On Answering Why-not Questions in Reverse Skyline Queries

Md. Saiful Islam, Rui Zhou and Chengfei Liu

Swinburne University of Technology, VIC 3122, Australia

{mdsaifulislam, rzhou, cliu}@swin.edu.au

***Abstract*—This paper aims at answering the so called why-not questions in reverse skyline queries. A reverse skyline query retrieves all data points whose dynamic skylines contain the query point. We outline the benefit and the semantics of answering why-not questions in reverse skyline queries. In connection with this, we show how to modify the why-not point and the query point to include the why-not point in the reverse skyline of the query point. We then show, how a query point can be positioned safely anywhere within a region (i.e., called safe region) without losing any of the existing reverse skyline points. We also show how to answer why-not questions considering the safe region of the query point. Our approach efficiently combines both query point and data point modification techniques to produce meaningful answers. Experimental results also demonstrate that our approach can produce high quality explanations for why-not questions in reverse skyline queries.**

I. Introduction

In recent years, why-not questions have received a considerable amount of attention in the database community in the hope of improving the usability of database systems. Today's database systems are highly efficient in terms of query execution time and resource usage. However, these systems are not usable for the end users to the same degree as they are proficient in underlying data management and query evaluation [1]. These days users expect systems to be more interactive and cooperative. That is, users are not satisfied only with receiving the query output, but also they want to know why the system returns only the current set of objects as output. In particular, users may want to know why a particular data object does not appear in the query output. Any database system that provides a good explanation for the missing objects in the query output, is very helpful for a user to understand her information need and thereby refine her initial query [2], [3].

There are three different aspects of answering "why-not questions" for query output. The first one is finding the causes why the expected data point does not appear in the query output. Chapman et al. [4] propose to return the query operator that filters out the desired data point from the query output in this direction. The second aspect is modifying the data point so that it appears in the query output in terms of the modified database as proposed by Huang et al. [5] for SPJ (Select-Project-Join) and by Herschel et al. [6] for SPJUA (SPJ-Union-Aggregation) queries, respectively. The third aspect is refining the initial query so that the why-not point appears in the refined query output as proposed by Tran et al. [2] for SPJA (SPJ-Aggregation) queries. In a recent work, He et al. [3] also propose a query refinement approach for answering why-not questions in top-k queries. In this paper, we study the problem of answering why-not questions in reverse skyline queries in light of the above aspects.

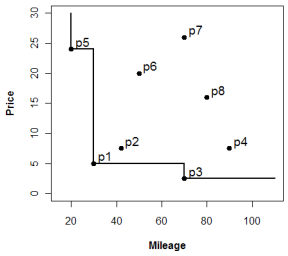
To introduce reverse skyline, we introduce dynamic skyline [7] first. Given a set of products P and a query point q as a customer's preference, a dynamic skyline query retrieves all products that are not dynamically dominated by other products from the customer's perspective. A product p_1 is considered as dynamically dominating another product p_2 with respect to a customer if p_1 compared with p_2 is closer to the customer's preference in at least one dimension and not farther to the customer's preference in the other dimensions. While regular skyline [8] prefers maximum or minimum values in each dimension, dynamic skyline prefers products closer to a given customer's preference. In other words, dynamic skyline adheres to the around-by semantics, under which a cheap product may not be necessarily preferable to an expensive one if the latter matches the customer's preference better. Based on dynamic skyline, a reverse skyline query retrieves information from the companies' perspectives. That is, given a set of products P, a query product q and a set of customer preferences C, a reverse skyline query according to q retrieves all customers that contain q in their dynamic skylines [9]. A reverse skyline query is used to measure the interestingness of a product in the market [10]. Consider the example in Fig.1(a): a database of cars and customer preferences are stored as tuples in a relation. Suppose pt_2 is a customer preference $c_2 \in C$, $pt_1, pt_3 - pt_8$ are cars $p_1, p_3 - p_8 \in P$, the dynamic skyline of c_2 can be found as $\{p_1, p_4, p_6\}$ (let us take the result for granted, visualized justification is in Section II). This means customer c_2 is interested in cars $\{p_1, p_4, p_6\}$. Now, a car dealer wants to put a car q(price:8.5K, mileage:55K) onto the market and see which customers are interested in this car. After careful examination, c_2's dynamic skyline becomes $\{p_1, p_4, p_6, q\}$ including q, which means c_2 is in the reverse skyline of q, so customer c_2 is a potential buyer of the car q. Similarly, c_2 is also in the reverse skylines of p_1, p_4 and p_6.

To answer why-not questions in reverse skyline queries, we aim to find out why a particular point is not in a reverse skyline, and what actions we should take to put the point into a reverse skyline. Let us illustrate the problem using an example. Consider again the car database, this time, let pt_1

978-1-4673-4910-9/13/$31.00 © 2013 IEEE

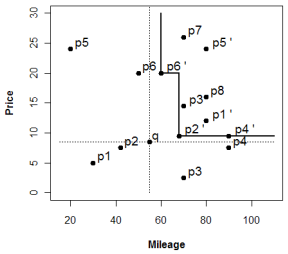
ID	Price($)	Mileage(m)
pt_1	5K	30K
pt_2	7.5K	42K
pt_3	2.5K	70K
pt_4	7.5K	90K
pt_5	24K	20K
pt_6	20K	50K
pt_7	26K	70K
pt_8	16K	80K

(a) Data points served as products and customers

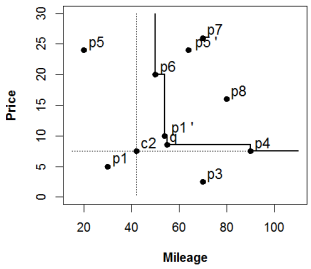


(b) Skyline

Fig. 1. An skyline query example



(a) Dynamic skyline of q



(b) Dynamic skyline of c_2

Fig. 2. Dynamic skylines of q (8.5K, 55K) and c_2

be a customer preference $c_1 \in C$, let $pt_2 - pt_8$ be the cars $p_2 - p_8 \in P$ on the market, suppose a car dealer wants to sell a car q(price:8.5K, mileage:55K), after a careful reverse skyline computation, we found c_1 is not in the reverse skyline of q, then the car dealer may want to know why c_1 is not interested in q. Firstly, we can explain the reason as that car p_2 is more interesting to customer c_1, because q is dynamically dominated by p_2 according to c_1. To go further, the car dealer may seek a negotiation with the customer and make q turn up in the dynamic skyline of the customer c_1 (i.e., c_1 becomes a reverse skyline point of q). This includes changing the price of the car q or persuading the customer to change her preference or both for the purpose of narrowing the gap. An important aspect here is that the car dealer might not want to lose existing customers who are already interested in the car q, therefore during the negotiation, it may be better to keep the existing q's reverse skyline points.

To sum up, this paper aims at answering *why-not* questions in reverse skyline queries. More specifically, we show how to modify the why-not point and query point to include the why-not point in the reverse skyline of the query point. To do so, we propose techniques that incur minimum changes to both the why-not point and the query point. We also show how to modify the why-not point and the query point while keeping its existing reverse skyline points. To the best of our knowledge, this is the first attempt ever made to answer why-not questions in reverse skyline queries. The main contributions of this paper are summarized as follows:

(1) We provide the semantics of answering why-not question in reverse skyline queries.
(2) Then, we show how to modify the why-not data point and the query point to include the why-not data point in the reverse skyline list of the query point, respectively.
(3) We also show how to modify both the query point and the why-not point while keeping existing reverse skyline.
(4) Finally, we present a detailed evaluation of the proposed scheme that demonstrates its effectiveness in both real and synthetic data sets.

Here is a road map of the paper. Section II describes preliminaries and terminology used in this paper. Section III describes the semantics of answering why-not questions in reverse skyline queries in detail. Section IV describes how to modify the why-not point. Section V describes how to modify the query point while keeping existing reverse skyline points. Section VI presents our experiments. Section VII presents related work and finally, Section VIII concludes our paper.

II. Preliminaries

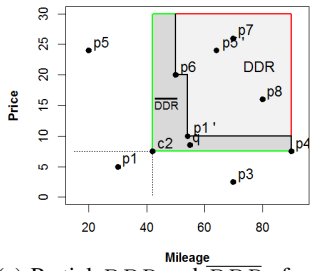
Let $D = (D^1, ..., D^d)$ be a d-dimensional data space, $P \subseteq D$ be the dataset of products and $C \subseteq D$ be the dataset of customer preferences. Each D^i refers to the universe of discourse for the i^{th} dimension. We assume that each D^i consists of numeric values only. A point $p \in P$ is represented as $p = \{p^1, p^2, ..., p^d\}$, where $p^i \in D^i$ and $i \in \{1, 2, ..., d\}$. Similarly, a point $c \in C$ is represented as $c = \{c^1, c^2, ..., c^d\}$, where $c^i \in D^i$ and $i \in \{1, 2, ..., d\}$. We contextualize the previous definitions of skyline [8], dynamic skyline [7] and reverse skyline [9] in this paper as follows.

Definition 1: (**Skyline**) Given a dataset of products P, the Skyline (SK) query retrieves all points p in P that are not dominated by others, and a point p_1 dominates another point p_2 (denoted by $p_1 \succ p_2$) iff (1) $\forall i \in \{1, ..., d\} : p_1^i \leq p_2^i$ and (2) $\exists j \in \{1, ..., d\} : p_1^j < p_2^j$. Equivalently, a point p_1 is in SK iff $\forall p_2 \not\equiv p_1, \exists i \in \{1, ..., d\} : p_1^i < p_2^i$.

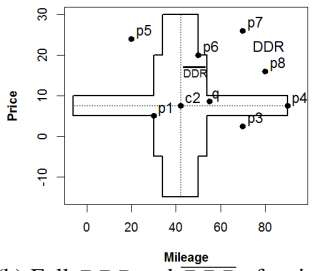
Without loss of generality, we assume that a smaller value is preferred in every dimension in the above definition. Consider the data points given in Fig. 1 (a) as P. Then, the skyline points of P, $SK = \{p_1, p_3, p_5\}$, are shown in Fig. 1(b). Point p_4 is not part of the skyline as it is dominated by p_1 and p_3.

Definition 2: (**Dynamic Skyline**) Given a query point q as customer preference and a dataset of products P, a Dynamic Skyline (DSL) query according to q retrieves all points $p \in P$ that are not dynamically dominated by others, and a point $p_1 \in P$ dynamically dominates $p_2 \in P$ with regard to the query point q (denoted by $p_1 \succ_q p_2$) iff (1) $\forall i \in \{1, ..., d\} : |q^i - p_1^i| \leq |q^i - p_2^i|$ and (2) $\exists j \in \{1, ..., d\} : |q^j - p_1^j| < |q^j - p_2^j|$. Equivalently, a point p_1 is in $DSL(q)$ iff $\forall p_2 \not\equiv p_1, \exists i \in \{1, ..., d\} : |q^i - p_1^i| < |q^i - p_2^i|$.

The dynamic skyline of a point q can be computed by any traditional skyline computing algorithm having all points $p \in P$ transformed to the new data space where point q is considered as the origin and the absolute distances to q are used as the mapping functions [9], [7]. The mapping function, f^i, is defined as $f^i(p^i) = |q^i - p^i|$. For example, consider the data points given in Fig. 1 (a) as P and the query point, $q(8.5K, 55K)$ as a customer preference. Then, the dynamic skyline of q, $DSL(q) = \{p_2, p_6\}$, is shown in Fig. 2(a). Here, point p_5 is transformed to p_5' w.r.t. q, p_1 to p_1', p_2 to p_2' and so on. Point p_1 is not in $DSL(q)$ as p_1 is dynamically dominated by p_2 w.r.t. q. It is verifiable from Fig. 1(b) and Fig. 2(a) that dynamic skyline adheres to around-by semantics (realized by

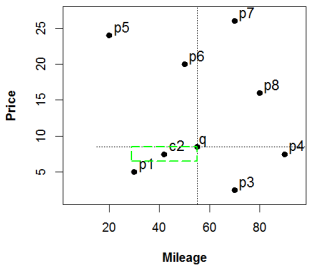


(a) Partial DDR and $\overline{DDR}$ of c_2 in the transformed space

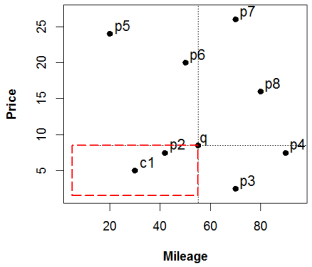


(b) Full DDR and $\overline{DDR}$ of c_2 in the original space

Fig. 3. Dynamic dominance and anti-dominance regions of c_2



(a) Window query of c_2



(b) Window query of c_1

Fig. 4. Window query of c_2 and c_1

coordinate-wise absolute differences to q), whereas traditional skyline is computed with respect to the origin zero.

Definition 3: (**Reverse Skyline**) Given a dataset of products P, a query point q as product and a dataset of customer preferences C, a Reverse Skyline (RSL) query according to q retrieves all points $c \in C$ where q is in the dynamic skyline of c. That is, a point $c_1 \in C$ is a reverse skyline point of q iff $\nexists p \in P$ such that (1) $\forall i \in \{1, ..., d\} : |c_1^i - p^i| \leq |c_1^i - q^i|$, and (2) $\exists j \in \{1, ..., d\} : |c_1^j - p^j| < |c_1^j - q^j|$. Equivalently, c_1 is in $RSL(q)$ iff $\forall p, \exists i \in \{1, ..., d\} : |c_1^i - q^i| < |c_1^i - p^i|$.

The $RSL(q)$ is realized by computing $DSL(c)$ for each point $c \in C$ and then by checking whether $DSL(c)$ contains q or not. For example, consider the data points $pt_1, pt_3 - pt_8$ given in Fig. 1(a) as the dataset of products P and pt_2 in Fig. 1(a) as the customer preference $c_2 \in C$ and the query point $q(8.5K, 55K)$. Then, c_2 is in the reverse skyline of q as q is in the dynamic skyline of c_2 as shown Fig. 2(b).

Definition 4: (**Dynamic Dominance Region**) The *dynamic dominance region* (DDR) of a point c contains the points dominated by at least one dynamic skyline point. We use $\overline{DDR}(c)$ to denote the absolute complement of $DDR(c)$ with respect to the universe. We refer $\overline{DDR}(c)$ as the dynamic anti-dominance region of c in this paper[1].

In general, $DSL(c)$ defines the border between $DDR(c)$ and $\overline{DDR}(c)$. If an arbitrary point q is positioned in $\overline{DDR}$ of a point c, then q will be in $DSL(c)$. This also eliminates all points from $DSL(c)$ that are dynamically dominated by q with regard to c. The DDR and $\overline{DDR}$ of c_2 in the transformed and original space are shown in Fig. 3(a) and Fig. 3(b). It is easily verifiable from Fig. 3 that, if the query point q appears within the $\overline{DDR}$ of $c \in C$, then c appears in the $RSL(q)$.

The computation of the reverse skyline of the query point q requires computation of the dynamic skyline of each point $c \in C$ (and thereby $\overline{DDR}(c)$), which can be bypassed by running a *window_query* centered at c [9]. The extent of the window is defined by the coordinate-wise distances to q. If the *window_query* returns no point, then c is included in $RSL(q)$. For example, given the query point q(8.5K, 55K) as product, the green dashed-rectangle in Fig. 4(a) represents the *window_query* of point c_2. This *window_query* returns an empty result and therefore, c_2 is in $RSL(q)$.

Now, consider the data points $pt_2 - pt_8$ given in Fig. 1(a) as the dataset of products P and data point pt_1 in Fig. 1(a) as the customer preference $c_1 \in C$ and the same query point $q(8.5K, 55K)$ as product. The *window_query* centered at point c_1 (red dashed-rectangle as shown in Fig. 4(b)) returns a non-empty result (i.e., $\{p_2\}$) for which c_1 is not in $RSL(q)$. In this paper, we study the problem of answering why-not question in reverse skyline queries. More specifically, we address why the data point c_1 is not in the reverse skyline of the given query-point, $q(8.5K, 55K)$. Then, we show how the data-point c_1 can be included in the reverse skyline of the query-point, q.

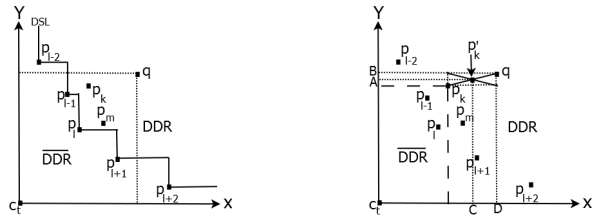
III. Semantics of Why-Not Question for Reverse Skyline Queries

There are three different aspects of answering why-not questions in reverse skyline queries as we discuss in Section I. These are: (1) finding the causes of why point c_t does not appear in the reverse skyline of the query point q; (2) modifying the why-not point c_t into c_t^* so that c_t^* appears in the reverse skyline of q (i.e., $c_t^* \in RSL(q)$); and (3) modifying the query point q into q^* so that c_t appears in the reverse skyline of q^* (i.e., $c_t \in RSL(q^*)$).

The first aspect of answering why-not questions in reverse skyline queries is finding the causes of why-not $c_t \in RSL(q)$. We already know that why point c_t does not appear in $RSL(q)$ because the $window\_query$ centered at c_t returns a non-empty result. In other words, the query point q does not appear in the dynamic skyline of c_t either. If we delete all points returned by $window\_query(c_t, q)$ from the data set P, c_t can appear in $RSL(q)$. For example, consider the data points $pt_2 - pt_8$ given in Fig. 1(a) as P and pt_1 in Fig. 1(a) as the why-not point c_1, we see that c_1 does not appear in $RSL(q)$ because the window query centered at c_1 returns $\{p_2\}$ (see Fig. 4(b)). This can be interpreted as "c_1 finds p_2 more interesting than q". We may find this kind of answer insightful in many real life cases. For example, if a company wants to investigate why their customers are not interested in its product 'X' anymore, then they may collect similar product information available in the market and store it in the database together with the user preferences, then query the product 'X' in the database and finally, find that there are other products in the market customers prefer more than the product 'X'. The company may then encounter the above by redesigning 'X' and/or modifying the different features of 'X' (e.g., packaging, price etc).

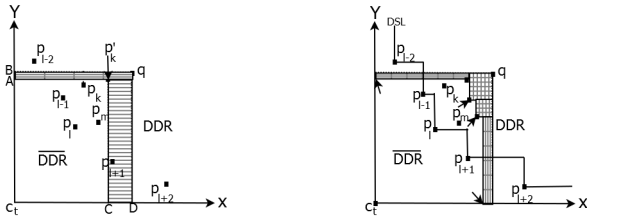
The second aspect of answering why-not questions in reverse skyline queries is modifying the why-not point c_t into c_t^* so that c_t^* appears in the reverse skyline of q. However, the

[1]Note that the term anti-dominance region is defined differently in [9].



(a) Frontiers of $c_t : \{p_k, p_m, p_{l+1}\}$ (b) Calculation of p'_k

Fig. 5. Window query and frontiers of c_t



(a) Movement of c_t in terms of p_k (b) Movement of why-not c_t

Fig. 6. Modification of why-not point c_t

modification of c_t into c_t^* should not incur lots of changes to the original point c_t, i.e., $|c_t - c_t^*|$ should be minimum. This kind of answer also has practical applications as we already discuss in Section I of this paper.

The third aspect of answering why-not questions in reverse skyline queries is modifying the query point q into q^* so that q^* appears in the dynamic skyline of c_t. Importantly, the modification of the query point q into q^* should be done in a way so that we do not lose any of the existing reverse skyline points of q, because, in many cases we do not want to lose existing reverse skyline points. For example, consider a company has a product 'X' in the market, if the company modifies many of its features, then many of the existing customers may not prefer this product, which is undesirable.

In this paper, we study only the second and third aspects in depth as they are computationally challenging. The first aspect is trivial to compute, because we just need to return the $window\_query$ result. In Section IV, we show how to move the why-not point c_t in the space to include it in the reverse skyline of the query point with minimum cost. In Section V, we first show how to move the query point q to include c_t in its reverse skyline list, and then we show how to move the query point q while keeping the existing reverse skyline.

IV. Modifying the why-not point

In this section, we describe how to move the why-not point $c_t \in C$ in the data space to include it in the reverse skyline of the query point q. We know that c_t is not in the reverse skyline list of q (i.e., $c_t \notin RSL(q)$) as q is not in the dynamic skyline list of c_t (i.e., $q \notin DSL(c_t)$). Now, we want to modify c_t into c_t^* in a way so that q appears in the dynamic skyline of c_t^* (i.e., $q \in DSL(c_t^*)$). Therefore, we formally define our why-not point modification problem as follows:

Definition 5: (**Moving the Why-not Point**) Given a dataset of products P, a query point q as product and a why-not point $c_t \in C$, modify c_t into c_t^* so that q appears in the dynamic skyline of c_t^*, i.e., $q \in DSL(c_t^*)$.

We observe that the why-not point c_t is not in the reverse skyline list of q as points $p \in P$ exist in the space between c_t and q as shown in Fig. 5 (a) and the result of the $window\_query$ centered at c_t is not empty. To move a why-not point, we need to find this set of points, say $\Lambda = \{p_{l-1}, p_l, p_k, p_m, p_{l+1}\} \subseteq P$ as shown in Fig 5(a). Some of its member points, for example $\{p_{l-1}, p_l, p_{l+1}\}$, are also included in $DSL(c_t)$, i.e., $\Lambda \cap DSL(c_t) \neq \emptyset$. An important property of Λ is that the deletion of its member points can include q in c_t's dynamic skyline and therefore can also include c_t in $RSL(q)$, as we can see from Fig. 5(a).

Lemma 1: The deletion of points $\in \Lambda$ from P can include c_t in $RSL(q)$.

Proof: The proof of the above lemma is obvious. This is because deletion of points $\in \Lambda$ from P ensures that q will be in $DSL(c_t)$ and the result of the $window\_query$ will be empty. Therefore, c_t will be in $RSL(q)$ according to the definition of reverse skyline and construction of $RSL(q)$. ■

The Λ can be retrieved by running a $window\_query$ centered at c_t, i.e., $\Lambda \leftarrow window\_query(c_t, q)$. Now, to find the movement of c_t, we only need to pick the frontiers, say $F = \{p_k, p_m, p_{l+1}\}$ from Λ as shown in Fig. 5. The property of this frontier point-set $F \subseteq \Lambda$, is given below:

$$\forall e_1 \in F, \nexists e_2 \in \Lambda : e_2 \succ_q e_1$$

The frontier point-set F can be calculated as follows: (1) F is initialized to Λ and (2) if for each $e_1 \in F$, $\exists e_2 \in F$ such that $e_2 \succ_q e_1$, then we remove e_1 from F. For each point e_1 in F, we need to make sure that all points that are dominated by e_1 w.r.t. q in the transformed space of c_t, will not be returned by the $window\_query$ centered at c_t^* again. Consider the frontier point $p_k \in F$. Now, we want to find the regions where c_t can be moved and the $window\_query$ centered at c_t^* will not return points that are dominated by p_k w.r.t. q including p_k itself. To do so, we need to make sure that c_t will be at least $\frac{1}{2} \times \mid q^i - p_k^i \mid$ far away from p_k in every dimension $i \in \{1, 2, ..., d\}$ so that q dominates p_k w.r.t. c_t^*, as shown in Fig. 5(b). Then, c_t can be moved in terms of p_k to the area $ABqDCp'_k$ as shown in Fig. 6(a). We need to compute these areas for all points in F and the intersecting area of them is the valid area where c_t can be moved, as shown in Fig. 6(b). The area where c_t can be moved in terms of p_k relies on the computation of p'_k. The general formula of this computation for each entry $e_l \in F$ is given below:

$$u_l^i = e_l^i + \frac{|e_l^i - q^i|}{2}, \forall i \in \{1, ..., d\} \quad (1)$$

The valid area computed by following the above technique gives us an infinite number of choices for c_t^* as c_t can be moved anywhere in the valid area. We want to reduce this infinite number of choices to only a few. Assume that $M = \{u_l\}$. Then, we sort M based on an arbitrary dimension, say i. Then, we update the entries of M by replacing each successive pair $u_l, u_{l+1} \in M$ by $u_{l,l+1}$ except the first and last one. The construction of $u_{l,l+1}$ is done as follows:

Algorithm 1 Modify Why-Not Point (c_t, q)

1: $\Lambda \leftarrow$ window_query(c_t, q);
2: $F \leftarrow \Lambda$;
3: **for** each $e_1 \in F$ **do**
4: **if** $\exists e_2 \in F$ such that $e_2 \succ_q e_1$ **then**
5: remove e_1 from F;
6: $M \leftarrow \{\}$; // M contains new locations for c_t
7: **for** each $e_l \in F$ **do**
8: $u_l^i = e_l^i + \frac{|e_l^i - q^i|}{2}, \forall i \in \{1, ..., d\}$;
9: Add u_l to M;
10: Sort M based on dimension i;
11: **for** $u_l, u_{l+1} \in M$ **do**
12: $u_{l,l+1}^i = \min\,(u_l^i, u_{l+1}^i), \forall i \in \{1, ..., d\}$;
13: **if** u_l is the first entry in M **then**
14: Replace u_{l+1} in M by $u_{l,l+1}$;
15: **else if** u_{l+1} is the last entry in M **then**
16: Replace u_l in M by $u_{l,l+1}$;
17: **else**
18: Replace u_l and u_{l+1} in M by $u_{l,l+1}$;
19: $u_1^i \leftarrow c_t^i$; // u_1 is the first entry in M
20: $u_{|M|}^{j \neq i} \leftarrow c_t^{j \neq i}$; // $u_{|M|}$ is the last entry in M

$$u_{l,l+1}^i = \min\,(u_l^i, u_{l+1}^i), \forall i \in \{1, ..., d\} \quad (2)$$

We argue that $u_{l,l+1}$ is a better choice for c_t^* than any location in the valid area that is dominated by $u_{l,l+1}$ ($u_{l,l+1}$ dominates u_l and u_{l+1} too) in terms of changes that are needed to be done on c_t (i.e., $|c_t - c_t^*|$) . Finally, we update the first entry, u_1 and the last entry, $u_{|M|}$ in M as follows:

$$u_1^i = c_t^i \text{ and } u_{|M|}^j = c_t^j, \text{ where } j \neq i \quad (3)$$

The points in M computed by following the above technique are marked with arrows in Fig. 6(b). Clearly, these locations are better choices for c_t^* in terms of $|c_t - c_t^*|$. It should also be noted that no two points in M dominate each other. The pseudo-code of all the above computational steps is given in Algorithm 1.

Example. Consider the data points $pt_2 - pt_8$ given in Fig. 1(a) as P, the query point $q(8.5K, 55K)$ and data point pt_1 given in Fig. 1(a) as why-not point c_1. The window query centered at c_1 returns $\Lambda = \{p_2\}$. The two different locations of c_1^* according to Algorithm 1 are $c_1^*(price, mileage) = \{(5K, 48.5K), (8K, 30K), \}$ as shown in Fig. 7. According to our first option (5K, 48.5K), we see that the customer c_1 has to modify her mileage preference from 30K to 48.5 K to be interested in car $q(8.5K, 55K)$ and for the second option, we see that c_1 has to pay at least 3K more to be interested in car q. Based on the above suggestions received from the system, the car dealer may now decide whether they should include c_1 in the plausible customer list of q or not.

Complexity Analysis. The complexity of modifying a why-not point is mainly dominated by the cost of checking pairwise dominance tests performed in steps 3-5 (i.e., $O(|\Lambda|^2)$) and sorting the entries in M performed in step 10 (i.e., $O(|M| \times \log_2|M|)$) in Algorithm 1. Steps 7-9 and steps 11-18 in Algorithm 1 require linear scan of the entries in F and M, respectively. We assume that the execution of $window\_query$ in step 1, computation of Eqns. (1) and (2) in step 8 and step 12, as well as computation of steps 19-20 can be done in constant time. Therefore, the overall complexity becomes $O(|\Lambda|^2) + |M| \times \log_2|M|)$.

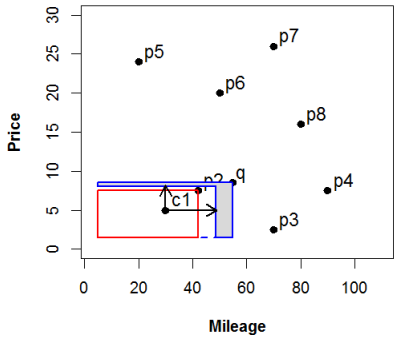


Fig. 7. Movement of why-not point $c_1(5K, 30K)$

V. Modifying the Query Point

In this section, we describe how to modify the query point q into q^* to include the why-not point $c_t \in C$ in the reverse skyline of q^* (i.e., $c_t \in RSL(q^*)$). Recall that to modify the why-not point c_t, we propose to move c_t towards q, but now to modify the query point q, naturally we aim to move q towards c_t. However, their computations are not symmetrical. For query point modification, we want to move q onto the dynamic skyline of c_t so that c_t becomes a reverse skyline point of q. But for why-not point modification, the solution is not moving c_t onto the dynamic skyline of q. Rather, we have moved c_t towards q in a way so that q can dynamically dominate all points returned by $window\_query(c_t, q)$. Next, we formally define our query-point modification problem in this section as follows:

Definition 6: (**Moving the Query Point**) Given a dataset of products P, a query point q as product and a why-not point $c_t \in C$, modify q into q^* so that c_t appears in the reverse skyline of q^*, i.e., $c_t \in RSL(q^*)$.

From Section II, we know that the query point q can be moved arbitrarily in the $\overline{DDR}(c_t)$ to include c_t in $RSL(q^*)$. However, this arbitrary movement may incur lots of changes to the query point q and it may also happen that we lose many of the existing reverse skyline points (existing customers), which is not always desirable. We find that it is possible to locate a safe region (defined in Definition 7), where the query point q can be moved without losing any existing reverse skyline point. Therefore, in this section, we show how to move the query point q with and without considering this safe region to include c_t in the reverse skyline list of q^*.

Definition 7: (**Safe Region**) A region in the data space is said to be safe, termed as *safe region* ($SR(q)$), for the query point q where q can be moved without losing any of the original reverse skyline points. That is, if q is modified to q^* by moving the query point q anywhere within $SR(q)$, then the following holds:

$$RSL(q) \cap RSL(q^*) = RSL(q), \text{ only if } q^* \in SR(q) \quad (4)$$

A. Moving the Query Point without Considering Safe Region

We already know from Section II that we can move the query point q within $\overline{DDR}$ of the why not point c_t to include c_t in $RSL(q)$. However, moving the query point arbitrarily within $\overline{DDR}(c_t)$ may incur lots of changes to q. Therefore, we need to consider only those locations within $\overline{DDR}(c_t)$ that can minimize the edit distance between the original q and the refined query point q^*. For example, it can be easily seen from Fig. 8 that q should be moved to the locations 'A-B-C-D' at least to include c_t in $RSL(q)$. But only locations 'A', 'B', 'C', and 'D' can possibly minimize $|q - q^*|$. Therefore, we want to compute only these locations within the $\overline{DDR}$ of c_t.

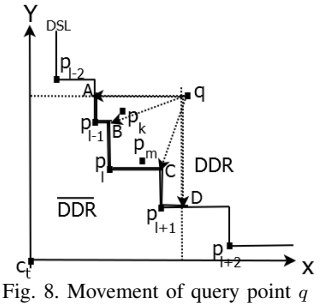


Fig. 8. Movement of query point q

Let $\Lambda = \{p_{l-1}, p_l, p_k, p_m, p_{l+1}\} \subseteq P$ be the data points whose deletion can include q in c_t's dynamic skyline. The Λ can be retrieved by running a window query centered at c_t, i.e., $\Lambda \leftarrow window\_query(c_t, q)$. Assume that $F \leftarrow \Lambda \cap DSL(c_t)$. That is, F contains only points that appear in both Λ and $DSL(c_t)$. The point-set F can be computed as follows: (1) F is initialized to Λ and (2) if for each $e_1 \in F$, $\exists e_2 \in F$ such that $e_1 \succ_{c_t} e_2$, then we remove e_2 from F. The above steps allow computing F without computing $DSL(c_t)$ and therefore, save a lot of computational time.

Then, we assign F to M. Now, we sort M based on an arbitrary dimension, say i. Then, we update the entries of M by replacing each successive pair $u_l, u_{l+1} \in M$ by $u_{l,l+1}$ except the first and last one. The construction of $u_{l,l+1}$ is done as follows:

$$u^i_{l,l+1} = \max\ (u^i_l, u^i_{l+1}), \forall i \in \{1, ..., d\} \tag{5}$$

Then, we update the first entry, $u_1$and the last entry, $u_{|M|}$ as

$$z_1 = q, z^i_1 = u^i_1, u_1 = z_1, z_{|M|} = q, z^j_{|M|} = u^{j \neq i}_{|M|}, u_{|M|} = z_{|M|} \tag{6}$$

Finally, M contains the locations of q^*.

Example. Consider the data points $pt_2 - pt_8$ given in Fig. 1(a) as P, the query point $q(8.5K, 55K)$ and data point p_1 given in Fig. 1(a) as why-not point c_1. The window query centered at c_1 returns $\Lambda = \{p_2\}$. The two different locations of q^* according to Algorithm 2 are $q^*(price, mileage) = \{(8.5K, 42K), (7.5K, 55K)\}$ as shown in Fig. 9. According to option $q^*(7.5K, 55K)$, the car dealer has to decrease the price of q at least 1K to make q interesting to customer c_1.

Complexity Analysis. The complexity of modifying a query-point without considering the safe region is the same as modifying a why-not point, and is mainly dominated by the cost of checking pairwise dominance tests performed in steps

Algorithm 2 Modify Query Point (c_t, q)

1: $\Lambda \leftarrow$ window_query(c_t, q);
2: $F \leftarrow \Lambda$;
3: **for** each $e_1 \in F$ **do**
4: **if** $\exists e_2 \in F$ such that $e_1 \succ_{c_t} e_2$ **then**
5: remove e_2 from F;
6: $M \leftarrow F$;
7: Sort M based on dimension i;
8: **for** $u_l, u_{l+1} \in M$ **do**
9: $u^i_{l,l+1} = \max\ (u^i_l, u^i_{l+1}), \forall i \in \{1, ..., d\}$;
10: **if** u_l is the first entry in M **then**
11: Replace u_{l+1} in M by $u_{l,l+1}$;
12: **else if** u_{l+1} is the last entry in M **then**
13: Replace u_l in M by $u_{l,l+1}$;
14: **else**
15: Replace u_l and u_{l+1} in M by $u_{l,l+1}$;
16: $z_1 \leftarrow q$; $z_{|M|} \leftarrow q$;
17: $z^i_1 \leftarrow u^i_1$; // u_1 is the first entry in M
18: Replace $u_1 \in M$ by z_1;
19: $z^j_{|M|} \leftarrow u^{j \neq i}_{|M|}$; // $u_{|M|}$ is the last entry in M
20: Replace $u_{|M|} \in M$ by $z_{|M|}$;

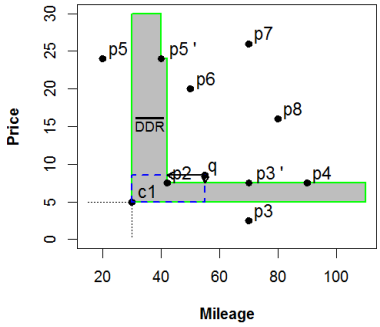


Fig. 9. Movement of query point $q(8.5K, 55K)$ for why-not point $c_1(5K, 30K)$

3-5 (i.e., $O(|\Lambda|^2)$) and sorting the entries in M performed in step 7 (i.e., $O(|M| \times \log_2|M|)$)) in Algorithm 2. Steps 8-15 in Algorithm 2 require linear scan of the entries in M. We assume that the execution of the $window\_query$ in step 1, the computation of Eqn. (5) in step 9 as well as compuation of steps 16-20 can be done in constant time. Therefore, the overall complexity becomes $O(|\Lambda|^2) + |M| \times \log_2|M|)$.

B. Moving the Query Point Considering Safe Region

Computing the safe region of the query point q relies on the fact that existing reverse skyline points include q in their dynamic skylines. That is, $\overline{DDR}$ of each point $c_l \in RSL(q)$ contains q. For example, $\overline{DDR}$ of $c_2 \in C$ includes q as shown in Fig. 3. If the query point q is moved to an arbitrary position within $\overline{DDR}(c_l)$, then q^* will again be in the dynamic skyline of c_l. Therefore, the following important lemma is our key to the construction of the safe region of q.

Lemma 2: The safe region of the query point q is the intersection of $\overline{DDR}$s of all points $c_l \in RSL(q)$. That is,

$$SR(q) = \bigcap_{c_l \in RSL(q)} \overline{DDR}(c_l) \tag{7}$$

Proof: Let $RSL(q)$ consists of k points as follows: $\{c_1, c_2, ..., c_k\}$. According to the definition of reverse skyline and $\overline{DDR}$, each $c_l \in RSL(q)$ contains q in its dynamic

Algorithm 3 Exact Safe Region ($RSL(q)$)

1: $SR(q) \leftarrow null$
2: **for** each $c_l \in RSL(q)$ **do**
3: Compute $\overline{DDR}(c_l)$;
4: **if** $SR(q) = null$ **then**
5: $SR(q) \leftarrow \overline{DDR}(c_l)$;
6: **else**
7: $SR(q) \leftarrow SR(q) \cap \overline{DDR}(c_l)$;

skyline and q must reside within $\overline{DDR}(c_l)$. If we move q arbitrarily within $\overline{DDR}(c_l)$, it will still be in the dynamic skyline of c_l. Therefore, if we take the intersection of $\overline{DDR}$s of all k points of $RSL(q)$ and move q in their intersecting region, q will still be in their dynamic skylines. Therefore, q can move arbitrarily anywhere in the intersecting region of $\overline{DDR}(c_l)$ and can retain the original reverse skyline. ■

Lemma 3: The safe region of the query point q computed by following Eqn. (7) is correct and exact.

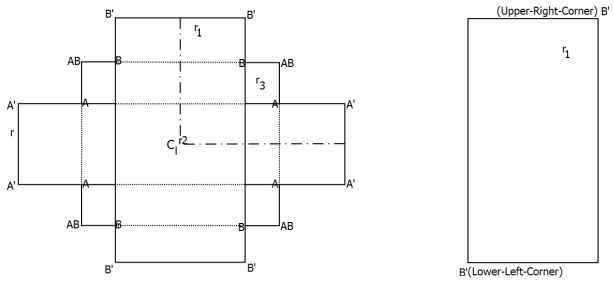
Proof: Suppose that the safe region, SR, of q computed by following the Eqn. (7) is not correct. Assume that the correct safe region of q is SR^*. According to the definition of safe region, this SR^* must be included in $\overline{DDR}$ of all points $c_l \in RSL(q)$. Then, the only way we can construct this SR^* is by taking the intersection of all $\overline{DDR}(c_l)$. Hence, $SR^* = SR$. Therefore, the SR of the query point q computed by following Eqn. (7) is correct and exact. ■

Computing the Safe Region. The steps for computing the exact safe region of the query point q are given in Algorithm 3. To find the intersection of all $\overline{DDR}(c_l)$, we represent each $\overline{DDR}(c_l)$ by a collection of rectangles (rectangles for 2D data points, cubes for 3D data points and so on).

Consider a particular reverse skyline point c_l ($c_l \in C$). We first compute the dynamic skyline points of c_l, $DSL(c_l) \subseteq P$ and assign it to M. Then, we sort M based on any dimension i. Then, we update the entries of M by replacing each successive pair $u_l, u_{l+1} \in M$ by $u_{l,l+1}$ ($u_{l,l+1}$ is computed by following Eqn. (5)) except the first and last one. The first entry is updated by shifting it's i^{th} dimensional value to the maximum value appearing in the i^{th} dimension in the dataset of products P. Similarly, the last entry is updated by shifting it's j^{th} ($j \neq i$) dimensional value to the maximum value appearing in the j^{th} dimension in the dataset of products P.

Considering each entry $u_l \in M$, then we form rectangles, whose extension are the coordinate-wise distances from the point c_l. It should be noted that a rectangle is represented by its lower-left and upper-right corner points only, as shown in Fig. 10 (b). Though the rectangles here have common space between them, it reduces the number of intersections needed to compute the safe region of q. The $\overline{DDR}$ of c_l is then represented by $|DSL(c_l)|+1$ rectangles, as shown Fig. 10(a). The step 3 in Algorithm 3 computes the above rectangle-based representation of $\overline{DDR}$ of each reverse skyline point c_l of q.

Finally, the safe region of the query point is computed by taking intersections of constituent rectangles of $\overline{DDR}(c_l)$ of all $c_l \in RSL(q)$ as shown in step 7 of Algorithm 3. For



(a) (b)

Fig. 10. Representation of $\overline{DDR}$ of a point c_l: $\overline{DDR}(c_l) = \{r_1, r_2, r_3\}$, where $DSL(c_l) = \{A, B\}$

example, assume that we have two reverse skyline points of q, i.e., $RSL(q) = \{c_1, c_2\} \subseteq C$ and the $\overline{DDR}$s of c_1 and c_2 are rectangles $\{r_{11}, r_{12}\}$ and $\{r_{21}, r_{22}\}$, respectively. Then, $SR(q)$ is computed as $r_{11} \cdot r_{21} + r_{11} \cdot r_{22} + r_{12} \cdot r_{21} + r_{12} \cdot r_{22}$, where $+$ and $\cdot$ represents the union and the intersection operation, respectively.

The safe region constructed in this section (i.e., following Eqn. (7)) can be, however, truncated/expanded to a smaller/greater one by limiting/relaxing certain product feature to achieve certain flexibility for practical applications. Because, the companies know about the certain feature range of the query product, they can modify. Though truncation/expansion of the safe region gives more flexibility, the companies may lose a few existing customers as a side effect.

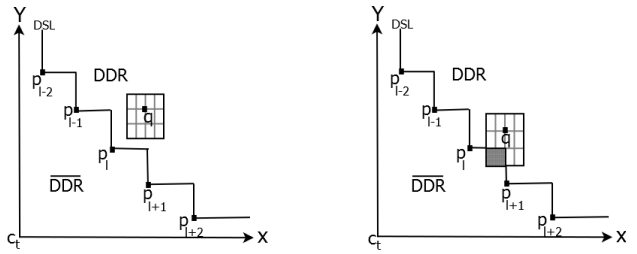
Example. Consider the data points given in Fig 1(a) as set of products P as well as set of customer preferences C. Then, the safe region of the query point $q(8.5K, 55K)$ for our example data given in Fig. 1(a) consists of two rectangles: (a) (price, mileage): $\{(7.5K, 50K), (10K, 58K)\}$ and (b) $\{(7.5K, 50K), (12.5K, 54K)\}$. That is, if we move q in these regions, none of the existing reverse skyline points $\{c_2, c_3, c_4, c_6, c_8\}$ will be lost.

Complexity Analysis. The computation of the safe region of the query point q requires computing the $\overline{DDR}$ of all $c_l \in RSL(q)$ and their representations. Then, finally computing the pairwise intersections of $\overline{DDR}$ of all $c_l \in RSL(q)$. Therefore, the overall complexity of computing the safe region of the query point q is $O(C \times (|DSL(c_l)| + 1)^{|RSL(q)|})$.

In the following, we adopt the query point as well as the data point modification approach to answer why-not questions in reverse skyline queries. That is, we want to modify the original query point q into q^* to retain the original reverse skyline points. Then, we want to modify the why-not point if necessary. We formally redefine our problem as follows:

Definition 8: (**Moving Both Points**) Given a data set of products P, a query point q and a why-not point $c_t \in C$, derive a new query point q^* and a new c_t^* where q^* appears in the dynamic skyline of c_t^*, i.e., $q^* \in DSL(c_t^*)$.

We already know that we can refine the query point q into q^* by moving q within $SR(q)$ while keeping its original reverse skyline points. However, the $\overline{DDR}$ of the why-not point c_t may or may not include q^* if we move the query-point q within $SR(q)$, as shown Fig. 11. If $\overline{DDR}(c_t)$ and $SR(q)$ overlap with each other, then we need to modify only the query point



(a)$SR(q) \cap \overline{DDR}(c_t) = \emptyset$ (b)$SR(q) \cap \overline{DDR}(c_t)! = \emptyset$

Fig. 11. Safe region of q and $\overline{DDR}$ of why-not point c_t

q, otherwise we need to modify the why-not point c_t too. Therefore, there are two different cases in why-not reverse skyline queries as shown in Table I. In the first case, we need to modify only the query point q. In the second case, we need to modify both the query point q and the why-not point c_t.

TABLE I
TWO CASES IN WHY-NOT REVERSE SKYLINE QUERIES

Cases	Modify c_t	Modify q	Comment
C1: $\overline{DDR}(c_t) \cap SR(q) \neq \emptyset$	no	yes	$c_t \in RSL(q^*)$
C2: $\overline{DDR}(c_t) \cap SR(q) = \emptyset$	yes	yes	$c_t^* \in RSL(q^*)$

However, it is possible to modify both the query point q and the why-not point c_t arbitrarily in the data space to include c_t^* in the reverse skyline of q^*. Then, we need to report the optimal (q^*, c_t^*) subject to the following:

$$\underset{(q^*,c_t^*)\in S}{\text{minimize}}\ cost(q^*, c_t^*) \tag{8}$$

The $cost(q^*,\ c_t^*)$ of an arbitrary answer (q^*, c_t^*) is defined as follows:

$$\begin{aligned} cost(q^*, c_t^*) &= cost(q, q^*) + cost(c_t, c_t^*) \\ &= \alpha\cdot \mid q - q^* \mid + \beta\cdot \mid c_t - c_t^* \mid \\ &= \sum_{i=1}^{d} \alpha_i \times \mid q^i - q^{*i} \mid + \sum_{i=1}^{d} \beta_i \times \mid c_t^i - c_t^{*i} \mid \end{aligned} \tag{9}$$

where $\alpha_i, \beta_i \in [0, 1]$. The α_i and β_i can be set based on how much we are willing to modify q and why-not c_t along the i^{th} dimension, respectively. But, solving the above equation is very difficult as there are an infinite number of pairs (q^*, c_t^*) in the data space that can minimize the cost function, $cost(q^*, c_t^*)$, optimally. Also, we do not want to lose existing reverse skyline points. Therefore, we allow the query point q to move only within the safe region of q, and assume that the cost of moving the query point q within the safe region of the query point q is zero. That is,

$$cost(q, q^*) = 0,\ \text{if } q^* \in SR(q) \tag{10}$$

Therefore, Eqn. (9) becomes as follows:

$$\underset{q^*\in SR(q)}{cost(q^*, c_t^*)} = \sum_{i=1}^{d} \beta_i \times \mid c_t^i - c_t^{*i} \mid \tag{11}$$

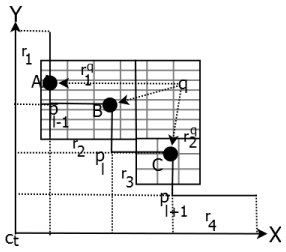


Fig. 12. Moving the query point q within $SR(q)$

From Eqn. (11), we conclude that the minimization of $cost(q^*, c_t^*)$ becomes the minimization of $cost(c_t, c_t^*)$ if q^* stays within the safe region of q (i.e., $q^* \in SR(q)$).

Now, assume that $\overline{DDR}(c_t)$ of why-not point c_t and $SR(q)$ of the query point q overlap with each other as shown in Fig. 12 (case C1 in Table I). Therefore, we need to modify only the query point q. The new location of q in the space must be anywhere in $\overline{DDR}(c_t) \cap SR(q)$. But, we consider only those locations that can minimize the edit distance between the original query point, q and the refined query point, q^*.

To test whether $\overline{DDR}(c_t)$ and $SR(q)$ overlap with each other or not, we first compute the rectangle(s) based representation of $\overline{DDR}(c_t)$ and perform intersection with $SR(q)$ ($SR(q)$ is also a collection of of rectangles). For example, consider the $\overline{DDR}(c_t)$ of a why-not point c_t as shown in Fig. 12. This $\overline{DDR}(c_t)$ of c_t here consists of four rectangles, $\{r_1, r_2, r_3, r_4\}$ and the $SR(q)$ of q here consists of two rectangles $\{r_1^q, r_2^q\}$. The result of intersection between $\overline{DDR}(c_t)$ and $SR(q)$ consists of three new rectangles $\{A, B, C\}$, which indicates that $\overline{DDR}(c_t)$ and $SR(q)$ overlap with each other. Finally, we compute the new locations of the query point q by locating the nearest point of these rectangles from q. These new locations are shown as big dots in Fig. 12 and marked with arrows originated from q. The pseudo-code of the above is shown in steps 1-6 in Algorithm 4.

Now, assume that the dynamic anti-dominance region of the why-not point c_t, $\overline{DDR}(c_t)$ and the safe region of the query point q, $SR(q)$ do not overlap with each other, i.e., $\overline{DDR}(c_t) \cap SR(q) = \emptyset$. Therefore, we need to modify both q and c_t (case C2 in Table I). But the query point can not move outside its safe region, $SR(q)$. This is because we want to keep the existing reverse skyline of q and we can move the query point q only within $SR(q)$ with zero cost. Therefore, we need to maximize the movement of q towards the why-not point c_t to minimize the movement of c_t. The movement of the query point q can be maximized by moving it up to the edges of $SR(q)$ towards c_t as shown in Fig. 13. To find the best new locations of q within its safe region, we first get the corner points of the constituent rectangle(s) of $SR(q)$ and assign them to E. Then, we transform these points into a new space considering c_t as their origin. Then, we perform the dominance test on these points. Then, we take only the non-dominated points and assign these points to Q. Then, considering each entry in Q as the refined query point q^*, we call Algorithm 1 to find the movement of c_t and collect all movements of c_t into M_c. Finally, the new locations in

Algorithm 4 Modify Query and Why-not Point (SR, c_t, q)

1: **if** $SR(q) \cap \overline{DDR}(c_t) \neq \emptyset$ **then**
2: $M_q \leftarrow \{\}$;
3: $OR(c_t, q) \leftarrow SR(q) \cap \overline{DDR}(c_t)$;
4: **for** each $rec_1 \in OR(c_t, q)$ **do**
5: $e_1 \leftarrow$ nearest_point(rec_1, q);
6: Add e_1 to M_q;
7: **else**
8: $E \leftarrow \{\}$;
9: **for** each $rec_1 \in SR(q)$ **do**
10: $E \leftarrow E \cup$ corner_points(rec_1);
11: $Q \leftarrow TS(E, c_t)$; // transformed space, c_t is origin
12: **for** $\exists e_1, e_2 \in Q$ such that $e_1 \succ_{c_t} e_2$ **do**
13: Remove e_2 from Q;
14: $M_c \leftarrow \{\}$;
15: **for** each $e_1 \in Q$ **do**
16: $T \leftarrow$ move_why_not_point(c_t, e_1); // Algorithm 1
17: $M_c \leftarrow M_c \cup T$;
//compute score s_1 of the entries $e_1 \in M_c$
18: **for** each $e_1 \in M_c$ **do**
19: $s_1 = \sum_{i=1}^{d} \beta_i \times \mid c_t^i - e_1^i \mid$; // Eqn. (11)
20: $M_c \leftarrow \forall e_1 \in M_c$ which has the lowest score s_1;

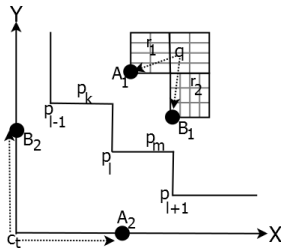


Fig. 13. Move both query point, q and why-not point, c_t

M_c are ranked according to the predefined weight vector β and the top ranked location(s) are returned. The pseudo-code of the above is given in Steps 7-20 in Algorithm 4.

Example. Consider the data points given in Fig. 1(a) as set of products P as well as set of customers C, query point $q(8.5K, 55K)$ as product and why-not point $c_7 \in C$, then the $\overline{DDR}$ of c_7 consists of four rectangles as follows:
$r_1^{c_7} = \{(2.5K, 60K), (49.5K, 80K)\}$,
$r_2^{c_7} = \{(16K, 50K), (36K, 90K)\}$,
$r_3^{c_7} = \{(20K, 20K), (32K, 120K)\}$, and
$r_4^{c_7} = \{(24K, 50K), (28K, 90K)\}$.
If we intersect the $\overline{DDR}$ of c_7 with $SR(q)$, we get: $\{(7.5K, 60K), (10K, 70K)\}$, which is the overlapped region between $\overline{DDR}(c_7)$ and $SR(q)$. If we move q in this overlapped region, customer c_7 will be included in $RSL(q)$. Therefore, the new location of q according to Algorithm 4 is $q^*(8.5K, 60K)$. Now, consider another why-not point $c_1 \in C$, the $\overline{DDR}$ of $c_1 \in C$ does not overlap with $SR(q)$. According to Algorithm 4, the best candidate of q^* within $SR(q)$ is $q^*(7.5K, 50K)$. Therefore, the new location of $c_1 \in C$ with respect to this q^* is $c_1^*(50K, 46)$.

Complexity Analysis. The complexity of modifying both query and why-not point is dominated by the computational cost of constructing the safe region of q ($O(C \times (|DSL(c_l)| + 1)^{|RSL(q)|})$) for step 1 and step 6 in Algorithm 4) and then checking whether it overlaps with the $\overline{DDR}$ of why-not point c_t or not ($O(C \times |SR(q)| \times (|DSL(c_t)| + 1)$ for step 3). We assume that computing the nearest point of the intersecting rectangles from q can be done in constant time in step 5 of Algorithm 4 and therefore, the complexity of steps 4-6 is $O(C \times |OR(c_t, q)|)$. Steps 9-10 in Algorithm 4 can be done in $O(C \times |SR(q)|)$, assuming that we can retrieve the corner points in constant time. Step 11 requires a linear scan of the points in Q. The dominance tests performed in steps 12-13 require $O(|Q|^2)$ time. Step 16 calls Algorithm 1 for each entry in Q. Finally, steps 18-20 require a linear scan of the entries in M_c and computing Eqn. 11 requires constant time.

TABLE II
DATA STATISTICS

Parameter	Ranges
Data Types	Real (Yahoo! Autos) and Synthetic (UN, CO, AC)
Data Size	50K, 100K, 200K
Size of RSL per Query	1-15

VI. EXPERIMENTS

We evaluate our proposed techniques for answering why-not questions in reverse skyline queries using real data, namely CarDB[2], of varying size (50K, 100K and 200K tuples). This is a six-dimensional dataset with attributes referring to Make, Model, Year, Price, Mileage and Location. The two numerical attributes Price and Mileage are considered in our experiments. Answering why-not questions in this dataset makes excellent sense in practice. The car sellers are often interested in finding potential customers to maximize the chance of a car being sold. An extended customer list (for targeted marketing) can be found by answering why-not questions in reverse skyline queries in such a dataset. We also present experimental results based on three types of synthetic data[3]: uniform (UN), correlated (CO) and anti-correlated (AC). A summary about the above datasets is given in Table II.

All experiments presented in this paper are performed on a Windows PC with 2.99 GHz CPU and 3.49 GB main memory. For each experiment we run queries with 1-15 reverse skyline points. The queries follow the distribution of the particular tested dataset. Each dataset is indexed by an R-tree[11], where the page size is set to 1536 bytes. We also implement the BBRS algorithm developed in [9] to compute the reverse skylines for the tested queries. All of our methods proposed in this paper are implemented in Java using the XXL library[12].

A. Effectiveness

In this section, we demonstrate the effectiveness of our proposed approach of answering why-not question in reverse skyline queries. More specifically, we compare the three different techniques proposed in this paper: (1) modifying only the why-not point (MWP), (2) modifying only the query point (MQP), and (3) modifying both the query point and the why-not point (MWQ). To do so, we first randomly select a data-point as an why-not-point for each reverse skyline query where each reverse skyline query has a different number of reverse

[2]Downloaded from autos.yahoo.com. The distribution of data is sparse.
[3]The distribution of data is dense and $|RSL(q)|$ is small in this dataset.

TABLE III
QUALITY OF RESULTS IN CARDB DATASETS

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 1$	0.573633056	0.077233219	0.000000000
$q_2, \|RSL(q_2)\| = 2$	0.208203022	0.779414615	0.000000000
$q_3, \|RSL(q_3)\| = 4$	0.089639196	0.506414971	0.075080839
$q_4, \|RSL(q_4)\| = 5$	0.062404909	0.198753477	0.047772136
$q_5, \|RSL(q_5)\| = 6$	0.130741812	0.447395608	0.129937287
$q_6, \|RSL(q_6)\| = 7$	0.108113847	1.007441591	0.108113847
$q_7, \|RSL(q_7)\| = 8$	0.123956143	1.224492322	0.123724504

(a) CarDB-50K dataset

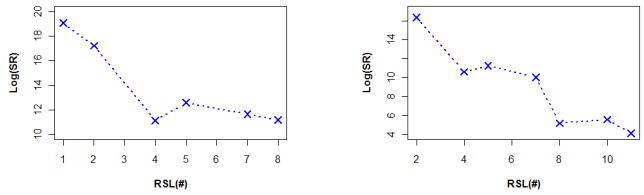
Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 1$	0.578931072	0.832696525	0.000000000
$q_2, \|RSL(q_2)\| = 2$	0.246704746	0.235568427	0.000000000
$q_3, \|RSL(q_3)\| = 4$	0.050311016	0.177055208	0.036971632
$q_4, \|RSL(q_4)\| = 5$	0.048688266	0.481636457	0.048361713
$q_5, \|RSL(q_5)\| = 7$	0.117195707	0.764211836	0.105643573
$q_6, \|RSL(q_6)\| = 8$	0.061553161	0.361610434	0.048393802
$q_7, \|RSL(q_7)\| = 10$	0.070768028	0.907164402	0.070754936
$q_8, \|RSL(q_8)\| = 11$	0.097542681	0.88364149	0.097542681

(b) CarDB-100K dataset

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 2$	0.274162197	0.966525902	0.270083225
$q_2, \|RSL(q_2)\| = 3$	0.371958746	1.280325545	0.339225107
$q_3, \|RSL(q_3)\| = 4$	0.230578336	1.10082634	0.211690534
$q_4, \|RSL(q_4)\| = 5$	0.038138603	0.243090389	0.02951842
$q_5, \|RSL(q_5)\| = 6$	0.135468879	1.413703538	0.135450054
$q_6, \|RSL(q_6)\| = 7$	0.089828649	0.642190348	0.089768971
$q_7, \|RSL(q_7)\| = 8$	0.028030879	0.210567316	0.028030879
$q_8, \|RSL(q_8)\| = 9$	0.079989249	0.542973513	0.079284723
$q_9, \|RSL(q_9)\| = 10$	0.044557559	0.400313076	0.033984709
$q_{10}, \|RSL(q_{10})\| = 12$	0.065886539	0.696464647	0.065886539
$q_{11}, \|RSL(q_{11})\| = 13$	0.009078408	0.26180871	0.009078408
$q_{12}, \|RSL(q_{12})\| = 14$	0.07612146	0.959489884	0.07612146
$q_{13}, \|RSL(q_{13})\| = 15$	0.007091629	0.254541311	0.007085752

(c) CarDB-200K dataset

skyline points (1-15). Then, we compute the cost of a solution by first normalizing the point using *min-max normalization* and then calculating its score (Eqn. 11) by assigning equal weight to each dimension (also $\sum \beta_i = 1$). For MQP, we compute the cost of the modified query point q^* as follows:



(a) CarDB-50K (b) CarDB-100K

Fig. 14. CarDB datasets: RSL size vs. Safe Region area

$$cost(q, q^*) = \alpha \cdot |q' - q^*| + \sum_{c_l \in RSL(q), c_l \notin RSL(q^*)} \beta \cdot |c_l - c_l^*|$$

where, q' is the closest point within $SR(q)$ to q^* w.r.t. $\alpha \cdot |q' - q^*|$. We also set $\alpha = \beta$. Finally, we compare the quality of the proposed methods (MWP, MQP and MWQ) by comparing the cost[4] of the best output received by each method.

1) MWQ vs. MWP: From Table III and Table IV, we see that the outputs returned by MWQ are less costly (at least equal) than MWP for both CarDB and synthetic datasets. If the $\overline{DDR}$ of why-not point overlaps with the safe region of the query point, we can also receive zero-cost output from MWQ as we see in the first two rows of Table III (a) and Table III (b). However, MWQ does not perform very well when the number of reverse skyline points of the query point increases (as we see in Table III and Table IV). This is because the safe region shrinks if the number of reverse skyline points

[4]A small difference is significant.

TABLE IV
QUALITY OF RESULTS IN SYNTHETIC DATASETS

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 1$	0.484858586	0.452025253	0.484858586
$q_2, \|RSL(q_2)\| = 2$	0.119975009	0.118691919	0.000000000
$q_3, \|RSL(q_3)\| = 4$	0.060102683	0.055560606	0.060102683

(a) UN-100K dataset

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 2$	0.054717866	0.121272727	0.045459596
$q_2, \|RSL(q_2)\| = 3$	0.163473577	0.621267677	0.15909596
$q_3, \|RSL(q_3)\| = 4$	0.114681736	0.550580808	0.111116162

(b) CO-100K dataset

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 1$	0.555555556	1.085893939	0.545459596
$q_2, \|RSL(q_2)\| = 2$	0.416850813	1.212181818	0.409095960
$q_3, \|RSL(q_3)\| = 3$	0.022625598	0.065699495	0.015156566
$q_4, \|RSL(q_4)\| = 4$	0.162777034	0.777888889	0.161621212

(c) AC-100K dataset

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 1$	0.045454545	0.042934343	0.042934343
$q_2, \|RSL(q_2)\| = 2$	0.310519291	0.772787879	0.181823232
$q_3, \|RSL(q_3)\| = 4$	0.099253142	0.449606061	0.095964646

(d) UN-200K dataset

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 2$	0.097412478	0.257616162	0.090924242
$q_2, \|RSL(q_2)\| = 4$	0.034838692	0.121323232	0.030308081

(e) CO-200K dataset

Queries	MWP	MQP	MWQ
$q_1, \|RSL(q_1)\| = 1$	0.070707071	0.237835394	0.068186869
$q_2, \|RSL(q_2)\| = 2$	0.109853207	0.287939394	0.101015152
$q_3, \|RSL(q_3)\| = 4$	0.042653853	0.171747475	0.040429293

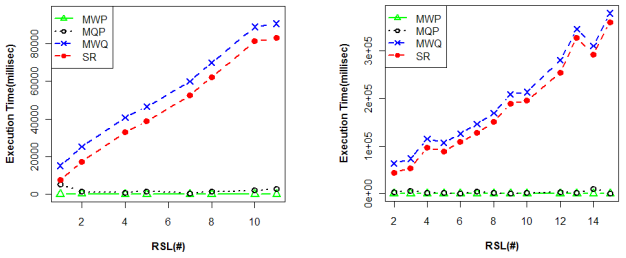
(f) AC-200K dataset

of the query point increases, as we see in Fig. 14. This also reflects real life scenarios where the companies might not like to decrease the price of a product (for example) if they already have sufficient number of customers interested in the product. Finally, the effectiveness of MWQ is the same as MWP when the safe region consists of the query point itself only, as we see from last two rows in Table III(a) and and Table III(b), and last four rows in Table III(c). Otherwise, MWQ always outperforms MWP.

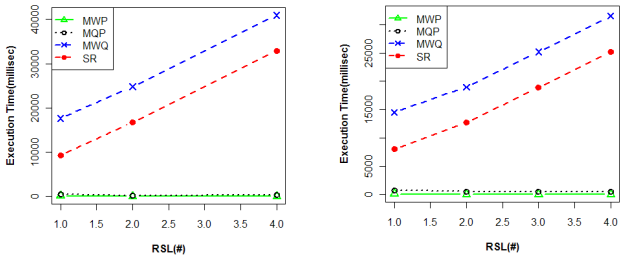
2) MWQ vs. MQP: MWQ provides cheaper solution than MQP in most cases as we see from Table III and Table IV. However, it is possible to receive cheaper solution from MQP than MWQ as we see from the first row of Table IV(a). This is because, the query point can move without restriction (can move outside of the safe region) in MQP, whereas the query point can only move within the safe region in MWQ. However, MQP does not guarantee of keeping the existing reverse skyline of the query point, which is a major disadvantage of MQP. On the other-hand, MWQ guarantees of keeping the existing reverse skyline of the query point, i.e., $RSL(q) \cap RSL(q^*) = RSL(q)$ (as MWQ does not move q outside of its $SR(q)$), and is desirable in practical applications.

B. Performance

The execution times of both MWP and MQP methods are quite smaller compared to MWQ, as we see from Fig. 15. The execution time of MWQ also increases as the number of reverse skyline points of the query point increases. This is because, we need to compute the DSL for each reverse skyline point to compute the safe region used for MWQ and computing the DSL of a point itself is a very costly operation [7]. This can also easily be observed from Fig. 15 that most of the execution time of MWQ is spent for computing the



(a) CarDB-100K (b) CarDB-200K



(c) UN-100K (d) AC-100K

Fig. 15. Execution time of MWP, MQP, Safe Region (SR) and MWQ in CarDB and synthetic datasets

safe region of the query point, $SR(q)$. An important aspect of computing the safe region of the query point is that we do not need to recompute it to answer another why-not question for the same query point. This property allows a user to inspect multiple why-not questions for a query once the safe region of the query point is computed by MWQ. Beside of this, we intend to find an approximated safe region that can be computed quickly by taking advantage of the precomputed approximated DSL of data-points [9].

1) Approximating the Safe Region: To find an approximated safe region of the query point, we pre-compute an approximated DSL for each data-point in C and store it (off-line). To approximate the DSL of the data-point, we first sort the points $\in DSL$ to a specific dimension and then, every $(|DSL|/k)^{th}$ point is drawn from the sorted sequence to store [9], where k is a constant. Now, when a query is submitted, we calculate its safe region from these precomputed approximated DSLs. However, we do not replace the successive pair u_l and u_{l+1} by $u_{l,l+1}$ here as we do for computing the exact safe region of the query point (Algorithm 3). But, to maximize the chance of overlap between the $\overline{DDR}$ of why-not point and the safe region of the query point, we always store the first and last point from the sorted sequence to approximate the DSL for each data-point as shown in Fig. 16.

2) Effect of Approximation: The execution time of MWQ for approximated DSLs (which are precomputed) dramatically reduces from mins to secs as we see in Fig. 17[5]. However, we may not receive results as good as the one returned by the original MWQ when the number of reverse skyline points increases as we see in Table V and Table VI. But, the result is no worse than the one received from MWP. The value of k is chosen empirically in our experiments for both datasets.

[5]The execution time includes the time spent for computing the best result.

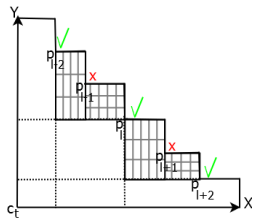
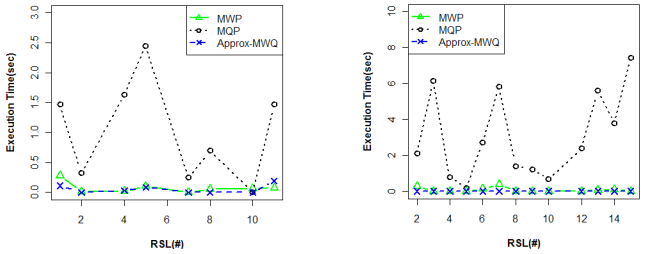
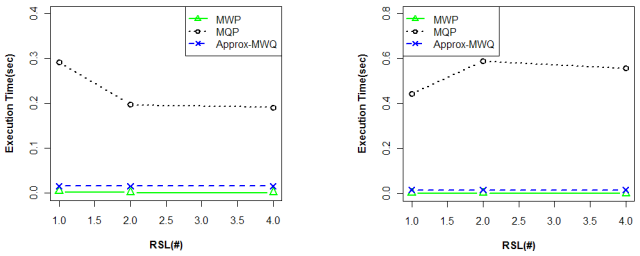


Fig. 16. Approximated DSL and $\overline{DDR}$ of c_t for k=3, and approximated $\overline{DDR}(c_t)$ misses the shaded region



(a) CarDB-100K (b) CarDB-200K



(c) UN-100K (d) UN-200K

Fig. 17. Execution time of MWP, MQP, and Approx-MWQ in CarDB and synthetic datasets

VII. Related Work and Discussion

Previous studies [5], [4], [6], [2] and [3] have addressed the issue of answering why-not questions in different data settings. In [5], Huang et al. and in [6], Herschel et al. propose to modify the original tuple values in the database so that why-not (missing) tuples become part of the (SPJ and SPJUA, respectively) query output. In [4], Chapman et al. propose to identify the culprit operator(s) that filters out the why-not (missing) tuple(s) from the query output. As a next step, Tran and Chan [2] answer why-not questions for SPJA queries through query refinement where they collect why-not (missing) tuples as feedback from the user. The authors exploit the idea of skyline queries to report the closest refined query with respect to the original one to minimize the distance between refined and original query. In [3], He et al. propose an approach to answer why-not questions on top-k queries through the modification of both k and/or weightings. Yet, before [5], [4], [6], [2] and [3], Motro [13] has discussed about the approaches for explaining empty answer for a query.

In user feedback-based query refinement techniques, only false positives (why) feedback have been emphasized in both database and information extraction areas before [14],[15]. In [14], Ma *et al.* model user feedback query refinement for both learning the structure of the query as well as learning the relative importance of query components, but they collect

TABLE V
EFFECT OF APPROX. IN COST FOR CARDB DATASETS

Queries	MWP	MQP	MWQ	Approx-MWQ$_{k=10}$
$q_1, \|RSL(q_1)\| = 1$	0.583105121	0.292546395	0.000000000	0.000000000
$q_2, \|RSL(q_2)\| = 2$	0.257558213	0.357151016	0.000000000	0.000000000
$q_3, \|RSL(q_3)\| = 4$	0.050311016	0.177055208	0.036971632	0.050311016
$q_4, \|RSL(q_4)\| = 5$	0.086962389	0.50232323	0.08839361	0.086962389
$q_5, \|RSL(q_5)\| = 7$	0.064378944	0.421775274	0.064355373	0.064378944
$q_6, \|RSL(q_6)\| = 8$	0.09346712	0.867555893	0.093459366	0.09346712
$q_7, \|RSL(q_7)\| = 10$	0.083575728	0.691113567	0.083559007	0.083575728
$q_8, \|RSL(q_8)\| = 11$	0.107217572	1.007932269	0.107204479	0.107217572

(a) CarDB-100K dataset

Queries	MWP	MQP	MWQ	Approx-MWQ$_{k=20}$
$q_1, \|RSL(q_1)\| = 2$	0.317222443	0.780333802	0.32123177	0.32123177
$q_2, \|RSL(q_2)\| = 3$	0.38737201	1.06368824	0.292690644	0.292690644
$q_3, \|RSL(q_3)\| = 4$	0.163688678	0.487682365	0.140717077	0.140717077
$q_4, \|RSL(q_4)\| = 5$	0.076933034	0.298898582	0.075045841	0.056923304
$q_5, \|RSL(q_5)\| = 6$	0.172445738	0.788075013	0.220154031	0.172445738
$q_6, \|RSL(q_6)\| = 7$	0.161279426	1.29542575	0.190381004	0.161279426
$q_7, \|RSL(q_7)\| = 8$	0.028030879	0.210567316	0.029396864	0.028030879
$q_8, \|RSL(q_8)\| = 9$	0.079989249	0.542973513	0.079284723	0.079989249
$q_9, \|RSL(q_9)\| = 10$	0.045925736	0.818973575	0.045906123	0.045925736
$q_{10}, \|RSL(q_{10})\| = 12$	0.040420303	0.539585348	0.04041207	0.040420303
$q_{11}, \|RSL(q_{11})\| = 13$	0.04782151	0.595096374	0.04782151	0.04782151
$q_{12}, \|RSL(q_{12})\| = 14$	0.062465558	0.800602413	0.057050319	0.062465558
$q_{13}, \|RSL(q_{13})\| = 15$	0.036071785	0.44020813	0.036065908	0.036071785

(b) CarDB-200K dataset

TABLE VI
EFFECT OF APPROX. IN COST FOR SYN. DATASETS

Queries	MWP	MQP	MWQ	Approx-MWQ$_{k=10}$
$q_1, \|RSL(q_1)\| = 1$	0.111111111	0.191954545	0.101015152	0.106065657
$q_2, \|RSL(q_2)\| = 2$	0.144572892	0.393979798	0.136378788	0.144572892
$q_3, \|RSL(q_3)\| = 4$	0.296552062	1.459651515	0.292944444	0.296552062

(a) UN-100K dataset

Queries	MWP	MQP	MWQ	Approx-MWQ$_{k=10}$
$q_1, \|RSL(q_1)\| = 2$	0.125424936	0.333393939	0.116166667	0.125424936
$q_2, \|RSL(q_2)\| = 3$	0.097817012	0.358641414	0.093439394	0.097817012
$q_3, \|RSL(q_3)\| = 4$	0.170237292	0.828358586	0.166671717	0.170237292

(b) CO-100K dataset

Queries	MWP	MQP	MWQ	Approx-MWQ$_{k=10}$
$q_1, \|RSL(q_1)\| = 1$	0.151515152	0.277813131	0.141419192	0.151515152
$q_2, \|RSL(q_2)\| = 2$	0.522911419	1.489949495	0.462126263	0.462126263
$q_3, \|RSL(q_3)\| = 3$	0.280323728	1.085944444	0.277782828	0.280323728
$q_4, \|RSL(q_4)\| = 4$	0.162777034	0.777888889	0.161621212	0.162777034

(c) AC-100K dataset

Queries	MWP	MQP	MWQ	Approx-MWQ$_{k=10}$
$q_1, \|RSL(q_1)\| = 1$	0.237373737	0.444479798	0.227277778	0.232328283
$q_2, \|RSL(q_2)\| = 2$	0.583246564	1.590969697	0.454550505	0.454550505
$q_3, \|RSL(q_3)\| = 4$	0.367259132	1.787949495	0.363651515	0.367259132

(d) UN-200K dataset

only false positive feedback from users. In [15], Liu et al. collect false positives (why tuples), again identified by users, to modify the initial rules in information extraction settings. In a very recent work, Islam et al. [16] propose a user feedback based query refinement framework for encountering both why and why-not questions in SPJ query output by exploiting the skyline operator. The authors consider minimizing the unexpected (why) tuples as well as maximizing the expected (why-not) tuples in the refined query output.

To the best of our knowledge, we are the first to address why-not questions in reverse skyline queries. The contributions mostly related to this paper are [17] and [18]. Given a "cost" column in Q, the authors in [17] create top-k profitable products that are not dominated by any product available in the market, P. The authors in [18] propose skyline distance as a measure of multidimensional competence and propose algorithms for computing the minimum cost of upgrading a point to the skyline given a cost function. In our work, we show how to make a product (query point) interesting to a customer (why-not point) by modifying product features (query attributes) and/or customer preferences. We also consider that modification of product features does not affect existing customers who are already interested in the product.

VIII. CONCLUSION

In this paper, we present the semantics of answering why-not questions in reverse skyline queries. In connection with this, we also show how to modify the why-not data point as well as the query point to answer why-not questions. Then, we show how to compute the safe region of a query-point where it can be moved while keeping its existing reverse skyline, and how to answer why-not questions considering the safe region of the query-point. Experimental results demonstrate the effectiveness of our approach in answering why-not questions in reverse skyline queries. As the construction of the exact safe region of the query point is time inefficient, we also show how to construct approximated safe region of the query point to answer why-not questions in reverse skyline queries by sacrificing the quality of results.

ACKNOWLEDGMENT

This work is supported by the grants of ARC Discovery Projects DP120102627 and DP110102407. We are grateful to the anonymous reviewers for their constructive comments.

REFERENCES

[1] H. V. 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] Q. T. Tran and C.-Y. Chan, "How to conquer why-not questions," in *SIGMOD*, 2010, pp. 15–26.
[3] Z. He and E. Lo, "Answering why-not questions on top-k queries," in *ICDE*, 2012, pp. 750–761.
[4] A. Chapman and H. V. Jagadish, "Why not?" in *SIGMOD*, 2009, pp. 523–534.
[5] J. Huang, T. Chen, A. Doan, and J. F. Naughton, "On the provenance of non-answers to queries over extracted data," *PVLDB*, vol. 1, no. 1, pp. 736–747, 2008.
[6] M. Herschel and M. A. Hernández, "Explaining missing answers to spjua queries," *PVLDB*, vol. 3, no. 1, pp. 185–196, 2010.
[7] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *SIGMOD*, 2003, pp. 467–478.
[8] S. Börzsönyi, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001, pp. 421–430.
[9] E. Dellis and B. Seeger, "Efficient computation of reverse skyline queries," in *VLDB*, 2007, pp. 291–302.
[10] P. M. Deshpande and D. P, "Efficient reverse skyline retrieval with arbitrary non-metric similarity measures," in *EDBT*, 2011, pp. 319–330.
[11] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger, "The r*-tree: An efficient and robust access method for points and rectangles," in *SIGMOD*, 1990, pp. 322–331.
[12] J. V. den Bercken, B. Blohsfeld, J.-P. Dittrich, J. Krämer, T. Schäfer, M. Schneider, and B. Seeger, "Xxl - a library approach to supporting efficient implementations of advanced database queries," in *VLDB*, 2001, pp. 39–48.
[13] A. Motro, "Flex: A tolerant and cooperative user interface to databases," *IEEE Trans. Knowl. Data Eng.*, vol. 2, no. 2, pp. 231–246, 1990.
[14] Y. Ma, S. Mehrotra, D. Y. Seid, and Q. Zhong, "Raf: An activation framework for refining similarity queries using learning techniques," in *DASFAA*, 2006, pp. 587–601.
[15] B. Liu, L. Chiticariu, V. Chu, H. V. Jagadish, and F. Reiss, "Automatic rule refinement for information extraction," *PVLDB*, vol. 3, no. 1, pp. 588–597, 2010.
[16] M. S. Islam, C. Liu, and R. Zhou, "User feedback based query refinement by exploiting skyline operator," in *Conceptual Modeling*, 2012, pp. 423–438.
[17] Q. Wan, R. C.-W. Wong, and Y. Peng, "Finding top-k profitable products," in *ICDE*, 2011, pp. 1055–1066.
[18] J. Huang, B. Jiang, J. Pei, J. Chen, and Y. Tang, "Skyline distance: a measure of multidimensional competence," *Knowledge and Information Systems*, 2012.