of q under* $\vec{w}$ *is bigger than* k'), *if* $\exists\, \vec{w}_i' \in W_m'$ *such that* $|\vec{w}_i - \vec{w}| < |\vec{w}_i' - \vec{w}_i|$ ($\vec{w}_i$ *is the original why-not weighting vector w.r.t.* $\vec{w}_i'$), *there exist another candidate tuple* (W_m'', k''), *where* W_m'' *contains* $\vec{w}$.

PROOF. If $\exists\, \vec{w}_i' \in W_m'$ such that $|\vec{w}_i - \vec{w}| < |\vec{w}_i' - \vec{w}_i|$, we can obtain a new W_m'' from W_m' by replacing all these $\vec{w}_i'$ with $\vec{w}$, and its corresponding k''. Although $k'' > k'$, $\Delta W_m'' < \Delta W_m'$. Thus, (W_m'', k'') is a candidate tuple for the final result including $\vec{w}$. □

According to Lemma 6, we can get the optimal refined W_m and k by examining the sample weighting vectors one by one. To be more specific, for every sample weighting vector, we compute its corresponding ranking of q. We also sort the whole sample weighting vectors in ascending order of the ranking of q. Next, we initialize a candidate tuple (W_m', k') to the first sample weighting vector and its corresponding ranking of q. For each remaining sample weighting vector $\vec{s}$, we examine whether it can contribute to the final result. Based on Lemma 6, if $\exists \vec{w}_i' \in W_m'$, $|\vec{w}_i - \vec{s}| < |\vec{w}_i - \vec{w}_i'|$, we replace all such $\vec{w}_i'$ with $\vec{s}$ and get a new (W_m'', k''). Thereafter, we obtain some candidate tuples (W_m', k'), and the one with the minimal penalty is the final answer.

Algorithm 2 Modifying W_m and k (MWK)

Input: an R-tree RT on a set P of data points, a query point q, a parameter k, a why-not weighting vector set W_m, a sample size $|S|$
Output: W_m' and k'
/* D is the set of points that dominate q; I is the set of points that are incomparable with q; HP is a min-heap; k'_{max} is the maximal value of k'; S is the set of sample weighting vectors; CW is a candidate W_m'; P_{min} is the penalty of the current optimal candidates W_m' and k'. */
1: initialize $k'_{max} = \infty$ and a min-heap $HP = \emptyset$
2: **FindIncom** (RT, q, HP, D, I)
3: sample $|S|$ weighting vectors from the hyperplanes formed by I and q, and add them to S
4: **for** each $\vec{s}_i \in S$ **do**
5: compute the ranking rs_i of q based on D and I
6: sort S in ascending order of rs_i values
7: **for** each weighting vector $\vec{w}_i \in W_m$ **do**
8: compute the ranking r_i of q based on D and I
9: **if** $k'_{max} < r_i$ **then** $k'_{max} = r_i$
10: initialize the CW with the first sample weighting vector in S
11: initialize $W_m' = W_m$, $k' = k'_{max}$, and $P_{min} = Penalty(W_m', k')$
12: **for** each remaining $\vec{s}_i \in S$ and its corresponding rs_i **do**
13: **if** $k'_{max} < rs_i$ **then break** // Lemma 4
14: **for** each $c\vec{w}_j \in CW$ and $\vec{w}_j \in W_m$ **do**
15: **if** $|\vec{s}_i - \vec{w}_j| < |c\vec{w}_j - \vec{w}_j|$ **then** $c\vec{w}_j = \vec{s}_i$ //updates CW using $\vec{s}_i$
16: **if** CW is updated **then**
17: **if** $Penalty(CW, rs_i) < P_{min}$ **then**
18: $W_m' = CW$, $k' = \max(k, rs_i)$, and $P_{min} = Penalty(W_m', rs_i)$
19: **return** W_m' and k'

Function FindIncom (RT, q, HP, D, I)
Input: an R-tree RT on a dataset P, a query point q, a min-heap HP, two point sets D and I
20: initialize sets $D = I = \emptyset$ and a min-heap $HP = \emptyset$
21: insert all root entries of RT into HP
22: **while** $HP \neq \emptyset$ **do**
23: de-heap the top entry e of HP
24: **if** e is a data point **then**
25: **if** e dominates q **then** add e to D
26: **else if** e is not dominated by q **then** add e to I
27: **else** // e is an intermediate node
28: **for** each child entry $e_i \in e$ **do**
29: **if** e_i is not dominated by q **then** insert e_i into HP

Based on the above discussion, we present our sampling based algorithm called MWK to **m**odify $\boldsymbol{W_m}$ and $\boldsymbol{k}$. The pseudo-code of MWK is shown in Algorithm 2. Initially, MWK invokes a function **FindIncom** that follows the branch-and-bound traversal to find the data point sets D and I (line 2), with its details depicted in lines 20-29 of Algorithm 2. Then, we sample $|S|$ weighting vectors from the hyperplanes formed by I and q, maintained by S (line 3). For every sample weighting vector $\vec{s}_i$, the algorithm computes the ranking rs_i of q, and then sorts vectors $\vec{s}_i$ in S based on ascending order of rs_i (lines 4-6). Thereafter, the maximum value of k' is obtained (lines 7-9) for pruning later. MWK then examines, for each sample weighting vector $\vec{s}_i$, whether $\vec{s}_i$ can contribute to the final result based on Lemma 6, and then gets the tuple (W_m', k') with the minimum penalty (lines 12-18). Theorem 2 below presents the time complexity of MWK algorithm.

THEOREM 2. The time complexity of MWK algorithm is $O(|RT| + |S| \times |W_m|)$, with $|S|$ the cardinality of a sample weighting vector set and $|W_m|$ the cardinality of a why-not weighting vector set.

PROOF. The time complexity of MWK is mainly determined by the computation of D and I as well as using the sample weighting vectors to get the optimal result. In the worst case, **FindIncom** has to traverse the whole R-tree RT to get D and I, with time complexity $O(|RT|)$. In addition, the time complexity of using the sample weighting vectors to get the optimal results is determined by the cardinality of the why-not weighting vector set and the sample size, i.e., $O(|S| \times |W_m|)$. Hence, the total time complexity of MWK is $O(|RT| + |S| \times |W_m|)$, and the proof completes. □

4.4 Modifying q, W_m, and k

The two solutions proposed above can return the refined query with the minimum penalty, but there might be some cases where the returned penalty is still beyond the manufacturers' or customers' limits of acceptability. Therefore, manufacturers (e.g., Apple) need to reach a compromise between what customers want and they can offer. In other words, both manufacturers and customers should change their preferences to narrow the gap, which can be addressed through bargaining, e.g., manufacturers and customers collaborate in finding an optimal solution [13]. Hence, in this subsection, we propose the third solution to refine the reverse top-k query by modifying both manufacturers' product (i.e., q) and customers' preferences (i.e., W_m and k).

Firstly, we present the penalty model to quantify the modifications of q, W_m, and k. We use Δq defined in Equation (1) and $\Delta(W_m, k)$ defined in Equation (4) to measure the cost of modifying q and (W_m, k). Weighting parameters γ and λ (with $\gamma + \lambda = 1$) are introduced to capture a user's tolerance to the changes of q and (W_m, k), respectively. Formally, the penalty model is defined in Equation (5), which is the sum of $Penalty(k)$ and $Penalty(W_m', k')$ representing the penalty of manufactures and that of customers respectively. Both Δq and $\Delta(W_m, k)$ have the values in the range of (0,1], and thus, there is no need to normalize them.

$$Penalty(q', W_m', k') = \gamma Penalty(q') + \lambda Penalty(W_m', k') \quad (5)$$

Note that, similar penalty functions have been used in industry, e.g., the sum score of the manufacturers and the customers for the final agreement, namely, *joint outcome*, is used to measure the bargaining solution [13], which further demonstrates that our penalty function is practical. For example, in Figure 1, if we modify q, $\vec{w}_1$, and $\vec{w}_4$ to q'(3.8, 3.8), $\vec{w}_1'$(0.135, 0.865), and $\vec{w}_4'$(0.8, 0.2) respectively, $\vec{w}_1'$ and $\vec{w}_4'$ become the reverse top-3 query result of q' with *penalty* = 0.06 ($\gamma = \lambda = 0.5$). Based on Equation (5), we formulate the problem of modifying q, W_m, and k as follows.

DEFINITION 10 (MODIFYING q, W_M, AND k). *Given a d-dimensional dataset P, a positive integer k, a query point q, and a why-not weighting vector set $W_m = \{\vec{w}_1, \vec{w}_2, \ldots, \vec{w}_n\}$ with $\forall \vec{w}_i \in W_m$, $q \notin TOPk(\vec{w}_i)$, the modification of q, W_m, and k is to find q', $W_m' = \{\vec{w}_1', \vec{w}_2', \ldots, \vec{w}_n'\}$, and k', such that (i) $\forall \vec{w}_i' \in W_m'$, $q' \in TOPk'(\vec{w}_i')$, and (ii) $Penalty(q', W_m', k')$ is minimized.*

For the third solution, we need to get a new tuple (q', W_m', k') whose penalty is minimized. There are two potential approaches. The first one is to find (W_m', k') and then determine the corresponding q'. The second method is to find the candidate q' and then the corresponding (W_m', k'). From MWK algorithm presented in section 4.3, we know that the optimal (W_m', k') can be obtained only when the query point q is fixed. This is because the set I which is used for the sampling is dependent on q. Thus, we adopt the second approach in our third solution. Since there are infinite

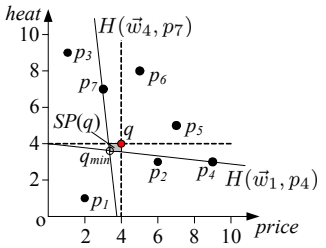


Figure 6: Example of the sample space of q

Algorithm 3 Modifying q, W_m, and k (MQWK)

Input: an R-tree RT on a set P of data points, a query point q, a parameter k, a why-not weighting vector set W_m, sample sizes $|S|$ and $|Q|$ for the sample weighting vector and the sample query point
Output: q', W_m', and k'
/* Q is a set of sample query points; *MinPenalty* is the penalty of the current optimal candidates q', W_m', and k'. */
1: initialize a set $Q = \emptyset$
2: q_{min} = **MQP**(RT, q, k, W_m)
3: Sample $|Q|$ query points from the space determined by q_{min} and q, and add them to Q
4: initialize *MinPenalty* =∞
5: **for** each query point $q_i \in Q$ **do**
6: (W_m'', k'') = **MWK**(RT, q_i, k, W_m, $|S|$)
7: **if** $Penalty(q_i, W_m'', k'') < MinPenalty$ **then**
8: $q' = q_i$, $W_m' = W_m''$, and $k' = k''$
9: $MinPenalty = Penalty(q', W_m', k')$
10: **return** q', W_m', and k'

candidate query points, it is impossible to evaluate all the potential candidates (q', W_m', k'). Hence, we again employ the sampling technique to modify q, W_m, and k. The basic idea is as follows. We first sample a set of candidate query points. For every sample query point q', we use MWK algorithm to find the optimal (W_m', k'). Finally, the tuple (q', W_m', k') with the smallest penalty is returned. In the sequel, we address issues: (i) how to sample query points, and (ii) how to invoke MWK repeatedly.

For the first issue, we need to find out the sample space of q and its sample size. Recall that, according to Definition 7, if the query point falls into the safe region of q, the why-not weighting vectors must appear in the reverse top-k query result. Thus, if we sample a query point (e.g., q') from the safe region, there is no need to modify (W_m, k), and the penalty of (q', W_m, k) must be larger than that of (q_{min}, W_m, k), in which q_{min} is the result returned by the first solution. Therefore, (q', W_m, k) cannot be the final result, and we should sample the query point out of the safe region. Furthermore, if we sample a query point (e.g., q'') out of the safe region, the corresponding refined tuple (q'', W_m'', k'') must satisfy $\Delta(W_m'', k'') > 0$. The tuple (q'', W_m'', k'') can be the optimal result only when $|q'' - q| < |q_{min} - q|$; otherwise, $Penalty(q'', W_m'', k'') > Penalty(q_{min}, W_m, k)$, and hence, it cannot be the final result. Therefore, we know that only the query point falling into the range $[q_{min}, q]$ is the qualified sample query point. Thus, the sample space of q is $SP(q) = \{q' \mid q_{min} < q' < q\}$. Take Figure 6 as an example, q_{min} is returned by the first solution, and the shaded area formed by q_{min} and q is the sample space of q. In addition, the sample size of query points $|Q|$ is also specified by users. For simplicity, we assume that the sample sizes of weighting vectors (used in MWK) and $|Q|$ are identical in our experiments.

The second issue is the iterative call of MWK algorithm. Recall that, the first step of MWK algorithm is to find the points that are

incomparable with the query point. Our third solution needs to call MWK for each sample query point to get the candidate (q', W_m', k'), which requires traversing the R-tree $|Q|$ times and hence incurs high cost accordingly. Thus, we employ the reuse technique to avoid repeated traversal of the R-tree. To this end, we use a heap to store the visited nodes for reusing unless they are expanded. Correspondingly, the function **FindIncom** needs to be revised as well. In particular, when **FindIncom** encounters a data point (lines 24-26 of Algorithm 2) or an intermediate node dominated by q (lines 27-29 of Algorithm 2), it has to be preserved for the reuse later.

Based on the above discussion, we present the algorithm called MQWK to **m**odify $\boldsymbol{q}$, $\boldsymbol{W_m}$, and $\boldsymbol{k}$. The pseudo-code of the algorithm is depicted in Algorithm 3. First of all, MQWK uses MQP algorithm to get the minimal q_{min} (line 2). Then, it samples $|Q|$ query points from the sample space determined by q_{min} and q, and adds them to the set Q (line 3). Next, for every sample query point q', MQWK computes the corresponding optimal (W_m', k') using MWK algorithm (line 6). Finally, the tuple (q', W_m', k') with the minimal penalty is returned (lines 7-10).

Next, we analyze the time complexity of MQWK algorithm.

THEOREM 3. The time complexity of MQWK algorithm is $O(|RT| \times |W_m| + d^3 \times L + |Q| \times (|RT| + |S| \times |W_m|))$.

PROOF. The time complexity of MQWK consists of the computation of q_{min} and the iterative call of MWK. The time complexity of q_{min} computation is equal to that of MQP, i.e., $O(|RT| \times |W_m| + d^3 \times L)$. The iterative call MWK takes $O(|Q| \times (|RT| + |S| \times |W_m|))$. Therefore, the total time complexity of MQWK is $O(|RT| \times |W_m| + d^3 \times L + |Q| \times (|RT| + |S| \times |W_m|))$. The proof completes. □

5. EXPERIMENTAL EVALUATION

In this section, we evaluate the effectiveness and efficiency of our proposed algorithms via extensive experiments, using both real and synthetic datasets.

5.1 Experimental Setup

In our experiments, we use two real datasets, i.e., *NBA* and *Household*. *NBA* contains 17K 13-dimensional points, where each point corresponds to the statistics of a NBA player in 13 categories such as the number of points scored, rebounds, assists, etc. *Household* is a 127K 6-dimensional dataset. Each tuple of the dataset represents the percentage of an American family's annual income spent on six types of expenditures (e.g., gas, electricity). We also create two synthetic datasets, i.e., *Independent* and *Anti-correlated*. In *Independent* dataset, all attribute values are generated independently using a uniform distribution; and *Anti-correlated* dataset denotes an environment in which points *good* in one dimension are *bad* in one or all of the other dimension(s).

We study the performance of the presented algorithms under various parameters, including dimensionality, dataset cardinality, k, actual ranking of q under W_m, the cardinality $|W_m|$ of a why-not weighting vector set, and the sample size. The ranges of the parameters and their default values are summarized in Table 1, which follows [14, 31]. Note that, in every experiment, only one factor varies, whereas others are fixed to their default values. The main performance metrics contain *total running time* (in seconds) and *penalty*. It is worth noting that, we fix $\alpha = \beta = \gamma = \lambda = 0.5$ when computing the penalty of the refined query. All experiments presented in this paper are conducted on a Windows PC with 2.8 GHz CPU and 4 GB main memory. Each dataset is indexed by an R-tree, where the page size is set to 4096 bytes. All algorithms proposed in the paper are implemented in C++.

Table 1. Parameter ranges and default values

Parameter	Range	Default
Dimensionality d	2, 3, 4, 5	3
Dataset cardinality $\|P\|$	10K, 50K, 100K, 500K, 1000K	100K
k	10, 20, 30, 40, 50	10
Actual ranking of q under W_m	11, 101, 1001, 10001	101
$\|W_m\|$	1, 2, 3, 4, 5	1
Sample size	100, 200, 400, 800, 1600	800

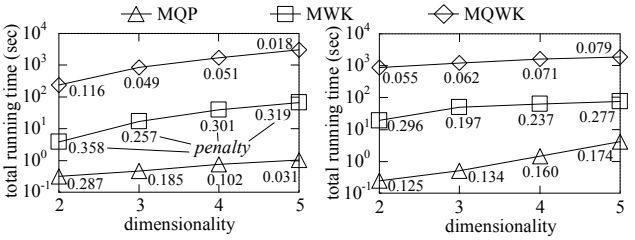
5.2 Performance Study

First, we investigate the impact of dimensionality d on the algorithms. We utilize synthetic datasets, where k = 10, $|P|$ = 100K, $|W_m|$ = 1, sample size is 800, actual ranking of q under W_m is 101 and d is in the range [2, 5], and report the efficiency of different algorithms in Figure 7, where each digit listed in every diagram refers to the *penalty* of one corresponding algorithm at a particular setting. In general, the performance of three algorithms degrades with the growth of dimensionality. This is because all three algorithms need to traverse the R-tree that has a poor efficiency in a high dimensional space, resulting in the degradation of three algorithms. Moreover, for MQP and MQWK, the quadratic programming takes more time in finding the optimal q' as d grows, which also leads to the degradation of MQP and MQWK. It is also observed that, all three algorithms return the answers with small penalty. However, the penalty changes in a relatively unstable trend. The reason is that, the penalty is only affected by the sample size, and other parameters have no influence on it.

Second, we vary the dataset cardinality $|P|$ from 10K to 1000K, and verify its effect on the algorithms using synthetic datasets, as shown in Figure 8. As expected, the total running time of three algorithms ascends as $|P|$ grows. Nevertheless, the penalties of MQP, MWK, and MQWK are small. This is because the larger the dataset cardinality is, the bigger the R-tree is. Thus, three algorithms need to traverse more R-tree nodes with the growth of $|P|$, incurring longer total running time. Hence, our algorithms might not be efficient for huge datasets.

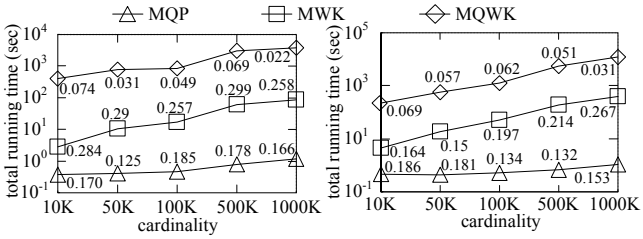
Third, we explore the influence of k on three algorithms, and report the results in Figure 9 using both real and synthetic datasets. It is observed that, all the algorithms degrade as k increases. The reason is that, k'_{max} ascends as k grows, and thus, MWK takes more time in getting the optimal tuple (W_m', k') using sample weighting vectors, which results in the degradation of MWK. For MQP algorithm, if the value of k becomes larger, the cost of finding the k-th point also ascends, and hence, the performance degrades. Since MQWK integrates MQP and MWK, it degrades as well. Again, the penalties of three algorithms are still small.

Then, we inspect the impact of actual ranking of q under the why-not weighting vector set W_m by fixing d at 3, $|P|$ at 100K, sample size at 800, $|W_m|$ = 1, and k = 10. Figure 10 depicts the results on both real and synthetic datasets. Clearly, the total running time of three algorithms increases while the penalties are small. For MWK algorithm, when the actual ranking of q under W_m grows, k'_{max} also ascends, incurring longer total running time. For MQP algorithm, if the ranking of q is low, L (defined in Theorem 1) becomes larger, and thus, the quadratic programming takes more time in finding q'. Based on the above two reasons,



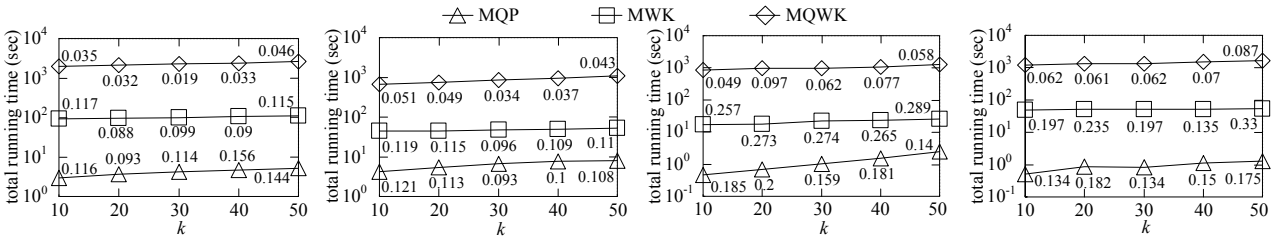
(a) *Independent* (b) *Anti-correlated*

Figure 7: WQRTQ cost vs. dimensionality



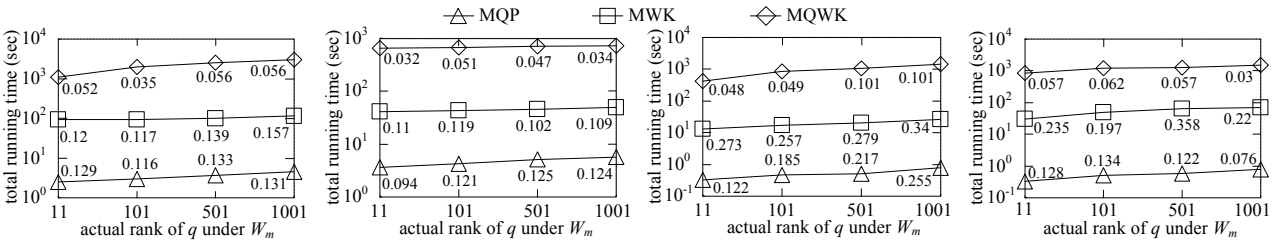
(a) *Independent* (b) *Anti-correlated*

Figure 8: WQRTQ cost vs. dataset cardinality



(a) *Household* (b) *NBA* (c) *Independent* (d) *Anti-correlated*

Figure 9: WQRTQ cost vs. *k*



(a) *Household* (b) *NBA* (c) *Independent* (d) *Anti-correlated*

Figure 10: WQRTQ cost vs. actual ranking under W_m

MQWK also degrades.

Next, we explore the influence of the cardinality $|W_m|$ of a why-not weighting vector set on the algorithms, and Figure 11 plots the results. We observe that MQP, MWK, and MQWK can find the optimal solution with small penalty. Again, the total running time of all algorithms increases gradually when $|W_m|$ ascends. The degradation of MWK is mainly caused by the second phase of the algorithm, i.e., using the sample weighting vectors to find the approximate optimal answer. The performance descent of MQP is due to the computation of the top k-th point for more why-not weighting vectors. Similarly, MQWK degrades as well.

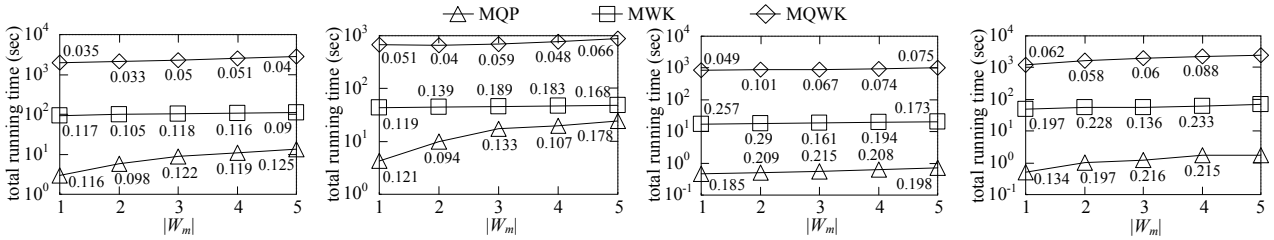
Finally, we evaluate the effect of sample size on the algorithms. To this end, we vary sample size from 100 to 1600 and fix other parameters to their default values. Figure 12 shows the results. The total running time of algorithms MQWK and MWK grows when sample size ascends, although the growth is relatively mild for MWK. This is because the algorithms take more time to examine the samples. Moreover, it is obvious that the penalty of algorithms MQWK and MWK drops as sample size grows. The reason behind is that the bigger the sample size, the higher the quality result. Note that, the penalty sometimes decreases very fast with increasing sample size, and sometimes it does not change. There are two potential reasons. First, it is caused by the randomness since the sample weighting vectors are randomly sampled from the sample space. Second, the different dataset distributions may also lead to this phenomenon. In addition, the total running time and the penalty of MQP algorithm do not change with the growth of sample size, because MQP dose not use the sampling technique.

In summary, from all the experimental results, we can conclude that our proposed algorithms, viz., MQP, MWK, and MQWK, are efficient, and scale well under a variety of parameters.

6. CONCLUSIONS

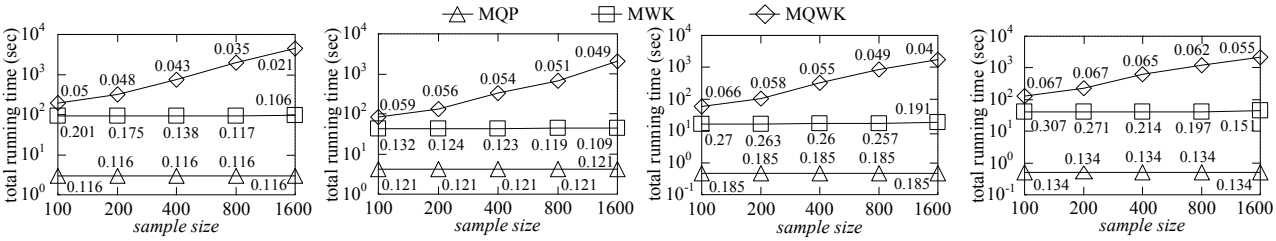
In this paper, for the first time, we study the problem of why-not questions on reverse top-k queries, which aims at explaining why the why-not weighting vector(s) is/are not in the results of reverse top-k queries. We propose a unified framework called WQRTQ to answer why-not questions on both monochromatic and bichromatic reverse top-k queries. WQRTQ consists of three solutions, i.e., (i) modifying a query point q, (ii) modifying a why-not weighting vector set W_m and a parameter k, and (iii) modifying q, W_m, and k. Furthermore, we utilize the quadratic programming, sampling method, and reuse technique to boost the performance of our algorithms. Extensive experiments with both real and synthetic data sets demonstrate the effectiveness and efficiency of our presented algorithms. In the future, we would like to explore why-not questions on reverse top-k queries over larger datasets.

Acknowledgements. This work was supported in part by the 973 Program No. 2015CB352502 and 2015CB352503, NSFC Grants No. 61379033 and 61472348, and the Fundamental Research Funds for the Central Universities.



(a) *Household* (b) *NBA* (c) *Independent* (d) *Anti-correlated*

Figure 11: WQRTQ cost vs. $|W_m|$



(a) *Household* (b) *NBA* (c) *Independent* (d) *Anti-correlated*

Figure 12: WQRTQ cost vs. sample size

7. REFERENCES

[1] S. J. Arnold, J. Handelman, and D. J. Tigert. The impact of a market spoiler on consumer preference structures (or, what happens when Wal-Mart comes to town). *J. Retailing and Consumer Services*, 5(1): 1–13, 1998.

[2] M. Berg, M. Kreveld, M. Overmars, and O. Schwarzkopf. Computational geometry: Algorithms and applications. *Springer*, 1997.

[3] D. Bhagwat, L. Chiticariu, W. C. Tan, and G. Vijayvargiya. An annotation management system for relational databases. *VLDB J.*, 14(4): 373–396, 2005.

[4] S. S. Bhowmick, A. Sun, and B. Q. Truong. Why not, WINE?: Towards answering why-not questions in social image search. In: *MM*, pages 917–926, 2013.

[5] N. Bidoit, M. Herschel, and K. Tzompanaki. Query-based why-not provenance with NedExplain. In: *EDBT*, pages 145–156, 2014.

[6] G. S. Carpenter and K. Nakamoto. Consumer preference formation and pioneering advantage. *J. Marketing Research*, 26(3): 285–298, 1989.

[7] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith. The onion technique: Indexing for linear optimization queries. In: *SIGMOD*, pages 391–402, 2000.

[8] A. Chapman and H. V. Jagadish. Why not? In: *SIGMOD*, pages 523–534, 2009.

[9] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides. Indexing reverse top-*k* queries in two dimensions. In: *DASFAA*, pages 201–208, 2013.

[10] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB J.*, 12(1): 41–58, 2003.

[11] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis. Answering top-*k* queries using views. In: *VLDB*, pages 451–462, 2006.

[12] S. Ge, L. H. U, N. Mamoulis, and D. W. Cheung. Efficient all top-*k* computation: A unified solution for all top-*k*, reverse top-*k* and top-*m* influential queries. *TKDE*, 25(5):1015–1027, 2013.

[13] K.-Y. Goh, H.-H. Teo, H. Wu, and K.-K. Wei. Computer-supported negotiations: An experimental study of bargaining in electronic commerce. In: *ICIS*, pages 104–116, 2000.

[14] Z. He and E. Lo. Answering why-not questions on top-*k* queries. In: *ICDE*, pages 750–761, 2012.

[15] M. Herschel. Wondering why data are missing from query results?: Ask conseil why-not. In: *CIKM*, 2213–2218, 2013.

[16] M. Herschel and M. Hernandez. Explaining missing answers to SPJUA queries. In: *VLDB*, pages 185–196, 2010.

[17] M. Herschel, M. A. Hernandez, and W. C. Tan. Artemis: A system for analyzing missing answers. In: *VLDB*, pages 1550–1553, 2009.

[18] V. Hristidis, N. Koudas, and Y. Papakonstantinou. PREFER: A system for the efficient execution of multi-parametric ranked queries. In: *SIGMOD*, pages 259–270, 2001.

[19] V. Hristidis and Y. Papakonstantinou. Algorithms and applications for answering ranked queries using ranked views. *VLDB J.*, 13(1): 49–70, 2004.

[20] J. Huang, T. Chen, A. H. Doan, and J. F. Naughton. On the provenance of non-answers to queries over extracted data. In: *VLDB*, pages 736–747, 2008.

[21] M. S. Islam, R. Zhou, and C. Liu. On answering why-not questions in reverse skyline queries. In: *ICDE*, pages 973–984, 2013.

[22] H. V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making database systems usable. In: *SIGMOD*, pages 13–24, 2007.

[23] C. Jin, R. Zhang, Q. Kang, Z. Zhang, and A. Zhou. Probabilistic reverse top-*k* queries. In: *DASFAA*, pages 406–419, 2014.

[24] J.-L. Koh, C.-Y. Lin, and A. L. P. Chen. Finding *k* most favorite products based on reverse top-*t* queries. *VLDB J.*, 23(4): 541–564, 2014.

[25] A. Meliou, W. Gatterbauer, K. F. Moore, and D. Suciu. WHY SO? or WHY NO? Functional causality for explaining query answers. In: *MUD*, pages 3-17, 2010.

[26] R. D. C. Monteiro and I. Adler. Interior path following primal-dual algorithms, part ii: Convex quadratic programming. *Math. Program.*, 44(1–3): 43–66, 1989.

[27] V. Padmanabhan, S. Rajiv, and K. Srinivasan. New products, upgrades, and new releases: A rationale for sequential product introduction. *J. Marketing Research*, 34(4): 456–472, 1997.

[28] W. C. Tan. Provenance in databases: Past, current, and future. *IEEE Data Eng. Bull.*, 30(4): 3–12, 2007.

[29] Y. Tao, V. Hristidis, D. Papadias, and Y. Papakonstantinou. Branch-and-bound processing of ranked queries. *Inf. Syst.*, 32(3): 424–445, 2007.

[30] Q. T. Tran and C. Y. Chan. How to ConQueR why-not questions. In: *SIGMOD*, pages 15–26, 2010.

[31] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Norvag. Monochromatic and bichromatic reverse top-*k* queries. *IEEE Trans. Knowl. Data Eng.*, 23(8): 1215–1229, 2011.

[32] A. Vlachou, C. Doulkeridis, and K. Norvag. Monitoring reverse top-*k* queries over mobile devices. In: *MobiDE*, pages 17–24, 2011.

[33] A. Vlachou, C. Doulkeridis, K. Norvag, and Y. Kotidis. Identifying the most influential data objects with reverse top-*k* queries. In: *VLDB*, pages 364–372, 2010.

[34] A. Vlachou, C. Doulkeridis, K. Norvag, and Y. Kotidis. Branch-and-bound algorithm for reverse top-*k* queries. In: *SIGMOD*, pages 481–492, 2013.

[35] M. Xie, L. V. S. Lakshmanan, and P. T. Wood. Efficient top-*k* query answering using cached views. In: *EDBT*, pages 489–500, 2013.

[36] D. Xin, C. Chen, and J. Han. Towards robust indexing for ranked queries. In: *VLDB*, pages 235–246, 2006.

[37] A. Yu, P. K. Agarwal, and J. Yang. Processing a large number of continuous preference top-*k* queries. In: *SIGMOD*, pages 397–408, 2012.

[38] C. Zong, X. Yang, B. Wang, and J. Zhang. Minimizing explanations for missing answers to queries on databases. In: *DASFAA*, pages 254–268, 2013.

[39] L. Zou and L. Chen. Dominant graph: An efficient indexing structure to answer top-*k* queries. In: *ICDE*, pages 536–545, 2008.