ANSWERING WHY-NOT QUESTIONS ON PREFERENCE QUERIES

ZHIAN HE

Ph.D

The Hong Kong Polytechnic University

2015

The Hong Kong Polytechnic University

Department of Computing

Answering Why-Not Questions on Preference Queries

Zhian He

A thesis submitted in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Dec 2014

CERTIFICATE OF ORIGINALITY

I hereby declare that this thesis is my own work and that, to the best of my knowledge and belief, it reproduces no material previously published or written, nor material that has been accepted for the award of any other degree or diploma, except where due acknowledgement has been made in the text.

.....

Zhian He

Dec 2014

Abstract

After decades of effort working on database performance, the quality and the usability of database systems have received more attention in recent years. Among all the studies that focus on improving databases' usability, the feature of explaining missing tuples in a query result, or the so-called "why-not" questions, has recently become an active topic. When using database system, users may sometimes feel frustrated if they find their expected tuples are not in the query results; and intuitively, they will ask *"why are my expected tuples not in the results?"* If a database system can give a good explanation for it, it would be very useful for users to understand and refine their queries.

In this dissertation, we study the problem of answering why-not questions on preference queries. Our motivation is that we know many users love to pose this 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. Therefore, we select three important kinds of preference queries (namely, top-k queries, top-k SQL queries and dominating queries) and develop algorithms to answer such "why-not" questions on each of them.

List of Publications

- Zhian He, Eric Lo. Answering Why-Not Questions on Top-k Queries. *Proceedings of the 28nd IEEE International Conference on Data Engineering*, ICDE 2012. (Selected as Bests of ICDE 2012)

- Zhian He, Petrie Wong, Ben Kao, Eric Lo, Reynold Cheng. Fast Evaluation of Iceberg Pattern-based Aggregate Queries. *Proceedings of the 22nd ACM International Conference on Information and Knowledge Management*, CIKM 2013.

- Petrie Wong, Zhian He, Eric Lo. Parallel Analytics as a Service. *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD 2013.

- Zhian He, Eric Lo. Answering Why-Not Questions on Top-k Queries. *IEEE Transactions on Knowledge and Data Engineering*, vol. 26, no. 6, pp. 1300-1315, June 2014.

- Petrie Wong, Zhian He, Feng Ziqiang, Wenjian Xu and Eric Lo. Thrifty: Offering Parallel Database as a Service using the Shared-Process Approach. *Proceedings of the ACM SIGMOD International Conference on Manage-*

ment of Data, SIGMOD 2015.

- Wai Kit Wong, Zhian He, Ben Kao , David W. Cheung, Rongbin Li, Siu Ming Yiu and Eric Lo. SDB: A Secure Query Processing System with Data Interoperability. (*Submitted to VLDB 2015*).

- Zhian He, Petrie Wong, Zhiqiang Feng, Ben Kao, Eric Lo, Reynold Cheng, Tatiana Aldyn-ool. IPBA: A System for Fast Evaluation of Iceberg Pattern-based Aggregate Queries. (*Submitted to TKDE*).

- Wenjian Xu, Zhian He, Eric Lo, Chi Yin Chow. Explaining Missing Answers to Top-k SQL Queries. (*Submitted to TKDE*).

Acknowledgements

It is a pleasure to thank the many people who helped me a lot during my PhD study.

It is difficult to overstate my gratitude to my PhD supervisor, Dr. Eric Lo. With his enthusiasm, his inspiration, his patience and his great efforts to explain things clearly and simply, he helped to make my research work more fun to me. His way to thinking and problem solving always inspired me. Throughout my thesis-writing period, he provided encouragement, sound advice, good teaching, and lots of good ideas. I would have been lost without him.

My parents have been a constant source of support − emotional and moral − during my postgraduate years, and this thesis would certainly not have existed without them. One of the most important persons who has been with me in every moment of my PhD work is my wife Shelly. I would like to thank her for the many sacrifices she has made to support me in undertaking my doctoral studies. Especially in those days that I was in bad physical condition. Without her careful tending and encouragement, I cannot carry through the work to the end.

I also thank Dr. Ben Kao, Dr. Shivnath Babu, and Dr. Ken Yiu, who have

guided me a lot in the way of doing research.

I am indebted to my many colleagues for providing a stimulating and joyful environment learning and growing. I am especially grateful to Ming-Hay Luk, Duncan Yung, Jianguo Wang, Yuanyuan Wang, Zhizhao Feng, Victor Liang, Jeppe Thomsen, Yu Li, Petrie Wong, Qiang Zhang, Ziqiang Feng, Bob Tong, Capital Li, Wenjian Xu, Chuanfei Xu, Bai Ran. Wenjian was particular helpful in running some parts of the experiments in this thesis.

Contents

List of Figures

Chapter 1

Introduction

After decades of effort working on database performance, recently the database research community has paid more attention to the issue of *database usability* [23], i.e., *how to make database systems and database applications more user friendly?* For example, to help end users query the database more easily, the features of keyword search [2, 17, 26] or form-based search [37] could be added to a database system to assist users to find their desired results. As another example, the features of query recommendation [3] or query autocompletion [30] could be added to a database system in order to help users to formulate SQL queries. Among all the studies that focus on improving database usability (e.g., SQL query auto-completion), the feature of explaining why some expected tuples are missing in a query result, or the so-called *"why-not?"* feature, is gaining momentum.

A why-not question is being posed to a database when a user wants to know why her expected tuples do not show up in the query result. Currently, end

users cannot directly sift through the dataset to determine "why-not?" because the query interface (e.g., web forms) restricts the types of query that they can express. When end users query the data through a database application and ask "why-not?" but do not find any means to get an explanation through the query interface, that would easily cause them to throw up their hands and walk away from the tool forever — the worst result that nobody, especially the database application developers who have spent months to build the database applications, want to see. Unfortunately, supporting the feature of why-not requires deep knowledge of various database query evaluation algorithms, which is beyond the capabilities of most database application developers. In view of this, recently, the database community has started to research techniques to answer why-not questions.

A few recent works have discussed techniques for answering why-not questions. So far, they have focused on SPJUA (Select+Project+Join+Union+Aggregate) queries and they cannot be used to answer why-not questions on *preference queries*. Faced with information overload, preference queries have been introduced to database systems to present users with the most preferred answers, instead of all the answers. Preference queries are useful in multi-criteria analysis and users would also ask "why-not?" in case their preferred answers are missing in the result. Unfortunately, techniques for answering why-not questions on SPJUA queries are insufficient for answering why-not questions on preference queries. In a preference query, a tuple can be included in the final result only when it can "beat" many other tuples in the database through tuple-to-tuple comparisons. In contrast, in a SPJUA query, whether a tuple can be included in the final result mainly depends on whether it can pass through the query predi-

cates, which is independent of most other tuples. The difference between the two answer spaces voids the use of existing techniques to answer why-not questions on preference queries — that signifies that there is a technology gap between why-not preference query processing and why-not SPJUA query processing.

In this dissertation, we want to remove the above technology gap by proposing techniques to answer why-not questions on preference queries. Specifically, we have selected the three most representative kinds of preference queries, namely, top-k queries, top-k SQL queries and dominating queries to be included in our study. There are three main challenges when explaining why-not questions on preference queries. The first challenge is "*what should the answer of a why-not question on preference query (i.e., the explanation) look like?*" One possible explanation type is "explain-by-query-refinement" — teaching the end user how to refine her query such that the missing tuple can go back to the result. The second challenge is: "*what is a good explanation and how to obtain that efficiently (i.e., short running time)?*" We address this by first understanding the problem complexity and then devising the corresponding algorithms. The third challenge is: "*how to evaluate our proposed solutions for this new kind of problem?*" We address this challenge by devising an evaluation metric and implementing our proposed solutions as a prototype, and using that for evaluation.

1.1 Why-Not Top-K Question

The first part of the dissertation is devoted to answering why-not questions on top-k queries. In a top-k query [20], a user needs to specify the k value and also a set of weightings $\vec{w}$ on the scoring attributes, such that only the top-k

objects are returned based on their ranking scores. A "why-not" question on a top-k query is asking why a specific object is not in the top-k result. In fact, answering why-not questions on top-k queries is very useful because while users love to use top-k queries when making multi-criteria decisions, they often feel frustrated when they are forced to quantify their preferences as a set of numeric weightings. Moreover, they would feel even more frustrated when their expected answers are missing in the query result.

For instance, a customer Mary is going to Paris for a holiday, and this is her first time to be there, so she needs to choose a hotel carefully from a list of hotels in Paris. Yet, the list is so long to read, therefore, Mary decides to look at the top-3 hotels based on a set of weightings she sets on the attributes "*Price*", "*Distance-to-Downtown*", and "*Rating-by-other-customers*". To her surprise, hotel Hilton, which is very famous around the world, and Mary's favourite, is not in the result. Now, Mary may feel frustrated: "*Why is my favourite hotel not in the top-3 result? Is that because there is no Hilton Hotel in Paris? Should I revise my weightings? Or my Hilton Hotel can be back to result if I simply revise my query to look for top-5 hotels instead of top-3 hotels?*" Chapter 3 is dedicated to answering why-not questions on this kind of queries.

1.2 Why-Not Top-K SQL Question

The second part of the dissertation is devoted to answering why-not top-k questions in the context of SQL. Specifically, a top-k query in SQL [21, 22, 25] appears as:

`SELECT` A_1, ..., A_m, $agg(\cdot)$

FROM T_1, ... T_k,

WHERE P_1 AND ... AND P_n

GROUP BY A_1, ..., A_m,

ORDER BY $f(\vec{w})$

LIMIT k

where each P_i is either a selection predicate "A_i *op* v" or a join predicate "A_j *op* A_k", where A_i is an attribute, v is a constant, and *op* is a comparison operator. Besides, $agg(\cdot)$ is an aggregation function and $f(\vec{w})$ is a ranking function with weighting vector $\vec{w}$. The weighting vector represents the user preference when making multi-criteria decisions.

There could be many reasons that the expected tuple is not in the top-k SQL result. It may be the k value is too small, or the weighting $\vec{w}$ favours other tuples rather than the expected one, or the selection condition is too restricted such that the expected tuple is filtered.

Compared with answering why-not top-k questions, answering why-not questions on top-k SQL queries is more challenging, since the additional SQL constructs such as selection, projection, join and aggregation could complicate the solution space. We solve this problem in Chapter 4.

1.3 Why-Not Dominating Question

The last part of the dissertation is to answer the same questions on the top-k dominating queries. Top-k dominating queries [38], or simply *dominating queries*, is a variant of top-k query that users may pose why-not questions on. While a

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 manager of a car brand may pose a top-3 dominating query about the best car models in the market. When there is no car of his brand in the result, she may want to know the reason.

The easiest way to make the missing object appear in the result is to increase the k value. This, however, is not enough to provide a good explanation. For example, in the above query, we may need to significantly increase the k value from 3 to 10000 to make the missing car model back to the result. On the contrary, the missing car model may be easily back to the top-3 result if we just slightly modify the values of it, e.g., decreasing its price a little bit. This kind of explanation is more useful for the manager to develop her marketing strategy.

In Chapter 5, we propose a hybrid explanation model (combining query-refinement and data-refinement) that modifies both the k value and the values of the missing objects to answer the why-not question on dominating queries.

1.4 Dissertation Outline

The rest of the dissertation is organized as follows. Chapter 2 presents the literature review. Chapter 3, Chapter 4 and Chapter 5 respectively present (i) the problem formulation, (ii) the problem analysis, (iii) the algorithms of an-

[1]The notion of *dominance* follows the tradition [7] 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.

swering why-not questions and (iv) cases studies and the experimental results on answering why-not questions on top-k queries, top-k SQL queries and dominating queries. Chapter 6 concludes the dissertation.

Chapter 2

Literature Review

Early works [28, 29] discussed mechanisms for explaining an empty answer for a database query. It points out that most database research mainly focuses on the performance issues, while the user experience of interaction between human and DBMS has not received enough attention. Especially, they all lack the ability to provide explanations to the user when her query returns an empty answer. Experience shows that empty answers are quite often interpreted by users as indication of mistakes. This happens when the user believes that the query should have matched some data, and therefore concludes that something goes wrong. If that is the case, she may react to check if there are any misspelled names in her query, or operations used incorrectly. She may also just conclude that she has insufficient understanding of the database. No matter in which case, the user interprets such empty answer as "mistake messages". Consequently, the user may end up trying different versions of this query in an attempt to understand the reason for its failure. This kind of frustrated experience is not uncommon

for many database users. Therefore, providing an explanation to empty answer is able to improve the user experience a lot. To handled that, [28, 29] propose to give an explanation by providing a set of more generalized queries than the original one to the user. For example, if the query: (Q1) select * from employee where gender=female and salary $>$ 30000 returns an empty answer, DBMS may explain it by providing a more generalized query: (Q2) select * from employee where salary $>$ 30000. The principle behind is that the original query formulated by the user is meaningful and expresses her intention. For example, she may believe that there are female employees who earn more than 30000 (otherwise why bother asked). Therefore, it is reasonable to assume that this user is even more confident that some employee (either males or females) indeed earn more than 30000. In other words, while the first query conveys the conceptions of the user about the database with some uncertainty, its more general queries convey user conceptions with a greater degree of confidence. As the first few attempts to explain empty answers to the user, the work in [28, 29] is simple but useful.

Name	Department	Age	Salary
Henry	IT	25	20000
Tom	Financial	34	34000
Peter	HR	27	10000
Mary	HR	27	30000
Sam	HR	20	9000
Ken	Sales	34	40000

Table 2.1. An example table: Employee

The concept of *why-not* was first formally discussed in [9]. Different from [28, 29] that focus on empty answers, a why-not question is more general. Specifically, the user may ask *why* a specific tuple is *not* in the query result even if the result is nonempty. As the first work, it proposes the *operator-identifying* approach to

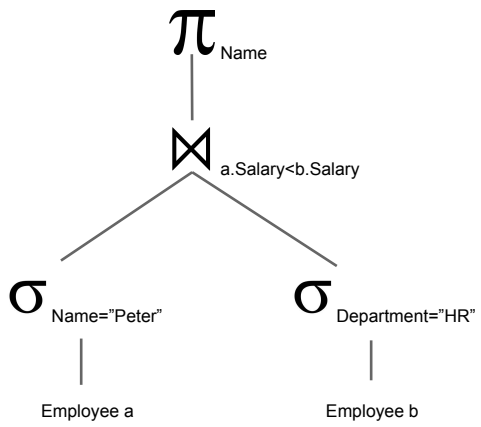


Figure 2.1. An example query to find all employees in the HR department having salary larger than Peter.

answer a user's why-not question on Select-Project-Join (SPJ) queries by telling her which query operator(s) eliminate her desired tuples. It is useful for the user to better understand the data behind and also further debug her query if necessary. Techniques in this work are designed for relational/workflow queries, and they draw heavily upon the formalisms and concepts of *lineage* set out by [1,13,14]. Table 2.1 shows an example employee table. It is used to illustrate the main idea in [9]. Assume a user issues the query (represented by a logical tree) in Figure 2.1, which aims to find all employees in the HR department having salary larger than $Peter$. The query result is $\{Mary\}$. The user wonders why Sam is not in the result. In this case, the method in [9] identifies that actually the Join operator eliminates her expected tuple, which implies that Sam does not have salary larger than $Peter$. If another user asks why $Henry$ is not in the result, the explanation will return the selection operator on the right hand side. The user can interpret that there is no person named $Henry$ in the "HR" department.

UID1	UID2
U1	U2
U2	U3

Table 2.2. Table Friend adopted from [18]

UID	Email	Name
U1	john@univ.edu	John
U2	jane@busy.com	Jane
U3	peter@home.de	Peter

Table 2.3. Table User adopted from [18]

After [9], this line of work has gradually expanded. In [19] and [18], the missing answers of SPJ [19] and SPJUA (SPJ + Union + Aggregation) queries [18] are explained by a *data-refinement* approach, i.e., tells the user how the data should be modified (e.g., adding a tuple) if she wants the missing answer back to the result. Consider two example tables $Friend$ and $User$ shown in Tables 2.2 and 2.3 adopted from [18]. A user may wonder why Peter is not a friend of John. A possible explanation is to add a new tuple $(U1, U3)$ in the $Friend$ table. In fact, there could be many other ways to create new tuples to answer the why-not question. [19] and [18] propose to find the explanation with the minimal changes of the database. For example, in [18], the method proposed first computes some possible ways of tuple insertions such that the missing answer(s) appear in the result when issuing the original query on the modified database. Then, it tries its best to prune redundant solutions such that the number of inserted tuples are as less as possible.

Following the above works, [35] proposes a new *query-refinement* approach to answer why-not question on SPJA queries. This approach explains the why-not question by telling the user how to revise her original SPJA queries so that

the missing answers can appear in 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. To illustrate their idea, an example is given as below.

pID	name
P1	"A"
P2	"B"
P3	"C"
P4	"D"
P5	"E"

Figure 2.2. Table Player adopted from [35]

pID	team	year	points	block	steal	rebound
P1	GSW	1993	2000	30	150	40
P2	SWA	1994	1600	35	200	281
P2	SEA	1995	1500	40	240	339
P3	CHI	1996	2500	45	250	361
P4	LAL	1997	1700	30	190	359

Figure 2.3. Table Regular adopted from [35]

Figure 2.2 and Figure 2.3 show a basketball data set. Assume a user want to find players who have "block" statistics no greater than 30 and "steal" statistics no greater than 150:

```
SELECT name
FROM Player P, Regular R
WHERE P.pID = R.pID and block ≤ 30 and steal ≤ 150
```

Referring to the join result of tables *Player* and *Regular* in Figure 2.4 , the output includes only one player "A". The user may wonder why player "B" is

pID	name	team	year	points	block	steal	rebound
P1	"A"	GSW	1993	2000	30	150	40
P2	"B"	SWA	1994	1600	35	200	281
P2	"B"	SEA	1995	1500	40	240	339
P3	"C"	CHI	1996	2500	45	250	361
P4	"D"	LAL	1997	1700	30	190	359

Figure 2.4. Player ⋈ Regular

not in the result. The following are two candidate refined queries that [35] will return to the user:

Q1:

SELECT name

FROM Player P, Regular R

WHERE P.pID = R.pID and $block \leq \mathbf{35}$ and $steal \leq \mathbf{200}$

Q2:

SELECT name

FROM Player P, Regular R

WHERE P.pID = R.pID and $block \leq \mathbf{35}$ and $steal \leq \mathbf{200}$ and $\boldsymbol{year} \leq \mathbf{1994}$

The results of Q1 and Q2 are $\{A, B, D\}$ and $\{A, B\}$ respectively. Both of these refined queries are considered to be good, because Q1 does the minimal and essential refinement to include "B" that only changes two constants. Compared with Q1, although Q2 adds one more predicate, its result contains no extra tuples.

It turns out that the query-refinement approach is more useful than the operator-identifying approach. Beyond merely identifying the culprit operator, the query-refinement approach can actually suggest one or more ways to "fix"

the original query such that the missing tuple(s) become present in the result. Besides, it is more useful than the data-refinement approach for those cases that the database is trusted and modifying the underlying data is meaningless (e.g., enterprise databases). Therefore, we adopt this approach in our problem.

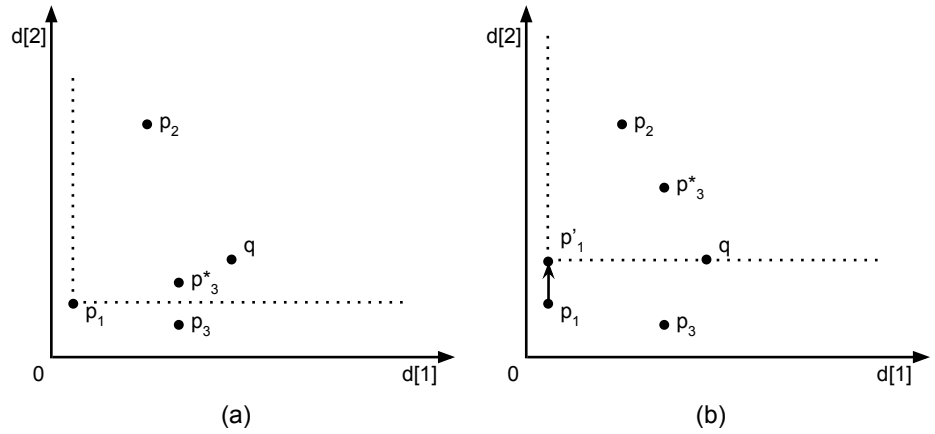


Figure 2.5. An example data set for why-not reverse skyline.

A recent work [27] is about answering why-not questions on *reverse skyline queries*, which is a special kind of preference queries. Given a set of data points D and a query point q, the *dynamic skyline* [31] of a data point $p \in D$ is the skyline of a transformed data space using p as the origin and the reverse skyline [15] returns the objects whose dynamic skyline contains the query point q. The why-not question to a reverse skyline query then asks about why a specific data point $p \in D$ is not in the reverse skyline of q. In [27], explanations using both data-refinement and query-refinement are discussed. In the data-refinement approach, the explanation is based on modifying the value of p such that p appears in the reverse skyline of q. In the query-refinement approach, the explanation is based on the modifying the query point q until p is in its reverse skyline. We illustrate their idea with an example. Figure 2.5(a) shows a set of data points

$D = \{p_1, p_2, p_3\}$ and a query point q. The dynamic skyline of p_1 are p_2 and p_3, while q is not the dynamic skyline of p_1 since it is dominated by p_3^* (it is the transformed point of p_3 in the transformed data space using p_1 as the origin). So, p_1 is not in the reverse skyline of q. Figure 2.5(b) shows an example modification based on data-refinement proposed in [27]. After modify p_1 to p_1', q appears to be the dynamic skyline of p_1'. Among all the possible modifications, [27] proposes to find the minimal one in terms of the value changes.

The latest work [11] is about answering why-not questions on spatial keyword top-k queries. Given a spatial location *loc* and a set of keywords *doc*, a top-k spatial keyword query [12], denoted by $q(loc, doc, k, \vec{w})$, returns the k best spatio-textual objects ranked according to their spatial distance to *loc* and textual similarity to *doc*, where $\vec{w} = |w_s \ w_t|$ is the weighting vector on the user's preference between spatial distance and textual similarity. Thus, the score of an object o can be calculated by the following equation:

$$Score(o, \vec{w}) = w_s \cdot (1 - SpatialDist(o, loc)) + w_t \cdot TextSim(o, doc) \qquad (2.1)$$

, where the spatial distance *SpatialDist* between two objects is normalized to the range [0,1].

The why-not question then ask why a specific object m is missing in the result of $q(loc, doc, k, \vec{w})$. To answer such a question, [11] adopts the query-refinement approach to modify the value k and $\vec{w}$ at the same time, such that m appears in the refined top-k spatial keyword query $q'(loc, doc, k', \vec{w}')$. Though it is similar to our why-not top-k problem, however, the method proposed in [11] only works for two dimensional weightings.

Beyond database queries, the why-not semantic has been extended to social image search [6], where images are socially tagged by their uploaders or viewers. The input of the social image search is a query Q with a set of keywords e.g. ("football", "apple"). Then, a search engine is responsible to return a list of images where the images annotated with the most *relevant* tags to the query are ranked higher. The why-not question then asks why images with a specific tag t_m are too few or even missing in the result. [6] proposes to relax the query by deleting some *selective* keywords in Q that are responsible for filtering majority of the relevant images related to t_m.

There are other related works about improving the usability of preference queries in the absence of why-not context. For example, in [39], they help 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. Technically speaking, the problem of answering why-not questions on preference queries like top-k queries has the challenges that we have to consider the alternation of both k and/or weightings and how to make a good balance between them using an efficient algorithm.

weighting	top-3 results
$\vec{w}_1 = \|0.3\ 0.7\|$	$\{p_1, p_2, p\}$
$\vec{w}_2 = \|0.5\ 0.5\|$	$\{p_1, p_2, p_3\}$
$\vec{w}_3 = \|0.8\ 0.2\|$	$\{p_1, p_3, p\}$

Table 2.4. An example candidate weighting set W

Another such related work is *reverse top-k queries* [36]. A reverse top-k query takes as input a top-k query, an existing object p, and a set of candidate

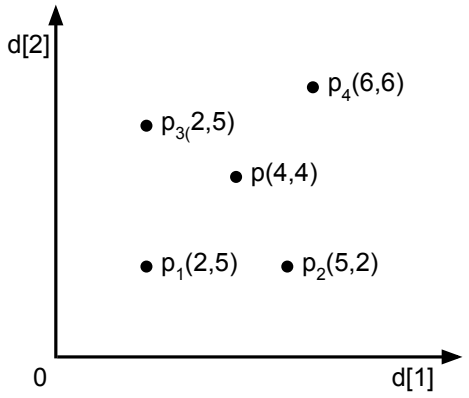


Figure 2.6. An example data set for reverse top-k.

weightings W. The output is a weighting $\vec{w} \in W$ that makes p in its top-k result. Figure 2.6 shows an example data set with five data points including the object p. Assume the user issues a top-3 query and a smaller score means a better ranking, then among the set of candidate weightings shown in Table 2.4, the result of reverse top-3 are weightings $\vec{w}_1$ and $\vec{w}_3$. Two solutions are given in [36]. 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 preference queries indeed does not require users to provide W. Furthermore, the problem of reverse top-k queries considers the alternation of weightings only. Answering why-not questions on preference queries, like top-k queries, in contrast, considers both the alternation of k and/or weightings. More importantly, our solution can practically solve the problem without any limitation.

Chapter 3

Why-Not Top-K Question

In this chapter, we present methods to answer why-not questions on top-k queries. We follow the latest *query-refinement* [35] approach that suggests users how to refine their top-k query in order to get their expected answers back. Note that the result of a top-k query depends on two sets of parameters: (1) the number of objects to be included in the result, i.e., the k value; and (2) the set of weightings $\vec{w}$ specified by the user. In some cases, the missing object can easily be back in the result if the k value is slightly increased (e.g., refining a top-3 query to be a top-5 query). However, there can be cases where the missing object will not be included in the result unless the k value is dramatically increased (e.g., refining a top-3 query to be a top-10000 query), and for those cases, modifying the set of weightings $\vec{w}$ instead of k may make more sense from the users perspective. Furthermore, there could be cases in which the best option is to slightly modify both the k value and the set of weightings $\vec{w}$ together, instead of significantly modify one of them alone.

To address this problem, we first define a penalty model to capture a user's tolerance to the changes of weightings $\vec{w}$ and k on her original query. Next, we show that finding the refined top-k query with the least penalty is actually computational expensive. Afterwards, we present an efficient algorithm that uses sampling to obtain the best approximate answer in reasonable time. We evaluate the quality and the performance of our method using real data and synthetic data.

3.1 Preliminary

In this section, we give a formal definition to the problem of answering *why-not* questions on a top-k query. Afterwards, we discuss the difficulty of solving the problem exactly.

3.1.1 Problem Statement

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**. 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 chapter, we accept the scoring function *score* as a linear function, where $score(\vec{p}, \vec{w}) = \vec{p} \cdot \vec{w}$, k as any positive integer, 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* $M = \{\vec{m}_1, \ldots, \vec{m}_j\}$, and hope the system to return her 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, we describe 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 is 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, 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:

$$Penalty = \lambda_k \Delta k + \lambda_w \Delta w \tag{3.1}$$

Note that the basic penalty model favours 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* [7], another refined query Q'_2, if both its refinements on k (i.e., Δk) and $\vec{w}$ (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 3.1 *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 know 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 $\Delta w \leq \sqrt{\sum w[i]^2 + \sum w_o[i]^2} \leq$ $\sqrt{1 + \sum w_o[i]^2}$. □

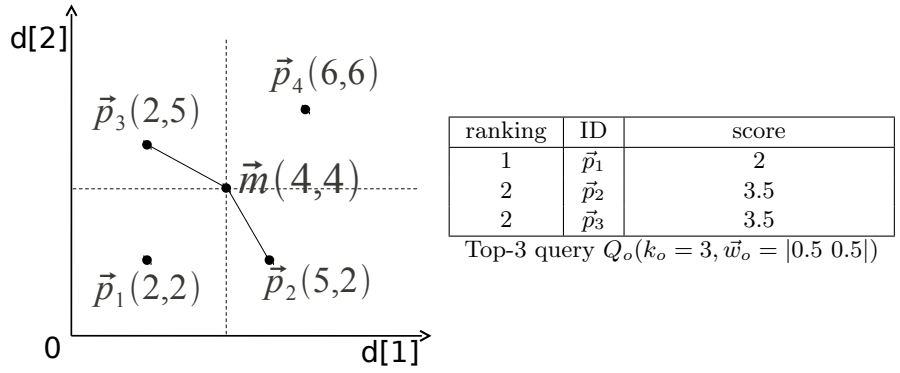
Now, we have a *normalized penalty function* as follows:

$$Penalty = \lambda_k \frac{\Delta k}{(r_o - k_o)} + \lambda_w \frac{\Delta w}{\sqrt{1 + \sum w_o[i]^2}} \tag{3.2}$$

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 and with the smallest penalty. In this chapter, we use Equation 3.2 as the penalty function. Nevertheless, our solution indeed 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[1] illustrated in Figure 3.1.

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

Figure 3.1. 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 $Q_o(k_o = 3, \vec{w}_o = |0.5\ 0.5|)$

Figure 3.2. A 2-D example

Let us give an example. Figure 3.2 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 $Q_o(k_o =$

[1]The number of choices and the pre-defined values for λ_k and λ_w, of course, could be adjusted. Furthermore, one might prefer to minimize the difference between the new result set and the original one, instead of minimizing the difference between the refined query and the original query. In that case, we suggest the user to choose the option where λ_w is a big value (i.e., *not* prefer modify the weightings) because [36] has pointed out that similar weightings generally lead to similar top-k results.

$3, \vec{w}_o = |0.5\ 0.5|$) 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 3.1 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 3.1. Example of candidate refined queries

Refined Query	Δk_i	Δw_i	Penalty
$Q'_1(4,\ \|0.5\ 0.5\|)$	1	0	0.5
$Q'_2(5,\ \|0.6\ 0.4\|)$	2	0.14	1.06
$Q'_3(3,\ \|0.7\ 0.3\|)$	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

3.1.2 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 $a[i] < b[i]$, and we say two objects are incomparable if no one can dominate the other. 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). Recall that our objective is to find a combination of k and weighting $\vec{w}_{r_i}$ that minimizes Equation 3.2. However, the weighting vector set W_{r_i} is actually either empty or a set of convex polytopes (Lemma 3.2). That means when W_{r_i} is not empty, then there are an infinite number of points $\vec{w}_{r_i} \in W_{r_i}$, which also makes the number of candidate answers infinite.

Lemma 3.2 *W_{r_i} is either empty or a set of convex polytopes.*

Proof. Given a data space and a missing object $\vec{m}$, assume there are k_d objects dominate $\vec{m}$, and the number of incomparable objects with $\vec{m}$ is n. To find the set of weighting vectors W_{r_i}, where $r_i = k_d + j$ and $j \in [1, n+1]$, we have to solve a set of linear inequality systems.

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 $\vec{e} \in E$ satisfies the following inequality:

$$\vec{e} \cdot \vec{w}_{r_i} < \vec{m} \cdot \vec{w}_{r_i} \tag{3.3}$$

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 $\vec{m}$:

$$\vec{f} \cdot \vec{w}_{r_i} \geq \vec{m} \cdot \vec{w}_{r_i} \tag{3.4}$$

Now we have a set of linear 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 a linear 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 linear inequality systems and W_{r_i} is essentially the union of the solutions of them.

The boundary theory of linear programming [32] shows that a linear inequality system is a *polyhedron*, which is the intersection of a finite set of half spaces. A polyhedron is either empty, unbounded, or a convex polytope. In our case, the polyhedrons are either empty or convex polytopes, because they are additionally bounded by the constraints $\sum w[i] = 1$ and $w[i] \in [0, 1]$. Therefore, the weighting vectors W_{r_i} is the union of a set of convex polytopes or an empty set if there are no solutions for all the inequality systems. □

As W_{r_i} is a set of convex polytopes with infinite points if it is not empty, the number of candidate answers in the answer space is also infinite. Therefore, we conclude that searching the optimal refined query for one missing object in an infinite answer space is unrealistic.[2] Moreover, the problem would not become easier when M has multiple missing objects.

[2] One exact solution that uses a quadratic programming (QP) solver [5] 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 3.2 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 at the worst case, so this exact solution is impractical.

3.2 Methodology

In this section, we present our method to answer *why-not* questions on top-k queries. 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, instead of considering the whole infinite answer space, we propose a special sampling-based algorithm that finds the best approximate answer.

3.2.1 Basic Idea

Let us start the discussion with an assumption that there is only one missing object $\vec{m}$. First, suppose we have a list of weighting vectors $S = [\vec{w}_o, \vec{w}_1, \vec{w}_2, \cdots, \vec{w}_s]$, where $\vec{w}_o$ is the weighting vector in the user's original query Q_o. 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 (e.g., [16], [8], [24]), 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. If $\vec{m}$ does not appear in the result of the first executed query, we report to the user that $\vec{m}$ does not exist in the database and the process terminates. Assuming $\vec{m}$ exists in the database, then 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.

In the following, we discuss where to get the list S of weighting vectors

(Section 3.2.2). Then, we discuss how large the list S should be (Section 3.2.3). Afterwards, we present the algorithm (Section 3.2.4). Finally, we present how to deal with multiple missing objects (Section 3.2.5).

3.2.2 Where to get weighting vectors?

In the basic idea of the algorithm, the size of S plays a crucial role in the algorithm efficiency and the solution quality. Having one more sample weighting in S, on the one hand, may increase the chance of having a better quality solution; on the other hand, that would definitely increase the number of progressive top-k operations by one and thus increase the running time. So, one of our objectives is to keep S as small as possible and at the same time put only high quality weightings (e.g., only those that may yield the optimal solution) into S.

Recall that if there are k_d objects dominate the missing object $\vec{m}$ and there are n objects incomparable with $\vec{m}$, the best and the worst ranking of $\vec{m}$ are $k_d + 1$ and $n + k_d + 1$, respectively. For these $n + 1$ possible rankings r_1, r_2, $\cdots$, r_{n+1} of $\vec{m}$, each ranking r_i is associated with a weighting vector set W_{r_i} such that for each weighting vector $\vec{w}_{r_i} \in W_{r_i}$, $\vec{m}$ ranks r_i-th in the corresponding refined query $Q'(r_i, \vec{w}_{r_i})$. So altogether there is a set $\mathcal{W}$ that contains $n + 1$ weighting vector sets, $\mathcal{W} = \{W_{r_1}, \cdots, W_{r_i}, \cdots, W_{r_{n+1}}\}$.

In the following, we are going to show that if the refined query $Q'_o(r_o, \vec{w}_o)$ is not the optimal answer, then the optimal refined query Q'_{opt} that minimizes Equation 3.2 in terms of Δk and Δw has a weighting vector $\vec{w}_{opt}$ on the *boundaries* of the weighting vector sets $\mathcal{W}$ (Theorem 3.1). Furthermore, refined queries with weightings on the boundaries of the weighting vector sets $\mathcal{W}$ would make

missing object $\vec{m}$ have a ranking no worse than other refined queries whose weightings not on the boundaries (Lemma 3.3). Therefore, in addition to the original weighting vector $\vec{w}_o$, the rest of the weighting vectors $\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_s$ in S should be sampled from the space formed by the boundaries of those $n+1$ weighting vector sets in $\mathcal{W}$.

Theorem 3.1 *If $Q'_o(r_o, \vec{w}_o)$ is not the optimal answer, then the optimal refined query Q'_{opt}, which minimizes Equation 3.2, has a weighting $\vec{w}_{opt}$ on the boundaries of the $n+1$ weighting vector sets in $\mathcal{W}$.*

Proof. According to Lemma 3.2, if a weighting vector set $W_{r_i} \in \mathcal{W}$ is not empty, then it is the union of a set of convex polytopes CP_{r_i}. So, the boundaries of W_{r_i} are essentially the union of the boundaries of each convex polytope in CP_{r_i}. Let $\mathcal{CP} = \bigcup CP_{r_i}$, Theorem 3.1 can be re-stated as follow: *if $Q'_o(r_o, \vec{w}_o)$ is not the optimal answer, then the optimal refined query Q'_{opt} has a weighting $\vec{w}_{opt}$ on the boundaries of $\mathcal{CP}$.*

Let $CP_{\setminus \vec{w}_o} \in \mathcal{CP}$ be the set of convex polytopes that do not contain $\vec{w}_o$. Further, let $CP_{\vec{w}_o} = \mathcal{CP} - CP_{\setminus \vec{w}_o}$ be the set of convex polytopes that contain $\vec{w}_o$. Since all the convex polytopes are disjoint (as no weighting can satisfy two different linear inequality systems described in Lemma 3.2 at the same time), $\vec{w}_o$ is in only one convex polytope. As such, $CP_{\vec{w}_o}$ essentially contains only one convex polytope.

Now, to prove the theorem, we need to prove the optimal refined query Q'_{opt} has a weighting $\vec{w}_{opt}$ on the boundaries of **((1))** $CP_{\setminus \vec{w}_o}$ or **((2))** $CP_{\vec{w}_o}$.

We now start with proving **((1))**. To do so, we first assume the optimal

refined query Q'_{opt} has a weighting $\vec{w}_{opt}$ NOT in $CP_{\vec{w}_o}$ or on its boundaries, and show that:

Lemma 3.3 *For all convex polytopes $cp \in \mathcal{CP}$, any refined query $Q'_b(r^b, \vec{w}_{r^b})$, whose weighting vector $\vec{w}_{r^b}$ on the boundaries of cp, has $\vec{m}$ ranks r^b-th under $\vec{w}_{r^b}$, which is not worse than rank r'-th (i.e., $r^b \leq r'$), where r' is the ranking of $\vec{m}$ under another refined query $Q'(r', \vec{w}_{r'})$, whose weighting vector $\vec{w}_{r'}$ not on the boundaries of cp (Lemma 3.3).*

Lemma 3.4 *For all convex polytopes $cp_{\setminus \vec{w}_o} \in CP_{\setminus \vec{w}_o}$, there exists a refined query $Q'_b(r^b, \vec{w}_{r^b})$, whose weighting vector $\vec{w}_{r^b}$ on the boundaries of $cp_{\setminus \vec{w}_o}$, has its $\Delta w < ||\vec{w}_{r'} - \vec{w}_o||_2$, where $\vec{w}_{r'}$ is the weighting vector of any other refined query $Q'(r', \vec{w}_{r'})$, whose weighting vector $\vec{w}_{r'}$ not on the boundaries of $cp_{\setminus \vec{w}_o}$ (Lemma 3.4).*

If both Lemma 3.3 and Lemma 3.4 hold, then **((1))** is true.

Proof of Lemma 3.3: We prove it by induction. First, recall that k_d is the number of objects that dominate the missing object $\vec{m}$. In the base case, we want to show that, when there is only one object $\vec{p}_1$ incomparable with $\vec{m}$ in the data space, the Lemma is true. When there is only one incomparable point $\vec{p}_1$, the whole weighting space is divided by the hyperplane H_1:$(\vec{p}_1 - \vec{m}) \cdot \vec{w} = 0$ into two convex polytopes $cp^<$ and $cp^>$; $cp^<$ is the convex polytope at the side $(\vec{p}_1 - \vec{m}) \cdot \vec{w} < 0$, $cp^>$ is the convex polytope at the side $(\vec{p}_1 - \vec{m}) \cdot \vec{w} > 0$, and hyperplane H_1 is the boundary of $cp^<$ and $cp^>$. We use $\mathcal{CP}_1$ to denote the whole set of convex polytopes at this moment. Now, consider a refined query $Q'(r', \vec{w}_{r'})$, whose $\vec{w}_{r'}$ in $cp^<$ but not on the boundary of $cp^<$, $\vec{m}$'s ranking $r' = k_d + 2$ because

$\vec{m}$ is dominated by $\vec{p}_1$ under $\vec{w}_{r'}$. Consider another refined query $Q'_b(r^b, \vec{w}_{r^b})$, whose $\vec{w}_{r^b}$ on the boundary H_1, $\vec{m}$'s ranking $r^b = k_d + 1$ because $\vec{m}$ and $\vec{p}_1$ have the same score. Finally, if the refined query $Q'(r', \vec{w}_{r'})$ has its weighting vector $\vec{w}_{r'}$ in $cp^>$ but not on the boundary of $cp^>$, $\vec{m}$ dominates $\vec{p}_1$ under $\vec{w}_{r'}$ and thus $\vec{m}$'s ranking remains as $r' = k_d + 1$. In the above, we can see that $r^b \leq r'$, thus the base case holds.

Assume Lemma 3.3 is still true when there are i objects incomparable with $\vec{m}$ in the data space and we use $\mathcal{CP}_i$ to denote the set of convex polytopes constructed by the corresponding i hyperplanes. Now, we want to show that the lemma is true when there are $i + 1$ incomparable objects in the data space.

When the $(i + 1)$-th incomparable object $\vec{p}_{i+1}$ is added, $\mathcal{CP}_i$ is divided into three sets of convex polytopes: $CP^<$, $CP^>$, and $CP^{\emptyset}$. $CP^<$ consists of any convex polytope $cp^<$ that is completely at the side of $(\vec{p}_{i+1} - \vec{m}) \cdot \vec{w} < 0$; $CP^>$ consists of any convex polytope $cp^>$ that is completely at the side of $(\vec{p}_{i+1} - \vec{m}) \cdot \vec{w} > 0$; and $CP^{\emptyset}$ consists of any convex polytope $cp^{\emptyset}$ that intersects the hyperplane H_{i+1}: $(\vec{p}_{i+1} - \vec{m}) \cdot \vec{w} = 0$. We want to show that the lemma is true for all the three sets of convex polytopes above.

For any $cp^< \in CP^<$, the refined query $Q'(r', \vec{w}_{r'})$, whose $\vec{w}_{r'}$ in $cp^<$ but not on the boundary of $cp^<$, the addition of $\vec{p}_{i+1}$ makes $\vec{m}$'s ranking r' increments by one, i.e., $r' \leftarrow r' + 1$, because $\vec{p}_{i+1}$ has a better score when the weighting vectors are at the side of $(\vec{p}_{i+1} - \vec{m}) \cdot \vec{w} < 0$. For the refined query $Q'_b(r^b, \vec{w}_{r^b})$, whose weighting vector $\vec{w}_{r^b}$ on the boundaries of $cp^<$, the addition of $\vec{p}_{i+1}$ makes $\vec{m}$'s ranking r^b increments by one, i.e., $r^b \leftarrow r^b + 1$. In this case, $r^b \leq r'$, the lemma still holds after $\vec{p}_{i+1}$ is added.

For any $cp^{>} \in CP^{>}$, the refined query $Q'(r', \vec{w}_{r'})$, whose $\vec{w}_{r'}$ in $cp^{>}$ but not on the boundary of $cp^{>}$, the addition of $\vec{p}_{i+1}$ does not change $\vec{m}$'s ranking r', because $\vec{p}_{i+1}$ has score worse than $\vec{m}$ when the weighting vectors at the side of $(\vec{p}_{i+1} - \vec{m}) \cdot \vec{w} > 0$. For the refined query $Q'_b(r^b, \vec{w}_{r^b})$, whose weighting vector $\vec{w}_{r^b}$ on the boundaries of $cp^{>}$, the addition of $\vec{p}_{i+1}$ also does not change $\vec{m}$'s ranking r^b. So, in this case, $r^b \leq r'$, the lemma still holds after the p_{i+1} is added.

For any $cp^{\emptyset} \in CP^{\emptyset}$, since it intersects hyperplane H_{i+1}, $cp^{\emptyset}$ is divided into two new convex polytopes $cp^{\emptyset}_{<}$ and $cp^{\emptyset}_{>}$, with H_{i+1} as their boundaries.

The proof for the case of $cp^{\emptyset}_{<}$ is similar to the case of any $cp^{<} \in CP^{<}$, with both r^b and r' get increased by one, so $r^b \leq r'$ is still true. For the case of $cp^{\emptyset}_{>}$, we also have $r^b \leq r'$ like the case of any $cp^{>} \in CP^{>}$. For the refined query $Q'_b(r^b, \vec{w}_{r^b})$, whose weighting vector $\vec{w}_{r^b}$ on the hyperplane H_{i+1}, as the added object $\vec{p}_{i+1}$ has the same score as $\vec{m}$ at this moment, $\vec{m}$'s ranking remains the same. And for those $Q'(r', \vec{w}_{r'})$, whose $w_{r'}$ not on H_{i+1}, may either keep $\vec{m}$'s ranking or increase its ranking by one just like the discussion above. Therefore, we can assert that $r^b \leq r'$, is true in all cases.

Proof of Lemma 3.4: Let $\vec{w}_{r*}$ be the weighting vector that is closest to $\vec{w}_o$ (i.e., the one with the optimal Δw), we prove the lemma by showing that, among all the weighting vectors in $cp_{\setminus \vec{w}_o}$, $\vec{w}_{r*}$ must be on the boundaries of $cp_{\setminus \vec{w}_o}$.

Assume $\vec{w}_{r^*}$ is in $cp_{\setminus \vec{w}_o}$ but not on its boundaries. So, $\vec{w}_{r^*}$ is an interior point of $cp_{\setminus \vec{w}_o}$ and there exists an open ball $B \subset cp_{\setminus \vec{w}_o}$ centered at $\vec{w}_{r^*}$ [5]. Since B is convex, we can find two points $\vec{w}_a$, $\vec{w}_b$ in B such that $\vec{w}_{r^*} = \frac{\vec{w}_a + \vec{w}_b}{2}$. As $\vec{w}_{r^*}$ is closest to $\vec{w}_o$, we know that $(\vec{w}_a - \vec{w}_{r^*}) \cdot (\vec{w}_o - \vec{w}_{r^*}) \leq 0$ (see Lemma 3.5). Because $\vec{w}_a = 2\vec{w}_{r^*} - \vec{w}_b$, we can substitute $\vec{w}_a$ into the inequality above and get

$(\vec{w}_{r^*} - \vec{w}_b) \cdot (\vec{w}_o - \vec{w}_{r^*}) \leq 0$. Now we have $(\vec{w}_b - \vec{w}_{r^*}) \cdot (\vec{w}_o - \vec{w}_{r^*}) \geq 0$, which contradicts Lemma 3.5 (below). Therefore, $\vec{w}_{r^*}$ must be on the boundaries of $cp_{\setminus \vec{w}_o}$.

Lemma 3.5 *Let $C \subset \mathbb{R}^n$ be a non-empty closed convex set. Then, for every point $\vec{x} \notin C$, $\vec{z}^* \in C$ is a closest point to $\vec{x}$ iif $(\vec{z} - \vec{z^*}) \cdot (\vec{x} - \vec{z^*}) \leq 0$ for all $\vec{z} \in C$. [5]*

Now, let us prove **((2))**, i.e., the optimal refined query Q'_{opt} has a weighting $\vec{w}_{opt}$ on the boundaries of $CP_{\vec{w}_o}$. To do so, we assume the optimal refined query Q'_{opt} has a weighting $\vec{w}_{opt}$ NOT in $CP_{\setminus \vec{w}_o}$ or its boundaries. We first show that, for any refined query $Q'(r', \vec{w}_{r'})$, whose weighting vector $\vec{w}_{r'}$ in $CP_{\vec{w}_o}$ but not on its boundaries, is dominated by $Q'_o(r_o, \vec{w}_o)$.

Given that $\vec{w}_{r'}$ is in $CP_{\vec{w}_o}$ but not on its boundaries, and also given that $\vec{w}_o$ is in $CP_{\vec{w}_o}$ we consider two cases, in which $\vec{w}_o$ is (i) *not on* and (ii) *on* $CP_{\vec{w}_o}$'s boundaries, respectively. In case (i), since both $\vec{w}_{r'}$ and $\vec{w}_o$ are in $CP_{\vec{w}_o}$ but not on its boundaries, they satisfy the same inequality system and thus we have $r' = r_o$. In case (ii), in which $\vec{w}_o$ is on $CP_{\vec{w}_o}$'s boundaries, we can apply Lemma 3.3 and thus we have $r_o \leq r'$. Combining two cases together we have $r_o \leq r'$ always holds. Note that when $\vec{w}_{r'} \neq \vec{w}_o$, $\Delta w'_r > \Delta w_o$. Hence, among all refined queries $Q'(r', \vec{w}_{r'})$, whose $\vec{w}_{r'}$ in $CP_{\vec{w}_o}$, only those with $\vec{w}_{r'}$ on the boundaries of $CP_{\vec{w}_o}$ have chances not to be dominated by $Q'_o(r_o, \vec{w}_o)$.

Finally, as $Q'_o(r_o, \vec{w}_o)$ is not the optimal answer (the given condition in Theorem 3.1), so we know that if the optimal weighting $\vec{w}_{opt}$ is not on the boundaries of $CP_{\setminus \vec{w}_o}$ (from **((1))**), then it is on the boundaries of $CP_{\vec{w}_o}$ (from **((2))**). Thus

Theorem 3.1 is proven. □

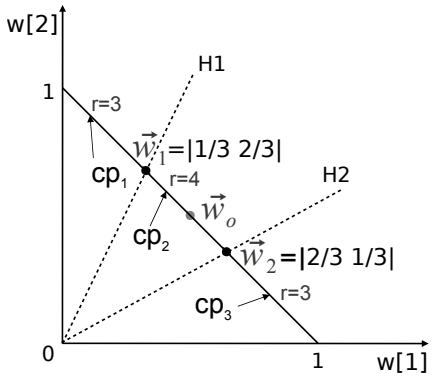


Figure 3.3. Convex polytopes for $\vec{m}$ shown under the weighting space

Here is the high level explanation of Lemma 3.4 and Lemma 3.5. According to the analysis in Section 3.1.2, the weighting space is divided into a set of polytopes W_{r_i} by a set of inequality systems, where $\vec{m}$ ranks r_i if we draw a weighting in W_{r_i}. Since all $\vec{w} \in W_{r_i}$ provide us the same ranking (i.e., same Δk), we only need to consider those with smaller Δw. Based on the *maximum principle* in convex optimization [5], those $\vec{w}$ with smaller Δw must be on the boundaries of these W_{r_i}.

Consider the dataset in Figure 3.2 as an example. Point $\vec{m}$, who is missing in the top-3 results of the original query Q_o, has two incomparable points $\vec{p}_2$ and $\vec{p}_3$. To find out the set of weighting vectors W_{r_i} that makes $\vec{m}$ ranks third, i.e., $r_i = 3$, we can solve the following inequality systems separately:

$$\begin{cases} \vec{p}_2 \cdot \vec{w}_{r_i} < \vec{m} \cdot \vec{w}_{r_i} \\ \vec{p}_3 \cdot \vec{w}_{r_i} \geq \vec{m} \cdot \vec{w}_{r_i} \end{cases} \tag{3.5}$$

$$\begin{cases} \vec{p}_2 \cdot \vec{w}_{r_i} \geq \vec{m} \cdot \vec{w}_{r_i} \\ \vec{p}_3 \cdot \vec{w}_{r_i} < \vec{m} \cdot \vec{w}_{r_i} \end{cases} \tag{3.6}$$

The union of the results above is $W_{r_i=3}$. In the systems above, we have two hyperplanes H_1: $(\vec{p}_3 - \vec{m}) \cdot \vec{w}_{r_i} = 0$ and H_2: $(\vec{p}_2 - \vec{m}) \cdot \vec{w}_{r_i} = 0$ that divide the weighting space like Figure 3.3. The two hyperplanes intersect the weighting constraint planes $\sum w_{r_i}[i] = 1$ and $w_{r_i}[i] \in [0, 1]$ and results in three convex polytopes cp_1, cp_2, and cp_3 (in 2-d case the polytopes are line segments). The union of cp_1 and cp_3 is the corresponding weighting vector set $W_{r_i=3}$; and cp_2 is the weighting vector set $W_{r_i=4}$. In this 2-D example, the intersections $\vec{w}_1 = |1/3\ 2/3|$ and $\vec{w}_2 = |2/3\ 1/3|$ between the two hyperplanes and the weighting constraint planes are the boundaries of the polytopes. Note that by some back-of-envelop calculation, we can derive that $\vec{m}$ ranks third under weightings $\vec{w}_1$ and $\vec{w}_2$. This aligns with Lemma 3.3, which states that the ranking of $\vec{m}$ on the boundaries (rank third) is not worse than its ranking not on the boundaries (rank fourth in cp_2 and third in cp_1 and cp_3).

3.2.3 How large the list of weighting vectors should be?

Having known that the list of weighting vectors S should be obtained from the boundaries of the weighting sets in $\mathcal{W}$, the next question is, given that there are still an infinite number of points (weightings) on the boundaries of the weighting sets in $\mathcal{W}$, *how many sample weightings from the boundaries should we put into S in order to obtain a good approximation answer?*

Recall that more sample weightings in S will increase the number of progres-

sive 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:

$$\begin{aligned} & 1-(1-T\%)^s \geq Pr \\ \Rightarrow \quad & (1-T\%)^s \leq 1-Pr \\ \Rightarrow \quad & log_{(1-T\%)}(1-T\%)^s \geq log_{(1-T\%)}(1-Pr) \\ \Rightarrow \quad & s \geq log_{(1-T\%)}(1-Pr) \end{aligned} \tag{3.7}$$

Equation 3.7 is general. In our algorithm, we use it based on a smaller sample space that contains high quality weightings. The sample size s is independent of the data size but controlled by two parameters: T% and Pr. Following our usual practice of not improving usability (i.e., why not queries) by increasing users' burden (e.g., specifying parameter values for λ_k, T%, and Pr), we make T%, and Pr as system parameters [3] and let users to override their values only when necessary.

3.2.4 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 3.2.1 for better ef-

[3]The default values are picked through a set of experiments, which guarantee that we can obtain good answers in reasonable time.

ficiency:

[PHASE-1] It first samples s weightings $\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_s$ from the boundaries of the weighting vector sets $\mathcal{W}$ and add them into S, which initially contains $\vec{w}_o$.

[PHASE-2] Next, for *some* weighting vectors $\vec{w}_i \in S$, it executes a progressive top-k query using $\vec{w}_i$ as the weighting until a stopping condition is met. Let us denote that operation as $r_i = \text{TOPK}(\vec{w}_i, \text{STOPPING-CONDITION})$. In the basic idea mentioned in Section 3.2.1, we have to execute $s+1$ progressive top-k queries for all $s+1$ weightings in S. In this section, we present a technique to skip many of those progressive top-k operations so as to improve the efficiency (Section 3.2.4-Technique (ii)). 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 (Section 3.2.4-Technique (i)). 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 $(k', \vec{w}')$ combination with the least penalty is formulated as a refined query $Q'(k', \vec{w}')$ and returned to the user as the why-not answer.

We first provide the details of PHASE-1. First, $\vec{w}_o$, the weighting vector in

the user's original query Q_o, is added to S. Next, we use the method in [33] to locate the set I of objects incomparable with $\vec{m}$. After that, we randomly pick a point $\vec{p_i}$ from I and use Gaussian-Jordan method [10] to efficiently find the intersection between the hyperplane $(\vec{p_i} - \vec{m}) \cdot \vec{w} = 0$ and the constraint plane $\sum w[i] = 1$. Then, we randomly pick a point (weighting) from the intersection. If all components of this weighting are non-negative ($w[i] \geq 0$), we add it to S. The process repeats until s weightings have been collected. As we show in the experiments, this phase can be done very efficiently because finding I for one (or a few) missing object(s) and solving plane intersections using Gaussian-Jordon method incur almost negligible costs.

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. Denoting that operation as $r_1 = \text{TOPK}(\vec{w}_1, \text{UNTIL-SEE-}\vec{m})$. 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 that a user specifies a top-2 query $Q_o(k_o = 2, \vec{w}_o)$ and a why-not question about missing object $\vec{m}$ is posed. Assume that the list of weightings S is $[\vec{w}_o, \vec{w}_1, \vec{w}_2, \vec{w}_3]$. Furthermore, assume that $\text{TOPK}(\vec{w}_o, \text{UNTIL-SEE-}\vec{m})$ is firstly executed 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, $Pen_{Q'_o}$, could be calculated using Equation 3.2. Remember that we want to find the refined query with the least penalty

Pen_{min}. So, at this moment, we set a penalty variable $Pen_{min} = Pen_{Q'_o}$.

According to our basic idea, 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.

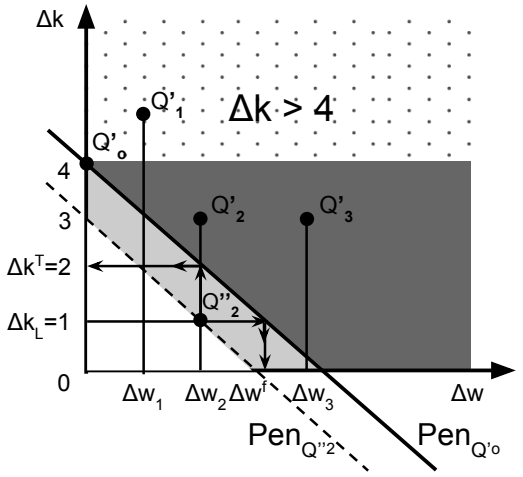


Figure 3.4. Example of answer space

Figure 3.4 illustrates the answer space of the example. The idea above essentially means that all other progressive top-k executions with $\vec{m}$ 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 any more.

While useful, we can actually be even more aggressive in many cases. Consider another candidate refined query, say, Q'_2, in Figure 3.4. Assume that

$r_2 = \text{TOPK}(\vec{w}_2, \text{UNTIL-SEE-}\vec{m}) = 5$ (i.e., $\Delta k_2 = 5 - 2 = 3$), which is not covered by the above technique (since $\Delta k_2 \not< 4$). However, Q'_2 can also stop early, as follows. In Figure 3.4, we show the normalized penalty Equation 3.2 as a slope $Pen_{min} = Pen_{Q'_o}$ that passes through the best refined query so far (currently Q'_o). All refined queries lie on the slope have the same penalty value as Pen_{min}. In addition, all refined queries that lie above the slope actually have penalty larger than Pen_{min}, and thus 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:

$$\begin{aligned} r^T &= \Delta k^T + k_o, \text{ where} \\ \Delta k^T &= \lfloor (Pen_{min} - \lambda_w \frac{\Delta w}{\sqrt{1 + \sum w_o[i]^2}}) \frac{r_o - k_o}{\lambda_k} \rfloor \end{aligned} \tag{3.8}$$

Equation 3.8 is a rearrangement of Equation 3.2 (with $Penalty = Pen_{min}$) and with the original top-k value k_o added.

Back to our example in Figure 3.4, 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 Pen_{min} (currently $Pen_{min} = Pen_{Q'_o}$) to obtain the corresponding Δk^T value, which is 2 in Figure 3.4. 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 Pen_{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 the first progressive top-k operation in which TOPK($\vec{w}_o$, 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 3.4). Recall that by projecting Δw_2 on to the slope of Pen_{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 pruning power of this technique increases when 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, Pen_{min} now is updated as $Pen_{Q''_2}$ and the slope Pen_{min} should be updated to pass through Q''_2 now (the dotted slope in Figure 3.4). Because Pen_{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 and earlier when the algorithm proceeds. With the same token, we also envision that the pruning power of this technique is stronger when we have a large λ_w (or small λ_k) because they make Δk^T decreases at a faster rate (see Equation 3.8).

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 entirely skipped, so that the overall running time can be further reduced.

The first part of the technique is based on the observation that similar weighting vectors may lead to similar top-k results [36]. Therefore, if a weighting $\vec{w_j}$ is similar to a weighting $\vec{w_i}$ and if operation TOPK($\vec{w_i}$, STOPPING-CONDITION) for $\vec{w_i}$ has already been executed, then the query result R_i of TOPK($\vec{w_i}$, STOPPING-CONDITION) 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 TOPK($\vec{w_j}$, STOPPING-CONDITION) 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 TOPK($\vec{w_o}$, 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 3.4, projecting $\vec{w_1}$ onto slope $Pen_{Q'_o}$ 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 $\vec{m}$, 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 TOPK($\vec{w_1}$, STOPPING-CONDITION) operation even without starting it.

The above caching technique is shown to be the most effective between similar weighting vectors [36]. 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 $\vec{m}$ (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 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 to $\max(k_d + 1 - k_o, 0)$ (By definition in Section 3.1, $\Delta k \geq 0$). So, this time, we project Δk_L onto slope Pen_{min} in order to determine the corresponding *maximum feasible* Δw value. Naming 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 $\vec{w}_i \in S$ has $||\vec{w}_i - \vec{w}_o||_2 > \Delta w^f$, TOPK($\vec{w}_i$, STOPPING-CONDITION) and all subsequent progressive top-k operations TOPK($\vec{w}_{i+1}$, STOPPING-CONDITION), ... , TOPK($\vec{w}_s$, STOPPING-CONDITION) could be skipped.

Reusing Figure 3.4 as an example and 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 $Pen_{Q'_o}$, we could determine the corresponding Δw^f value. So, when the algorithm finishes executing 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 when the algorithm proceeds. For instance, in Figure 3.4, if Q_2'' has been executed, slope Pen_{min} is changed from slope $Pen_{Q_o'}$ to slope $Pen_{Q_2''}$. Projecting Δk_L onto the new Pen_{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 idea is presented in Algorithm 3.1. It is self-explanatory and mainly summarizes what we have discussed above, so we do not give it a walkthrough here.

3.2.5 Multiple Missing Objects

To deal with multiple missing objects in a why-not question, we have to modify our algorithm a little bit. 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 $\vec{m}_i$ 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 next modification to the algorithm is related to PHASE-1 — finding good weightings and put them into S. First, the set I should now consist of incomparable points of all objects in M'. Second, we should randomly select a hyperplane $(\vec{p_i} - \vec{m_i}) \cdot \vec{w} = 0$, where $\vec{p_i}$ is a point in I. After that, as usual, we sample a point on the intersection of the hyperplanes plus the constraint plane $\sum w[i] = 1$ $(0 \leq w[i] \leq 1)$. That way, the whole method still obeys Theorem 3.1.

The modification related to Technique (i) is as follows. For the condition

Algorithm 3.1 Answering Why-not Question on a Top-K Query

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 $Q'(k', \vec{w}')$

Set list of weighting $S = [\vec{w}_o]$;
Result of a top-k query R_o;
Rank of missing object r_o;
$(R_o, r_o) \leftarrow$ TOPK$(\vec{w}_o,$ UNTIL-SEE-$\vec{m})$
if $r_o = \emptyset$ **then**
  return "$\vec{m}$ is not in the D"
end if
Phase 1:
Use [33] to determine the number of points k_d that dominate $\vec{m}$ and the set of points I incomparable with $\vec{m}$;
$\Delta k_L = \max(k_d + 1 - k_o, 0)$;
Determine s from $T\%$ and Pr using Equation 3.7;
Sample s weightings from the hyperplanes boundaries constructed by I and $\vec{m}$ and add them to S;
Sort S according to their Δw_i values;
Phase 2:
$R \leftarrow (R_o, \vec{w}_o)$; //Cache the results
$Pen_{min} \leftarrow Penalty(r_o, \vec{w}_o)$;
$\Delta k_L \leftarrow max(k_d + 1 - k_o, 0)$; //Calculate the lower bound ranking of m
$\Delta w^f = (Pen_{min} - \lambda_k \frac{\Delta k_L}{r_o - k_o}) \frac{\sqrt{1+\sum w_o[i]^2}}{\lambda_w}$; //Project Δk_L to determine early termination point Δw^f
for all $\vec{w}_i \in S$ **do**
  if $\Delta w_i > \Delta w^f$ **then**
    break; //Technique (ii) — early algorithm termination
  end if
  $\Delta k^T \leftarrow \lfloor (Pen_{min} - \lambda_w \frac{\Delta w_i}{\sqrt{1+\sum w_o[i]^2}}) \frac{r_o - k_o}{\lambda_k} \rfloor$;
  $r^T \leftarrow k^T + k_o$;
  if there exist r^T objects in some $R_i \in R$ having scores better $\vec{m}$ under $\vec{w}_i$ **then**
    continue; //Technique (ii) — use cached result to skip a progressive top-k
  end if
  $(R_i, r_i) \leftarrow$ TOPK$(\vec{w}_i,$ UNTIL-SEE-$\vec{m}$ OR UNTIL-RANK-$r^T)$; //Technique (i) — stopping a progressive top-k early
  $Pen_i \leftarrow Penalty(r_i, \vec{w}_i)$;
  $R \leftarrow R \cup (R_i, \vec{w}_i)$;
  if $Pen_i < Pen_{min}$ **then**
    $Pen_{min} \leftarrow Pen_i$;
    $\Delta w^f = (Pen_{min} - \lambda_k \frac{\Delta k_L}{r_o - k_o}) \frac{\sqrt{1+\sum w_o[i]^2}}{\lambda_w}$;
  end if
end for
Phase 3:
return the best refined query $Q'(k', \vec{w}')$ whose penalty=Pen_{min};

UNTIL-SEE-$\vec{m}$, it should now be UNTIL-SEE-ALL-OBJECTS-IN-M'. For example, r_o in Algorithm 3.1 Line 7 should now refer to the ranking of the missing object with the highest score. 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) is 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 $\max(|DOM_1 \cup DOM_2 \cup \cdots \cup DOM_n \cup M'| - k_o, 0)$.

3.3 Experiments

We evaluate our proposed solution using both synthetic and real data. By default, we set the system parameters $T\%$ and Pr as 0.2% and 0.8, respectively (resulting in a sample size of 800 weightings). 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. We adopt [24] as our progressive top-k algorithm.

3.3.1 Case Study

We use the NBA data set in the case study. 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).

For comparison, we also implemented a version of our algorithm in which weightings are randomly sampled from the whole weighting space. We refer to that version as WWS. In the following, we present several interesting cases:

Case 1 (Finding the top-3 centers in NBA history). The first case was to find the top-3 centers in the NBA history. Therefore, we issued a top-3 query Q_1 with equal weighting (0.2) on five attributes PTS, REB, BLK, FG, and FT. The initial 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	1	0.58	0.52

Because we were curious why Yao Ming was not in the result, we issued a why-not question $\{\{\text{Yao Ming}\}, Q_1\}$ using the "Prefer modify weighting" option, since we wanted to see Yao in top-3. In 156ms, our algorithm returned a refined query Q'_1 with $k'_1 = 3$ and $\vec{w}'_1 = |0.0243\ 0.0024\ 0.0283\ 0.0675\ 0.8775|$. The refined query essentially indicated that we should have put more weights on a center's free-throw (FT) ability if we wish to see Yao in the top-3 result. The corresponding result of Q'_1 was:

Rank	Player	PTS	REB	BLK	FG	FT
1	Abdul-jabbar	25	11	2	0.55	0.72
2	Hakeem Olajuwon	22	11	3	0.51	0.71
3	Yao Ming	19	9	2	0.52	0.83

The penalty value of Q'_1 was 0.069. As a comparison, WWS returned another refined query Q'^{WWS}_1, using 154ms. However, Q'^{WWS}_1 was a top-7 query that

uses another set of weighting (Yao ranked 7-th). The penalty of $Q_1'^{WWS}$ was 0.28, which was four times worse than Q_1'.

Case 2 (Finding the top-3 guards in NBA history). The second case was to find the top-3 guards in the NBA history. Therefore, 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 were surprised why Kobe Bryant was not in the result. So, we posed a why-not question {{Kobe Bryant}, Q_2} using the "Prefer modify weighting" option, since we wanted to see Kobe Bryant in top-3. In 163ms, our algorithm returned a refined query Q_2' with $k_2' = 3$ and $\vec{w}_2' = |0.0129\ \ 0.0005\ \ 0.0416\ \ 0.2316\ \ 0.3769\ \ 0.3364|$. The corresponding result of Q_2' was:

Rank	Player	PTS	AST	STL	FG	FT	3PT
1	Michael Jordan	30	5	2	0.49	0.83	0.32
2	Pete Maravich	24	5	1	0.44	0.82	0.66
3	Kobe Bryant	25	5	2	0.45	0.83	0.34

The penalty of Q_2' was 0.035. As a comparison, WWS returned a refined query $Q_2'^{WWS}$, in 161ms. However, $Q_2'^{WWS}$ was a top-4 query (Kobe Bryant ranked 4-th), which conflicted with our "Prefer modify weighting" option. Thus, $Q_2'^{WWS}$'s penalty was 0.2, which was more than five times worse than Q_2'.

Case 3 (Finding the top-3 players in NBA history). The third case was to find the top-3 players in the NBA history. Therefore, we issued a top-3

query Q_3 with equal weighting ($\frac{1}{8}$) on all eight numeric attributes. 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

Amazingly, Michael Jordan was missing in the result. To understand why, we issued a *why-not* question {{Michael Jordan}, Q_3}, using the "Prefer modify k" option, because we insisted that Michael Jordan should have a top ranking without twisting the weightings much. Using 150ms, our algorithm returned a refined query Q'_3 with $k'_3 = 5$ and $\vec{w}'_3 = \vec{w}_o$, with the following result:

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
4	Bob Pettit	26	16	3	0	0	0.43	0.76	0
5	Michael Jordan	30	6	5	2	1	0.49	0.83	0.32

The refined query Q'_3 essentially means that our initial weightings were reasonable but we should have looked for the top-5 players instead. In this case, both versions of our algorithms came up with the same refined query in 150ms.

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 the NBA history. Therefore, we issued another why-not query {{Michael Jordan, Shaquille O'neal}, Q_3}, using the "Never mind" option. In 166ms, our algorithm returned a refined query Q''_3 with $k' = 5$ and $\vec{w}'_3 = |0.138\ 0.0847\ 0.0639\ 0.1066\ 0.2481\ 0.1143\ 0.1231\ 0.1212|$. The corresponding result of Q''_3 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

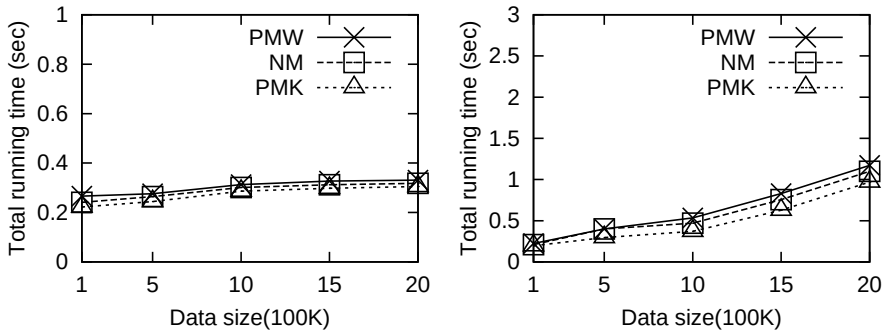
The penalty of Q_3'' was 0.2. WWS returned another refined query $Q_3''^{WWS}$, in 164ms. However, $Q_3''^{WWS}$ was a top-5 query with penalty 0.27, which was higher than Q_3''.

3.3.2 Performance

We next turn our focus to the performance of our algorithm. We present experimental results based on three types of synthetic data: uniform (UN), correlated (CO) and anti-correlated (AC). Since, the experiment results between UN and CO are very similar, we only present the results of UN and AC here. Table 3.2 shows the parameters we varied in the experiments. The default values are in bold faces. The default top-k query Q_o 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 $\vec{w}_o$.

Table 3.2. Parameters setting

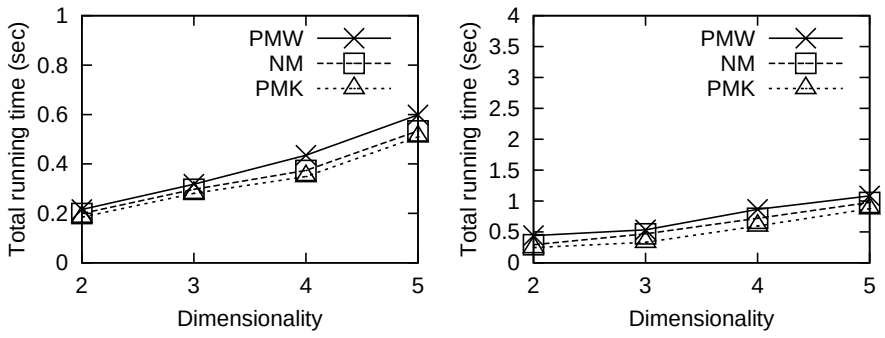
Parameter	Ranges
Data size	100K, 500K, **1M**, 1.5M, 2M
Dimension	2, **3**, 4, 5
k_o	5, **10**, 50, 100
Actual ranking of $\vec{m}$ under Q_o	11, **101**, 501 ,1001
$T\%$	0.3%, 0.25%, **0.2%**, 0.15%, 0.1%
Pr	0.5, 0.6, 0.7, **0.8**, 0.9
$\|M\|$	**1**, 2, 3, 4, 5



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

Figure 3.5. Varying data size

Varying Data Size. Figure 3.5 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. The running time scales linearly, but at a faster rate, on AC data because of the general fact that progressive top-k operations on anti-correlated data takes a longer time to finish [24].

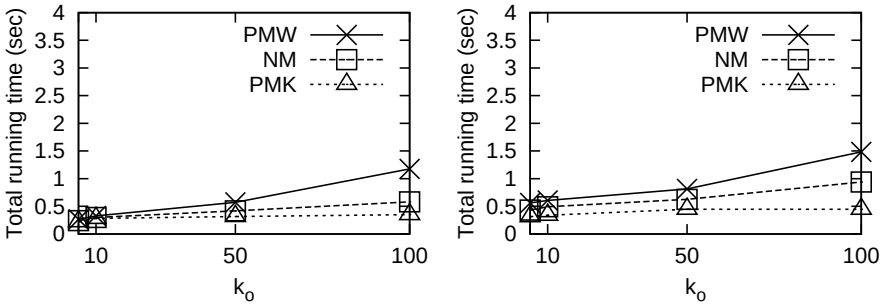


(a) Uniform Data (b) Anti-correlated Data

Figure 3.6. Varying query dimension

Varying Query Dimension. Figure 3.6 shows the running time of our

algorithm using top-k queries in different number of query dimensions. In general, answering why-not questions for top-k queries in a higher query dimension needs more time because the execution time of a progressive top-k operation increases if a top-k query involves more attributes. From the figure, we see that our algorithm scales well with the number of dimensions.

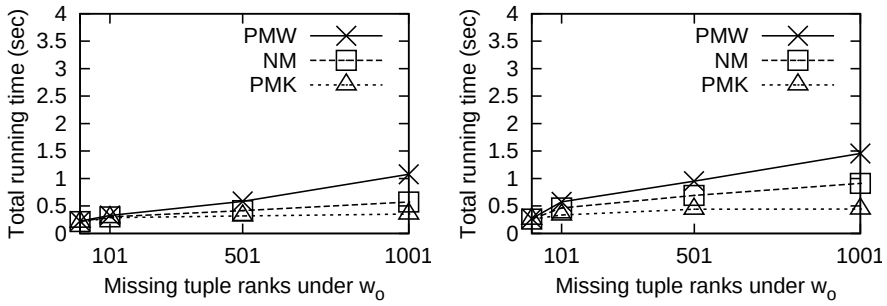


(a) Uniform Data (b) Anti-correlated Data

Figure 3.7. Varying k_o

Varying k_o. Figure 3.7 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 51-th 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 501-th 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 3.7 shows that our algorithm scales well with k_o. The running time of our algorithm increases very little under the PMK option. Recall that in Section 3.2.4 we mentioned that the effectiveness of our pruning techniques is more pronounced when the PMK option is used. In this experiment, when we

scaled up k_o to a large value, the algorithm running time became higher. As such, the stronger pruning effectiveness of the PMK option became more obvious in the experimental result.

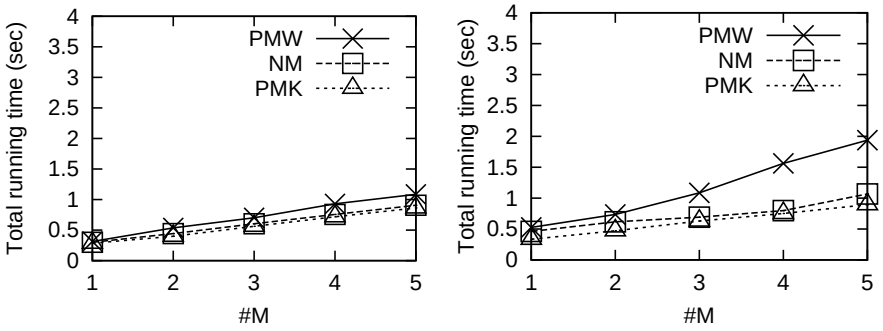


(a) Uniform Data (b) Anti-correlated Data

Figure 3.8. Varying the ranking of the missing object

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 experiment, the default top-10 query is used. We asked four individual why-not questions about why the object that ranked 11-th, 101-th, 501-th, and 1001-th, respectively, is missing in the result. Figure 3.8 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 see it in the result and thus the overall running time must increase. Again, because our pruning techniques are especially effective when the PMK option is used, the running time of our algorithm increases very little under that option.

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

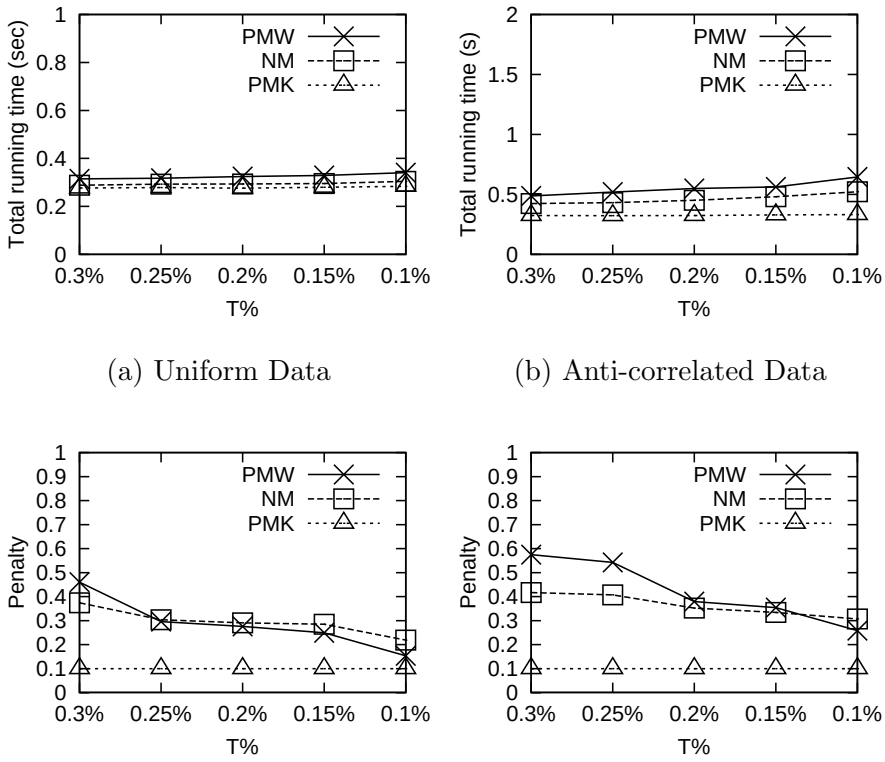


(a) Uniform Data (b) Anti-correlated Data

Figure 3.9. Varying $|M|$

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 3.9 shows that our algorithm scales linearly with respect to different sizes of M.

Varying $T\%$. We 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 3.10 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 0.3% ($|S| = 536$) to accepting refined queries that are within the best 0.1% ($|S| = 1609$). From Figures 3.10(a) and 3.10(b), we can see that, with our effective pruning techniques, the running time of our algorithm do not increase much when the guarantee is more stringent. However, from Figures 3.10(c) and 3.10(d), we can see that the solution quality of the algorithm does improve when T decreases, under options PMW and NM. The



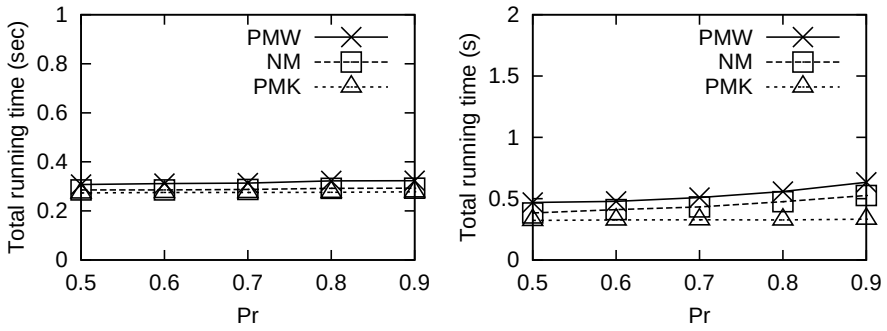
(a) Uniform Data (b) Anti-correlated Data

(c) Uniform Data (d) Anti-correlated Data

Figure 3.10. Varying T%

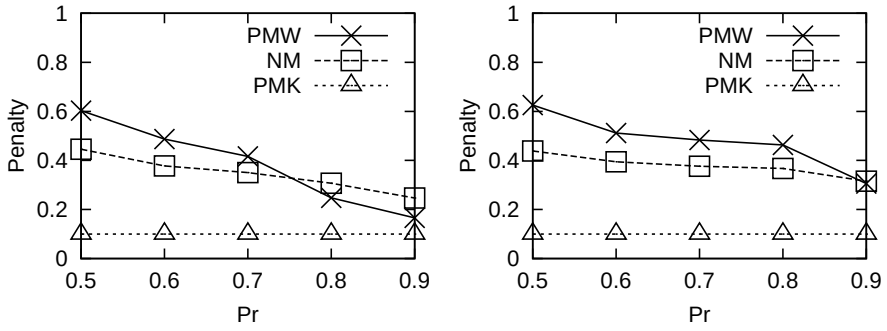
solution quality of our algorithm under the option PMK does not change when T increases because the refined query $Q'_o(r_o, \vec{w_o})$ has a very small Δk under that option. Option PMK prefers *not* to change the weighting. So, the query $Q'_o(r_o, \vec{w_o})$, which has its $\Delta w = 0$, is the best refined query over all different values of T. This explains why the solution quality remains constant under that option.

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%



(a) Uniform Data

(b) Anti-correlated Data



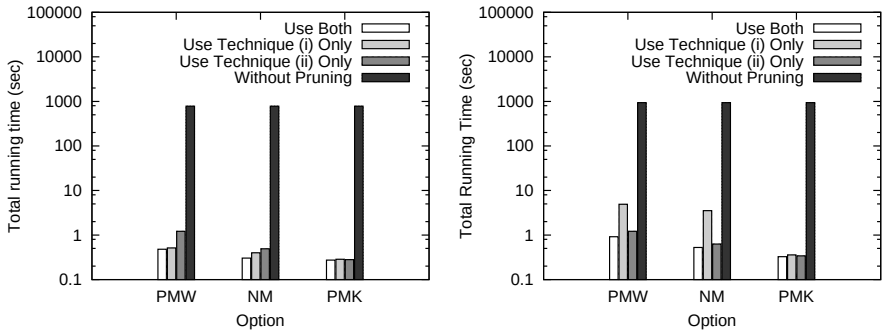
(c) Uniform Data

(d) Anti-correlated Data

Figure 3.11. Varying Pr

above. That is because both parameters are designed for controlling the quality of the approximate solutions. In Figure 3.11, we can see that when we vary Pr from 0.5 ($|S| = 347$) to 0.9 ($|S| = 1151$), the running times remain roughly constant. However, the solution quality does improve a lot, except under the option PMK because of the same reason we described above.

Effectiveness of Pruning Techniques. Finally, we investigate the effectiveness of the two pruning techniques we used in our algorithm. Figure 3.12 shows the performance of our algorithm using only Technique (i), only Technique



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

Figure 3.12. Pruning Effectiveness

(ii), both, and none, under the default setting. The pruning effectiveness of both techniques are also very promising. Without using any pruning technique, the algorithm requires a running time of about 1000 seconds. A progressive top-k operation is more costly in anti-correlated data than in uniform data. Therefore, the effectiveness of Technique (ii), which prunes the entire progressive top-k operations, is relatively stronger than Technique (i) in AC data than in UN data. Using the same token, this explains why Technique (i) is relatively more effective than Technique (ii) in the UN data.

3.4 Chapter Summary

In this chapter, we have studied the problem of answering why-not questions on top-k queries. Our target is to give an explanation to a user who is wondering why her expected answers are missing in the query result. By returning the user a refined query with (approximate) minimal changes to the k value and her weightings, the user could get not only her desired query, but also learn what

was/were wrong with her initial query. Case studies and experimental results show that our approach returns high quality explanations to users efficiently.

Chapter 4

Why-Not Top-K SQL Question

In Chapter 3, we discussed the answering of why-not questions on top-k queries in the absence of other SQL constructs such as selection, projection, join, and aggregation. In this chapter, we extend our algorithm to support why-not top-k questions in the context of SQL by adopting the query-refinement approach as our explanation model. In [35], the query-refinement approach has been adopted on why-not SPJA questions. 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. In this chapter, we adopt the above "similar" and "precise" metrics.

Considering additional SQL constructs on why-not top-k questions could complicate the solution space. As an example, consider table U in Figure 4.1(a) and the following top-3 SQL query:

ID	A	B
P1	240	60
P2	235	60
P3	340	70
P4	100	70
P5	**140**	**100**
P6	150	50

(a) An example table U

ID	0.5*A+0.5*B
P3	205
P1	150
P2	147.5
P5	**120**
P6	100
P4	85

(b) Ranking under original weightings

Figure 4.1. Motivation Example

Q_o:

```
SELECT U.ID
FROM U
WHERE U.A ≥ 205
ORDER BY 0.5∗U.A + 0.5∗U.B
LIMIT 3
```

Figure 4.1(b) shows the ranking scores of all tuples in U and the top-3 results are {P3, P1, P2}. Assuming that we are interested in asking *why* P5 is *not* in the top-3, we see that using SPJA query modification techniques in [35] to modify only the SPJ construct (e.g., modifying `WHERE` clause to be `U.A` $\geq$ `140`) cannot include P5 in the top-3 result (because P5 indeed ranks 4-th under the current weighting $\vec{w} = |0.5\ 0.5|$). Using our preliminary top-k query modification technique in Chapter 3 to modify only the top-k construct (e.g., modifying k to be 4) cannot work either because P5 is filtered by the `WHERE` clause. This motivates us to develop holistic solutions that consider the modification of both SPJA constructs and top-k constructs in order to answer why-not questions on

top-k SQL queries. For the example above, the following *refined query* Q' is one good candidate answer:

Q':

SELECT U.ID

FROM U

WHERE U.A $\geq$ **140**

ORDER BY 0.5∗U.A + 0.5∗U.B

LIMIT **4**

Q' is precise because it includes no extra tuple and is similar to Q_o because only essential edits were carried out: (1) modifying from `U.A` $\geq$ `205` to `U.A` $\geq$ `140`, and (2) modifying k from 3 to 4.

In this chapter, we show that finding the best explanations (i.e. the best refined queries) is actually computationally expensive. 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.

4.1 Why-Not Top-K SPJ Question

In this section, we first focus on answering why-not questions on top-k SQL queries with SPJ clauses. We will extend the discussion to why-not top-k SQL queries with `GROUP BY` and aggregation in the next section.

4.1.1 The Problem and The Explanation Model

We consider a top-k SPJ query Q with a set of Select-Project-Join clauses SPJ, a monotonic scoring function f and a weighting vector $\vec{w} = |w[1]\ w[2]\ \cdots\ w[d]|$, where d is the number of attributes in the scoring function. For simplicity, we assume a larger value means a better score (and rank) and the weighting space subject to the constraints $\sum w[i] = 1$ and $0 \leq w[i] \leq 1$. We only consider conjunctions of predicates $P_1 \wedge \ldots \wedge P_n$, where each P_i is either a selection predicate "A_j *op* v" or a join predicate "A_j *op* A_k", where A is an attribute , v is a constant, and *op* is a comparison operator. For simplicity, our discussion focuses on $\geq$ comparison because generalizing our discussion to other comparison operators is straightforward. The query result would then be a set of k tuples whose scores are the largest (in case tuples with the same scores are tie at rank k-th, only one of them is returned).

Initially, a user issues an original top-k SPJ query $Q_o(SPJ_o, k_o, \vec{w}_o)$ on a dataset D. After she gets the query result, denoted as R_o, she may pose a *why-not* question with a set of *missing tuples* $Y = \{y_1, \ldots, y_l\}$ $(l \geq 1)$, where y_i has the same set of projection attributes as Q_o. In this chapter, we adopt the query-refinement approach in [35] so that the system returns the user a *refined query* $Q'(SPJ', k', \vec{w}')$, whose result R' includes Y and R_o, i.e., $\{Y \cup R_o\} \in R'$. It is possible that there are indeed no refined queries Q' that can include Y (e.g., Y includes a missing tuple whose expected attribute values indeed do not exist in the database). For those cases, the system will report to the user about her error.

There are possibly multiple refined queries for being the answers to a why-

not question $\{Y, Q_o\}$. We thus use ΔSPJ, Δk, and Δw to measure the quality of a refined query Q', where $\Delta k = k' - k_o$, $\Delta w = ||\vec{w}' - \vec{w}_o||_2$, and ΔSPJ is defined based on four different types of edit operations of SPJ clauses adopted in [35]:

- (e_1) modifying the constant value of a selection predicate in the `WHERE` clause.
- (e_2) adding a selection predicate in the `WHERE` clause.
- (e_3) adding/removing a join predicate in the `WHERE` clause.
- (e_4) adding/removing a relation in the `FROM` clause.

Following [35], we do not allow other edit operations such as changing the projection attributes (because users usually have a clear intent about the projection attributes). Note that there is no explicit edit operation for removing a selection predicate, since it is equivalent to modifying the constant value in the predicate to cover the whole domain of the attribute. Furthermore, we also do not consider modifying the joins to include self-join. Let c_i denote the cost of the edit operation e_i, and we follow [35] to set $c_1 = 1, c_2 = 3, c_3 = 5, c_4 = 7$. So, $\Delta SPJ = \sum_{1 \leq i \leq 4}(c_i \times n_i)$, where n_i is the number of edit operations e_i used to obtain the refined query Q'. In order to capture a user's tolerance to the changes of SPJ clauses, k, and $\vec{w}$ on her original query Q_o, we first define a basic penalty model that sets the penalties λ_{spj}, λ_k and λ_w to ΔSPJ, Δk and Δw, respectively, where $\lambda_{spj} + \lambda_k + \lambda_w = 1$:

$$Penalty = \lambda_{spj}\Delta SPJ + \lambda_k \Delta k + \lambda_w \Delta w \tag{4.1}$$

A	B
P1	Alice
P2	Bob
P3	Chandler
P4	Daniel
P5	Eagle
P6	Fabio
P7	Gary
P8	Henry

(a) Table T_1

A	C	D	E
P1	90	400	80
P2	60	290	60
P3	90	200	100
P4	50	300	70
P4	80	100	210
P5	70	250	70
P6	50	280	50
P7	100	500	100

(b) Table T_2

A	F	G	H
P1	60	200	70
P2	100	250	90
P3	90	300	90
P7	80	300	100
P8	60	200	60

(c) Table T_3

Figure 4.2. Running example: data set D

Note that the basic penalty model is able to capture both the *similar* and *precise* requirements. Specifically, a refined query Q' that minimizes *Penalty* implies it is similar to the original query Q_o. To make the result precise (having fewer extra tuples), we can set a larger penalty to λ_k.

The basic penalty model, however, has a drawback because Δk generally could be a large integer (as large as $|D|$) whereas Δw and ΔSPJ are generally smaller. One possible way to mitigate this discrimination is to normalize them respectively.

We normalize ΔSPJ using the maximum editing cost ΔSPJ_{max}. This cost refers to the editing cost of obtaining a refined SPJ query Q^{SPJ}_{max}, whose (1) SPJ constructs most deviated from the SPJ constructs of original query Q_o (based on the four types of edit operations e_1 to e_4) and (2) with a query result that includes all missing tuples Y and the original query result R_o.

Example 4.1 *Figure 4.2 shows an example database D with three base tables T_1, T_2, and T_3. Assume a user has issued the following top-2 SQL query:*

A	B	C	D	E	0.5*D+0.5*E
P7	Gary	100	**500**	100	300
P1	Alice	90	**400**	80	240
P4	Daniel	50	300	70	185
P2	Bob	60	290	60	175
P6	Fabio	50	280	50	165
P5	Eagle	70	250	70	160
P4	Daniel	80	100	210	155
P3	Chandler	90	200	100	150

Figure 4.3. $T_1 \bowtie T_2$ **ranked under** $\vec{w}_o = |0.5\ 0.5|$

Q_o:

SELECT B

FROM T_1, T_2

WHERE $T_1.A = T_2.A$ *AND* $D \geq 400$

*ORDER BY 0.5*D + 0.5*E*

LIMIT 2

By referring to Figure 4.3, the join result of $T_1 \bowtie T_2$*, the result* R_o *of the top-2 query is:* {*Gary, Alice*}*. Assuming the missing tuples set* Y *of the why-not question is* {*Chandler*}*. Then, the refined SPJ query whose SPJ constructs most deviate from the original query* Q_o *but with a query result that includes all missing tuples* Y *and the original query result* R_o *is:*

Q^{SPJ}_{max}:

SELECT B

FROM T_1, T_2, $\boldsymbol{T_3}$

WHERE $T_1.A = T_2.A$ *AND* $\boldsymbol{T_1.A = T_3.A}$

AND $\mathbf{C} \geq \mathbf{50}$ *AND* $D \geq \mathbf{100}$ *AND* $\mathbf{E} \geq \mathbf{50}$

AND $\mathbf{F} \geq \mathbf{60}$ *AND* $\mathbf{G} \geq \mathbf{200}$ *AND* $\mathbf{H} \geq \mathbf{60}$

According, $\Delta SPJ_{max} = 1 \times c_1 + 5 \times c_2 + 1 \times c_3 + 1 \times c_4 = 1 + 5 \times 3 + 5 + 7$ *= 28.* □

We normalize Δk using $(r_o - k_o)$, where r_o is the worst rank among all tuples in $Y \cup R_o$ of a refined top-j SQL query Q^{SPJ}_{min}, whose (1) SPJ constructs least deviated from the original query Q_o (measured by c_1 to c_4), (2) using the original weighting $\vec{w_o}$, (3) with a query result that includes all missing tuples Y and the original query result R_o, and (4) the modification of j is minimal.

To explain why r_o is a suitable value to normalize Δk, we first remark that we could normalize Δk using the cardinality of the join result of Q_o because that is the worst possible rank. But to get a more reasonable rank, we look at Equation 4.1. First, to obtain the "worst" but reasonable value of Δk, we can assume that we do not modify the weighting, leading to condition (2) above. Similarly, we do not want to modify the SPJ constructs so much but we hope the SPJ constructs at least do not filter out the missing tuples Y and the original query result R_o, leading to conditions (1) and (3) above. So, based on Example 4.1, Q^{SPJ}_{min} is:

Q^{SPJ}_{min}:
SELECT B
FROM T_1, T_2
WHERE T_1.A = T_2.A AND D $\geq$ **200**
ORDER BY 0.5*D + 0.5*E
LIMIT **7**

We note that the following is **not** Q^{SPJ}_{min} although it also satisfies conditions (1) to (3) because its modification of j is from 2 to 8, which is not minimal

(condition 4) comparing with the true Q_{min}^{SPJ} above:

SELECT B

FROM T_1, T_2

WHERE T_1.A = T_2.A AND D $\geq$ **100**

ORDER BY 0.5*D + 0.5*E

LIMIT **8**

So, based on Q_{min}^{SPJ}, Δk will be normalized by $(r_o - k_o) = (7 - 2) = 5$.

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 we have proven that Δw is always smaller than $\sqrt{1 + \sum w_o[i]^2}$ by Lemma 3.1 in Chapter 3.

Summarizing the above discussion about normalizing Equation 4.1, our *normalized penalty function* is as follows:

$$Penalty = \lambda_{spj} \frac{\Delta SPJ}{\Delta SPJ_{max}} + \lambda_k \frac{\Delta k}{(r_o - k_o)} + \lambda_w \frac{\Delta w}{\sqrt{1 + \sum w_o[i]^2}} \tag{4.2}$$

The problem definition is as follows. Given a *why-not* question $\{Y, Q_o\}$, where Y is a set of missing tuples and Q_o is the user's initial query with result R_o, our goal is to find a refined top-k SQL query $Q'(SPJ', k', \vec{w}')$ that includes $Y \cup R_o$ in the result with the smallest penalty. In this chapter, we use Equation 4.2 as the penalty function. Nevertheless, our solution works for all kinds of monotonic (with respect to all ΔSPJ, Δk and Δw) penalty functions. For better usability, we do not explicitly ask users to specify the values for λ_{spj}, λ_k and λ_w. Instead, we follow our work in Chapter 3 so that users are prompted to

Table 4.1. Example of candidate refined queries

Refined Query	ΔSPJ_i	Δk_i	Δw_i	Penalty
$Q'_1(\{T_1 \ltimes_A T_2,\ D \geq 200\}, 7, \lvert 0.5\ 0.5\rvert)$	1	5	0	0.35
$Q'_2(\{T_1 \ltimes_A T_2,\ D \geq 100\}, 8, \lvert 0.5\ 0.5\rvert)$	1	6	0	0.41
$Q'_3(\{T_1 \ltimes_A T_2,\ D \geq 200\}, 7, \lvert 0.6\ 0.4\rvert)$	1	5	0.14	0.38
$Q'_4(\{T_1 \ltimes_A T_2,\ D \geq 200 \wedge C \geq 90\}, 3, \lvert 0.5\ 0.5\rvert)$	4	1	0	0.11
$Q'_5\{T_1 \ltimes_A T_2,\ D \geq 200\}, 3, \lvert 0.2\ 0.8\rvert)$	1	1	0.42	0.20

answer a simple multiple-choice question as illustrated in Figure 4.4.[1]

Question / Choice	*Prefer modifying k or your weightings?*
Prefer modify SPJA (PMSPJ)	$\lambda_{spj} = 0.1$, $\lambda_k = 0.45$, $\lambda_w = 0.45$
Prefer modify k (PMK)	$\lambda_{spj} = 0.45$, $\lambda_k = 0.1$, $\lambda_w = 0.45$
Prefer modify weightings (PMW)	$\lambda_{spj} = 0.45$, $\lambda_k = 0.45$, $\lambda_w = 0.1$
Never mind (NM); Default	$\lambda_{spj} = 1/3$ $\lambda_k = 1/3$, $\lambda_w = 1/3$

Figure 4.4. A multiple-choice question for freeing users to specify λ_{spj}, λ_k and λ_w

Assume the default option "Never mind" is chosen. Table 4.1 lists some examples of refined queries that could be the answer of the why-not question to Q_o in Example 4.1. According to the above discussion, we have $\Delta SPJ_{max} = 28$, $r_o - k_o = 5$ and $\sqrt{1 + \sum w_o[i]^2} = 1.2$ Among those refined queries, Q'_1 dominates Q'_2 because its Δk is smaller than that of Q'_2 and the other dimensions are equal. The best refined query in the example is Q'_4 ($Penalty$=0.11). At this point, readers may notice that the best refined query is in the *skyline* of the answer space of three dimensions: (1) ΔSPJ_i, (2) Δk_i, (3) Δw_i. Later, we will show how to exploit properties like this to obtain better efficiency in our algorithm.

[1]The number of choices and the pre-defined values for λ_{spj}, λ_k and λ_w, of course, could be adjusted. For example, to satisfy the *precise* requirement stated in [35], we suggest the user to choose the option where λ_k is a large value.

4.1.2 Problem Analysis

Answering a why-not question is essentially searching for the best refined SPJ clause and weighting in (1) the space $\boldsymbol{\mathcal{S}}_{SPJ}$ of all possible modified SPJ clauses and in (2) the space $\boldsymbol{\mathcal{S}}_w$ of all possible weightings, respectively. It is not necessary to search for k because once the best set of SPJ clauses and the best weighting $\vec{w}$ are found, the value of k can be accordingly set as the rank of the missing tuple (if there are multiple missing tuples, set as the worst rank of all missing tuples). The search space $\boldsymbol{\mathcal{S}}_{SPJ}$ can be further divided into two: (1a) the space of the query schemas $\boldsymbol{\mathcal{S}}_{QS}$ and (1b) the space of all the selection conditions $\boldsymbol{\mathcal{S}}_{sel}$. A query schema QS represents the set of relations in the `FROM` clause and the set of join predicates in the `WHERE` clause. A selection condition *sel* represents the set of selection predicates in the `WHERE` clause.

First, the space $\boldsymbol{\mathcal{S}}_{QS}$ is $O(2^n)$, where n is the number of relations in the database. Second, given a particular query schema QS_i, the space $\boldsymbol{\mathcal{S}}_{sel}^{QS_i}$ is $O(\prod_{j=1}^{m_i}(|A_j|+1))$, where m_i is the number of attributes in QS_i, $|A_j|$ is the number of distinct values in attribute A_j in QS_i, and $|A_j|+1$ takes into account an attribute A_j can optionally be or not to be added to the selection condition. Therefore, the space $\boldsymbol{\mathcal{S}}_{SPJ}$ is $O(\sum_{QS_i \in \boldsymbol{\mathcal{S}}_{QS}}(\boldsymbol{\mathcal{S}}_{sel}^{QS_i}))$. Finally, the space $\boldsymbol{\mathcal{S}}_w$ is infinite. Hence, it is obvious that finding the exact best refined query from $\boldsymbol{\mathcal{S}}_{SPJ} \times \boldsymbol{\mathcal{S}}_w$ is impractical.

4.1.3 The Solution

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.

Let us start the discussion by illustrating our basic idea under the assumption that there is only one missing tuple y. First, we observe that not every query schema can generate a query whose results contain y. For example 4.1, if the missing tuple y is (Henry), then the query schema $T_1 \bowtie T_2$ does not include y. In this case, no matter how we exhaust the search space $\boldsymbol{\mathcal{S}}_{sel}$ and $\boldsymbol{\mathcal{S}}_w$, we cannot generate a valid refined query. In other words, during the search for the good why-not answer, if we can filter out the query schema $T_1 \bowtie T_2$ first, the subsequent search in $\boldsymbol{\mathcal{S}}_{sel}$ and $\boldsymbol{\mathcal{S}}_w$ can be eliminated. Hence, we enumerate the space $\boldsymbol{\mathcal{S}}_{QS}$ first. Similarly, we should enumerate $\boldsymbol{\mathcal{S}}_{sel}$ before $\boldsymbol{\mathcal{S}}_w$, since some predicates in $\boldsymbol{\mathcal{S}}_{sel}$ may filter out the tuples that result in y.

When enumerating $\boldsymbol{\mathcal{S}}_{QS}$, there could be multiple query schemas that satisfy the requirement of generating a query whose results contain y. Our idea here is similar to [35], which starts from the original query schema QS_o, carries out incremental modification to QS_o (using edit operations e_3 and e_4), and stops once we have found a query schema QS' for which queries based on that can include $y \cup R_o$ in the result. If no such a query schema is found, we report to the user that no refined query can answer her why-not question.

Once the target query schema QS' is found, we next enumerate all possible selection conditions $\boldsymbol{\mathcal{S}}_{sel}^{QS'}$ that can be derived from QS' with a set S_w of weighting vectors. The set S_w includes a random sample S of vectors $\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_s$ from the weighting space $\boldsymbol{\mathcal{S}}_w$ and the original weighting $\vec{w}_o$. That is, $S_w = \vec{w}_o \cup S$. For each selection condition $sel_i \in \boldsymbol{\mathcal{S}}_{sel}^{QS'}$ and each weighting $\vec{w}_j \in S_w$, we formulate a refined top-k SQL query Q'_{ij} and execute it using a progressive top-k SQL

algorithm (e.g., [21,22,25]), which progressively reports each top rank tuple one-by-one, until all tuples in $y \cup R_o$ come forth to the result set at ranking r_{ij}. So, after $|\boldsymbol{\mathcal{S}}_{sel}^{QS'}| \cdot |S_w|$ progressive top-k SQL executions, we have $|\boldsymbol{\mathcal{S}}_{sel}^{QS'}| \cdot |S_w|$ refined queries. Finally, using r_{ij} as the refined k', sel_i and QS' to formulate the refined SPJ clauses SPJ', and $\vec{w}_j$ as the refined weighting $\vec{w}'$, the refined query $Q'(SPJ', k', \vec{w}')$ with the least penalty is returned to the user as the why-not answer.

The basic idea above incurs many progressive top-k SQL executions. In the following, we present our algorithm in detail. The algorithm consists of three phases and includes different optimization techniques in order to reduce the execution time.

[PHASE-1] In this phase, we start from the original query schema QS_o and do incremental modification to QS_o (using edit operations e_3 and e_4) and stop until we find a query schema QS' for which queries based on that can include $y \cup R_o$ in the result. As all the subsequent considered refined queries will be based on QS', we materialize the join result J based on QS' in order to avoid repeated computation of the same join result based on QS' in the subsequent phase. If no such a query schema is found, we report to the user that no refined query can answer her why-not question. Finally, we randomly sample s weightings $\vec{w}_1, \vec{w}_2, \cdots, \vec{w}_s$ from the weighting space and add them into S_w in addition to $\vec{w}_o$.

[PHASE-2] Next, for a **subset** $S_{sel}^{QS'} \subseteq \boldsymbol{\mathcal{S}}_{sel}^{QS'}$ of selection conditions that can be derived from QS' and weighting vectors $\vec{w}_j \in S_w$, we execute a progressive

top-k SQL query on the materialized join result J using a selection condition $sel_i \in S_{sel}^{QS'}$ and a weighting $\vec{w_j} \in S_w$ until a **stopping condition** is met. From now on, we denote a progressive top-k SQL execution as:

$$r_{ij} = \text{TOPK}(sel_i, \vec{w}_j, \text{STOPPING-CONDITION})$$

where r_{ij} denotes the rank when all $y \cup R_o$ come forth to the result under selection condition sel_i and weighting $\vec{w_j}$.

In the basic idea mentioned above, we have to execute $|\boldsymbol{\mathcal{S}}_{sel}^{QS'}| \cdot |S_w|$ progressive top-k SQL executions. In Section 4.1.3.1, we will illustrate that we can just focus on a much smaller subset $S_{sel}^{QS'} \subseteq \boldsymbol{\mathcal{S}}_{sel}^{QS'}$ without jeopardizing the quality of the answer. Consequently, the number of progressive top-k SQL executions could be largely reduced to $|S_{sel}^{QS'}| \cdot |S_w|$.

Furthermore, the original stopping condition in our basic idea is to proceed the TOPK execution on J until all tuples in $y \cup R_o$ come forth to the result[2]. However, if some tuples in $y \cup R_o$ rank very poorly under some weighting $\vec{w_j}$, the corresponding progressive top-k SQL operation may be quite slow because it has to access many tuples in the materialized join result J. In Section 4.1.3.2, we present a much more aggressive and effective stopping condition that makes most of those operations stop early before $y \cup R_o$ comes forth to the result.

Finally, we present a technique in Section 4.1.3.3 that can identify some weightings in S_w whose generated refined queries have poorer quality, thereby skipping the TOPK execution for those weightings to gain better efficiency.

[2]There could be multiple tuples in J that match y. In this case, we randomly choose one in J.

[PHASE-3] Using r_{ij} as the refined k', sel_i and QS' as the refined SPJ clauses SPJ', and $\vec{w}_j$ as the refined weighting $\vec{w}'$, the refined query $Q'(SPJ', k', \vec{w}')$ with the least penalty is returned to the user as the why-not answer.

4.1.3.1 Excluding selection predicates that could not yield good refined queries

In the basic idea, we have to enumerate all possible selection conditions based on the query schema QS' chosen in Phase-1. As mentioned, if there are m attributes in QS' and $|A_i|$ is the number of distinct values in attribute A_i in that query schema, there would be $O(\prod_1^m(|A_i|+1))$ possible selection conditions. However, the following theorem can help us to exclude selection conditions that could not yield good refined queries:

Theorem 4.1 *If we need to modify the original selection condition by modifying the constant value v_i of its selection predicate P in the form of $A_i \geq v_i$ (edit operation e_1), or adding a selection predicate P in the form of $A_i \geq v_i$ (edit operation e_2) to the selection condition, we can simply consider only one possibility of P, which is $A_i \geq v_i^{min}$, where v_i^{min} is the minimum value of attribute A_i among tuples in $y \cup R_o$, because P in any other form would not lead to better refined top-k SQL queries whose penalties are better than P as $A_i \geq v_i^{min}$.*

Proof. Let $sel' = \{A_1 \geq v_1, \cdots, A_l \geq v_l\}$ $(l \leq m)$ be the selection condition after modification and particularly $A_i \geq v_i'$ be a predicate P' in sel' whose gets

modified/added from the original selection condition.

First, v_i' in P' has to be smaller than or equal to v_i^{min} or otherwise some tuples in $y \cup R_o$ would get filtered away.

Second, comparing a predicate P: $A_i \geq x$, with x as any value smaller than v_i^{min}, and the predicate P_{min}: $A_i \geq v_i^{min}$, these two predicates incur the same ΔSPJ to the original selection condition.

Third, as the predicate P: $A_i \geq x$, is less restrictive than the predicate P_{min}: $A_i \geq v_i^{min}$, more tuples could pass P. So, given a weighting $\vec{w}$, the worst rank of $y \cup R_o$ under P cannot be better than that under P_{min}. That implies Δk under P would not be better than that under P_{min} as well. Therefore, we conclude that top-k SQL queries based on P would not lead to better refined top-k SQL queries based on P_{min}. □

Consider Example 4.1 again. The query schema QS' that can include back the missing tuple (Chandler) would lead to a join result like Figure 4.3. Originally, we have to consider eight predicates when dealing with attribute D, which are $D \geq 500$, $D \geq 400$, $D \geq 300$, $D \geq 290$, $D \geq 280$, $D \geq 250$, $D \geq 200$, and $D \geq 100$. By using Theorem 4.1, we just need to consider $D \geq 200$ because among $y \cup R_o = \{$Chandler, Gary, Alice$\}$, their attribute values of D are 200, 400, and 500, with 200 as the minimum. Similar for attribute E, by using Theorem 4.1, we just need to consider $E \geq 80$. The above discussion can be straightforwardly generalized to other comparisons including $\leq, <, >$.

4.1.3.2 Stopping a progressive top-k SQL operation earlier

In PHASE-2 of our algorithm, the basic idea is to execute the progressive top-k SQL query until $y \cup R_o$ come forth to the result, with rank r_{ij}. In the following, we show that it is actually possible for a progressive top-k SQL execution to stop early even before $y \cup R_o$ come forth to the result. Techniques here are adjusted based on Technique (i) in Section 3.2.4 to support SPJ clauses.

Consider an example where a user specifies a top-k SQL query $Q_o(SPJ_o, k_o = 2, \vec{w}_o)$ on a data set D and poses a why-not question about a missing tuple y. Assume that the list of weightings S_w is $[\vec{w}_o, \vec{w}_1, \vec{w}_2, \vec{w}_3]$ and TOPK($sel_1, \vec{w}_o$, UNTIL-SEE-$\{y \cup R_o\}$) is first executed, and y's actual ranking under sel_1 and $\vec{w}_o$ is 7. Now, we have our first candidate refined query $Q'_{1o}(sel_1, r_{1o} = 7, \vec{w}_o)$, with $\Delta k = 7 - 2 = 5$ and $\Delta w = ||\vec{w}_o - \vec{w}_o||_2 = 0$. The corresponding penalty, denoted as, $Pen_{Q'_{1o}}$, could be calculated using Equation 4.2. Remember that we want to find the refined query with the least penalty Pen_{min}. So, at this moment, we set penalty $Pen_{min} = Pen_{Q'_{1o}}$.

According to our basic idea outlined above, we should next execute another progressive top-k SQL using the selection condition sel_1 and another weighting vector, say, $\vec{w}_1$, until $y \cup R_o$ come forth to the result with a rank r_{11}. However, we notice that the skyline property in the answer space can help to stop that operation earlier, even before $y \cup R_o$ are seen. Given the first candidate refined query $Q'_{1o}(sel_1, r_{1o} = 7, \vec{w}_o)$ with $\Delta w = 0$ and $\Delta k = 5$, any other candidate refined queries Q'_{1j} with $\Delta k > 5$ must be dominated by Q'_{1o}. In our example, after the first executed progressive top-k SQL execution, all the subsequent progressive top-k SQL executions with sel_1 can stop once $y \cup R_o$ do not show up in the top-7

tuples.

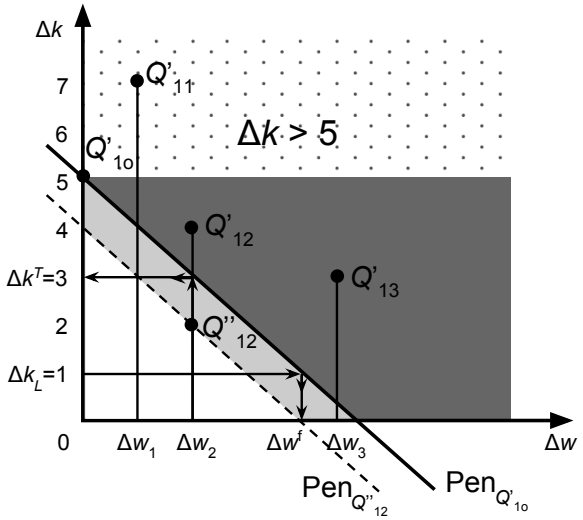


Figure 4.5. Example of answer space under selection condition sel_1

Figure 4.5 illustrates the answer space of the example. The idea above essentially means a progressive top-k SQL execution can stop if $y \cup R_o$ do not show up in result after returning the top-7 tuples (i.e., $\Delta k > 5$; see the dotted region). For example, consider Q'_{11} in Figure 4.5 whose weighting is $\vec{w}_1$ and $y \cup R_o$ show up in the result only when $k = 9$, i.e., $\Delta k = 9 - 2 = 7$. So, the progressive top-k SQL associated with Q'_{11} can stop after $k = 7$, because after that Q'_{11} has no chance to dominate Q'_{1o} anymore. In other words, the progressive top-k SQL associated with Q'_{11} do not wait to reach $k = 9$ where $y \cup R_o$ come forth to the result but stop early when $k = 7$.

While useful, we can actually be even more aggressive in many cases. Consider another candidate refined query, say, Q'_{12}, in Figure 4.5. Assume that $r_{12} = \text{TOPK}(sel_1, \vec{w}_2, \text{UNTIL-SEE-}\{y \cup R_o\}) = 6$ (i.e., $\Delta k = 6 - 2 = 4$), which is not covered by the above technique (since $\Delta k \not< 5$). However, Q'_{12} can also stop early, as follows. In Figure 4.5, we show the normalized penalty Equation 4.2

as a slope $Pen_{min} = Pen_{Q'_{1o}}$ that passes through the best refined query so far (currently Q'_{1o}). All refined queries that lie on the slope have the same penalty value as Pen_{min}. In addition, all refined queries that lie above the slope actually have a penalty larger than Pen_{min}, and thus are dominated by Q'_{1o}. Therefore, similar to the skyline discussion above, we can determine an even tighter *threshold ranking* r^T for stopping the subsequent progressive top-k SQL operations with sel_1:

$$\begin{aligned} &r^T = \Delta k^T + k_o, \text{ where} \\ &\Delta k^T = \lfloor (Pen_{min} - \lambda_{spj}\frac{\Delta SPJ}{\Delta SPJ_{max}} - \lambda_w \frac{\Delta w}{\sqrt{1+\sum w_o[i]^2}}) \frac{r_o - k_o}{\lambda_k} \rfloor \end{aligned} \tag{4.3}$$

Equation 4.3 is a rearrangement of Equation 4.2 (with $Penalty = Pen_{min}$). Back to our example in Figure 4.5, given that the weighting of candidate refined query Q'_{12} is $\vec{w}_2$, we can first compute its Δw_2 value. Then, we can project Δw_2 onto the slope Pen_{min} (currently $Pen_{min} = Pen_{Q'_{1o}}$) to obtain the corresponding Δk^T value, which is 3 in Figure 4.5. That means, if we carry out a progressive top-k SQL operation using sel_1 as the predicates and $\vec{w}_2$ as the weighting, and if $y \cup R_o$ still do not appear in result after the top-5 tuples ($r^T = \Delta k^T + k_o = 3 + 2 = 5$) are seen, then we can stop it early because the penalty of Q'_{12} is worse than the penalty Pen_{min} of the best refined query (Q'_{1o}) seen so far.

Following the discussion above, we now have two early stopping conditions for the progressive top-k SQL algorithm: UNTIL-SEE-$\{y \cup R_o\}$ and UNTIL-RANK-r^T. Except for the first progressive top-k SQL operation which TOPK($sel_1, \vec{w}_o$, UNTIL-SEE-$\{y \cup R_o\}$) must be used, the subsequent progressive top-k SQL operations with sel_1

can use "UNTIL-SEE-$\{y \cup R_o\}$ OR UNTIL-RANK-r^T" as the stopping condition. We remark that the conditions UNTIL-RANK-r^T and UNTIL-SEE-$\{y \cup R_o\}$ are both useful. For example, assume that the actual worst rank of $y \cup R_o$ under sel_1 and $\vec{w}_2$ is 4, which gives it a $\Delta k_{12} = 2$ (see Q''_{12} in Figure 4.5). Recall that by projecting Δw_2 onto the slope of $Pen_{min} = Pen_{Q'_{1o}}$, we can stop the progressive top-k SQL operation after $r^T = 3 + 2 = 5$ tuples have been seen. However, using the condition UNTIL-SEE-$\{y \cup R_o\}$, we can stop the progressive top-k SQL operation when all $y \cup R_o$ show up at rank four. This drives us to use "UNTIL-SEE-$\{y \cup R_o\}$ 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''_{12} has been executed, the best refined query seen so far should be updated as Q''_{12} (because its penalty is better than Q'_{1o}). Therefore, Pen_{min} now is updated as $Pen_{Q''_{12}}$ and the slope Pen_{min} should be updated to pass through Q''_{12} now (the dotted slope in Figure 4.5). Because Pen_{min} is continuously decreasing, Δk^T and thus the threshold ranking r^T would get smaller and smaller and the subsequent progressive top-k SQL operations can terminate even earlier while the algorithm proceeds.

The above early stopping technique can be applied to the subsequent selection conditions. In our example, after selection condition sel_1 and turning to consider selection condition sel_2 with the set of weightings S_w, we can derive r^T based on Equation 4.3 by simply reusing the Pen_{min} obtained from sel_1.

4.1.3.3 Skipping progressive top-k SQL operations

In PHASE-2 of our algorithm, the basic idea is to execute progressive top-k SQL queries for *all* selection conditions in $S_{sel}^{QS'}$ and *all* weightings in S_w. After the discussion in 4.1.3.2, we know that some progressive top-k SQL executions can early stop. We now illustrate how some of those executions could be skipped entirely, so that the overall running time can be reduced further. Techniques here are adjusted based on technique (ii) in Section 3.2.4 to support SPJ clauses.

The first pruning opportunity is based on the observation from [36] that under the same selection condition sel_i, similar weighting vectors (measured using their cosine similarity) may lead to top-k SQL results with more common tuples. Therefore, if an operation $\text{TOPK}(sel_i, \vec{w_j}, \text{UNTIL-SEE-}\{y \cup R_o\})$ for $\vec{w_j}$ has already been executed, and if a weighting $\vec{w_l}$ is similar to $\vec{w_j}$, then we can use the query result R_{ij} of $\text{TOPK}(sel_i, \vec{w_j}, \text{UNTIL-SEE-}\{y \cup R_o\})$ to deduce the smallest k value for sel_i and $\vec{w_l}$. Let k' be the deduced k value for sel_i and $\vec{w_l}$. If the deduced k' is larger than the threshold ranking r^T, then we can skip the entire $\text{TOPK}(sel_i, \vec{w_l}, \text{STOPPING-CONDITION})$ operation.

We illustrate the above by reusing our running example. Assume that we have cached the result sets of executed progressive top-k SQL queries. Let R_{1o} be the result set of the first executed query $\text{TOPK}(sel_1, \vec{w}_o, \text{UNTIL-SEE-}\{y \cup R_o\})$ and $R_o = \{t_5, t_6\}$. Assume that $R_{1o} = [t_1, t_2, t_3, t_4, t_5, t_6, y]$. Then, when we are considering the next weighting vector, say, $\vec{w}_1$, in S_w, we first follow Equation 4.3 to calculate the threshold ranking r^T. In Figure 4.5, projecting $\vec{w}_1$ onto slope $Pen_{Q'_{1o}}$ we get $r^T = 4 + 2 = 6$. Next we calculate the scores of all tuples in R_{1o} using $\vec{w}_1$ as the weighting. More specifically, let us denote the tuple in $y \cup R_o$

under weighting vector $\vec{w}_1$ as t_{bad} if it has the worst rank among $y \cup R_o$. In the example, assume under $\vec{w}_1$, the scores of t_1, t_2, t_3 and t_4 are still better than t_{bad}, then k' is at least $4 + 3 = 7$. Since k' is worse than $r^T = 6$, we can skip the entire TOPK($sel_1, \vec{w}_1$, STOPPING-CONDITION) operation.

The above caching technique is shown to be the most effective between similar weighting vectors [36]. Therefore, we design the algorithm in a way that *the list of weightings S_w is sorted according to their corresponding Δw_i values* (of course, $\vec{w_o}$ is in the head of the list since $\Delta w_o = 0$). In addition, the technique is general so that the cached result for a specific selection condition sel_i can also be used to derive the smallest k' value of y and R_o for another selection condition sel_j. As long as sel_i and sel_j are similar, the chance that we can deduce k' from the cached result that leads to TOPK operation pruning is also higher. So, we design the algorithm in a way that sel_i is enumerated in increasing order of Δsel as well.

The second pruning opportunity is to exploit the best possible ranking of $y \cup R_o$ (under all possible weightings) to set up an early termination condition *for some weightings*, so that after a certain number of progressive top-k SQL operations have been executed under sel_i, operations associated with some other weightings for the same sel_i can be skipped.

Recall that the best possible ranking of $y \cup R_o$ is k_o+1, since $|y \cup R_o| = k_o+1$. Therefore, the lower bound of Δk, denoted as Δk_L equals 1. So, this time, we project Δk_L onto slope Pen_{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 "$y \cup R_o$ has $\Delta k < \Delta k_L$", which is impossible. As our algorithm is designed

to examine weightings in their increasing order Δw values, when a weighting $\vec{w}_j \in S_w$ has $||\vec{w}_j - \vec{w}_o||_2 > \Delta w^f$, TOPK($sel_i, \vec{w}_j$, STOPPING-CONDITION) and all subsequent progressive top-k SQL operations TOPK($sel_i, \vec{w}_l$, STOPPING-CONDITION) where $l \geq j + 1$ could be skipped.

Reuse Figure 4.5 as an example. By projecting $\Delta k_L = 1$ onto the slope $Pen_{Q'_{1o}}$, we could determine the corresponding Δw^f value. So, when the algorithm finishes executing a progressive top-k SQL operation for weighting $\vec{w}_2$, the algorithm can skip all the remaining weightings and proceed to examine the next selection condition.

As a remark, we would like to point out that the pruning power of this technique also increases while the algorithm proceeds. For instance, in Figure 4.5, if Q''_{12} has been executed, slope Pen_{min} is changed from slope $Pen_{Q'_{1o}}$ to slope $Pen_{Q''_{12}}$. Projecting Δk_L onto the new Pen_{min} slope would result in a smaller Δw^f, which in turn increases the chance of eliminating more weightings.

The last pruning opportunity is to set up an early termination condition *for the whole algorithm.* In fact, since we are sorting sel_i in their increasing order of Δsel, as soon as we encounter a $\Delta SPJ > (Pen_{min} - \lambda_k \frac{\Delta k_L}{r_o - k_o}) \frac{\Delta SPJ_{max}}{\lambda_{spj}}$, we can skip all subsequent progressive top-k SQL operations and terminate the algorithm. This equation is a rearrangement of the following equation.

$$Pen_{min} < \lambda_{spj} \frac{\Delta SPJ}{\Delta SPJ_{max}} + \lambda_k \frac{\Delta k_L}{r_o - k_o} \tag{4.4}$$

The pseudo-code of the complete algorithm is presented in Algorithm 4.1. It is self-explanatory and mainly summarizes what we have discussed above, so

we do not give it a full walkthrough here.

4.1.3.4 How large should be the list of weighting vectors?

Given that there are an infinite number of points (weightings) in the weighting space, 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 3.7 to determine the sample size.

4.1.3.5 Multiple Missing Tuples

To handle multiple missing tuples $Y = \{y_1, \cdots, y_l\}, l > 1$, we just need little modification to the algorithm. Specifically, a refined query needs to ensure $\{Y \cup R_o\}$ (instead of $y \cup R_o$) come forth to the result. When generating the candidate selection conditions, we need to consider the minimum attribute values for all tuples in $Y \cup R_o$. Finally, the stopping condition should be changed to UNTIL-SEE-$\{Y \cup R_o\}$.

4.2 Why-Not Top-K SPJA Question

In this section, we extend the discussion to why-not top-k SQL queries with `GROUP BY` and aggregation.

Algorithm 4.1 Answering a Why-not Top-K SPJ Question

Input:

The dataset D; original top-k SQL query $Q_o(SPJ_o, k_o, \vec{w}_o)$; missing tuple y; penalty settings λ_{spj}, λ_k, λ_w; $T\%$ and Pr; edit cost for SPJ clauses: c_1, c_2, c_3 and c_4

Output:

A refined query $Q'(SPJ', k', \vec{w}')$

Phase 1:

Obtain QS' and J by doing incremental modification to the original query schema QS_o

if QS' does not exist **then**

  return "cannot answer the why-not question"

end if

Construct $S_{sel}^{QS'}$ based on Section 4.1.3.1

Sort $S_{sel}^{QS'}$ according to their Δsel value

Determine s from $T\%$ and Pr according to Section 4.1.3.4;

Sample s weightings from the weighting space and add them and $\vec{w}_o$ into S_w;

Sort S_w according to their Δw values;

Phase 2:

$R \leftarrow \emptyset$

$Pen_{min} \leftarrow \infty$

$\Delta w_f \leftarrow \infty$

for all $sel_i \in S_{sel}^{QS'}$ **do**

  if $\Delta SPJ_i > (Pen_{min} - \frac{\lambda_k}{r_o - k_o})\frac{\Delta SPJ_{max}}{\lambda_{spj}}$ **then**

    break; //Section 4.1.3.3 — early algorithm termination

  end if

  for all $\vec{w}_j \in S_w$ **do**

    if $\Delta w_j > \Delta w^f$ **then**

      break; //Technique 4.1.3.3 — skipping weightings

    end if

    compute r^T based on Equation 4.3

    if there exist $r^T - |y \cup R_o| + 1$ objects in some $R_{ij} \in R$ having scores better than the worst rank of $y \cup R_o$ under $\vec{w}_j$ **then**

      continue; //Section 4.1.3.3 — use cached result to skip a progressive top-k SQL

    end if

    $(R_{ij}, r_{ij}) \leftarrow$ TOPK$(sel_i, \vec{w}_j,$ UNTIL-SEE-$\{y \cup R_o\}$ OR UNTIL-RANK-$r^T)$; //Section 4.1.3.2 — stopping a progressive top-k SQL early

    Compute Pen_{ij} based on Equation 4.2

    $R \leftarrow R \cup (R_{ij}, sel_i, \vec{w}_j)$;

    if $Pen_{ij} < Pen_{min}$ **then**

      $Pen_{min} \leftarrow Pen_{ij}$;

      update Δw^f according to Section 4.1.3.3

    end if

  end for

end for

Phase 3:

Return the best refined query $Q'(SPJ', k', \vec{w}')$ whose penalty=Pen_{min};

4.2.1 The Problem and The Explanation Model

Initially, a user issues an original top-k SPJA query $Q_o(SPJA_o, k_o, \vec{w}_o)$ on a dataset D. After she gets the result R_o, she may pose a *why-not* question about a set of *missing groups* $Y = \{g_1, \ldots, g_l\}$ $(l \geq 1)$, where g_i has the same set of projection attributes as Q_o. Then, the system returns the user a *refined query* $Q'(SPJA', k', \vec{w}')$, whose result R' includes Y and R_o, i.e., $\{Y \cup R_o\} \in R'$. If there are indeed no refined queries Q' that can include Y, the system will report to the user about her error.

For the SPJA clauses, we adopt the four edit operations in Section 4.1 whereas the penalty function is similar to Equation 4.2:

$$Penalty = \lambda_{spja} \frac{\Delta SPJA}{\Delta SPJA_{max}} + \lambda_k \frac{\Delta k}{(r_o - k_o)} + \lambda_w \frac{\Delta w}{\sqrt{1+\sum w_o[i]^2}} \tag{4.5}$$

The calculation of $\Delta SPJA$ is the same as ΔSPJ. The cost of $\Delta SPJA_{max}$ refers to the editing cost of obtaining a refined SPJA query Q^{SPJA}_{max}, whose definition is the same as Q^{SPJ}_{max} except that it needs to include all missing *groups* Y and the original result groups R_o.

The problem definition is as follows. Given a *why-not* question $\{Y, Q_o\}$, where Y is a set of missing groups and Q_o is the user's initial query with result R_o, our goal is to find a refined top-k SQL query $Q'(SPJA', k', \vec{w}')$ that includes $Y \cup R_o$ in the result with the smallest penalty with respect to Equation 4.5. Again, we do not explicitly ask users to specify the values for λ_{spja}, λ_k and λ_w, but prompt users to answer a simple multiple-choice question listed in Figure 4.4.

4.2.2 Problem Analysis

Since we allow the same set of edit operations as top-k SPJ queries, answering why-not top-k SPJA questions is also computationally expensive.

4.2.3 The Solution

The following changes are required to extend Algorithm 4.1 to handle why-not top-k SPJA questions.

First, when materializing the join result J of QS' (line 2), we in addition sort J using the group-by attributes so as to facilitate the subsequent grouping step.

Second, we do not apply Theorem 4.1. Consider the following top-k SPJA query based on the data set in Figure 4.2.

Q_o:
SELECT B
FROM T_1, T_2
WHERE T_1.A = T_2.A AND D $\geq$ 400
GROUP BY B
ORDER BY AVG(0.5*D + 0.5*E)
LIMIT 2

The result R_o is : {Gary, Alice}. Let us assume the why-not question is asking for the missing group "Daniel". From Figure 4.3, we see that the group "Daniel" is composed by two base tuples, with a group score average as $(185 + 155)/2 = 170$, which ranks 3-rd. If we follow Theorem 4.1 (using the

minimum D's values among Gary, Alice, and Daniel) to modify the selection condition $D \geq 400$ to $D \geq 100$, the top-4 would then become:

Group By B	AVG(0.5*D + 0.5*E)	Rank
Gary	300	1
Alice	240	2
Bob	175	3
Daniel	170	4

If we do not follow Theorem 4.1, we can modify the selection condition, say, $D \geq 400$ to $D \geq 300$, we can get a better refined query (in terms of k) because the missing group Daniel can now rank 3-rd because the group Bob has been filtered by the selection condition.

Lastly, when using the caching technique described in Section 4.1.3.3, we cannot just cache the resulting group or otherwise we do not have the tuples that contributed to that group to derive the new score using another weighting. Therefore, for that technique, we also cache the base tuples for each resulting group.

4.3 Experiments

We evaluate our proposed solution using both synthetic and real data. The real data is the NBA data set whose schema is shown in Figure 4.6. It contains statistics of all NBA players from 1973-2009 with four tables: (i) *Player* (4051 tuples) records players' name and their career start year, (ii) *Career* (4051 tuples) stores players' performance in their whole career, (iii) *Regular* (21961 tuples) and

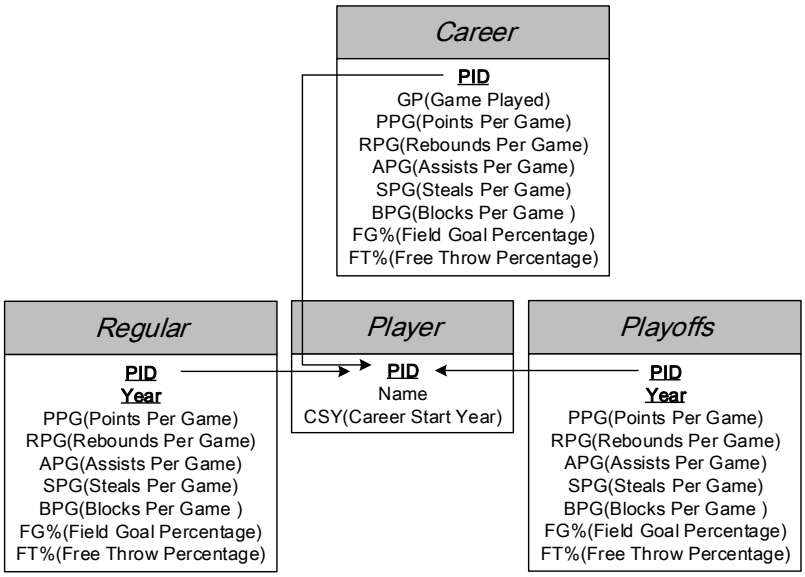


Figure 4.6. Schema of the NBA data set

(iv) *Playoffs* (8341 tuples) contain players' performance year-by-year in regular games and playoffs games, respectively.

By default, we set the system parameters $T\%$ and Pr as 0.5% and 0.7, respectively, resulting in a sample size of 241 weighting vectors. The algorithms are implemented in C++ and the experiments are run on a Ubuntu PC with Intel 3.4GHz i7 processor and 16GB RAM.

4.3.1 Case Study

Case 1 (Finding the top-3 players who have played at least 1000 games in NBA history). The first case study was to find the top-3 players with at least 1000-game experience in the NBA history. Therefore, we issued a query Q_1:

Q_1:

SELECT P.Name

FROM Player P, Career C

WHERE P.PID = C.PID AND C.GP >= 1000

ORDER BY ($\frac{1}{7}$ * C.PPG + $\frac{1}{7}$ * C.RPG + $\frac{1}{7}$ * C.APG + $\frac{1}{7}$ * C.SPG
+ $\frac{1}{7}$ * C.BPG + $\frac{1}{7}$ * C.FG% + $\frac{1}{7}$ * C.FT%) DESC

LIMIT 3

The initial result was[3]:

Rank	**Name**	**GP**	**PPG**	**RPG**	**APG**	**SPG**	**BPG**	**FG%**	**FT%**
1	Michael Jordan	1072	30.1	6.2	5.3	2.3	0.8	49.7	83.5
2	Oscar Robertson	1040	25.7	7.5	9.5	0	0	48.5	83.8
3	Kareem Abdul-jabbar	1560	24.6	11.2	3.6	0.7	2.0	55.9	72.1

We were surprised that Magic Johnson, who has won 5 NBA championships and 3 NBA Most Valuable Player (MVP) in his career, was not in the result. So we issued a why-not question $\{\{\text{Magic Johnson}\}, Q_1\}$.

Using our algorithm, we got a refined query Q_1' in 10.2ms (modifications are in **bold face**):

[3]We attach extra columns to the results for better understanding.

Q_1':

SELECT P.Name

FROM Player P, Career C

WHERE P.PID = C.PID AND C.GP >= **906**

ORDER BY ($\frac{1}{7}$ * C.PPG + $\frac{1}{7}$ * C.RPG + $\frac{1}{7}$ * C.APG + $\frac{1}{7}$ * C.SPG
+ $\frac{1}{7}$ * C.BPG + $\frac{1}{7}$ * C.FG% + $\frac{1}{7}$ * C.FT%) DESC

LIMIT **4**

Its new top-k result was:

Rank	**Name**	**GP**	**PPG**	**RPG**	**APG**	**SPG**	**BPG**	**FG%**	**FT%**
1	Michael Jordan	1072	30.1	6.2	5.3	2.3	0.8	49.7	83.5
2	Magic Johnson	906	19.5	7.2	11.2	1.9	0.4	52.0	84.8
3	Oscar Robertson	1040	25.7	7.5	9.5	0	0	48.5	83.8
4	Kareem Abdul-jabbar	1560	24.6	11.2	3.6	0.7	2.0	55.9	72.1

The refined query essentially hinted us that our original selection predicate C.GP >= 1000 (number of games played) eliminated Magic Johnson (who got a brilliant records without playing a lot of games).

We can see that using our early top-k query modification technique in Chapter 3 to modify only the top-k construct (be the value k or the weighting) cannot work because Magic Johnson is filtered by the predicate 'C.GP>=1000'. Using the SPJA query modification techniques in [35] to modify only the SPJ construct (e.g., modifying the selection predicate to be 'GP>=906') cannot include Magic Johnson in the top-3 result either (because we at least need to modify k to 4 in order to include Magic Johnson and the original result).

Case 2 (Finding the top-3 players who performed best in year 2004). In this case, we first look up the top-3 players in year 2004:

Q_2:

SELECT P.Name

FROM Player P, Regular R, Playoffs O

WHERE P.PID = R.PID AND P.PID = O.PID

AND R.Year = 2004 AND O.Year = 2004

ORDER BY ($\frac{1}{7}$ * R.PPG + $\frac{1}{7}$ * R.RPG +$\frac{1}{7}$ * R.APG
+ $\frac{1}{7}$ * R.SPG + $\frac{1}{7}$ * R.BPG + $\frac{1}{7}$ * R.FG%
+ $\frac{1}{7}$ * R.FT%) DESC

LIMIT 3

The initial result was: {Dirk Nowitzki, Allen Iverson, Steve Nash}. We wondered why Kobe Bryant was missing in the answer, so we posed a why-not question {{Kobe Bryant}, Q_2}.

Our algorithm returned the following refined query in 18.1ms:

Q_2':

SELECT P.Name

FROM Player P, Regular R ~~, Playoffs O~~

WHERE P.PID = R.PID ~~AND P.PID = O.PID~~

AND R.Year = 2004 ~~AND O.Year = 2004~~

ORDER BY (**0.2125** * R.PPG + **0.0902** * R.RPG
+ **0.1955** * R.APG + **0.0844** * R.SPG
+ **0.1429** * R.BPG + **0.0732** * R.FG%
+ **0.2013** * R.FT%) DESC

LIMIT **4**

The refined query essentially hinted us that Kobe Bryant did not play Play-Offs in year 2004. We checked back the data, and we were confirmed with that the fact Kobe Bryant's host team, LA Lakers, failed to enter the Playoffs in that year. The result of Q_2' was: {Allen Iverson, Dirk Nowitzki, Steve Nash, **Kobe Bryant**}. As an interpretation, in addition to eliminating the PlayOff records, the refined weighting Q_2' indicates that, in order to include Kobe Bryant in the result, we should weigh the players' scoring/assisting/free throwing abilities higher than the other abilities.

Case 3 (Finding the top-3 players who performed best in their early career). Our last case is to find the top-3 players who performed best in their first three years:

Q_3:

```
SELECT P.Name
FROM Player P, Regular R
WHERE P.PID = R.PID AND R.Year < P.CSY + 3
GROUP BY P.Name
ORDER BY AVG(1/7 * R.PPG + 1/7 * R.RPG + 1/7 * R.APG
             + 1/7 * R.SPG + 1/7 * R.BPG + 1/7 * R.FG%
             + 1/7 * R.FT%) DESC
LIMIT 3
```

The initial result was: {Michael Jordan, Magic Johnson, Kareem Abdul-jabbar}. In this case, we were interested in why Dirk Nowitzki, a player that was internationally famous in his early career, was missing in the result. So, we issued a why-not question {{Dirk Nowitzki}, Q_3}.

Our algorithm returned the following refined query in 48.3m:

Q_3':

SELECT P.Name

FROM Player P, Regular R

WHERE P.PID = R.PID AND R.Year < P.CSY + **6**

GROUP BY P.Name

ORDER BY AVG($\frac{1}{7}$ * R.PPG + $\frac{1}{7}$ * R.RPG + $\frac{1}{7}$ * R.APG
+ $\frac{1}{7}$ * R.SPG + $\frac{1}{7}$ * R.BPG + $\frac{1}{7}$ * R.FG%
+ $\frac{1}{7}$ * R.FT%) DESC

LIMIT **5**

The refined query essentially hinted us Dirk Nowitzki being early famous does not imply he had excellent performance in his early career. In fact, Dirk Nowitzki was in the top-5 if we accounted for the first six years of the players' career. Our first wrong impression of Dirk Nowitzki (Q_3) was probably due to Dirk Nowitzki was not an America player, which made him famous in his early career.

4.3.2 Performance

We next turn the focus to the performance of our algorithms. We used TPC-H data in the experiment. We selected 10 TPC-H queries that can let us extend as top-k SQL queries with minor modifications. The list of modifications of the selected queries is shown in Table 4.2.

Table 4.3 shows the parameters we varied in the experiments. The default

$Q2$	SELECT s_acctbal, s_name, n_name, p_partkey, p_mfgr, s_address, s_phone, s_comment FROM part, supplier, partsupp, nation, region WHERE [...] ORDER BY **(0.5 * p_retailprice + 0.5 * ps_supplycost) DESC LIMIT 10**
$Q3$	SELECT l_orderkey, sum(**0.5 * l_extendedprice * (1 - l_discount) + 0.5 * o_totalprice**) as amount, o_orderdate, o_shippriority FROM customer, orders, lineitem WHERE [...] GROUP BY l_orderkey, o_orderdate, o_shippriority ORDER BY amount DESC **LIMIT 10**
$Q9$	SELECT nation, o_year, sum(amount) as sum_profit FROM (SELECT n_name as nation, extract(year from o_orderdate) as o_year, **0.5*l_extendedprice*(1-l_discount)-0.5*ps_supplycost*l_quantity** as amount FROM part, supplier, lineitem, partsupp, orders, nation WHERE [...]) as profit GROUP BY nation, o_year ORDER BY **sum_profit DESC LIMIT 10**
$Q10$	SELECT c_custkey, c_name, sum(**0.5 * l_extendedprice * (1 - l_discount) + 0.5 * c_acctbal**) as amount, n_name, c_address, c_phone, c_comment FROM customer, orders, lineitem, nation WHERE [...] GROUP BY c_custkey, c_name, c_acctbal, c_phone, n_name, c_address, c_comment ORDER BY amount DESC **LIMIT 10**
$Q11$	SELECT ps_partkey, sum(**0.5 * ps_supplycost * ps_availqty + 0.5 * s_acctbal**) as value FROM partsupp, supplier, nation WHERE ps_suppkey=s_suppkey AND s_nationkey=n_nationkey AND n_name='GERMANY' GROUP BY [...] ORDER BY value DESC **LIMIT 10**
$Q12$	SELECT **o_orderkey, sum(0.5 * l_extendedprice * (1 - l_discount) + 0.5 * o_totalprice)** as score FROM orders, lineitem WHERE [...] GROUP BY **o_orderkey** ORDER BY score DESC **LIMIT 10**
$Q13$	SELECT **c_custkey, sum(0.5 * c_acctbal + 0.5 * o_totalprice) as score** FROM customer, orders WHERE [...] GROUP BY **c_custkey** ORDER BY **score** DESC **LIMIT 10**
$Q16$	SELECT p_brand, p_type, p_size, **sum(0.5 * ps_supplycost + 0.5 * p_retailprice) as value** FROM partsupp, part WHERE [...] GROUP BY p_brand, p_type, p_size ORDER BY **value**, p_brand, p_type, p_size DESC **LIMIT 10**
$Q18$	SELECT c_name, p_custkey, p_orderkey, o_orderdate, o_totalprice, sum**(0.5 * l_extendedprice * (1 - l_discount) + 0.5 * o_totalprice) as value** FROM customer, orders, lineitem WHERE [...] GROUP BY c_name, c_custkey, o_orderkey, o_orderdate, o_totalprice ORDER BY **value**, o_totalprice, o_orderdate DESC 10
$Q20$	SELECT s_name, s_address, **sum(0.5 * s_acctbal + 0.5 * ps_supplycost) as value** FROM supplier, nation WHERE [...] ORDER BY s_name, **value** DESC **LIMIT 10**

Table 4.2. Modified TPC-H Queries (modified parts are in bold face; [...] means that clause is exactly the same as the original TPC-H queries)

Table 4.3. Parameter Setting

Parameter	Range
Data size	1G, 2G, **3G**, 4G, 5G
T%	10%, 5%, 1%, **0.5%**, 0.1%
Pr	0.1, 0.3, 0.5, **0.7**, 0.9
Preference option	PMSPJ, PMW, PMK, **NM**
k_o	5, **10**, 50, 100
r_o	**101**, 501, 1001
$\|Y\|$	**1**, 2, 3, 4, 5

values are in bold faces. The default weighting is $\vec{w}_o = |\frac{1}{d} \ \ldots \ \frac{1}{d}|$, where d is the number of attributes in the ranking function. By default, the why-not question asks for a missing tuple/group that is ranked $(10 * k_o + 1)$-th under $\vec{w}_o$.

Effectiveness of Optimization Techniques. In this experiment, we investigate the effectiveness of (i) excluding unnecessary selection condition (Section 4.1.3.1), (ii) the early stopping (Section 4.1.3.2) and (iii) skipping (Section 4.1.3.3) used in our algorithm. Figure 4.7 shows the performance of our algorithm using only (i), only (ii), only (iii), all, and none, under the default setting. The effectiveness of the techniques is very promising when they are applicable (in particular, Theorem 1 is not used when the queries contain GROUP-BY). Without using any optimization techniques, the algorithm requires a running time of roughly 1000 seconds on these TPC-H queries. However, our algorithm runs about two to three orders faster when our optimization techniques are all enabled.

Varying data size. Figure 4.8(a) shows the running time of our algorithm under different data size (i.e., scale factor of TPC-H). We can see our algorithm for answering why-not questions scales linearly with the data size for all queries.

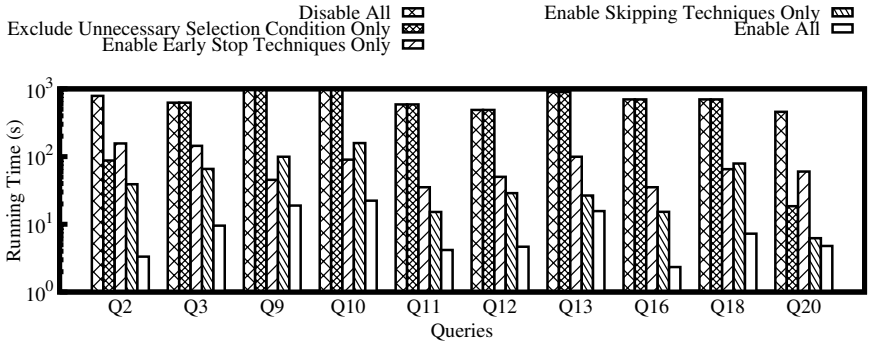
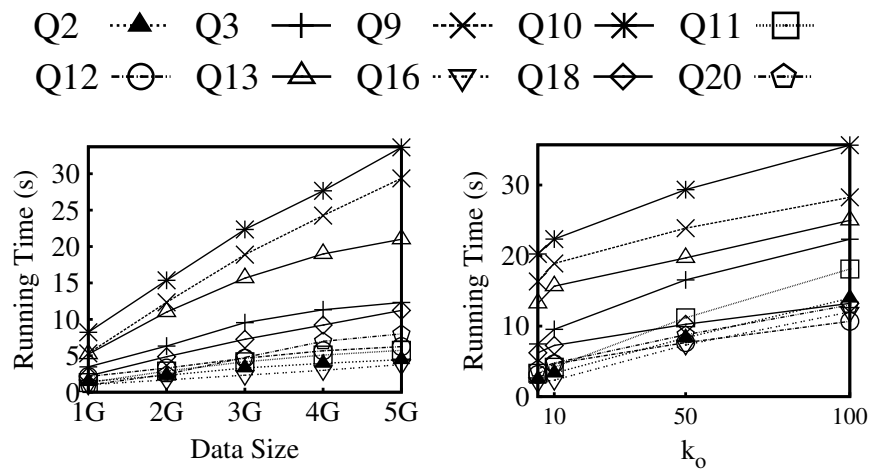


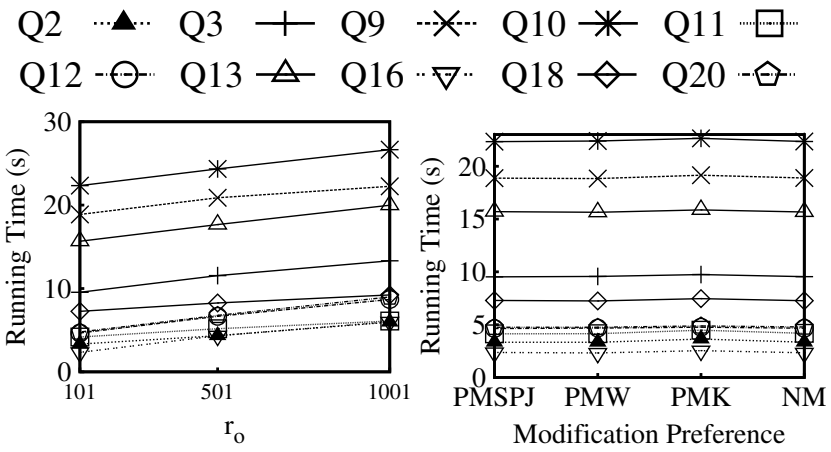
Figure 4.7. Effectiveness of optimization techniques



(a) Varying data size (b) Varying k_o

Figure 4.8. Varying parameters

Varying k_o. Figure 4.8(b) shows the running time of our algorithm using top-k SQL 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 tuple 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 tuple 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 SQL operation also increases with k_o. Figure 4.8(b) shows that our algorithm scales well with k_o.



(a) Varying r_o (b) Varying preference

Figure 4.9. Varying parameters

Varying the missing tuple to be inquired. We next study the performance of our algorithm by posing why-not questions with missing tuples from different rankings (i.e., r_o). In this experiment, we set $k_o = 10$ and asked three individual why-not questions about why the tuple that ranked 101st, 501st, and 1001st, respectively, is missing in the result. Figure 4.9(a) shows that our algorithm scales well with the ranking of the missing tuple. Of course, when the missing tuple has a worse ranking under the original weighting $\vec{w_o}$, the progressive top-k SQL operation should take a longer time to discover it in the result and thus the overall running time must increase.

Varying preference option. We next study the performance of algorithm under different user preference on changing SPJA constructs, k and $\vec{w}$. These values can be system parameters or user specified as stated in Section 4.1.1. In Figure 4.9(b), it is good to show that our algorithm is insensitive to various preference options. In all cases, our algorithm can return answers very efficiently.

Varying the number of missing tuples $|Y|$. We also study the perfor-

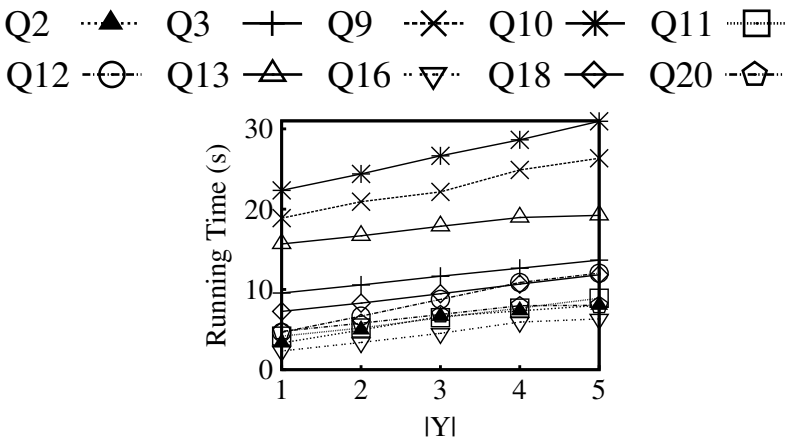
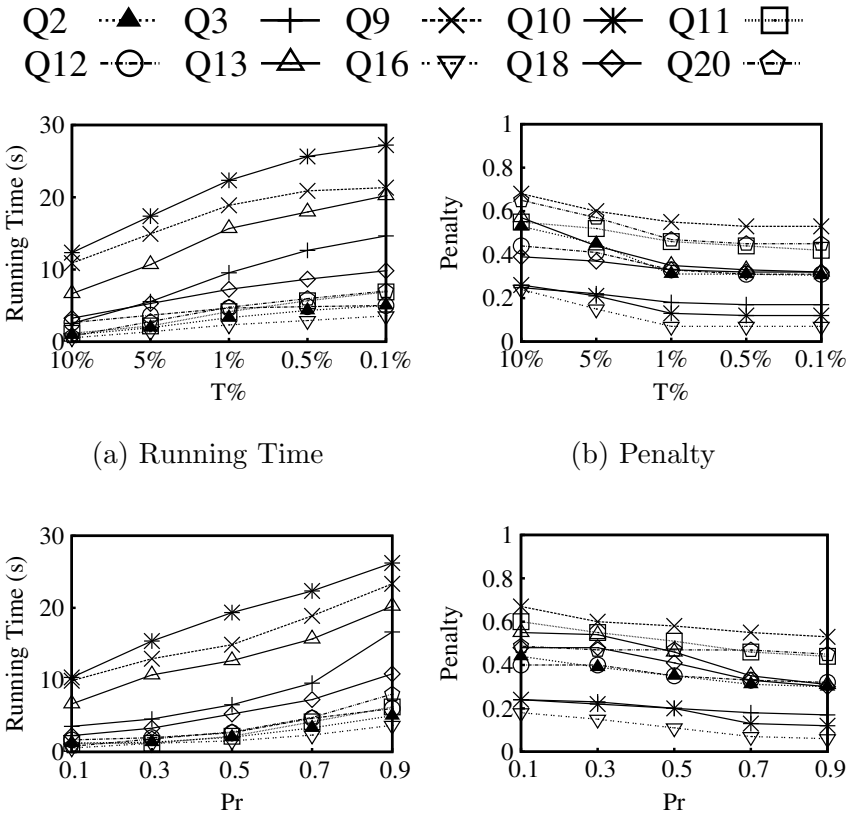


Figure 4.10. Varying parameters

mance of our algorithm by posing why-not questions with different number of missing tuples. In this experiment, a total of five why-not questions are asked for each TPC-H query. In the first question, one missing tuple that ranked 101-th under $\vec{w}_o$ is included in Y. In the second question, two missing tuples that respectively ranked 101-th and 201-th under $\vec{w}_o$ are included in Y. The third to the fifth questions are constructed similarly. Figure 4.10 shows that our algorithm scales linearly with respect to different size of Y.

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. Figures 4.11(a) and 4.11(b) show 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 Figure 4.11(a), we can see that the running time of our algorithm increases when the guarantee is more stringent. However, from Figure 4.11(b), we can see that the solution quality of the algorithm improves when T increases, until T reaches 1% where



(a) Running Time (b) Penalty

(c) Running Time (d) Penalty

Figure 4.11. Varying $T\%$ or Pr

further increases the sample size cannot yield any significant improvement.

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 Figures 4.11(c) and 4.11(d), 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.

4.4 Chapter Summary

In this chapter, we have studied the problem of answering why-not questions on top-k SQL queries. Our target is to give an explanation to a user who is wondering why her expected answers are missing in the query result. We return to the user a refined query that can include the missing expected answers back to the result. Our case studies and experimental results show that our solutions efficiently return very high quality solutions.

Chapter 5

Why-Not Dominating Question

Different from a top-k query, a dominating query [38] 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).

In this chapter, we define a hybrid explanation model that combines query refinement and data refinement to answer "why-not" dominating questions. We show that finding the best explanation is actually computational expensive. Afterwards, we present efficient evaluation algorithms that can obtain the best approximate explanation 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.

5.1 Preliminary

5.1.1 Problem Statement

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 [18, 19] here. That is, we may either adjust the value of k, the values of $\vec{m}_1, \cdots, \vec{m}_j$, or both[1].

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:

$$Penalty(k', M') = \lambda_k \frac{\Delta k}{r_o - k_o} + \lambda_c \frac{\Delta c}{\sum_{i=1}^{j} \|\vec{m}_i - \vec{l}\|_2} \tag{5.1}$$

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

[1]Our data-refinement approach is slightly different from [19]—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.

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 5.1. Similar to why-not top-k questions, we prompt users to answer a simple multiple-choice question like the one in Figure 3.1 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).

5.1.2 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

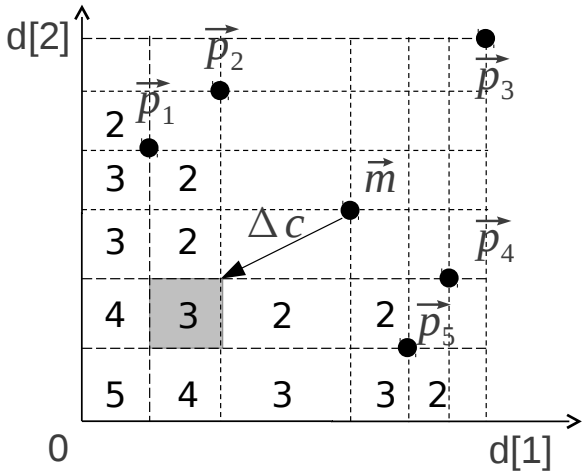


Figure 5.1. An example data space with grids

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 5.1 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 5.1, 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 $\vec{p}_2, \vec{p}_3, \vec{p}_4$).[2] 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 5.1 makes $\vec{m}$ rank 1-st, with the highest score 3, so $\Delta k = 0$). (2) Since we hope to

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

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 5.1). (3) Finally, the best answer is the corner of the grid whose Δk and Δc minimize Equation 5.1.

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.

5.2 Methodology

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

5.2.1 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 3.2.1). 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 query evaluation algorithm (e.g., [34]) 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.

5.2.2 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 5.1.1) and upper bound value of dimension i is $m[i]$. Figure 5.2 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 $\vec{m}$.

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 5.2, 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-

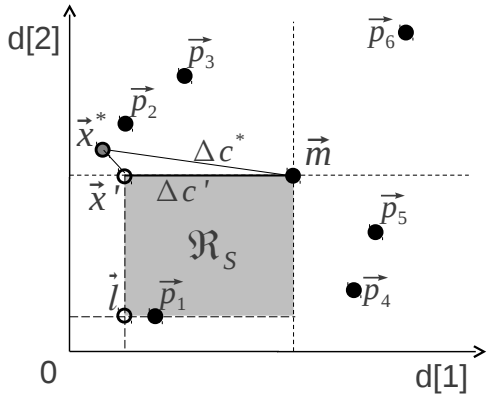


Figure 5.2. Restricted sample space $\mathcal{R}_s$

query-modified-value pair with least penalty, is located in $\mathcal{R}_s$:

Theorem 5.1 *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:

$$x'[i] = \quad m[i] \quad \text{if } x^*[i] > m[i]; \tag{5.2}$$

$$x'[i] = \quad l[i] \quad \text{if } x^*[i] < l[i]; \tag{5.3}$$

$$x'[i] = \quad x^*[i] \quad \text{if } l[i] \leq x^*[i] \leq m[i]; \tag{5.4}$$

For example, Figure 5.2 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 $x'[i] \leq x^*[i]$ in all dimensions, then $\vec{x}'$ certainly has a score no worse than $\vec{x}^*$, since it equals or dominates $\vec{x}^*$;

(2) If $x'[j] \leq x^*[j]$ 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:

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

$\forall i \in \mathcal{D}', x^*[i] \leq p[i] < x'[i] = l[i]$ (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. □

5.2.3 How large the list of sample values 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 3.7 to determine the sample size.

5.2.4 Algorithm

The algorithm is based on the basic idea mentioned in Section 5.2.1, 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., [34]) 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 $(L, r_o) = \text{DOMINATING}(\text{UNTIL-SEE-}\vec{m})$. 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 3.2.4) 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., [33]) or by posing a simple range query on an R-tree [4]. 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: $r_i = \text{COMPUTE-RANK}(\vec{m}, \vec{x}_i)$.

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 3.2.4) 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 $\text{COMPUTE-RANK}(\vec{m}, \vec{x}_i)$ operation can be entirely skipped.

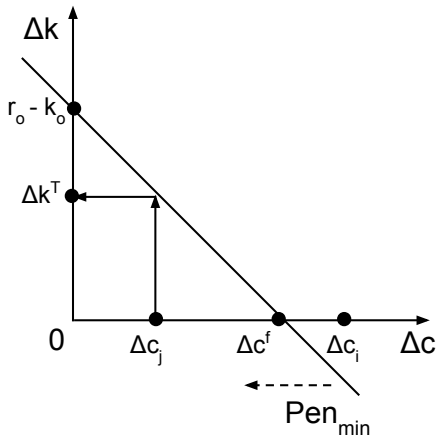


Figure 5.3. Example of answer space

First, let the original k value be k_o. In PHASE-1, we computed one candidate answer during the $(L, r_o) = \text{DOMINATING}(\text{UNTIL-SEE-}\vec{m})$ 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 5.3 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 Pen_{min} seen so far, and thus the corresponding COMPUTE-RANK operation can be skipped. Once again, when a better sample (candidate answer) is found, Pen_{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 5.1).

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 5.3) and obtain a threshold Δk^T and a threshold ranking r^T:

$$\begin{aligned} \Delta k^T &= \lfloor (Pen_{min} - \lambda_c \frac{\Delta c_j}{\|\vec{l} - \vec{m}\|_2}) \frac{r_o - k_o}{\lambda_k} \rfloor \\ r^T &= \Delta k^T + k_o \end{aligned} \tag{5.5}$$

r^T is the minimum ranking that $\vec{x}_j$ should achieve; otherwise, its penalty cannot be better than the current best answer. With r^T, we try to see if any

processed sample $\vec{x}_i$ dominates $\vec{x}_j$ in the data space. If yes, it means $\vec{x}_i$'s score and rank are no worse than $\vec{x}_j$, hence, $r_i \leq r_j$. Thus, if $r_i \geq r^T$, then $r_j \geq r^T$, so, $\vec{x}_j$ can be pruned because its penalty cannot be better than the penalty Pen_{min} achieved by the current best answer. The pseudocode of the algorithm is shown in Algorithm 5.1.

5.2.5 Multiple Missing Objects

To deal with multiple missing objects $M = \{\vec{m}_1, ..., \vec{m}_j\}$ in a why-not question, we can modify the algorithm as follows.

First, the initial dominating query stops only when all missing objects in M are seen, i.e., replace line 6 in Algorithm 5.1 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 3.7.

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.

Algorithm 5.1 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: $\langle Q'(k'), \vec{m} = \vec{x}_{best} \rangle$

Result list L of the first dominating query;
Rank r_o of the missing object;
Phase 1:
$(L, r_o) \leftarrow$ DOMINATING(UNTIL-SEE-$\vec{m}$);
if $r_o = \emptyset$ **then**
  return "$\vec{m}$ **is not in** D**"**;
end if
Determine s from $T\%$ and Pr using Equation 3.7;
Compute the lower bound point $\vec{l}$;
Find the restricted sample space $\mathcal{R}_s$ that bounded by $\vec{l}$ and $\vec{m}$;
Sample s points from $\mathcal{R}_s$ and add them into S;
Phase 2:
$Pen_{min} \leftarrow Penalty(r_o, \vec{m})$;
$\Delta c^f = Pen_{min} * \frac{\|\vec{l}-\vec{m}\|_2}{\lambda_c}$; //Find the Δc-intercept
$B \leftarrow \emptyset$; //The buffer stores the set of samples $\vec{b}$ with their ranking value r_b
for all $\vec{x}_i \in S$ **do**
  if $\Delta c_i > \Delta c^f$ **then**
    continue; //Technique (b) — skip COMPUTE-RANK operation
  end if
  $\Delta k^T \leftarrow \lfloor (Pen_{min} - \lambda_c \frac{\Delta c_i}{\|\vec{l}-\vec{m}\|_2}) \frac{r_o - k_o}{\lambda_k} \rfloor$;
  $r^T \leftarrow \Delta k^T + k_o$;
  if there exist objects $\vec{b} \in B$, such that $\vec{b}$ dominates $\vec{x}_i$ and $r_b \geq r^T$ **then**
    continue; //Technique (b) — use cached result to skip COMPUTE-RANK operation
  end if
  $r_i \leftarrow$ COMPUTE-RANK$(\vec{m}, \vec{x}_i)$; //Technique (a)
  $B \leftarrow B \cup (\vec{x}_i, r_i)$;
  if $Pen_i < Pen_{min}$ **then**
    $Pen_{min} \leftarrow Pen_i$;
    $\Delta c^f = Pen_{min} * \frac{\|\vec{l}-\vec{m}\|_2}{\lambda_c}$;
  end if
end for
Phase 3:
Return the $\langle k' = r_i, \vec{m} = \vec{x}_i \rangle$ pair whose penalty=Pen_{min};

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.

5.3 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.

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

5.3.1 Case Study

Case 1 (Finding the top-3 centers in NBA history). The following shows the result of a top-3 dominating query Q_1 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_1} 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_2 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_2 } 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_3 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: $\{\{\text{LeBron James, Kobe Bryant}\}, Q_3\}$ using the "Never Mind" option. Within 260ms, we got a refined top-4 query, with the following modified statistics of LeBron James and Kobe Bryant:

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	LeBron James	29	12	9	3	1	0.5	0.77	0.34
4	Kobe Bryant	25	6	5	2	1	0.5	0.85	0.5

The following are the original statistics of Lebron James and Kobe Bryant for readers' reference:

Player	PTS	REB	AST	STL	BLK	FG	FT	3PT
LeBron James	28	7	7	2	1	0.47	0.73	0.32
Kobe Bryant	25	5	5	2	1	0.45	0.83	0.34

The penalty of this answer is 0.16.

The naive sampling method return a refined answer with $k' = 7$ and penalty 3 times larger than the previous method.

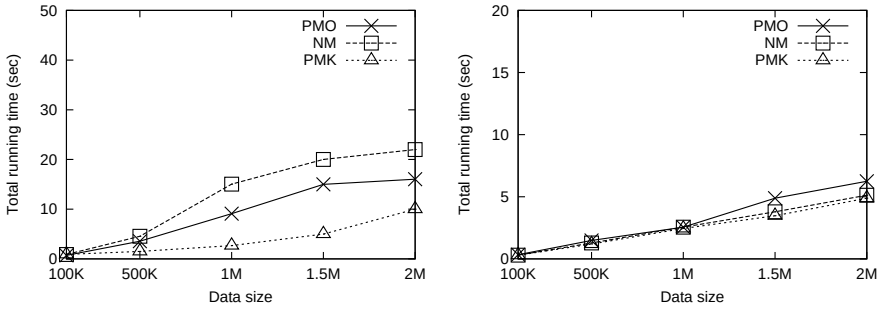
5.3.2 Performance

We next evaluate the performance of Algorithm 5.1. We present experimental results based on three types of synthetic data: uniform (UN), correlated

Table 5.1. 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

(CO) and anti-correlated (AC). Since, the experiment results between UN and CO are very similar, we only present the results of UN and AC here. Table 5.1 shows the parameters we varied in the experiments. The default values are in bold faces. By default, the why-not question asks for a missing object that is ranked $(10 * k_o + 1)$-th in the original database.

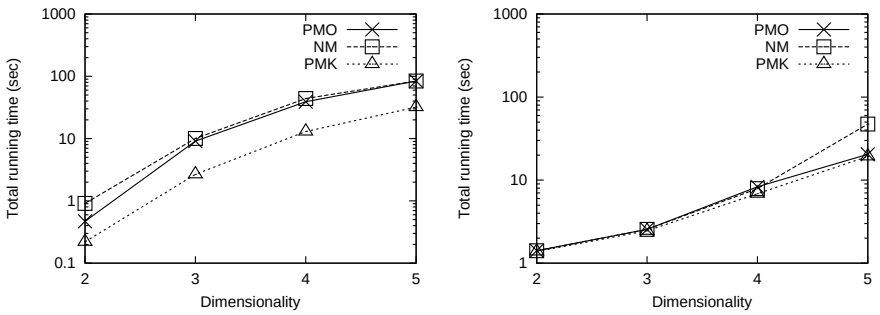


(a) Uniform Data (b) Anti-correlated Data

Figure 5.4. Varying data size vs. Time

Varying Data Size. Figure 5.4 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

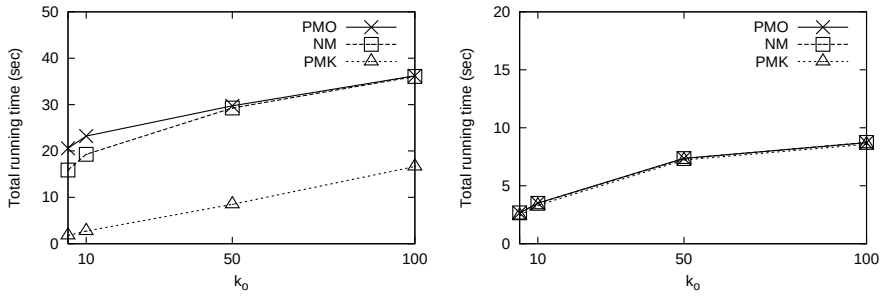
data is higher than on anti-correlated data in answering why-not dominating questions. In uniform data, objects with top rankings usually have large scores (i.e., dominating many objects), so it takes a relatively longer time to find a sample data value for $\vec{m}$ that makes it dominate those objects 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

Figure 5.5. Varying query dimension vs. Time

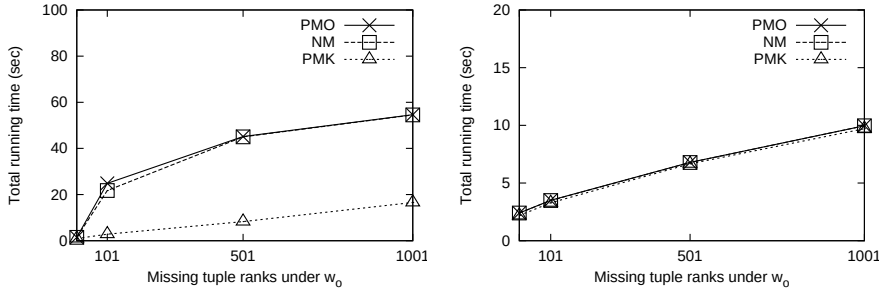
Varying Query Dimension. Figure 5.5 shows the running time of our algorithm of answering why-not questions on top-k dominating queries with different numbers of query dimensions. We can see that answering why-not questions for queries with more dimensions needs more time because the execution times of the DOMINATING function and the COMPUTE-RANK function increase with the number of dimensions.



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

Figure 5.6. Varying k_o vs. Time

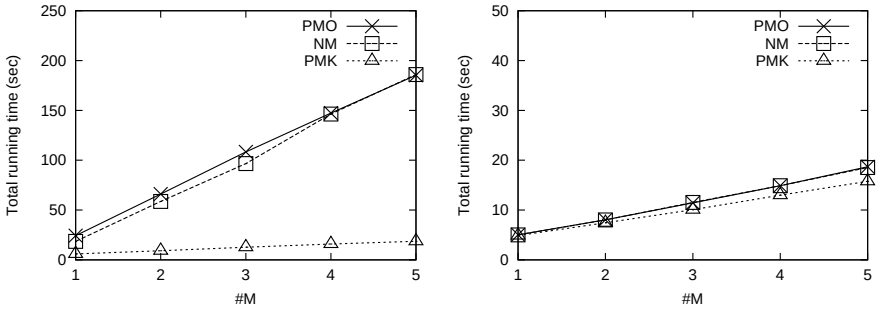
Varying k_o. Figure 5.6 shows the running time of our algorithm using top-k queries with different k_o values. Recall that when a top-5 query ($k_o = 5$) is used, the corresponding why-not question is to ask why the object in rank 51st is missing. And when a top-50 query ($k_o = 50$) is used, the corresponding 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 functions increase with k. Figure 5.6 shows that our algorithm scales well with k_o.



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

Figure 5.7. Varying the ranking of the missing object vs. Time

Varying the missing object to be inquired. Figure 5.7 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.

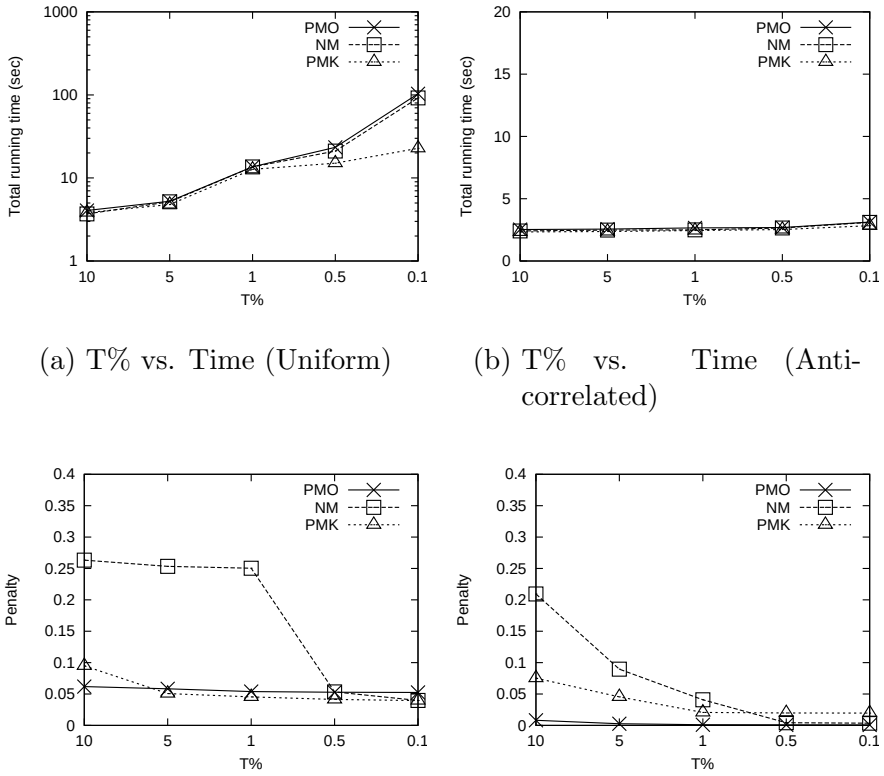


(a) Uniform Data (b) Anti-correlated Data

Figure 5.8. Varying $|M|$ vs. Time

Varying the size of $|M|$. Figure 5.8 shows the results when we vary the number of missing objects in a why-not question. In the first question, one missing object that ranked 101-th is included in M. In the second question, two missing objects that respectively ranked 101-th and 201-th are included in M. The third to the fifth questions are constructed similarly. 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 5.9 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



(a) T% vs. Time (Uniform)

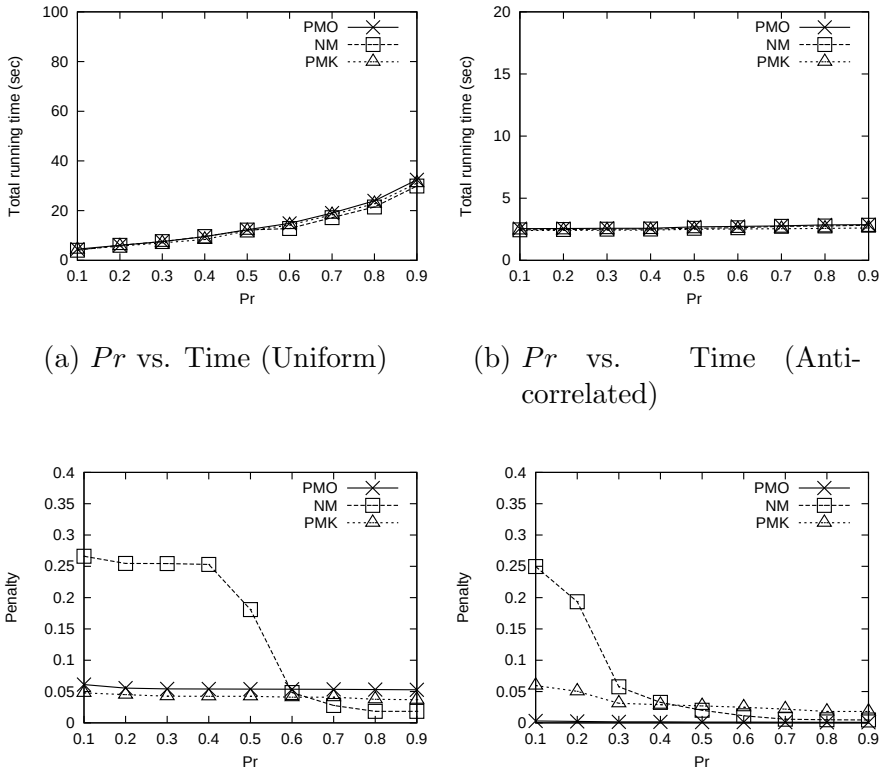
(b) T% vs. Time (Anti-correlated)

(c) T% vs. Penalty (Uniform)

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

Figure 5.9. Varying T% vs. Time/Penalty

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 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)

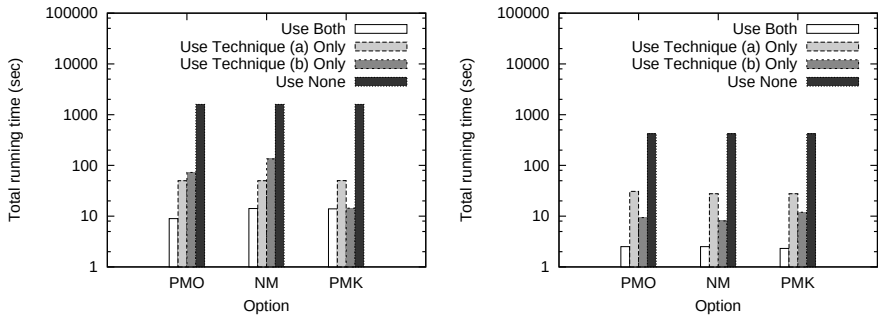
(c) Pr vs. Penalty (Uniform)

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

Figure 5.10. 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 5.10, 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

Figure 5.11. Optimization Effectiveness

Effectiveness of Optimization Techniques. Finally, we investigate the effectiveness of the two optimization techniques we used in our algorithm. Figure 5.11 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.

5.4 Chapter Summary

In this chapter, we have studied the problem of answering why-not questions on 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 this problem is different from answering why-not top-k questions and why-not top-k SQL questions, we use a different explanation model for top-k dominating queries. Specifically, 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.

Chapter 6

Conclusion

Answering why-not questions is a very useful feature to improve the usability of a database system. It can help users a lot to reason the query results and debug their queries, which greatly improves the user experience when they are interacting with the database system. As an important kind of database query, preference queries are urgently in need of this feature as well. In this dissertation, we have presented our explanation models and efficient algorithms for answering why-not questions on preference queries.

6.1 Contribution

The first contribution of this dissertation is the explanation models that we proposed for different preference queries. Identifying the best explanation model is not easy, since an unsuitably chosen model may not provide useful information to do reasoning at all. In answering why-not top-k question and why-not top-k

SQL question, the query-refinement approach we adopted captures the *similar* and *precise* requirements of refined queries, which makes the refined queries best match users' original intention. For the why-not dominating question, it is not enough to just refine the original query. Therefore, we propose a hybrid explanation model that combines both data-refinement and query-refinement to provide insightful explanations. Cases studies on real life data demonstrate the usefulness of our explanation models.

The second contribution of this dissertation is the new evaluation metrics that we proposed to quantify the quality of the explanations, namely, the penalty models for different preference queries. These metrics capture user's preferences on the changes of different aspects of refined queries, e.g. the value k and weightings at the same time. They help to formulate the complicated problems as optimization problems. They also guarantee that only the answer with the best quality will be returned to users.

The final contribution of this dissertation is the efficient algorithms that we proposed for answering why-not questions on different preference queries. In particular, we first study the complexity of each problem. Based on these analysis, we observe important properties from each problem, and use them to derive several useful theorems to exclude those answers with poor quality. Besides, we develop effective optimization techniques to significantly reduce the algorithms' running time. Extensive experiments on large synthetic data and real data show that our algorithms can return high quality answers efficiently.

6.2 Possibilities for Future Work

There are many interesting avenues for future work. One of the most important future work is to integrate our proposed methods into existing database systems. How to integrate our algorithms as access methods that are recognized by query optimizer would be an important topic.

Another important future work is to support Top-k SQL queries with more SQL constructs, like HAVING clause. A simple idea is to add two more edit operators to capture the modification of the HAVING clause: (e1) modifying the constant value of a having predicate; (e2) adding/removing a having predicate. In terms of the algorithm, we may need to adjust it a little bit. For example, when enumerating candidate answers, the algorithm needs to consider the extra having predicates such that it may early stop a PROGRESS operation as soon as it finds the missing group is being filtered.

The last important future work is to support preference queries with non-numeric attributes. In that case, user's preferences are not specified in numeric numbers. Instead, it is represented in more complex constraints such as "I like A better than B". It is an open question that what would be the best explanation model for this kind of preference queries. It may require us to come up with a totally new explanation model and also totally different algorithms.

Bibliography

[1] Parag Agrawal, Omar Benjelloun, Anish Das Sarma, Chris Hayworth, Shubha U. Nabar, Tomoe Sugihara, and Jennifer Widom. Trio: A system for data, uncertainty, and lineage. In *VLDB*, pages 1151–1154, 2006.

[2] S. Agrawal, S. Chaudhuri, and G. Das. DBXplorer: A System for Keyword-Based Search over Relational Databases. In *ICDE*, pages 5–16, 2002.

[3] J. Akbarnejad, G. Chatzopoulou, M. Eirinaki, S. Koshy, S. Mittal, D. On, N Polyzotis, and J. S. V. Varman. SQL QueRIE Recommendations. In *PVLDB*, volume 3, pages 1597–1600, 2010.

[4] 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*, page 1990, 322-331.

[5] Leonard D. Berkovitz. *Convexity and Optimization in* $\mathbb{R}^n$. A Wiley-Interscience pubication, 2002.

[6] Sourav S Bhowmick, Aixin Sun, and Ba Quan Truong. Why not, WINE?: Towards Answering Why-Not Questions in Social Image Search. In *MM*, pages 917–926, 2013.

[7] S. Borzsonyi, D. Kossmann, and K. Stockek. The Skyline Operator. *ACM Trans. Database System*, 25(2):129–178, 2000.

[8] Y. C. Chang, L. Bergman, V. Castelli, M.L. Lo C.S. Li, and J.R. Smith. The Onion Technique: Indexing for Linear Optimization Queries. In *SIGMOD*, page 2000, 391-402.

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

[10] Steven C. Chapra and Raymond P. Chanale. *Numerical methods for Engineers*. McGRAW-HILL, 2010.

[11] Lei Chen, Xin Lin, Haibo Hu, Christian S. Jensen, and Jianliang Xu. Answering Why-not Questions on Spatial Keyword Top-k Queries? In *ICDE*, 2015.

[12] Gao Cong, Christian S. Jensen, and Dingming Wu. Efficient Retrieval of the Top-k Most Relevant Spatial Web Objects. *VLDB*, 2(1):337–348, 2009.

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

[14] Yingwei Cui, Jennifer Widom, and Janet L. Wiener. Tracing the lineage of view data in a warehousing environment. *ACM Trans. Database Syst.*, 25(2):179–227, 2000.

[15] E. Dellis and B. Seeger. Efficient Computation of Reverse Skyline Queries. In *VLDB*, pages 291–302, 2007.

[16] R. Faginm, Amnon Lotem, and M. Naor. Optimal aggregation algorithm for middleware. *J. Comput. Syst. Sci.*, 64(4):614–656, 2003.

[17] Liu Fang, Yu Clement T., Meng Weiyi, and Chowdhury: Abdur. Effective keyword search in relational databases. In *SIGMOD*, pages 563–574, 2006.

[18] M. Herschel and M. A. Hernández. Explaining Missing Answers to SPJUA Queries. In *PVLDB*, pages 185–196, 2010.

[19] J. Huang, T. Chen, A-H. Doan, and J. F. Naughton. On the Provenance of Non-Answers to Queries over Extracted Data. In *PVLDB*, pages 736–747, 2008.

[20] I. F. Ilyas, G. Beskales, and M. A. Soliman. A survey of top-k processing techniques in relational database systems. *ACM Comput. Surv.*, 40(4), 2008.

[21] Ihab F. Ilyas, Walid G. Aref, and Ahmed K. Elmagarmid. Supporting top-k join queries in relational databases. *VLDB Journal*, 13(3):207–221, 2004.

[22] Ihab F. Ilyas, Rahul Shah, G Aref, Walid, Jeffrey Scott Vitter, and Ahmed K. Elmagarmid. Rank-aware Query Optimization. In *ACM SIGMOD*, pages 203–214, 2004.

[23] H.V. Jagadish, A. Chapman, A. Elkiss, M. Jayapandian, Y. Li, A. Nandi, and C. Yu. Making Database Systems Usable. In *SIGMOD*, pages 13–24, 2007.

[24] Zou L and L. Chen. Dominant Graph: An Efficient Indexing Structure to Answer Top-K Queries. In *ICDE*, page 2008, 536-545.

[25] Chengkai Li, Kevin Chen-Chuan Chang, and Ihab F Ilyas. Supporting ad-hoc ranking aggregates. In *ACM SIGMOD*, 2006.

[26] Qin Lu, Jeffrey Xu Yu, and Chang Lijun. Keyword Search in Databases: The Power of RDBMS. In *SIGMOD*, pages 681–694, 2009.

[27] Islam Md. Saiful, Zhou Rui, and Liu Chengfei. On Answering Why-not questions in Reverse Skyline Queries. In *ICDE*, 2013.

[28] A. Motro. Query Generalization: A Method for Interpreting Null Answers. In *Expert Database Workshop*, pages 597–616, 1984.

[29] A. Motro. SEAVE: A Mechanism for Verifying User Presuppositions in Query Systems. *ACM Trans. Inf. Syst.*, 4(4):312–330, 1986.

[30] M. Balazinska N. khoussainova, Y.C. Kwon and D. Suciu. Snipsuggest: Context-Aware Autocompletion for SQL. In *PVLDB*, volume 4, pages 22–33, 2010.

[31] D. Papadias, Tao Yufei, G. Fu, and B. Seeger. An Optimal and Progressive Algorithm for Skyline Queries. In *SIGMOD*, pages 467–478, 2003.

[32] R. Saigal. *Linear Programming: A Modern Integrated Analysis*. Springer, 1995.

[33] Y. F. Tao, X. K. Xiao, and J. Pei. Efficient Skyline and Top-k Retrieval in Subspaces. *IEEE Trans. Knowl. Data Eng.*, 19(8):1072–1088, 2007.

[34] Eleftherios Tiakas, Apostolos N. Papadopoulos, and Yannis Manolopoulos. Progressive processing of subspace dominating queries. *VLDB J.*, 20(6):921–948, 2011.

[35] Q. T. Tran and Chee-Yong Chan. How to ConQueR Why-not Questions. In *SIGMOD*, pages 15–26, 2010.

[36] A. Vlachou, C. Doulkeridis, Yannis Kotidis, and K. Nørvåg. Reverse Top-K Queries. In *ICDE*, page 2010, 365-376.

[37] H. Wu, Guoliang Li, C. Li, and Lizhu Zhou. Seaform: Search-As-You-Type in Forms. In *PVLDB*, volume 3, pages 1565–1568, 2010.

[38] Man Lung Yiu and Nikos Mamoulis. Efficent Processing of Top-k Dominating Queries on Multi-Dimensional Data. In *VLDB*, pages 541–552, 2007.

[39] F. Zhao, K-L Tan G. Das, and A. K. H. Tung. Call to Order: A Hierarchical Browsing Approach to Eliciting Users' Preference. In *SIGMOD*, page 2010, 27-38.