Answering Why-Not Questions on Top-K Queries

Zhian He, Eric Lo

Abstract—After decades of effort working on database performance, the quality and the usability of database systems have received more attention in recent years. In particular, the feature of explaining missing tuples in a query result, or the so-called "why-not" questions, has recently become an active topic. In this paper, we study the problem of answering why-not questions on top-k queries. Our motivation is that we know many users love to pose that kinds of queries when they are making multi-criteria decisions. However, they would also want to know *why* if their expected answers do *not* show up in the query results. In this paper, we develop algorithms to answer such why-not questions efficiently. Case studies and experimental results show that our algorithms are able to return high quality explanations efficiently.

Index Terms—Why-Not, Top-K, Usability, Dominating.

1 Introduction

Although the performance of database systems has gained dramatic improvement in the past decades, they also become more and more difficult to use than ever [1]. In recent years, there is a growing effort to improve the usability of database systems. For example, to help end users query the database more easily, the feature of keyword search [2] or form-based search [3] could be added to a database system to assist users to find their desired results. As another example, the feature of query recommendation [4] or query auto-completion [5] could be added to a database system in order to help users to formulate SQL queries. Among all the studies that focus on improving databases' usability, the feature of explaining missing tuples in database queries, or the so-called "why-not" [6] questions, has recently received growing attentions.

A why-not question is being posed when a user wants to know *why* her expected tuples do *not* show up in the query result. Recently, a certain effort has worked on answering why-not questions on traditional relational/SQL queries (e.g., [6], [7], [8], [9]). However, none of those can answer why-not questions on preference queries like top-k queries yet.

Answering why-not questions on top-k queries is useful because users love to pose top-k queries when making multi-criteria decisions. However, they may feel lost when their expected answers are missing in the query result and they may want to know *why*: "*Is it because I have set k too small?*", "*Or I have set my weightings badly?*, "*Or because of both?*"

Top-k dominating queries [10], or simply *dominating queries*, is a variant of top-k query that users may pose why-not questions on. While a top-k dominating query frees users from specifying the set of weightings by ranking the objects based on the number of (other) objects that they could *dominate* (e.g., if object x dominates[1] nine objects while object y dominates four objects, then x ranks higher than y), users may still want to know *why* their expected answers *not* in the query result. For example, the agent of Jeremy Lin, a hot NBA player this year, may pose a top-100 dominating query about the best guards in NBA history. When Lin is not in the result, his agent may want to know the reason: "*Is that I have set my k too small?*".

In this paper, we define the explanation models for both why-not top-k questions and why-not dominating questions. We show that finding the best explanations is actually computationally expensive for both. Afterwards, we present efficient evaluation algorithms that can obtain the best approximate explanations in reasonable time. We present case studies to demonstrate our solutions. We also present experimental results to show that our solutions return high quality solutions efficiently. A preliminary version of this paper appears in [12], in which we discuss only why-not top-k questions.

The rest of the paper is organized as follows. Section 2 presents the related work. Section 3 and Section 4 respectively present (i) the problem formulation, (ii) the problem analysis, and (iii) the algorithms of answering why-not questions on top-k queries and dominating queries. Section 5 presents both the case studies and the experimental results. Section 6 concludes the paper.

2 Related Work

To the best of our knowledge, we are the first to address why-not questions on top-k queries and dominating queries. Early works [13], [14] discussed mechanisms for explaining a null answer for a database query. The concept of why-not is first formally discussed in [6]. That work answers a user's why-not question on Select-Project-Join (SPJ) queries by telling her which query operator(s) eliminated her desired

The research was supported by grants 521012E and 520413E from Hong Kong RGC.

• *Z. He and E. Lo are with the Department of Computing, Hong Kong Polytechnic University, Hong Kong.*
E-mail: {cszahe, ericlo}@comp.polyu.edu.hk

1. The notion of *dominance* follows the tradition [11] in which an object a *dominates* b if all attribute values of a are no worse than all attributes of b and at least one attribute of a is better than b.

tuples. After that, this line of work is gradually expanded. In [7] and [8], the missing answers of SPJ [7] and SPJUA (SPJ + Union + Aggregation) queries [8] are explained by a *data-refinement* approach, i.e., it tells the user how the data should be modified (e.g., adding a tuple) if she wants the missing answer back to the result. The latest result is [9], which uses a *query-refinement* approach that tells the user how to revise her original SPJA queries so that the missing answers can return to the result. They define that a good refined query should be (a) *similar* — have few "edits" comparing with the original query (e.g., modifying the constant value in a selection predicate is a type of edit; adding/removing a join predicate is another type of edit) and (b) *precise* — have few extra tuples in the result, except the original result plus the missing tuples.

A related work is [15], which helps users to quantify their preferences as a set of weightings. Its solution is based on presenting users a set of objects to choose, and try to infer the users' weightings based on the objects that they have chosen. In the why-not paradigm, users are quite clear with which are the missing objects and our job is to explain to them why those objects are missing.

Another related work is *reverse top-k queries* [16]. A reverse top-k query takes as input a top-k query, a missing object $\vec{m}$, and a set of candidate weightings W. The output is a weighting $\vec{w} \in W$ that makes $\vec{m}$ in its top-k result. Two solutions are given in [16]. The first one insists users to provide W as input, which slightly limits its practicability. The second one does not require users to provide W, however, it only works when the top-k queries involve two attributes. Although the problems look similar, why-not questions on top-k queries indeed does not require users to provide W.

3 WHY-NOT TOP-K QUESTION

3.1 The Problem and The Explanation Model

Given a database of n objects, each object $\vec{p}$ with d attribute values can be represented as a point $\vec{p} = |p[1]\ p[2]\ \cdots\ p[d]|$ in a d-dimensional **data space** $\mathcal{R}^d$. For simplicity, we assume that all attribute values are numeric and a smaller value means a better score. A top-k query is composed of a scoring function, a result set size k, and a weighting vector $\vec{w} = |w[1]\ w[2]\ \cdots\ w[d]|$. In this paper, we accept the scoring function as any monotonic function and we assume the **weighting space** subject to the constraints $\sum w[i] = 1$ and $0 \leq w[i] \leq 1$. The query result would then be a set of k objects whose scores are the smallest (in case objects with the same scores are tie at rank k-th, only one of them is returned).

Initially, a user specifies a top-k query $q_o(k_o, \vec{w}_o)$. After she gets the result, she may pose a *why-not* question on q_o with a set of *missing objects* $M = \{\vec{m}_1, \ldots, \vec{m}_j\}$. For top-k queries, we follow the query-refinement approach in [9] so that the system returns the user a *refined top-k query* $q'(k', \vec{w}')$ such that all objects in M appear in the result of q' under the same scoring function. (A special case is that a missing object $\vec{m}_i$ is indeed not in the database; more on this later.) We use Δk and Δw to measure the quality of the refined query, where $\Delta k = max(0, k' - k_o)$ and $\Delta w = ||\vec{w}' - \vec{w_o}||_2$. We define Δk this way to deal with the possibilities that a refined query may obtain a k' value smaller than the original k_o value. For instance, assume a user has issued a top-10 query and the system returns a refined top-3 query with a different set of weightings. We regard Δk as 0 in this case because the user essentially does not need to change her original k.

In order to capture a user's tolerance to the changes of k and $\vec{w}$ on her original query q_o, we define a basic penalty model that sets the penalties λ_k and λ_w to Δk and Δw, respectively, where $\lambda_k + \lambda_w = 1$, is as follows:

$$Penalty(k', \vec{w}') = \lambda_k \Delta k + \lambda_w \Delta w \tag{1}$$

Note that the basic penalty model captures the *similar* requirement stated in [9]. However, it favors changing weightings more than changing k because Δk could be a large integer whereas Δw is generally small. One possible way to mitigate this discrimination is to normalize them respectively. To do so, we normalize Δk using $(r_o - k_o)$, where r_o is the rank of the missing object $\vec{m}$ under the original weighting vector $\vec{w}_o$. To explain this, we have to consider the **answer space** of why-not queries, which consists of two dimensions: Δk and Δw. Obviously, a refined query q'_1 is better than, or *dominates*, another refined query q'_2, if both its refinements on k (i.e., Δk) and w (i.e., Δw) are smaller than that of q'_2. For a refined query q' with $\Delta w = 0$, its corresponding Δk must be $r_o - k_o$. Any other possible refined queries with $\Delta w > 0$ and $\Delta k > (r_o - k_o)$ must be dominated by q' in the answer space. In other words, a refined query with $\Delta w > 0$ must have its Δk values smaller than $r_o - k_o$ or else it is dominated by q' and could not be the best refined query. Therefore, $r_o - k_o$ is the largest possible value for Δk and we use that value to normalize Δk. Similarly, let the original weighting vector $\vec{w}_o = |w_o[1]\ w_o[2]\ \cdots\ w_o[d]|$, we normalize Δw using $\sqrt{1 + \sum w_o[i]^2}$, because:

Lemma 1. *In our concerned weighting space, given $\vec{w}_o$ and an arbitrary weighting vector $\vec{w} = |w[1]\ \ w[2]\ \ \cdots\ \ w[d]|$, $\Delta w \leq \sqrt{1 + \sum w_o[i]^2}$.*

Proof: First, we have $\Delta w = \|\vec{w} - \vec{w}_o\|_2 = \sqrt{\sum (w[i] - w_o[i])^2}$. Since $w[i]$ and $w_o[i]$ are both nonnegative, then we can have $\sqrt{\sum (w[i] - w_o[i])^2} \leq \sqrt{\sum (w[i]^2 + w_o[i]^2)} = \sqrt{\sum w[i]^2 + \sum w_o[i]^2}$. It is easy to see that $\sum w[i]^2 \leq (\sum w[i])^2$. As $\sum w[i] = 1$, we know that $\sum w[i]^2 \leq 1$. Therefore, we have $\Delta w \leq \sqrt{\sum w[i]^2 + \sum w_o[i]^2} \leq \sqrt{1 + \sum w_o[i]^2}$. □

Now, we have a *normalized penalty function* as follows:

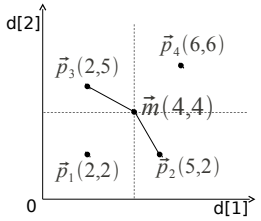
$$Penalty(k', \vec{w}') = \lambda_k \frac{\Delta k}{(r_o - k_o)} + \lambda_w \frac{\Delta w}{\sqrt{1 + \sum w_o[i]^2}} \tag{2}$$

The problem definition is as follows. Given a *why-not* question $\{M, q_o\}$, where M is a non-empty set of missing objects and q_o is the user's initial query, our goal is to find a refined top-k query $q'(k', \vec{w}')$ that includes M in the result with the smallest penalty. In this paper, we use Equation 2 as the penalty function. Nevertheless, our solution works for all kinds of monotonic (with respect to both Δk and Δw) penalty functions. For better usability, we do not explicitly

ask users to specify the values for λ_k and λ_w. Instead, users are prompted to answer a simple multiple-choice question[2] illustrated in Figure 1.

Choice \ Question	*Prefer modifying k or your weightings?*
Prefer modify k (PMK)	$\lambda_k = 0.1$, $\lambda_w = 0.9$
Prefer modify weightings (PMW)	$\lambda_k = 0.9$, $\lambda_w = 0.1$
Never mind (NM); Default	$\lambda_k = 0.5$, $\lambda_w = 0.5$

Fig. 1. A multiple-choice question for freeing users to specify λ_k and λ_w



ranking	ID	score
1	$\vec{p}_1$	2
2	$\vec{p}_2$	3.5
2	$\vec{p}_3$	3.5

Top-3 query $q_o(k_o = 3, \vec{w}_o = |0.5\ 0.5|)$

Fig. 2. A 2-D example

Let us give an example. Figure 2 shows a 2-D data set with five data objects $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$, $\vec{p}_4$, and $\vec{m}$. Assume a user has issued a top-3 query $q_o(k_o = 3, \vec{w}_o = |0.5\ 0.5|)$, with the ranking function: $w_o[1] \cdot p[1] + w_o[2] \cdot p[2]$ and wonders why the point $\vec{m}$ is missing in the query result. So, she would like to find the reason by declaring a *why-not* question $\{\{\vec{m}\}, q_o\}$, using the default penalty preference "Never mind" ($\lambda_k = \lambda_w = 0.5$).

In the example, $\vec{m}$ ranks 4-th under $\vec{w}_o$, so we get $r_o = 4$, $r_o - k_o = 4 - 3 = 1$, and $\sqrt{1 + \sum w_o[i]^2} = 1.2$. Table 1 lists some example refined queries that include $\vec{m}$ in their results. Among those, refined query q'_1 dominates q'_2 because both its Δk and Δw are smaller than that of q'_2. The best refined query in the example is q'_3 (penalty=0.12). At this point, readers may notice that the best refined query is located on the *skyline* of the answer space. Later, we will show how to exploit properties like this to obtain better efficiency in our algorithm.

TABLE 1
Example of candidate refined queries

Refined Query	Δk_i	Δw_i	Penalty
q'_1(4, $\|0.5\ 0.5\|$)	1	0	0.5
q'_2(4, $\|0.6\ 0.4\|$)	1	0.14	0.558
q'_3(3, $\|0.7\ 0.3\|$)	0	0.28	0.12
q'_4(3, $\|0.2\ 0.8\|$)	0	0.42	0.175
q'_5(3, $\|0.21\ 0.79\|$)	0	0.41	0.17
q'_6(3, $\|0.22\ 0.78\|$)	0	0.4	0.167

3.2 Problem Analysis

First, let us consider there is only one missing object $\vec{m}$ in the why-not question. In the data space, we say an object $\vec{a}$ dominates object $\vec{b}$, if $a[i] \leq b[i]$ for $i = [1, \cdots, d]$ and there exists at least one $a[i] < b[i]$. In a data set, if there are k_d objects dominating $\vec{m}$ and n objects incomparable with $\vec{m}$, then the ranking of $\vec{m}$ could be $(k_d + 1), \cdots, (k_d + n + 1)$. For these $n+1$ possible rankings $r_1, r_2, \cdots, r_{n+1}$ of $\vec{m}$, each ranking r_i has a corresponding set W_{r_i} of weighting vectors, where each weighting vector $\vec{w}_{r_i} \in W_{r_i}$ makes $\vec{m}$ ranks r_i-th. As such, any refined queries $q'(r_i, \vec{w}_{r_i})$ are also *candidate answers* (because when $k = r_i$, the missing tuple $\vec{m}$ is in the result, with rank r_i). For each ranking value r_i, we can actually find out the set of $\vec{w}$ which make $\vec{m}$ ranks r_i-th as follows.

First, we define the scoring function using the form: $score(\vec{p}, \vec{w})$. We use I to stand for the set of incomparable objects with respect to $\vec{m}$. Now, we arbitrarily put $j-1$ objects from I into a new set E, and put the rest into another set F. Let any object $\vec{e} \in E$ satisfy the following inequality:

$$score(\vec{e}, \vec{w}_{r_i}) < score(\vec{m}, \vec{w}_{r_i}) \tag{3}$$

which means E is the set of objects that have scores better than $\vec{m}$. Similarly, any object $\vec{f} \in F$ is an object that has score not better than $\vec{m}$:

$$score(\vec{f}, \vec{w}_{r_i}) \geq score(\vec{m}, \vec{w}_{r_i}) \tag{4}$$

Now we have a set of inequalities for all objects in E and F. Together with the constraints $\sum w_{r_i}[i] = 1$ and $w_{r_i}[i] \in [0, 1]$, we can get an inequality system. The solution of this inequality system is a set of weighting vectors that make $\vec{m}$ rank (k_d+j)-th. In fact, there are C^n_{j-1} such inequality systems and W_{r_i} is essentially the union of the solutions of them.

We then discuss the property of W_{r_i}. If the score function is a linear function, then it will become a set of linear inequalities. Then W_{r_i} is formed by a set of convex polytopes. In this case, one exact solution that uses a quadratic programming (QP) solver [17] is as follows: For each ranking value $r_i = k_d + j$, $j \in [1, n+1]$, we can compute $\Delta k = \max(r_i - k_o, 0)$. In order to make Equation 2 as small as possible under this r_i, we have to find a $\vec{w}_{r_i} \in W_{r_i}$ such that $||\vec{w}_{r_i} - \vec{w}_o||_2$ is minimized. Since general QP solver requires the solution space be convex, we have to first divide W_{r_i} into C^n_{j-1} convex polytopes. Each convex polytope corresponds to a quadratic programming problem. After solving all these quadratic programming problems, the best $\vec{w}_{r_i}$ could then be identified. For all ranking r_i to be considered, there are $\sum_{j=1}^{n+1} C^n_{j-1} = 2^n$ (n is the number of incomparable objects with $\vec{m}$) quadratic programming problems in the worst case, so this exact solution is impractical.

Since the problem would not become easier when M has multiple missing objects or when the score function is non-linear, we conclude that searching for the optimal answer is unrealistic.

3.3 Answering Why-Not Top-K Questions

According to the problem analysis presented above, finding the best refined query is computationally difficult. Therefore, we trade the quality of the answer with the running time. Specifically, we propose a sampling-based algorithm that finds the best approximate answer.

2. The number of choices and the pre-defined values for λ_k and λ_w, of course, could be adjusted. For example, to satisfy the *precise* requirement stated in [9], we suggest the user to choose the option where λ_w is a big value (i.e., to prefer PMK, rather than PMW) because [16] has pointed out that similar weightings generally lead to similar top-k results.

3.3.1 Basic Idea

Let us start the discussion with an assumption that there is only one missing object $\vec{m}$. First, we execute a progressive top-k query q'_o based on the weighting vector $\vec{w}_o$ in the user's original query q_o, using any progressive top-k query evaluation algorithm (e.g., [18], [19], [20]), and stop when $\vec{m}$ comes forth to the result set with a ranking r_o. If $\vec{m}$ does not appear in the query result, we report to the user that $\vec{m}$ does not exist in the database and the process terminates.

If $\vec{m}$ exists in the database, we next randomly sample a list of weighting vectors $S = [\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_s]$ from the weighting space. For each weighting vector $\vec{w}_i \in S$, we formulate a progressive top-k query q'_i using $\vec{w}_i$ as the weighting. Each query q'_i is executed by a progressive top-k algorithm, which progressively reports each top ranking object one-by-one, until the missing object $\vec{m}$ comes forth to the result set with a ranking r_i. So, after $s+1$ progressive top-k executions, we have $s+1$ refined queries $q'_i(r_i, \vec{w}_i)$, where $i = o, 1, 2, \ldots, s$, with missing object $\vec{m}$ known to be rank r_i-th exactly. Finally, the refined query $q'_i(r_i, \vec{w}_i)$ with the least penalty is returned to the user as the answer.

3.3.2 How large should be the list of weighting vectors?

Given that there are an infinite number of points (weightings) in the weighting space, *how many sample weightings should we put into S in order to obtain a good approximation of the answer?*

Recall that more sample weightings in S will increase the number of progressive top-k executions and thus the running time. Therefore, we hope S to be as small as possible while maintaining good approximation. We say a refined query is *the best-$T\%$ refined query* if its penalty is smaller than $(1-T)\%$ refined queries in the whole (infinite) answer space, and we hope the probability of getting at least one such refined query is larger than a certain threshold Pr:

$$1 - (1 - T\%)^s \geq Pr \tag{5}$$

Equation 5 is general. The sample size s is independent of the data size but is controlled by two parameters: $T\%$ and Pr. Following our usual practice of not improving usability (i.e., why not queries) by increasing the users' burden (e.g., specifying parameter values for $T\%$, and Pr), we make $T\%$, and Pr system parameters and let users override their values only when necessary.

3.3.3 Algorithm

To begin, let us first outline the three core phases of the algorithm, which is slightly different from the basic idea mentioned in Section 3.3.1 for better efficiency:

[PHASE-1] We first execute a progressive top-k query until $\vec{m}$ appears in the result, with rank r_o, using the original weighting $\vec{w}_o$. Let us denote that operation as r_o=TOPK($q'_o(\vec{w}_o)$, UNTIL-SEE-$\vec{m}$). If $\vec{m}$ exists in the database, next we randomly sample s weightings $\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_s$ from the weighting space and add them into S.

[PHASE-2] Next, for **some** weighting vectors $\vec{w}_i \in S$, we execute a progressive top-k query using $\vec{w}_i$ as the weighting until a stopping condition is met. Let us denote that operation as r_i = TOPK($q'_i(\vec{w}_i)$, STOPPING-CONDITION). In the basic idea mentioned in Section 3.3.1, we have to execute s progressive top-k queries for all s weightings in S. In this section, we present a technique to skip many of those progressive top-k operations so as to improve the algorithm's efficiency (Technique (ii) below). In addition, the stopping condition in the basic idea is to proceed until the missing object $\vec{m}$ comes forth to the result. However, if $\vec{m}$ ranks very poorly under some weighting $\vec{w}_i$, the corresponding progressive top-k operation may be quite slow because it has to access many tuples in the database. In this section, we present a much more aggressive and effective stopping condition that makes most of those operations stop early even before $\vec{m}$ is seen (Technique (i) below). These two techniques together can significantly reduce the overall running time of the algorithm.

[PHASE-3] Using r_i as the refined k', $\vec{w}_i$ as the refined weighting $\vec{w}'$, the $(k', \vec{w}_i)$ combination with the least penalty is formulated as a refined query $q'(k', \vec{w}_i)$ and returned to the user as the why-not answer.

Technique (i) — Stopping a progressive top-k operation earlier

In PHASE-2 of our algorithm, the basic idea is to execute the progressive top-k query until $\vec{m}$ appears in the result, with rank r_i. We denote that operation as r_i = TOPK($q'_i(\vec{w}_i)$, UNTIL-SEE-$\vec{m}$). In the following, we show that it is actually possible for a progressive top-k execution to stop early even before $\vec{m}$ shows up in the result.

Consider an example where a user specifies a top-2 query $q_o(k_o = 2, \vec{w}_o)$ and poses a why-not question about missing object $\vec{m}$. Assume that the list of weightings S is $[\vec{w}_1, \vec{w}_2, \vec{w}_3]$ and TOPK($q'_o(\vec{w}_o)$, UNTIL-SEE-$\vec{m}$) is first executed in PHASE-1 and $\vec{m}$'s actual ranking under $\vec{w}_o$ is 6. Now, we have our first candidate refined query $q'_o(r_o = 6, \vec{w}_o)$, with $\Delta k_o = 6 - 2 = 4$ and $\Delta w_o = 0$. The corresponding penalty, denoted as, $P_{q'_o}$, could be calculated using Equation 2. Remember that we want to find the refined query with the least penalty P_{min}. So, at this moment, we set a penalty variable $P_{min} = P_{q'_o}$.

According to our basic idea outlined in Section 3.3.1, we should execute another progressive top-k using weighting vector, say, $\vec{w}_1$, until $\vec{m}$ shows up in the result set with a ranking r_1. However, we notice that the skyline property in the answer space can help to stop that operation earlier, even before $\vec{m}$ is seen. Given the first candidate refined query $q'_o(r_o = 6, \vec{w}_o)$ with $\Delta w_o = 0$ and $\Delta k_o = 4$, any other candidate refined queries q'_i with $\Delta k_i > 4$ must be dominated by q'_o. In our example, since the first executed progressive top-k execution, all the subsequent progressive top-k executions can stop once $\vec{m}$ does not show up in the top-6 tuples.

Figure 3 illustrates the answer space of the example. The idea above essentially means that all other progressive top-k executions with m does not show up in top-6, i.e., $\Delta k_i > 4$ (see the dotted region), e.g., q'_1, can stop early at top-6, because after that, they have no chance to dominate q'_o anymore.

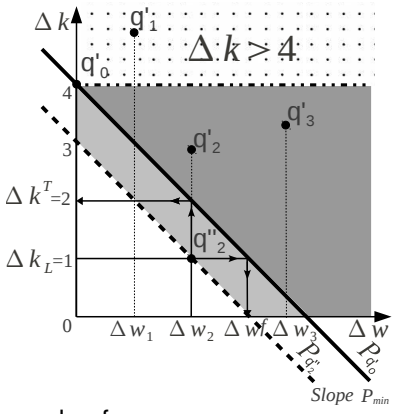


Fig. 3. Example of answer space

While useful, we can actually be even more aggressive in many cases. Consider another candidate refined query, say, q_2', in Figure 3. Assume that $r_2 = \text{TOPK}(q_2'(\vec{w}_2), \text{UNTIL-SEE-}\vec{m}) = 5$ (i.e., $\Delta k_2 = 5 - 2 = 3$), which is not covered by the above technique (since $\Delta k_2 \not\prec 4$). However, q_2' can also stop early, as follows. In Figure 3, we show the normalized penalty Equation 2 as a slope P_{min} that passes through the best refined query so far (currently q_o'). All refined queries that lie on the slope have the same penalty value as P_{min}. In addition, all refined queries that lie above the slope actually have a penalty larger than P_{min}, and thus are dominated by q_o' in this case. Therefore, similar to the skyline discussion above, we can determine an even tighter *threshold ranking* r^T for stopping the subsequent progressive top-k operations even earlier:

$$r^T = \Delta k^T + k_o, \text{ where}$$
$$\Delta k^T = \lfloor (P_{min} - \lambda_w \frac{\Delta w}{\sqrt{1 + \sum w_o[i]^2}}) \frac{r_o - k_o}{\lambda_k} \rfloor \quad (6)$$

Equation 6 is a rearrangement of Equation 2 (with $Penalty(k', \vec{w}') = P_{min}$). Back to our example in Figure 3, given that the weighting of candidate refined query q_2' is $\vec{w}_2$, we can first compute its Δw_2 value. Then, we can project Δw_2 onto the slope P_{min} (currently $P_{min} = P_{q_o'}$) to obtain the corresponding Δk^T value, which is 2 in Figure 3. That means, if we carry out a progressive top-k operation using $\vec{w}_2$ as the weighting, and if $\vec{m}$ still does not appear in result after the top-4 tuples ($r^T = \Delta k^T + k_o = 2 + 2 = 4$) are seen, then we can stop it early because the penalty of q_2' is worse than the penalty P_{min} of the best refined query (q_o') seen so far.

Following the discussion above, we now have two early stopping conditions for the progressive top-k algorithms: UNTIL-SEE-$\vec{m}$ and UNTIL-RANK-r^T. Except for the first progressive top-k operation in PHASE-1 in which $\text{TOPK}(q_o'(\vec{w}_o), \text{UNTIL-SEE-}\vec{m})$ must be used, the subsequent progressive top-k operations can use "UNTIL-SEE-$\vec{m}$ OR UNTIL-RANK-r^T" as the stopping condition. We remark that the conditions UNTIL-RANK-r^T and UNTIL-SEE-$\vec{m}$ are both useful. For example, assume that the actual ranking of $\vec{m}$ under $\vec{w}_2$ is 3, which gives it a $\Delta k_2 = 1$ (see q_2'' in Figure 3). Recall that by projecting Δw_2 on to the slope of P_{min}, we can stop the progressive top-k operation after $r^T = 2 + 2 = 4$ tuples have been seen. However, using the condition UNTIL-SEE-$\vec{m}$, we can stop the progressive top-k operation when $\vec{m}$ shows up at rank three. This drives us to use "UNTIL-SEE-$\vec{m}$ OR UNTIL-RANK-r^T" as the stopping condition.

Finally, we remark that the optimization power of this technique increases while the algorithm proceeds. For example, after q_2'' has been executed, the best refined query seen so far should be updated as q_2'' (because its penalty is better than q_o'). Therefore, P_{min} now is updated as $P_{q_2''}$ and the slope P_{min} should be updated to pass through q_2'' now (the dotted slope in Figure 3). Because P_{min} is continuously decreasing, Δk^T and thus the threshold ranking r^T would get smaller and smaller and the subsequent progressive top-k operations can terminate even earlier while the algorithm proceeds. By the same token, we also envision that the optimization power of this technique is stronger when we have a large λ_w (or small λ_k) because Δk^T will decrease at a faster rate (see Equation 6).

Technique (ii)—Skipping progressive top-k operations

In PHASE-2 our algorithm, the basic idea is to execute progressive top-k queries for *all* weightings in S. We now illustrate how some of those executions could be skipped entirely, so that the overall running time can be reduced further.

The first part of the technique is based on the observation from [16] that similar weighting vectors (measured using their cosine similarity) may lead to top-k results with more (tuples) overlap. Therefore, if a weighting $\vec{w_j}$ is similar to a weighting $\vec{w_i}$ and if operation $\text{TOPK}(q_i'(\vec{w_i}))$ for $\vec{w_i}$ has already been executed, then the query result R_i of $\text{TOPK}(q_i'(\vec{w_i}))$ could be exploited to deduce the highest ranking of the missing object $\vec{m}$ under $\vec{w_j}$. If the deduced highest ranking of $\vec{m}$ is worse than the threshold ranking r_T, then we can skip the entire $\text{TOPK}(q_j'(\vec{w_j}))$ operation.

We illustrate the above by reusing our running example. Assume that we have cached the result sets of executed progressive top-k queries. Let R_o be the result set of the first executed query $\text{TOPK}(q_o'(\vec{w}_o), \text{UNTIL-SEE-}\vec{m})$ and $R_o = [\vec{p}_1, \vec{p}_2, \vec{p}_3, \vec{p}_4, \vec{p}_5, \vec{m}]$. Then, when we are considering the next weighting vector, say, $\vec{w}_1$, in S, we first follow Technique (i) to calculate the threshold ranking r^T. In Figure 3, projecting $\vec{w}_1$ onto slope $P_{q_o'}$ we get $r^T = 3 + 2 = 5$. Next we calculate the scores of all objects in R_o using $\vec{w}_1$ as the weighting. Assume that the scores of $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$, $\vec{p}_4$, and $\vec{p}_5$ are also smaller (better) than $\vec{m}$ under $\vec{w}_1$. In this case, we know the rankings of $\vec{p}_1$, $\vec{p}_2$, $\vec{p}_3$, $\vec{p}_4$, and $\vec{p}_5$ are all smaller (better) than the ranking of m, i.e., the ranking of $\vec{m}$ is at least $5 + 1 = 6$, which is worse than $r^T = 5$. So, we can skip the entire $\text{TOPK}(q_1'(\vec{w}_1))$ operation even without starting it.

The above caching technique is shown to be the most effective between similar weighting vectors [16]. Therefore, we design the algorithm in a way that the list of weightings S is sorted according to their corresponding Δw_i values (of course, $\vec{w_o}$ is the in the head of the list since $\Delta w_o = 0$).

The second part of the technique is to exploit the best possible ranking of m (under all possible weightings) to set up an early termination condition *for the whole algorithm*, so that after a certain number of progressive top-k operations

Algorithm 1 Answering a Why-not Top-K Question

Input:
The dataset D; original top-k query $q_o(k_o, \vec{w}_o)$; missing object $\vec{m}$; penalty settings λ_k, λ_w; $T\%$ and Pr

Output:
A refined query $q'(k', \vec{w}')$

1: Result R of a top-k query;
2: Rank of missing object under original weighting: r_o;
Phase 1:
3: $(R_o, r_o) \leftarrow$ TOPK$(q'_o(\vec{w}_o),$ UNTIL-SEE-$\vec{m})$
4: **if** $r_o = \emptyset$ **then**
5: **return "**$\vec{m}$ **is not in** D**"**
6: **end if**
7: Use [21] to determine the number of points k_d that dominate $\vec{m}$;
8: Determine s from $T\%$ and Pr using Equation 5;
9: Sample s weightings from the weighting space and add them into S;
10: Sort S according to their Δw_i values;
Phase 2:
11: $R \leftarrow (R_o, \vec{w_o})$; //Cache the results
12: $P_{min} \leftarrow Penalty(r_o, \vec{w}_o)$;
13: $\Delta k_L \leftarrow \max(k_d + 1 - k_o, 0)$; //Calculate the lower bound ranking of m
14: $\Delta w^f = (P_{min} - \lambda_k \frac{\Delta k_L}{r_o - k_o}) \frac{\sqrt{1+\sum w_o[i]^2}}{\lambda_w}$; //Project Δk_L to determine early termination point Δw^f
15: **for all** $\vec{w}_i \in S$ **do**
16: **if** $\Delta w_i > \Delta w_f$ **then**
17: **break**; //Technique (ii) — early algorithm termination
18: **end if**
19: $\Delta k^T \leftarrow \lfloor (P_{min} - \lambda_w \frac{\Delta w_i}{\sqrt{1+\sum w_{oi}^2}}) \frac{r_o - k_o}{\lambda_k} \rfloor$;
20: $r^T \leftarrow \Delta k^T + k_o$;
21: **if** there exist r^T objects in some $R_i \in R$ having scores better than $\vec{m}$ under $\vec{w}_i$ **then**
22: **continue**; //Technique (ii) — use cached result to skip a progressive top-k
23: **end if**
24: $(R_i, r_i) \leftarrow$ TOPK$(q'_o(\vec{w}_i),$ UNTIL-SEE-$\vec{m}$ OR UNTIL-RANK-$r^T)$; //Technique (i) — stopping a progressive top-k early
25: $P_i \leftarrow Penalty(r_i, \vec{w}_i)$;
26: $R \leftarrow R \cup (R_i, \vec{w}_i)$;
27: **if** $P_i < P_{min}$ **then**
28: $P_{min} \leftarrow P_i$;
29: $\Delta w^f = (P_{min} - \lambda_k \frac{\Delta k_L}{r_o - k_o}) \frac{\sqrt{1+\sum w_{oi}^2}}{\lambda_w}$;
30: **end if**
31: **end for**
Phase 3:
32: Return the best refined query $q'(k', \vec{w}')$ whose penalty=P_{min};

have been executed, the algorithm can terminate early without executing the subsequent progressive top-k operations.

Recall that the best possible ranking of $\vec{m}$ is k_d+1, where k_d is the number of objects that dominate $\vec{m}$. Therefore, the lower bound of Δk, denoted as Δk_L equals $\max(k_d + 1 - k_o, 0)$. By definition, as outlined in Section 3.1, we have $\Delta k \geq 0$. So, this time, we project Δk_L onto slope P_{min} in order to determine the corresponding *maximum feasible* Δw value. We name that value as Δw^f. For any $\Delta w > \Delta w^f$, it means "$\vec{m}$ has $\Delta k < \Delta k_L$", which is impossible. As our algorithm is designed to examine the weightings in their increasing order of Δw values, when a weighting $w_i \in S$ has $||w_i - w_o|| > \Delta w^f$, TOPK$(q'_i(\vec{w}_i))$ and all subsequent progressive top-k operations TOPK$(q'_{i+1}(\vec{w}_{i+1}))$, ... , TOPK$(q'_s(\vec{w}_s))$ could be skipped.

Reusing Figure 3 as an example, we assume that the number of objects that dominated $\vec{m}$ is 2. By projecting $\Delta k_L = \max(k_d + 1 - k_o, 0) = 1$ onto the slope $P_{q'_o}$, we could determine the corresponding Δw^f value. So, when the algorithm finishes executing a progressive top-k operation for weighting $\vec{w}_2$, PHASE-2 of the algorithm can terminate at that point because all the subsequent Δw_i are larger than Δw^f.

As a final remark, we would like to point out that the pruning power of this technique also increases while the algorithm proceeds. For instance, in Figure 3, if q''_2 has been executed, slope P_{min} is changed from slope $P_{q'_o}$ to slope $P_{q''_2}$. Projecting Δk_L onto the new P_{min} slope would result in a smaller Δw^f, which in turns increases the chance of terminating PHASE-2 earlier.

The pseudo-code of the complete algorithm is presented in Algorithm 1. It is self-explanatory and mainly summarizes what we have discussed above, so we do not give it a walkthrough here.

3.4 Multiple Missing Objects

To deal with multiple missing objects in a why-not question, we have to modify our algorithm a little. First, we do a simple filtering on the set of missing objects M. Specifically, among all the missing objects in M, if there is a missing object $\vec{m}_i$ that dominates another one $\vec{m}_j$ in the data space, then we can remove the dominating object $\vec{m}_i$ from M for the reason that every time $\vec{m}_j$ appears into the top-k result, $\vec{m}_i$ is certainly in the result as well. So, we only need to consider $\vec{m}_j$.

Let M' be the set of missing objects after the filtering step. The sampling method we used here is identical to the case where there is only one missing object.

The modification related to Technique (i) is as follows. For the condition UNTIL-SEE-$\vec{m}$, it should now be UNTIL-SEE-ALL-OBJECTS-IN-M'. For example, r_o in Algorithm 1 Line 3 should now refer to the ranking of the missing object with the worst ranking. The threshold ranking r^T for the condition UNTIL-RANK-r^T should also be computed based on the above r_o instead.

The modifications related to Technique (ii) are as follows. We now have to identify the lower bound of Δk_L for a set of missing objects M' instead of a single missing object. With a set of missing objects $M' = \{\vec{m}_1, \cdots \vec{m}_n\}$, we use DOM_i to represent the set of objects that dominate $\vec{m}_i$. So, Δk_L for M' is $\max(|DOM_1 \cup DOM_2 \cup \cdots \cup DOM_n \cup M'| - k_o, 0)$.

4 Why-Not Top-k Dominating Question

4.1 The Problem and The Explanation Model

Different from a top-k query, a top-k dominating query is composed of a result set size k and a special score function, which scores an object $\vec{p}$ by the number of points that it can dominate. The query result is then the top-k objects with the highest scores (in case objects with the same scores are tie at rank k-th, only one of them is returned).

Initially, a user poses a top-k dominating query $q_o(k_o)$. After she gets the result, she may pose a *why-not* question on q_o with a set of missing objects $M = \{\vec{m}_1, \cdots, \vec{m}_j\}$. Different from why-not questions on top-k queries, if using only the query-refinement approach here, we can only modify the value of k

in order to make M appear in the result. That may result in a refined query whose k's value is increased significantly if there are some missing objects that are actually dominated by many points. As such, we also use the data-refinement approach [7], [8] here. That is, we may either adjust the value of k, the values of $\vec{m}_1, \cdots, \vec{m}_j$, or both[3].

The answer of a why-not question on a dominating query consists of a new value k' and a new value $\vec{m}_i'$ for each object $\vec{m}_i \in M$. We use Δk and Δc to measure the quality of the why-not answer, where $\Delta k = max(0, k' - k_o)$ and $\Delta c = \sum_{i=1}^{j} \|\vec{m}_i' - \vec{m}_i\|_2$. Again, to capture user's tolerance on the change of k and on the change of data values of the missing objects, the corresponding normalized penalty function is defined as follows:

$$Penalty(k', M') = \lambda_k \frac{\Delta k}{r_o - k_o} + \lambda_c \frac{\Delta c}{\sum_{i=1}^{j} \|\vec{m}_i - \vec{l}\|_2} \quad (7)$$

where λ_k is again the penalty of modifying k and λ_c is the penalty of modifying the data values, $\lambda_k + \lambda_c = 1$. Again, Δk and Δc are normalized using their largest possible values, respectively. For Δc, we can show that its largest possible value is $\sum_{i=1}^{j} \|\vec{m}_i - \vec{l}\|_2$, where $\vec{l} = |l[1] \ \cdots \ l[d]|$ is the *lower bound point*, whose value $l[i]$ is the lowest value of dimension i among all the objects in the original d-dimensional dataset. The proof is pretty simple: to make a missing object $\vec{m}_i \in M$ rank better, we need to decrease some of its attribute values such that it can dominate more points than before. Since $\vec{l}$ dominates all the points in the dataset, we can surely make $\vec{m}_i$ rank first if we modify its value to be same as $\vec{l}$. As such we arrive the normalizing factor to be $\sum_{i=1}^{j} \|\vec{m}_i - \vec{l}\|_2$.

Now, formally, the problem is: Given a *why-not* question $\{M, q_o(k_o)\}$, where M is a non-empty set of missing objects, $q_o(k_o)$ is the user's initial top-k dominating query, our goal is to find a new value k' and a value replacement M' for M, such that all the objects in M' appear in the result of refined dominating query $q'(k')$ with the smallest penalty based on Equation 7. Similar to why-not top-k questions, we prompt users to answer a simple multiple-choice question like the one in Figure 1 to determine the values for λ_k and λ_c. The choices are respectively (i) *Prefer modify k* (PMK) ($\lambda_k = 0.1$, $\lambda_c = 0.9$), (ii) *Prefer modify objects' values* (PMO) ($\lambda_k = 0.9$, $\lambda_c = 0.1$), and (iii) *Never mind* (NM) ($\lambda_k = 0.5$, $\lambda_c = 0.5$; default).

4.2 Problem Analysis

First, consider the case where there is only one missing object $\vec{m}$ in the why-not question. On the surface, it seems the solution space (and thus the number of candidate answers) are infinite because $\vec{m}$ can move to anywhere in the data space. However, we can actually have a deeper analysis by dividing the data space $\mathcal{R}^d$ into a set $\mathcal{G}$ of grids, where points within the same grid have the same score (i.e., they dominate the same number of (other) data points). Figure 4 shows an example. In the figure, there are six data points $\vec{p}_1, ..., \vec{p}_5$ and $\vec{m}$. Assume the original query q_o is a top-1 dominating query $q_o(k_o = 1)$ and the why-not question asks why $\vec{m}$ not in the result. In Figure 4, the number within a grid $g \in \mathcal{G}$ denotes the score of $\vec{m}$ if it is *entirely* in g. For example, if $\vec{m}$ falls into the highlighted grid (but not on its boundaries), $\vec{m}$'s score is 3 (it dominates 3 points, which are $\vec{p}_2, \vec{p}_3, \vec{p}_4$).[4] With the grids and their scores, we do not need to consider all possible points in the data space when computing the best value modification and k's value modification, but instead use the following simple method: (1) we consider moving $\vec{m}$ to each grid; by doing so, the new ranking of $\vec{m}$ can be computed by comparing its new score with the scores of the other data points. As such, we can also deduce the corresponding Δk value assuming $\vec{m}$ is moved to that grid (e.g., moving $\vec{m}$ to the highlighted grid in Figure 4 makes $\vec{m}$ rank 1-st, with the highest score 3, so $\Delta k = 0$). (2) Since we hope to minimize both Δk and Δc and all data points in the same grid share the same Δk value, the corresponding Δc of a grid is then the minimum distance between that grid and $\vec{m}$ (see Figure 4). (3) Finally, the best answer is the corner of the grid whose Δk and Δc minimize Equation 7.

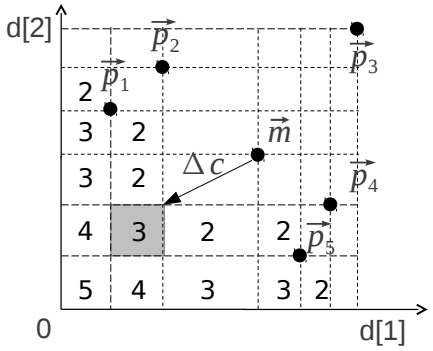


Fig. 4. An example data space with grids

Now, we can see that the complexity of this exact method depends on the number of grids. Since there are N^d grids for a d-dimensional data set with N data points, the complexity of this exact method is $O(N^d)$ in the worst case and the problem would not be easier if there are multiple missing tuples. This motivates us to look for approximate solutions.

4.3 Answering Why-Not Top-k Dominating Questions

Since finding the best refined query with minimal data modification is computationally difficult, our solution here is also a sampling-based algorithm.

4.3.1 Basic Idea

The basic idea for answering why-not top-k dominating questions is similar to the idea of answering top-k why-not questions (Section 3.3.1). Let us start with the case where there is only one missing object $\vec{m}$. First, we execute a top-k dominating query q_o' using a progressive top-k dominating

3. Our data-refinement approach is slightly different from [7]—the latter tries to choose a value already in the database while we may suggest a data value that may not be in the database.

4. If $\vec{m}$ lies on the boundaries, its score follows the largest one.

query evaluation algorithm (e.g., [22]) and stop when $\vec{m}$ comes forth to the result set with a ranking r_o. If $\vec{m}$ does not appear in the query result, we report to the user that $\vec{m}$ does not exist in the database and the process terminates.

If $\vec{m}$ exists in the database, we draw a list of data value samples $S = [\vec{x}_1, \vec{x}_2, \cdots, \vec{x}_s]$. For each data value sample $\vec{x}_i \in S$, we modify $\vec{m}$'s values to be $\vec{x}_i$ and then execute a progressive top-k dominating query until $\vec{m}$ comes forth to the result set with a ranking r_i. So, after $s+1$ progressive top-k dominating executions, we have $s+1$ "refined queries and modified values" pairs: $\langle q'_o(r_o), \vec{m} = \vec{m}\rangle$, $\langle q'_1(r_1), \vec{m} = \vec{x}_1\rangle$, ... , $\langle q'_s(r_s), \vec{m} = \vec{x}_s\rangle$. Finally, the pair with the least penalty is returned to the user as the answer. Next, we discuss where to get the list S of sample data values.

4.3.2 Where to draw sample values?

In the following, we show that the best answer (which minimizes the penalty function) is located within a restricted (smaller) region $\mathcal{R}_s$ of the data space and thus we should draw samples from $\mathcal{R}_s$ instead of from the whole data space $\mathcal{R}^d$. The restricted (smaller) sample space $\mathcal{R}_s$ is essentially a hyper-rectangle (bounding box) whose lower bound value of dimension i is $l[i]$ (recall that $\vec{l}$ is the lower bound point, $l[i]$ is the value of $\vec{l}$ on dimension i; see Section 4.1) and upper bound value of dimension i is $m[i]$. Figure 5 shows an example restricted sample space, $\mathcal{R}_s$, from a 2-d data space. $\mathcal{R}_s$ is the highlighted rectangle bounded by $\vec{l}$ and the missing object $\vec{m}$.

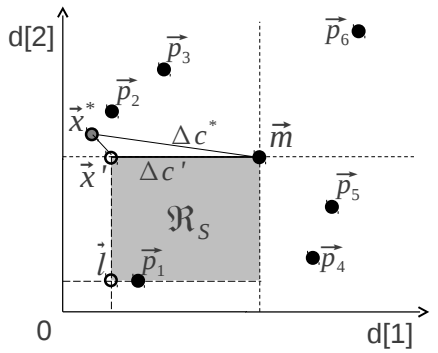


Fig. 5. Restricted sample space $\mathcal{R}_s$

On the surface, drawing samples from $\mathcal{R}_s$ sounds obvious because setting $\vec{m}$ to be some values in $\mathcal{R}_s$ can make $\vec{m}$ dominates more points, rendering it to improve its ranking. However, setting $\vec{m}$ to be some values outside $\mathcal{R}_s$ can also improve $\vec{m}$'s ranking. For example, in Figure 5, if we set $\vec{m}$'s value to be $\vec{x}^*$, $\vec{m}$'s score can be increased from 1 (dominating $\vec{p_6}$ only) to 3 (dominating $\vec{p_2}, \vec{p_3}, \vec{p_6}$), too. In the following, we show that the best answer, i.e., the refined-query-modified-value pair with least penalty, is located in $\mathcal{R}_s$:

Theorem 1. *For any sample $\vec{x}^* \notin \mathcal{R}_s$, there exists at least one sample $\vec{x}' \in \mathcal{R}_s$, such that the corresponding new rankings r^* of $\vec{m}$ by setting $\vec{m} = \vec{x}^*$, and the corresponding new rankings r' of $\vec{m}$ by setting $\vec{m} = \vec{x}'$, follow:*

$$Penalty(r', \{\vec{m} = \vec{x}'\}) < Penalty(r^*, \{\vec{m} = \vec{x}^*\})$$

Proof: Let $\overline{\mathcal{R}_s}$ be the complement of $\mathcal{R}_s$ and let $\vec{x}^* \in \overline{\mathcal{R}_s}$. We try to find a point $\vec{x}' \in \mathcal{R}_s$ that satisfies the penalty inequality stated in the theorem. Such a point $\vec{x}'$ can be found in $\mathcal{R}_s$ whose dimension i (i.e., $x'[i]$) has the value as follows:

$$x'[i] = \quad m[i] \quad \text{if } x^*[i] > m[i]; \tag{8}$$

$$x'[i] = \quad l[i] \quad \text{if } x^*[i] < l[i]; \tag{9}$$

$$x'[i] = \quad x^*[i] \quad \text{if } l[i] \le x^*[i] \le m[i]; \tag{10}$$

For example, Figure 5 shows a sample point $\vec{x}^* \notin \mathcal{R}_s$ and its corresponding sample points $\vec{x}'$ in $\mathcal{R}_s$ determined by the rules above. Now, it is quite easy to see that the score of (i.e., the number of objects dominated by) $\vec{x}'$ is no worse than $\vec{x}^*$ in all cases:

(1) If $x'[i] \le x^*[i]$ in all dimensions, then $\vec{x}'$ certainly has a score no worse than $\vec{x}^*$, since it equals or dominates $\vec{x}^*$;

(2) If $x'[j] \le x^*[j]$ only in some dimensions $\mathcal{D}$, but $x'[i] > x^*[i]$ in the remaining dimensions $\mathcal{D}'$, $\vec{x}^*$ still cannot dominate more points than $\vec{x}'$ because that would imply there exists at least one object $\vec{p}$ in the database that is dominated by $\vec{x}^*$ but not by $\vec{x}'$, resulting in the following contradiction:

$\forall j \in \mathcal{D}, x'[j] \le x^*[j] \le p[j]$

$\forall i \in \mathcal{D}', x^*[i] \le p[i] < x'[i] = l[i]$ (contradiction comes here because $l[i]$ is the lower bound and $p[i]$ cannot $< l[i]$).

As such, we know the ranking r' of $\vec{m}$ by setting $\vec{m} = \vec{x}'$ is no worse than the ranking r^* of $\vec{m}$ by setting $\vec{m} = \vec{x}^*$. Since Δk is measured as the change from the original ranking k_o to the new ranking, we know sample $\vec{x}'$ does not lead to a larger Δk than sample $\vec{x}^*$.

With the example, it is easy to see that the change of data value Δc between $\vec{x}'$ and $\vec{m}$ is also smaller than that between $\vec{x}^*$ and $\vec{m}$ because:

For those dimensions that $x^*[i] > m[i] = x'[i]$, we have $(x'[i] - m[i])^2 < (x^*[i] - m[i])^2$;

For those dimensions that $x^*[i] < l[i] = x'[i]$, we have $(x'[i] - m[i])^2 < (x^*[i] - m[i])^2$ as well.

For those dimensions that $x^*[i] = x'[i]$, we have $(x^*[i] - m[i])^2 = (x'[i] - m[i])^2$;

Because $\vec{x}^* \neq \vec{x}'$, we conclude that $\|\vec{x}^* - \vec{m}\|_2 > \|\vec{x}' - \vec{m}\|_2$.

Since both Δk and Δc of sample $\vec{x}'$ are no worse than that of sample $\vec{x}^*$, the theorem is proved.

□

4.3.3 How large the list of weighting vectors should be?

Although we have shown that we can draw higher quality samples from a restricted sample space, there is still an infinite number of samples in there. Therefore, we adopt the same idea as in why-not top-k processing and look for *the best-T% answer* if its penalty is smaller than $(1-T)\%$ answers in the whole (infinite) answer space, and hope that the probability of getting at least one such answer is larger than a threshold Pr. Since the logic here is exactly the same as the logic in why-not top-k processing, we can use Equation 5 to determine the sample size.

4.3.4 Algorithm

The algorithm is based on the basic idea mentioned in Section 4.3.1, with optimizations added to improve the efficiency. It consists of three phases:

[PHASE-1] The algorithm first executes a progressive top-k dominating query evaluation algorithm (e.g., [22]) to locate the list L of objects, together with their scores, in rank 1, 2, 3, ... , until the missing object $\vec{m}$ shows up in the result in rank r_o-th. Let us denote that operation as (L, r_o) = DOMINATING(UNTIL-SEE-$\vec{m}$). After that, it samples s data values $\vec{x}_1, \vec{x}_2, \ldots, \vec{x}_s$ from the restricted sample space $\mathcal{R}_s$ and adds them into S.

[PHASE-2] Next, for **some** data value sample $\vec{x}_i \in S$, we modify $\vec{m}$'s values to be $\vec{x}_i$ and then determine the ranking r_i of $\vec{m}$ after the value modification. Note that the ranking r_i basically can be determined by executing a progressive top-k dominating algorithm once again on the database (as in the basic idea). We discuss in Technique (a) below to illustrate a much efficient way to determine the ranking r_i, without actually invoking the progressive top-k dominating algorithm. Furthermore, we discuss how techniques similar to Technique (ii) in why-not top-k processing (Section 3.3.3) can be applied here to skip ranking calculations for some data value samples.

[PHASE-3] After PHASE-2, we should have $s+1$ "refined queries and modified values" pairs: $\langle q'_o(r_o), \vec{m} = \vec{m}\rangle$, $\langle q'_1(r_1), \vec{m} = \vec{x}_1\rangle$, ... , $\langle q'_{s+1}(r_{s+1}), \vec{m} = \vec{x}_{s+1}\rangle$. The pair with the least penalty is returned to the user as the answer.

Technique (a) — Efficient ranking computation for a sample point

Here we describe a method to efficiently compute the ranking r_i of $\vec{m}$ if setting $\vec{m}$'s value to $\vec{x}_i$. First, we compute the new score of $\vec{m}$ (i.e., the number of objects dominated by $\vec{m}$) when its values equal to sample $\vec{x}_i$. This step can be easily done by any skyline-related algorithm (e.g., [21]) or by posing a simple range query on an R-tree. Next, we update the scores of all objects in L (stored in PHASE-1) as the value of $\vec{m}$ is changed to $\vec{x}_i$. Note that we do not need to update the scores of objects not in L because they were either dominated by $\vec{m}$ or incomparable with $\vec{m}$. So, their scores would not get changed. For the objects in L that do not dominate $\vec{m}$, their scores are unchanged because if they did not dominate $\vec{m}$ before, they also cannot dominate $\vec{m}$ now (because $\vec{m}$ gets a better value $\vec{x}_i$). Only for those objects in L that dominate $\vec{m}$, we check whether every such object dominates $\vec{x}_i$ (which is $\vec{m}$'s new value), if yes, its score is unchanged; otherwise its score is reduced by one. With all the updated scores in place, we can easily determine the new ranking r_i of $\vec{m}$. We represent this operation as: r_i = COMPUTE-RANK$(\vec{m}, \vec{x}_i)$.

Technique (b). Skipping COMPUTE-RANK *operations*

We now discuss how to apply techniques similar to the techniques (ii) in why-not top-k query processing (Section 3.3.3) to identify a sample $\vec{x}_i$ that must result in answers that are dominated by some processed samples, so that that sample and its associated COMPUTE-RANK$(\vec{m}, \vec{x}_i)$ operation can be entirely skipped.

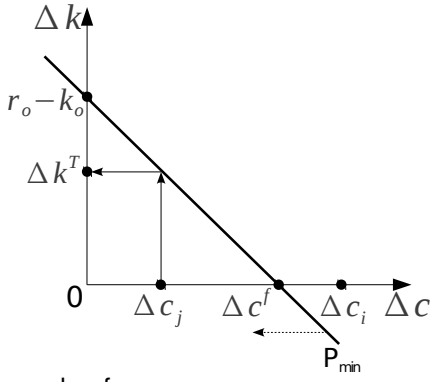


Fig. 6. Example of answer space

First, let the original k value be k_o. In PHASE-1, we computed one candidate answer during the (L, r_o) = DOMINATING(UNTIL-SEE-$\vec{m}$) operation. That candidate answer has $\Delta k = r_o - k_o$ and $\Delta c = 0$, which is the best candidate answer with the least penalty at that point. Figure 6 illustrates the answer space with that candidate answer. The penalty function that passes through that answer is also shown in the figure. We call the Δc-intercept Δc^f as the *maximum feasible* Δc *value*, meaning any sample x_i whose $\Delta c_i > \Delta c^f$ has a penalty larger than the minimum penalty P_{min} seen so far, and thus the corresponding COMPUTE-RANK operation can be skipped. Once again, when a better sample (candidate answer) is found, P_{min} is updated and the penalty function is moved towards the origin (with the same slope) and results in a smaller Δc^f. Therefore, we can expect that its pruning power becomes stronger and stronger during the process and especially strong when we have a large λ_c or small λ_k (i.e., option *Prefer Modify* k) because Δc^f will decrease at a faster rate (Equation 7).

If a sample $x_j \in S$ has $\Delta c_j < \Delta c^f$ and cannot be pruned by the above, we can try to project it to the penalty function (see Figure 6) and obtain a threshold Δk^T and a threshold ranking r^T:

$$\Delta k^T = \lfloor (P_{min} - \lambda_c \frac{\Delta c_j}{\|\vec{l} - \vec{m}\|_2}) \frac{r_o - k_o}{\lambda_k} \rfloor$$
$$r^T = \Delta k^T + k_o \quad (11)$$

r^T is the minimum ranking that x_j should achieve; otherwise, its penalty cannot be better than the current best answer. With r^T, we try to see if any processed sample x_i dominates x_j in the data space. If yes, it means x_i's score and rank are no worse than x_j, hence, $r_i \leq r_j$. Thus, if $r_i \geq r^T$, then $r_j \geq r^T$, so, x_j can be pruned because its penalty cannot be better than the penalty P_{min} achieved by the current best answer. The pseudocode of the algorithm is shown in Algorithm 2.

4.3.5 Multiple Missing Objects

To deal with multiple missing objects $M = \{m_1, ..., m_j\}$ in a why-not question, we can modify the algorithm as follows.

Algorithm 2 Answering a Why-not Top-K Dominating Question

Input:
The dataset D; original top-k dominating query $q_o(k_o)$; missing object $\vec{m}$; penalty settings λ_k, λ_d; $T\%$ and Pr

Output:
Refined query and new $\vec{m}$'s values: $\langle q'(k'), \vec{m} = \vec{x}'_{best} \rangle$

1: Result list L of the first dominating query;
2: Rank r_o of the missing object;
Phase 1:
3: $(L, r_o) \leftarrow$ DOMINATING(UNTIL-SEE-$\vec{m}$);
4: **if** $r_o = \emptyset$ **then**
5: **return "$\vec{m}$ is not in D"**;
6: **end if**
7: Determine s from $T\%$ and Pr using Equation 5;
8: Compute the lower bound point $\vec{l}$;
9: Find the restricted sample space $\mathcal{R}_s$ that bounded by $\vec{l}$ and $\vec{m}$;
10: Sample s points from $\mathcal{R}_s$ and add them into S;
Phase 2:
11: $P_{min} \leftarrow Penalty(r_o, \vec{m})$;
12: $\Delta c^f = P_{min} * \frac{\|\vec{l}-\vec{m}\|_2}{\lambda_c}$; //Find the Δc-intercept
13: $B \leftarrow \emptyset$; //The buffer stores the set of samples $\vec{b}$ with their ranking value r_b
14: **for all** $\vec{x}_i \in S$ **do**
15: **if** $\Delta c_i > \Delta c^f$ **then**
16: **continue**; //Technique (b) — skip COMPUTE-RANK operation
17: **end if**
18: $\Delta k^T \leftarrow \lfloor (P_{min} - \lambda_c \frac{\Delta c_i}{\|\vec{l}-\vec{m}\|_2}) \frac{r_o - k_o}{\lambda_k} \rfloor$;
19: $r^T \leftarrow \Delta k^T + k_o$;
20: **if** there exist objects $\vec{b} \in B$, such that $\vec{b}$ dominates $\vec{x}_i$ and $r_b \geq r^T$ **then**
21: **continue**; //Technique (b) — use cached result to skip COMPUTE-RANK operation
22: **end if**
23: $r_i \leftarrow$ COMPUTE-RANK$(\vec{m}, \vec{x}_i)$; //Technique (a)
24: $B \leftarrow B \cup (\vec{x}_i, r_i)$;
25: **if** $P_i < P_{min}$ **then**
26: $P_{min} \leftarrow P_i$;
27: $\Delta c^f = P_{min} * \frac{\|\vec{l}-\vec{m}\|_2}{\lambda_c}$;
28: **end if**
29: **end for**
Phase 3:
30: Return the $\langle k' = r_i, \vec{m} = \vec{x}'_i \rangle$ pair whose penalty=P_{min};

First, the initial dominating query stops only when all missing objects in M are seen, i.e., replace line 3 in Algorithm 2 with the use of DOMINATING(UNTIL-SEE-ALL-OBJECTS-IN-M). In addition, we set r_o be the rank of the object in M with the worst ranking.

Second, during PHASE-1, we determine the corresponding restricted sample space $\mathcal{R}_s^i$ for each missing object $m_i \in M$. Then, each sample $\vec{\mathcal{X}}_i$ is in the form $(\vec{x}_1, ..., \vec{x}_j)$, where $\vec{x}_i$ is sampled from $\mathcal{R}_s^i$. The number of such "compound" sample $\vec{\mathcal{X}}_i$ still follows Equation 5.

During the COMPUTE-RANK operation (Technique (a)), we update the scores of all missing objects to be the sampled values in that compound sample, i.e., $m_i = \vec{x}_i, \forall i = [1..j]$. Then, more or less similar to the case where only one missing object is in M, the new score of each missing object is first computed. Afterwards, only objects in L that dominate some objects in M may need to reduce their scores accordingly.

Technique (b) is largely the same, except that Δc_i now refers to the aggregated difference between the original value and the modified value of each missing object. When a compound sample $\vec{\mathcal{X}}_j$ cannot be pruned after comparing its Δc_i with the maximum feasible Δc^f value, we may also try to compute its corresponding threshold ranking r^T. Then, we see if any processed sample x_i dominates any sample in $\vec{\mathcal{X}}_j$. If yes, we check if its corresponding ranking r_i is larger than $r^T + (j-1)$. We add $(j-1)$ as to account for the fact that there are j missing objects, and prune the compound sample $\vec{\mathcal{X}}_j$ if so.

5 EXPERIMENTS

We evaluate our proposed solution using both synthetic and real data. The real data is the NBA data set. The NBA data set contains 21961 game statistics of all NBA players from 1973-2009. Each record represents the career performance of a player: player name (Player), points per game (PTS), rebounds per game (REB), assists per game (AST), steals per game (STL), blocks per game (BLK), field goal percentage (FG), free throw percentage (FT), and three-point percentage (3PT).

By default, we set the system parameters $T\%$ and Pr as 0.5% and 0.8, respectively, resulting in a sample size of 322. The algorithms are implemented in C++ and the experiments are run on a Ubuntu PC with Intel 2.67GHz i5 Dual Core processor and 4GB RAM.

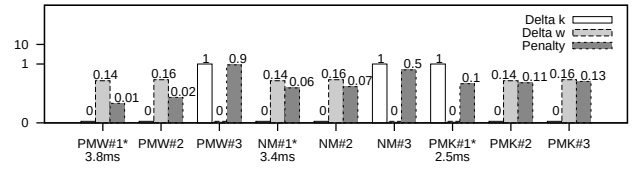
5.1 Why-Not Top-K Question

5.1.1 Case Study

Case 1 (Finding the top-3 centers in NBA history). The first case is to find the top-3 centers in NBA history. Therefore, we first issued a top-3 query q_1 with the following scoring function: $\frac{1}{5} * PTS + \frac{1}{5} * REB + \frac{1}{5} * BLK + \frac{1}{5} * FG + \frac{1}{5} * FT$. The initial result was:

Rank	Player	PTS	REB	BLK	FG	FT
1	W. Chamberlain	30	23	0	0.53	0.51
2	K. Abdul-Jabbar	25	11	2	0.55	0.72
3	Shaquille O'Neal	25	11	2	0.58	0.52

Because we were curious why Bill Russell, who won 11 champions, was not in the result, we issued a why-not question $\{\{\text{Bill Russell}\}, q_1\}$. The following shows three refined queries for each option: PMK (prefer modify k), PMW (prefer modify weighting), NM (Never mind). The ones that marked with * are the answers returned by our algorithms. The others are answers generated from others samples, we put two of them there for comparison purpose.



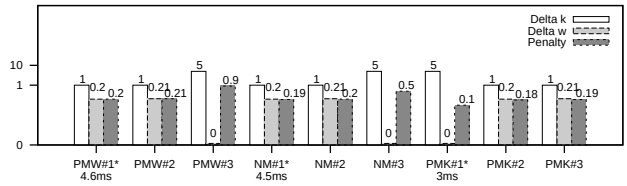
For example, using the option PMK, in 2.5ms, we got a refined query q'_1 with $k' = 4$ and $\vec{w}' = \vec{w}_o$, resulting in a penalty of 0.1, and the new top-k result was:

Rank	Player	PTS	REB	BLK	FG	FT
1	W. Chamberlain	30	23	0	0.53	0.51
2	Abdul-jabbar	25	11	2	0.55	0.72
3	Shaquille O'neal	25	11	2	0.58	0.52
4	Bill Russell	15	22	0	0.43	0.56

Case 2 (Finding the top-3 guards in NBA history). The second case is to find the top-3 guards in NBA history. We issued a top-3 query q_2 with equal weighting ($\frac{1}{6}$) on six attributes PTS, AST, STL, FG, FT, and 3PT. The initial result was:

Rank	Player	PTS	AST	STL	FG	FT	3PT
1	Michael Jordan	30	5	2	0.49	0.83	0.32
2	LeBron James	28	7	2	0.47	0.73	0.32
3	Oscar Robertson	26	10	0	0.48	0.83	0

We wondered why Kobe Bryant was missing in the answer, so we posed a why-not question $\{\{\text{Kobe Bryant}\},q_2\}$. The following is a quick visual of the quality of the answers of our results and two random samples:



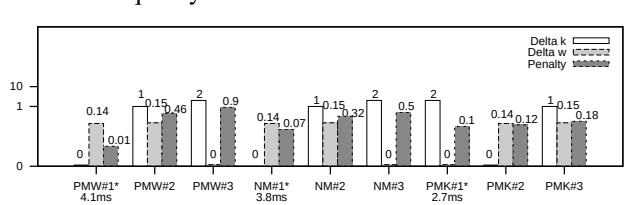
For example, using the option "Never Mind", in 4.5ms, we got a refined query q_2' with $k' = 4$ and $\vec{w} = |0.083\ 0.037\ 0.243\ 0.101\ 0.397\ 0.123|$. The corresponding new result was:

Rank	Player	PTS	AST	STL	FG	FT	3PT
1	Michael Jordan	30	5	2	0.49	0.83	0.32
2	LeBron James	28	7	2	0.47	0.73	0.32
3	Allen Iverson	27	6	2	0.42	0.78	0.31
4	Kobe Bryant	25	5	2	0.45	0.83	0.34

Case 3 (Finding the top-3 players in NBA history). The third case is to find the top-3 players in NBA history. Therefore, we issued a top-3 query q_4 with equal weighting ($\frac{1}{8}$) on all eight numeric attributes: $\frac{1}{8} * PTS + \frac{1}{8} * REB + \frac{1}{8} * AST + \frac{1}{8} * STL + \frac{1}{8} * BLK + \frac{1}{8} * FG + \frac{1}{8} * FT + \frac{1}{8} * 3PT$. The initial result was:

Rank	Player	PTS	REB	AST	STL	BLK	FG	FT	3PT
1	W. Chamberlain	30	23	4	0	0	0.53	0.51	0
2	LeBron James	28	7	7	2	1	0.47	0.73	0.32
3	Elgin Baylor	27	14	4	0	0	0.43	0.77	0

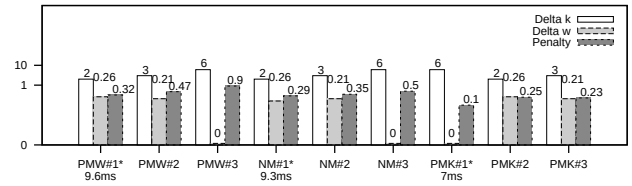
We wondered why Michael Jordan was missing. So, we issued a *why-not* question $\{\{\text{Michael Jordan}\}, q_4\}$. A quick visual of the quality of our answers and others is:



For example, using the "Prefer modify k" option, in 2.7ms, our algorithm returned a refined query q_4' with $k' = 5$ and $\vec{w}' = \vec{w}_o$, with the following result: {W. Chamberlain, LeBron James, Elgin Baylor, Bob Pettit, Michael Jordan}.

The refined query q_4' essentially means that our initial weightings were reasonable but we should have looked for the top-5 players instead.

As a follow up, we were also interested in understanding why both Michael Jordan and Shaquille O'neal were not the top-3 players in NBA history. Therefore, we issued another why-not query $\{\{\text{Michael Jordan, Shaquille O'neal}\}, q_4\}$, the quick visual is as follows:



For example, using the "Never mind" option, in 9.3ms, our algorithm returned a refined query q_4'' with $k' = 5$ and $\vec{w}_3' = |0.148\ 0.0747\ 0.0539\ 0.1066\ 0.2581\ 0.1143\ 0.1231\ 0.1212|$. The corresponding result of q_4'' was:

Rank	Player	PTS	REB	AST	STL	BLK	FG	FT	3PT
1	W. Chamberlain	30	23	4	0	0	0.53	0.51	0
2	Michael Jordan	30	6	5	2	1	0.49	0.83	0.32
3	LeBron James	28	7	7	2	1	0.47	0.73	0.32
4	Abdul-jabbar	25	11	4	1	2	0.55	0.72	0.05
5	Shaquille O'neal	25	11	3	1	2	0.58	0.52	0.25

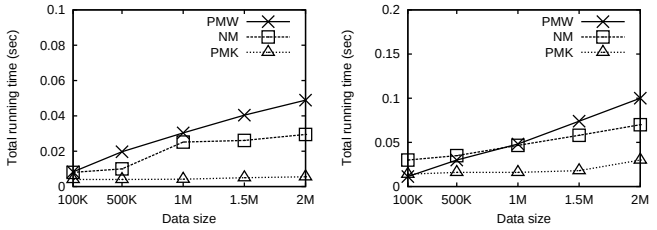
5.1.2 Performance

TABLE 2
Parameter settings

Parameter	Ranges
Data size	100K, 500K, **1M**, 1.5M, 2M
Dimension	2, **3**, 4, 5
k_o	5, **10**, 50, 100
Actual ranking of $\vec{m}$ under q_o	11, **101**, 501 ,1001
$T\%$	10%, 5%, 1%, **0.5%**, 0.1%
Pr	0.1, 0.2, 0.3, 0.4,0.5, 0.6, 0.7, **0.8**, 0.9
$\|M\|$	**1**, 2, 3, 4, 5

We next turn our focus to the performance of Algorithm 1. We present experimental results based on three types of synthetic data: uniform (UN), correlated (CO) and anti-correlated (AC). Since the experimental conclusion of CO is similar to the experimental conclusions of UN and AC, we only present the results of UN and AC here. Table 2 shows the parameters we varied in the experiments. The default values are in bold face. The default top-k query q_o uses a monotonic linear function and has a setting of: $k = k_o$, $\vec{w}_o = |\frac{1}{d}\ \cdots\ \frac{1}{d}|$, where d is the number of dimensions (attributes involved). By default, the why-not question asks for a missing object that is ranked $(10 * k_o + 1)$-th under $\vec{w}_o$.

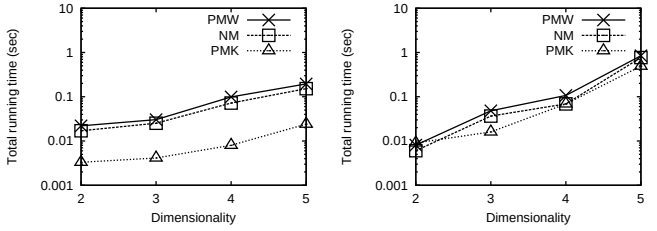
Varying Data Size. Figure 7 shows the running time of our algorithm under different data sizes, using different penalty options (PMK stands for "Prefer modifying k", PMW stands for "Prefer modifying weighting", NM stands for "Never mind") We can see our algorithm for answering why-not questions scales linearly with the data size. In general, the



(a) Uniform Data (b) Anti-correlated Data

Fig. 7. [Why-Not TopK] Varying data size vs. Time

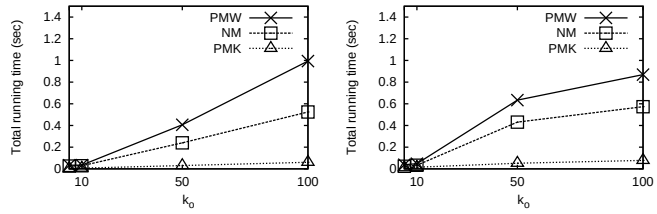
running times on AC data is longer than on uniform data because progressive top-k operations on anti-correlated data takes a longer time to finish [20]. The performance of our algorithm using the PMK option is generally better than using the other options because of the reason we discussed in Section 3.3.3, i.e., a larger λ_w makes our optimization techniques even stronger.



(a) Uniform Data (b) Anti-correlated Data

Fig. 8. [Why-Not TopK] Varying query dimension vs. Time

Varying Query Dimension. Figure 8 shows the running time of our algorithm using top-k queries with different numbers of query dimensions. We can see that answering why-not questions for top-k queries of higher dimensions needs more time because the execution time of a progressive top-k operation increases if a top-k query involves more attributes.

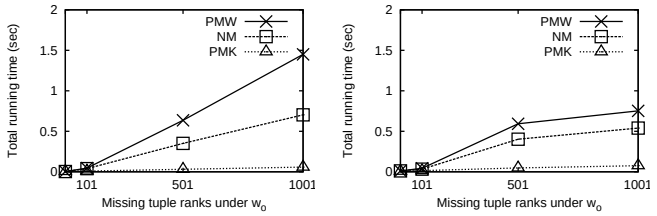


(a) Uniform Data (b) Anti-correlated Data

Fig. 9. [Why-Not TopK] Varying k_o vs. Time

Varying k_o. Figure 9 shows the running time of our algorithm using top-k queries with different k_o values. In this experiment, when a top-5 query ($k_o = 5$) is used, the corresponding why-not question is to ask why the object in rank 51st is missing. Similarly, when a top-50 query ($k_o = 50$) is used, the corresponding why-not question is to ask why the object in rank 501st is missing. Naturally, when k_o increases, the time to answer a why-not question should also increase because the execution time of a progressive top-k operation also increases with k. Figure 9 shows that our algorithm scales well with k_o.

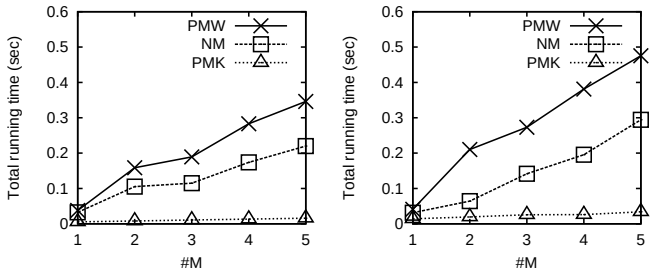
Varying the missing object to be inquired. We next study the performance of our algorithm by posing why-not questions with missing objects from different rankings. In this



(a) Uniform Data (b) Anti-correlated Data

Fig. 10. [Why-Not TopK] Varying the ranking of the missing object vs. Time

experiment, the default top-10 query is used. We asked four individual why-not questions about why the object that ranked 11th, 101st, 501st, and 1001st, respectively, is missing in the result. Figure 10 shows that our algorithm scales well with the ranking of the missing object. Of course, when the missing object $\vec{m}$ has a worse ranking under the original weighting $\vec{w_o}$, the progressive top-k operation should take a longer time to discover it in the result and thus the overall running time must increase. Again, because our optimization techniques are especially effective when the PMK option is used, the running time of our algorithm increases very little under that option.

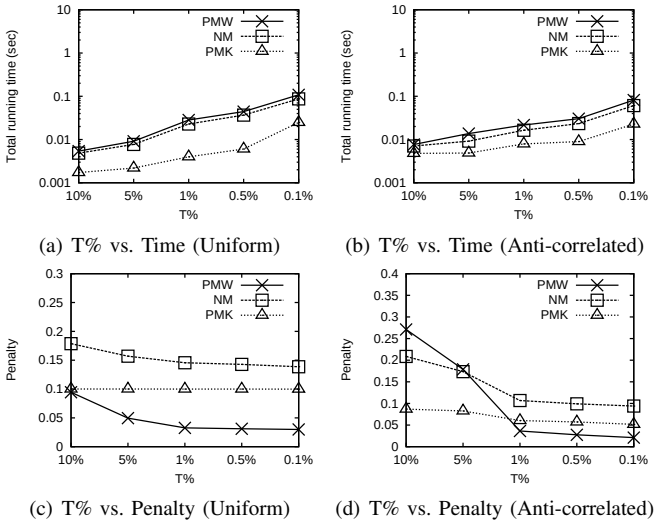


(a) Uniform Data (b) Anti-correlated Data

Fig. 11. [Why-Not TopK] Varying $|M|$ vs. Time

Varying the size of M. We also study the performance of our algorithm by posing why-not questions with different numbers of missing objects. In this experiment, the default top-10 query is used and five why-not questions are asked. In the first question, one missing object that ranked 101-th under $\vec{w_o}$ is included in M. In the second question, two missing objects that respectively ranked 101-th and 201-th under $\vec{w_o}$ are included in M. The third to the fifth questions are constructed similarly. Figure 11 shows that our algorithm scales linearly with respect to different sizes of M.

Varying $T\%$. We would also like to know how the performance and solution quality of our algorithm vary when we look for refined queries with different quality guarantees. Figure 12 shows the running time of our algorithm and the penalty of the returned refined queries when we changed from accepting refined queries that are within the best 10% ($|S| = 16$) to accepting refined queries that are within the best 0.1% ($|S| = 1609$). From Figures 12(a) and 12(b), we can see that the running time of our algorithm increases when the guarantee is more stringent. However, from Figures 12(c) and 12(d), we can see that the solution quality of the algorithm improves when T increases, until T reaches 1%



(a) T% vs. Time (Uniform) (b) T% vs. Time (Anti-correlated)

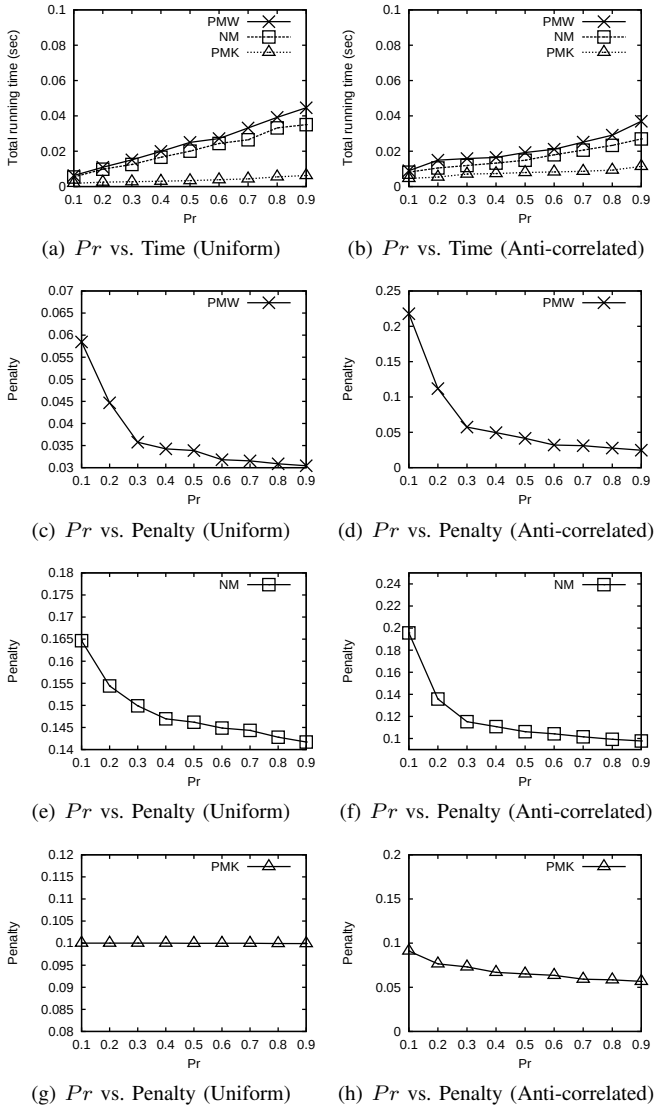
(c) T% vs. Penalty (Uniform) (d) T% vs. Penalty (Anti-correlated)

Fig. 12. [Why-Not TopK] Varying T% vs. Time/Penalty

where further increases the sample size cannot yield any significant improvement. The solution quality of our algorithm under the option PMK on uniform data is constant because option PMK prefers *not* to change the weighting. So, the query $(r_o, \vec{w_o})$, which has $\Delta w = 0$, is the best refined query over all different values of T. This explains why the solution quality remains constant.

Varying Pr. The experimental result of varying Pr, the probability of getting the best-$T\%$ of refined queries, is similar to the results of varying $T\%$ above. That is because both parameters are designed for controlling the quality of the approximate solutions. In Figure 13, we can see that when we vary Pr from 0.1 ($|S| = 22$) to 0.9 ($|S| = 460$), the running times increase mildly. However, the solution quality also increases gradually, except under the option PMK on uniform data, because of the same reason we described above.

Effectiveness of Optimization Techniques. Finally, we investigate the effectiveness of the two optimization techniques we used in our algorithm. Figure 14 shows the performance of our algorithm using only Technique (i), only Technique (ii), both, and none, under the default setting. The effectiveness of both techniques is also very promising. Without using any optimization technique, the algorithm requires a running time of about 6 seconds on uniform data and about 730 seconds on anti-correlated data. However, our algorithm respectively runs about two and four orders faster on uniform data and anti-correlated data when our optimization techniques are enabled.



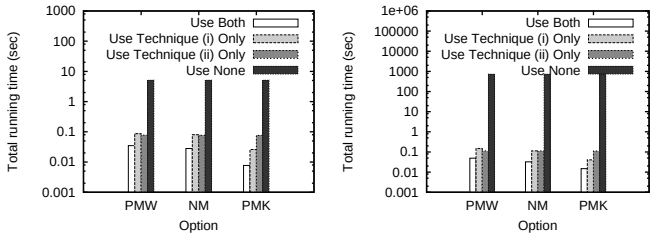
(a) Pr vs. Time (Uniform) (b) Pr vs. Time (Anti-correlated)

(c) Pr vs. Penalty (Uniform) (d) Pr vs. Penalty (Anti-correlated)

(e) Pr vs. Penalty (Uniform) (f) Pr vs. Penalty (Anti-correlated)

(g) Pr vs. Penalty (Uniform) (h) Pr vs. Penalty (Anti-correlated)

Fig. 13. [Why-Not TopK] Varying Pr vs. Time/Penalty



(a) Uniform Data (b) Anti-correlated Data

Fig. 14. [Why-Not TopK] Optimization Effectiveness

5.2 Why-Not Top-K Dominating Question

In this section, we repeat the case study and performance study using top-k dominating queries.

5.2.1 Case Study

Case 1 (Finding the top-3 centers in NBA history). The following shows the result of a top-3 dominating query q_5 posed on five attributes: PTS, REB, BLK, FG, and FT.

Rank	Player	PTS	REB	BLK	FG	FT
1	Yao Ming	19	9	2	0.52	0.83
2	Brook Lopez	13	8	2	0.53	0.79
3	Bob Lanier	20	10	1	0.51	0.76

We wondered why Kareem Abdul-Jabbar, the famous center in Los Angeles Lakers, was missing in the result, and so we posed a why-not question {{Kareem Abdul-Jabbar},q_5} using the "Prefer modify k" option because Abdul-Jabbar is already retired and it made no sense to modify his attribute values

anymore. In 30ms, we got an answer without modifying the attribute value of Abdul-Jabbar, but the k value was suggested to increase from 3 to 5:

Rank	Player	PTS	REB	BLK	FG	FT
1	Yao Ming	19	9	2	0.52	0.83
2	Brook Lopez	13	8	2	0.53	0.79
3	Bob Lanier	20	10	1	0.51	0.76
4	Dan Issel	20	8	1	0.5	0.79
5	Kareem Abdul-Jabbar	25	11	2	0.55	0.72

Case 2 (Finding the top-3 guards in NBA history). Next, we posed a top-3 dominating query q_6 on attributes PTS, AST, STL, FG, FT, and 3PT to look for the top-3 guards in NBA history. The original result was:

Rank	Player	PTS	AST	STL	FG	FT	3PT
1	Steve Nash	14	8	1	0.48	0.9	0.43
2	Mark Price	15	7	1	0.47	0.9	0.4
3	Jeff Hornacek	15	5	1	0.49	0.87	0.4

Since Kobe Bryant was not in the result and we hoped to discover what aspects of his game Kobe Bryant could improve in order to appear in the result, we asked a why-not question $\{\{$Kobe Bryant$\}$, q_6 $\}$ using the "Prefer modify object's value" option. In 54ms, we got the answer below, with Bryant becomes the top-1 guard in the NBA:

Rank	Player	PTS	AST	STL	FG	FT	3PT
1	Kobe Bryant	28	5	2	0.47	0.87	0.43
2	Steve Nash	14	8	1	0.48	0.9	0.43
3	Mark Price	15	7	1	0.47	0.9	0.4

The original data value of Kobe Bryant is shown below:

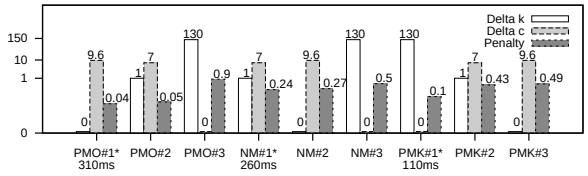
Player	PTS	AST	STL	FG	FT	3PT
Kobe Bryant	25	5	2	0.45	0.83	0.34

We can see that Kobe Bryant can perhaps improve his field goal (FG), free throw (FT), or three-point (3PT) shooting percentages in order to be ranked as the best guard in NBA history.

Case 3 (Finding the top-3 players in NBA history). The last case is again to find the top-3 players in NBA history. We posed a top-3 dominating query q_7 on all eight attributes. The initial result was:

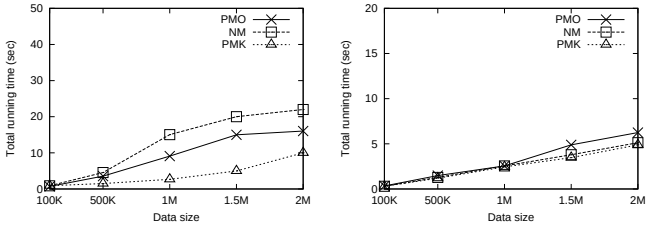
Rank	Player	PTS	REB	AST	STL	BLK	FG	FT	3PT
1	Larry Bird	24	10	6	2	1	0.49	0.88	0.38
2	Dirk Nowitzki	23	9	3	1	1	0.47	0.87	0.37
3	Cris Mullin	18	4	3	2	1	0.5	0.86	0.38

We were curious that why LeBron James and Kobe Bryant were missing. So, we issued a why-not question: $\{\{$LeBron James, Kobe Bryant$\}$, $q_7\}$. The following shows a visual of the quality of the answers returned by our algorithm (marked with *) and two answered generated from random samples:



5.2.2 Performance

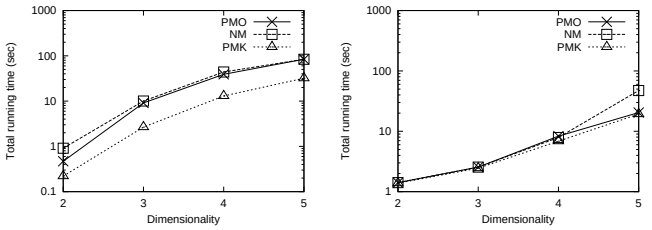
We next evaluate the performance of Algorithm 2. The experiment settings and the default values follow Table 2.



(a) Uniform Data (b) Anti-correlated Data

Fig. 15. [Why-Not Dominating] Varying data size vs. Time

Varying Data Size. Figure 15 shows the running time of our algorithm under different data sizes, using different penalty options (PMK stands for "Prefer modifying k", PMO stands for "Prefer modifying object's values", NM stands for "Never mind"). We can see that our algorithm scales linearly with the data size. Note that it is normal that the running time of algorithm on the uniform data is higher than on anti-correlated data in answering why-not dominating questions. In uniform data, objects with top rankings usually have large scores (i.e., dominating many objects), so it takes a relatively longer time to find a sample data value for $\vec{m}$ that makes it dominate those objects with high scores, thereby increasing the algorithm's running time. In contrast, objects in anti-correlated data usually have low scores (i.e., dominating few objects). Therefore it is relatively easy to find a sample data value for $\vec{m}$ that makes it dominate those objects with low scores. Again, our optimization techniques are especially effective when the PMK option is used, so the running time of our algorithm under that option is generally faster, but the effect is less obvious in anti-correlated data because our algorithm runs especially fast on anti-correlated data.

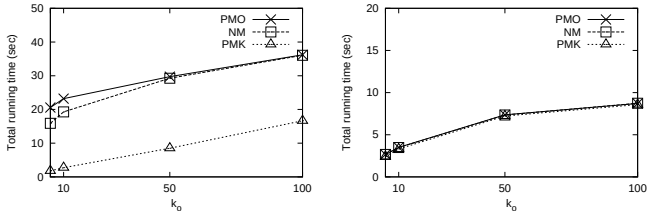


(a) Uniform Data (b) Anti-correlated Data

Fig. 16. [Why-Not Dominating] Varying query dimension vs. Time

Varying Query Dimension. Figure 16 shows the running time of our algorithm of answering why-not questions on top-k dominating queries with different numbers of query dimensions. We can see that answering why-not questions for queries with more dimensions needs more time because the execution times of the DOMINATING function and the COMPUTE-RANK function increase with the number of dimensions.

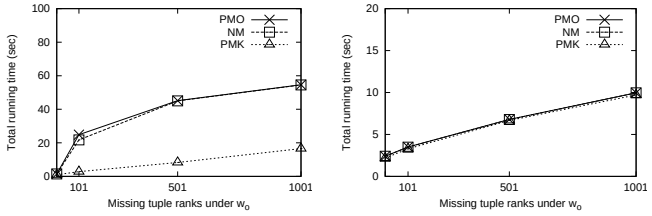
Varying k_o. Figure 17 shows the running time of our algorithm using top-k queries with different k_o values. Recall that when a top-5 query ($k_o = 5$) is used, the corresponding why-not question is to ask why the object in rank 51st is missing. And when a top-50 query ($k_o = 50$) is used, the corresponding why-not question is to ask why the object in rank 501st is missing. So, naturally, when k_o increases, the time to answer a why-not question also increases because the execution times of both DOMINATING and COMPUTE-RANK



(a) Uniform Data (b) Anti-correlated Data

Fig. 17. [Why-Not Dominating] Varying k_o vs. Time

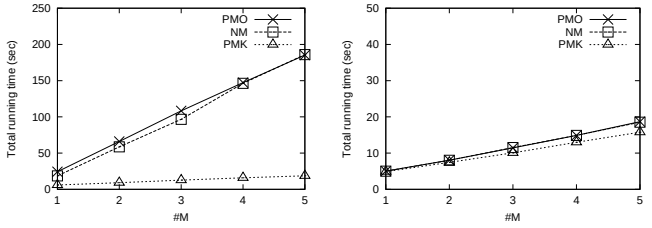
functions increase with k. Figure 17 shows that our algorithm scales well with k_o.



(a) Uniform Data (b) Anti-correlated Data

Fig. 18. [Why-Not Dominating] Varying the ranking of the missing object vs. Time

Varying the missing object to be inquired. Figure 18 shows the results when we vary the ranking of the missing object of the default top-10 query. We can see that our algorithm scales well with the missing object's ranking.

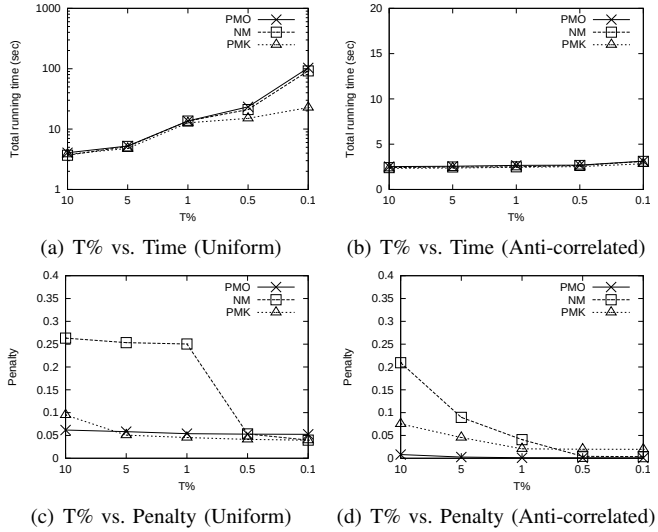


(a) Uniform Data (b) Anti-correlated Data

Fig. 19. [Why-Not Dominating] Varying $|M|$ vs. Time

Varying the size of M. Figure 19 shows the results when we vary the number of missing objects in a why-not question. We can see that our algorithm scales well with the number of missing objects. Once again the running times increase gradually when the PMK option is used and all three options are all very efficient when the data is anti-correlated.

Varying $T\%$. Figure 20 shows the running time of our algorithm and the penalty of the returned refined queries when we changed from accepting refined queries that are within the best 10% ($|S| = 16$) to accepting refined queries that are within the best 0.1% ($|S| = 1609$). The running time of our algorithm increases when the guarantee is more stringent, although the increase is relatively mild for the anti-correlated dataset. The solution quality of the algorithm improves when T increases and remains steady beyond a certain sample size. The solution quality for the NM option increases significantly from $T = 1\%$ to $T = 0.5\%$ on the uniform data because NM

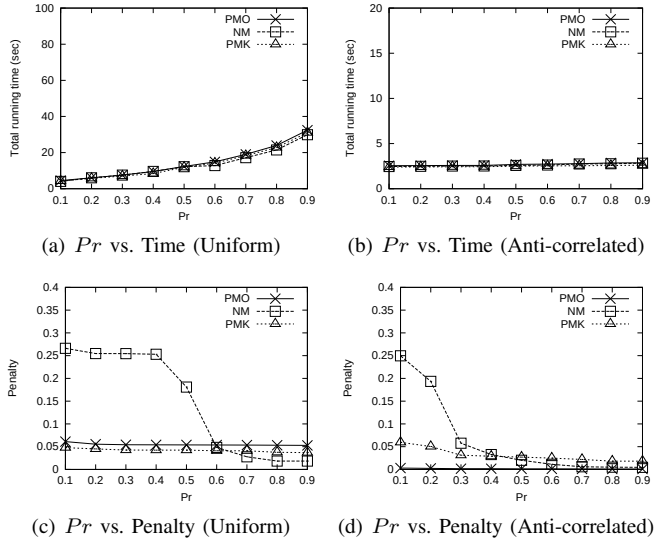


(a) T% vs. Time (Uniform) (b) T% vs. Time (Anti-correlated)

(c) T% vs. Penalty (Uniform) (d) T% vs. Penalty (Anti-correlated)

Fig. 20. [Why-Not Dominating] Varying T% vs. Time/Penalty

does not put any preference on Δk and Δc. Generally, NM is more difficult than the other two options (which have clear preferences on either minimizing Δk or Δc) to find a good answer. Therefore, it needs a larger sample size in order to reach a stable state. In this experiment, NM reaches its stable state when $T\% = 0.5\%$.

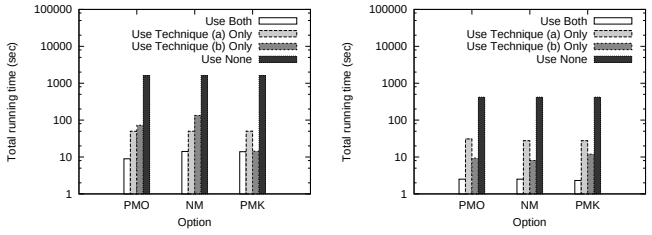


(a) Pr vs. Time (Uniform) (b) Pr vs. Time (Anti-correlated)

(c) Pr vs. Penalty (Uniform) (d) Pr vs. Penalty (Anti-correlated)

Fig. 21. [Why-Not Dominating] Varying Pr

Varying Pr. The experimental result of varying Pr, the probability of getting the best-$T\%$ of refined queries, is similar to the results of varying $T\%$ above. That is because both parameters are designed for controlling the quality of the approximate solutions. In Figure 21, we can see that when we vary Pr from 0.1 ($|S| = 22$) to 0.9 ($|S| = 460$), the running times increase mildly on the uniform data and remain roughly constant on the anti-correlated data. However, the solution quality also improves mildly, except that when the option NM is used, the solution quality improves sharply at $Pr = 0.5$ (uniform data) and $Pr = 0.3$ (anti-correlated data) because of the same reason we described above.



(a) Uniform Data (b) Anti-correlated Data

Fig. 22. [Why-Not Dominating] Optimization Effectiveness

Effectiveness of Optimization Techniques. Finally, we investigate the effectiveness of the two optimization techniques we used in our algorithm. Figure 22 shows the performance of our algorithm using only Technique (a), only Technique (b), both, and none, under the default setting. The effectiveness of both techniques are also very promising. Without using any optimization technique, the algorithm requires about 1500 seconds and 400 seconds on uniform dataset and anti-correlated dataset, respectively. But when our optimization techniques are enabled, the algorithm runs about two orders of magnitude faster — it requires only about 10 seconds and 2 seconds on uniform dataset and anti-correlated dataset, respectively.

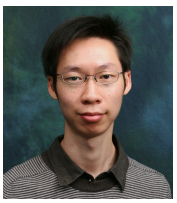
6 Conclusion

In this paper, we have studied the problem of answering why-not questions on two types of top-k queries: the basic top-k query where users need to specify the set of weightings, and the top-k dominating query where users do not need to specify the set of weightings because the ranking function ranks an object higher if it can dominate more objects. Our target is to give an explanation to a user who is wondering why her expected answers are missing in the query result. Since the problems are different, we use different explanation models for top-k queries and top-k dominating queries. For the former, we return the user a refined query with approximately minimal changes to the k value and their weightings. For the latter, we return the user a refined query with approximately minimal changes to the k value and the missing objects' data values. Our case studies and experimental results show that our solutions efficiently return very high quality solutions. In future work, we will study the issues about non-numeric attributes.

References

[1] H. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu, "Making Database Systems Usable," in *SIGMOD*, 2007, pp. 13–24.
[2] S. Agrawal, S. Chaudhuri, and G. Das, "DBXplorer: A System for Keyword-Based Search over Relational Databases," in *ICDE*, 2002, pp. 5–16.
[3] H. Wu, G. Li, C. Li, and L. Zhou, "Seaform: Search-As-You-Type in Forms," in *PVLDB*, vol. 3, no. 2, 2010, pp. 1565–1568.
[4] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N. Polyzotis, and J. S. V. Varman, "SQL QueRIE Recommendations," in *PVLDB*, vol. 3, no. 2, 2010, pp. 1597–1600.
[5] M. B. N. Khoussainova, Y.C. Kwon and D. Suciu, "Snipsuggest: Context-Aware Autocompletion for SQL," in *PVLDB*, vol. 4, no. 1, 2010, pp. 22–33.
[6] A. Chapman and H. Jagadish, "Why not?" in *SIGMOD*, 2009, pp. 523–534.
[7] J. Huang, T. Chen, A.-H. Doan, and J. F. Naughton, "On the Provenance of Non-Answers to Queries over Extracted Data," in *PVLDB*, 2008, pp. 736–747.
[8] M. Herschel and M. A. Hernández, "Explaining Missing Answers to SPJUA Queries," in *PVLDB*, 2010, pp. 185–196.
[9] Q. T. Tran and C.-Y. Chan, "How to ConQueR Why-not Questions," in *SIGMOD*, 2010, pp. 15–26.
[10] M. L. Yiu and N. Mamoulis, "Efficent Processing of Top-k Dominating Queries on Multi-Dimensional Data," in *VLDB*, 2007, pp. 541–552.
[11] S. Borzsonyi, D. Kossmann, and K. Stocker, "The Skyline Operator," *ACM Trans. Database System*, vol. 25, no. 2, pp. 129–178, 2000.
[12] Z. He and E. Lo, "Answering Why-Not Questions on Top-K Queries," in *ICDE*, 2012.
[13] A. Motro, "Query Generalization: A Method for Interpreting Null Answers," in *Expert Database Workshop*, 1984, pp. 597–616.
[14] A. Motro, "SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems," *ACM Trans. Inf. Syst.*, vol. 4, no. 4, pp. 312–330, 1986.
[15] F. Zhao, K.-L. T. G. Das, and A. K. H. Tung, "Call to Order: A Hierarchical Browsing Approach to Eliciting Users' Preference," in *SIGMOD*, 27-38, p. 2010.
[16] A. Vlachou, C. Doulkeridis, Y. Kotidis, and K. Nørvåg, "Reverse Top-K Queries," in *ICDE*, 365-376, p. 2010.
[17] L. D. Berkovitz, *Convexity and Optimization in* $\mathbb{R}^n$. A Wiley-Interscience pubication, 2002.
[18] R. Faginm, A. Lotem, and M. Naor, "Optimal aggregation algorithm for middleware," *J. Comput. Syst. Sci.*, vol. 64, no. 4, pp. 614–656, 2003.
[19] Y. C. Chang, L. Bergman, V. Castelli, M. L. C.S. Li, and J. Smith, "The Onion Technique: Indexing for Linear Optimization Queries," in *SIGMOD*, 391-402, p. 2000.
[20] Z. L and L. Chen, "Dominant Graph: An Efficient Indexing Structure to Answer Top-K Queries," in *ICDE*, 536-545, p. 2008.
[21] Y. F. Tao, X. K. Xiao, and J. Pei, "Efficient Skyline and Top-k Retrieval in Subspaces," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 8, pp. 1072–1088, 2007.
[22] E. Tiakas, A. N. Papadopoulos, and Y. Manolopoulos, "Progressive processing of subspace dominating queries," *VLDB J.*, vol. 20, no. 6, pp. 921–948, 2011.

Andy He is currently a PhD candidate in the Department of Computing, Hong Kong Polytechnic University. He obtained his Bachelor degree in 2010 from Zhejiang University. His research focuses on database usability and query processing.



Eric Lo received a PhD degree in 2007 from ETH Zurich. He is currently an assistant professor in the Department of Computing, Hong Kong Polytechnic University. His research interests include query processing, benchmarking, OLAP, and database usability.