UNIVERSITY OF TEXAS AT ARLINGTON

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

PH.D. DISSERTATION

Algorithms For Building Compact Representatives And Processing Ranking Queries

By:

ABOLFAZL ASUDEH

Advisor:

DR. GAUTAM DAS

Committee Members:

DR. GERGELY ZÁRUBA

DR. CHENGKAI LI

DR. RAMEZ ELMASRI

16-OCT-2017

To my grandfather, Mohammad Asudeh.

تقدیم به پدربزرگ، محمد آسوده

ACKNOWLEDGEMENTS

I would like to thank my supervising professor Dr. Gautam Das for his constant support, patience, invaluable advice, brilliant ideas, and for teaching me the team work and research. I thank Dr. Gergely Zaruba who always took my back and supported me in past 5+ years, like a good friend. I also want to thank Dr. Changkai Li for the time he spent helping me, and Dr. Ramez Elmasri for his advice.

I want to extend my appreciation to my kind collaborators, especially Dr. Nan Zhang, Dr. Nick Koudas, and Dr. Divesh Srivastava, my current and past lab-mates, my teachers, and the CSE department for the support during my PhD.

I cannot be grateful enough to my parents who sacrificed their life for my success, my brother Omid, my sisters Zeinab and Bita, and my best friend and wife Hadis.

1 Abstract

Ranked retrieval model has rapidly replaced the traditional Boolean retrieval model as the de facto way for query processing when a large portion of (big) data matches a given query. Returning all the query results in these cases is not efficient nor informative. Unlike the Boolean retrieval model, the ranked retrieval model orders the matching tuples according to an often proprietary ranking function and returns the top-k of them. In this dissertation, we study ranked retrieval model and propose exact and approximate algorithms for (i) building representatives for fast query processing, and (ii) online processing of ranking queries. We study the problem both in the general cases and in the special environment of web databases, a natural fit for the ranked retrieval model.

We start the dissertation by building representatives that serve as indices for ranking query processing. A critical observation is that skyline, also known as Pareto-optimal, (resp. k sky-band) is a set that contains the top-1 (resp. top-k) for every possible ranking function following the monotonic order of attribute values. Thus, first, we study the problem crowdsourcing Pareto-optimal object finding, in the case where objects do not have explicit attributes and preference relations on objects are strict partial orders. Then, we initiate the research into the novel problem of skyline discovery over hidden web databases, which enables a wide variety of innovative third-party applications over one or multiple web databases.

A major problem with the ranking queries representatives, i.e., skyline and convex hull, is that as in real-world applications the representative can be a significant portion of the data, its performance in the ranking query processing is greatly reduced. Thus, computing a subset limited to r tuples that minimize the user's dissatisfaction with the result from the limited set is of interest. We make several fundamental theoretical as well as practical advances in developing such a compact set.

Finally, considering the limitations of top-k indices, while focusing on the client-server databases, we propose query reranking third-party service that uses public interface of the database to enable the on-the-fly processing of ranking queries.

Table of Contents

1 Abstract **3**

2 Introduction **8**

2.1 Pareto-Optimal Object Finding by Pairwise Comparisons 8

2.2 Discovering the Skyline of Web Databases 9

2.3 Regret-ratio Minimizing Set 9

2.4 Query Reranking Service 10

3 Pareto-Optimal Object Finding by Pairwise Comparisons **11**

3.1 General Framework 16

3.1.1 Question Selection 20

3.1.2 Resolving Unusual Contradictions in Question Outcomes . 24

3.2 Micro-Ordering in Question Selection 26

3.2.1 Random Question (RandomQ) 26

3.2.2 Random Pair (RandomP) 27

3.2.3 Pair with Fewest Remaining Questions (FRQ) 29

3.3 Experiments 33

3.3.1 Efficiency and Scalability 33

3.3.2 Experiments Using a Real Crowdsourcing Marketplace . . 37

3.4 Related Work 38

3.5 Final Remarks 39

4 Discovering the Skyline of Web Databases **41**

4.1 Preliminaries 45

4.1.1 Model of Hidden Database 45

4.1.2 Taxonomy of Attribute Search Interface 47

4.1.3 Problem Definition 49

4.2 Skyline Discovery for SQ-DB 50

4.2.1 Key Idea: Algorithm SQ-DB-SKY 52

4.2.2 Query-Cost Analysis 56

4.3 Skyline Discovery for RQ-DB 61

4.3.1 Key Idea: Algorithm RQ-DB-SKY . . . 62
4.3.2 Query-Cost Analysis . . . 66
4.4 Skyline Discovery for PQ-DB . . . 67
4.4.1 2D Case . . . 67
4.4.2 Higher-D Case: Negative Results . . . 71
4.4.3 Algorithm PQ-DB-SKY . . . 77
4.5 Skyline Discovery for Mixed-DB . . . 83
4.5.1 Overview . . . 84
4.5.2 Details for Leveraging Two-Ended Ranges . . . 85
4.5.3 Algorithm MQ-DB-SKY . . . 86
4.6 Extensions . . . 87
4.6.1 Anytime Property . . . 87
4.6.2 Sky Band . . . 88
4.7 Experimental Evaluation . . . 91
4.7.1 Experimental Setup . . . 91
4.7.2 Experiments over Real-World Dataset . . . 94
4.7.3 Online Demonstration . . . 97
4.8 Related Work . . . 99
4.9 Final Remarks . . . 100

5 Regret-ratio Minimizing Set: A Compact Maxima Representative 101
5.1 Motivation . . . 101
5.2 Technical Highlights . . . 103
5.2.1 Summary of Contributions . . . 104
5.3 Preliminaries . . . 105
5.3.1 Problem Definition . . . 108
5.4 2D Regret-ratio Minimizing Set . . . 108
5.4.1 Graph Modeling . . . 109
5.4.2 Baseline Solution . . . 113
5.4.3 Dynamic Programming Algorithm . . . 114
5.5 HD Regret-ratio Minimizing Set . . . 116
5.5.1 Problem with Existing Heuristic Solution . . . 116

5.5.2 Conceptual Model . . . 117
5.5.3 Matrix Discretization . . . 118
5.5.4 HD-RRMS Algorithm . . . 122
5.5.5 Practical HD-RRMS Algorithm . . . 125
5.6 Discussion . . . 127
5.6.1 Top-k Extension . . . 127
5.6.2 Alternative Matrix Discretization . . . 128
5.7 Experiments . . . 128
5.7.1 Experimental Setup . . . 128
5.7.2 Two-dimensional Experimental Result . . . 132
5.7.3 High-dimensional Experimental Result . . . 135
5.8 Related Work . . . 141
5.9 Final Remarks . . . 143

6 Query Reranking As A Service **144**
6.1 Preliminaries . . . 149
6.1.1 Database Model . . . 149
6.1.2 Problem Definition . . . 150
6.2 1D-RERANK . . . 152
6.2.1 Baseline Solution and Its Problem . . . 152
6.2.2 1D-RERANK . . . 155
6.3 MD-RERANK . . . 162
6.3.1 Problem with TA over 1D-RERANK . . . 163
6.3.2 MD-Baseline . . . 165
6.3.3 MD-Binary . . . 168
6.3.4 MD-RERANK . . . 173
6.4 discussions . . . 173
6.5 Experimental Evaluation . . . 175
6.5.1 Experimental Setup . . . 175
6.5.2 1D Experiments . . . 177
6.5.3 MD Experiments . . . 181
6.6 Related Work . . . 183

6.7 Final Remarks 184

7 List of Publications **185**

2 Introduction

In this dissertation, we present efficient algorithms for building compact representatives and processing ranking queries. When a large portion of (big) data matches a given query, returning all the query results in these cases is not efficient nor informative Thus, as a natural fit, ranked retrieval model has become the de facto way for query processing in these environments. This model orders the matching tuples according to an often proprietary ranking function and returns the top-k of them. Focusing on the ranked retrieval model, we propose exact and approximate algorithms for (i) building representatives for fast query processing, and (ii) online processing of ranking queries. We study the problem both in the general cases and in the special environment of web databases, a natural fit for the ranked retrieval model. The following are the studies problems in this dissertation.

2.1 Pareto-Optimal Object Finding by Pairwise Comparisons

Crowdsourcing Pareto-optimal object finding has applications in public opinion collection, group decision making, and information exploration. Departing from prior studies on crowdsourcing skyline and ranking queries, it considers the case where objects do not have explicit attributes and preference relations on objects are strict partial orders. The partial orders are derived by aggregating crowdsourcers' responses to pairwise comparison questions. The goal is to find all Pareto-optimal objects by the fewest possible questions. It employs an iterative question-selection framework. Guided by the principle of eagerly identifying non-Pareto optimal objects, the framework only chooses candidate questions which must satisfy three conditions. This design is both sufficient and efficient, as it is proven to find a short terminal question sequence. The framework is further steered by two ideas—macro-ordering and micro-ordering. By different micro-ordering heuristics, the framework is instantiated into several algorithms with varying power in pruning questions. Experiment results using both real crowdsourcing marketplace and simulations exhibited not only orders of magnitude reductions in questions when compared with a brute-force approach, but also close-to-optimal performance from

the most efficient instantiation.

2.2 Discovering the Skyline of Web Databases

Many web databases are "hidden" behind proprietary search interfaces that enforce the top-k output constraint, i.e., each query returns at most k of all matching tuples, preferentially selected and returned according to a proprietary ranking function. In this paper, we initiate research into the novel problem of skyline discovery over top-k hidden web databases. Since skyline tuples provide critical insights into the database and include the top-ranked tuple for every possible ranking function following the monotonic order of attribute values, skyline discovery from a hidden web database can enable a wide variety of innovative third-party applications over one or multiple web databases. Our research in the paper shows that the critical factor affecting the cost of skyline discovery is the type of search interface controls provided by the website. As such, we develop efficient algorithms for three most popular types, i.e., one-ended range, free range and point predicates, and then combine them to support web databases that feature a mixture of these types. Rigorous theoretical analysis and extensive real-world online and offline experiments demonstrate the effectiveness of our proposed techniques and their superiority over baseline solutions.

2.3 Regret-ratio Minimizing Set

Finding the maxima of a database based on a user preference, especially when the ranking function is a linear combination of the attributes, has been the subject of recent research. A critical observation is that the convex hull is the subset of tuples that can be used to find the maxima of any linear function. However, in real world applications the convex hull can be a significant portion of the database, and thus its performance is greatly reduced. Thus, computing a subset limited to r tuples that minimizes the regret ratio (a measure of the user's dissatisfaction with the result from the limited set versus the one from the entire database) is of interest. In this paper, we make several fundamental theoretical as well as practical advances

in developing such a compact set. In the case of two dimensional databases, we develop an optimal linearithmic time algorithm by leveraging the ordering of skyline tuples. In the case of higher dimensions, the problem is known to be NP-complete. As one of our main results of this paper, we develop an approximation algorithm that runs in linearithmic time and guarantees a regret ratio, within any arbitrarily small user-controllable distance from the optimal regret ratio. The comprehensive set of experiments on both synthetic and publicly available real datasets confirm the efficiency, quality of output, and scalability of our proposed algorithms.

2.4 Query Reranking Service

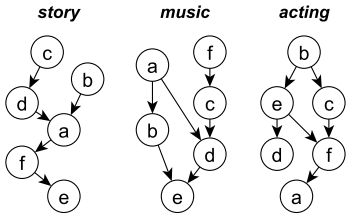
The ranked retrieval model has rapidly become the de facto way for search query processing in client-server databases, especially those on the web. Despite of the extensive efforts in the database community on designing better ranking functions/mechanisms, many such databases in practice still fail to address the diverse and sometimes contradicting preferences of users on tuple ranking, perhaps (at least partially) due to the lack of expertise and/or motivation for the database owner to design truly effective ranking functions. This paper takes a different route on addressing the issue by defining a novel *query reranking problem*, i.e., we aim to design a third-party service that uses nothing but the public search interface of a client-server database to enable the on-the-fly processing of queries with any user-specified ranking functions (with or without selection conditions), no matter if the ranking function is supported by the database or not. We analyze the worst-case complexity of the problem and introduce a number of ideas, e.g., on-the-fly indexing, domination detection and virtual tuple pruning, to reduce the average-case cost of the query reranking algorithm. We also present extensive experimental results on real-world datasets, in both offline and live online systems, that demonstrate the effectiveness of our proposed techniques.

3 Pareto-Optimal Object Finding by Pairwise Comparisons

The growth of user engagement and functionality in crowdsourcing platforms has made computationally challenging tasks unprecedentedly convenient. The subject of our study is one such task—crowdsourcing *Pareto-optimal object finding*. The concept of Pareto-optimal objects resembles that of *skyline objects* [1], but there are several critical differences which shall be discussed later in this section, § 3.4, and Table 4. Consider a set of objects O and a set of criteria C for comparing the objects. An object $\mathsf{x} \in O$ is *Pareto-optimal* if and only if x is not dominated by any other object, i.e., $\nexists \mathsf{y} \in O$ such that $\mathsf{y} \succ \mathsf{x}$. An object y dominates x (denoted $\mathsf{y} \succ \mathsf{x}$) if and only if x is not better than y by any criterion and y is better than x by at least one criterion, i.e., $\forall c \in C : \mathsf{x} \not\succ_c \mathsf{y}$ and $\exists c \in C : \mathsf{y} \succ_c \mathsf{x}$. If x and y do not dominate each other (i.e., $\mathsf{x} \not\succ \mathsf{y}$ and $\mathsf{y} \not\succ \mathsf{x}$), we denote it by $\mathsf{x} \sim \mathsf{y}$. The *preference (better-than) relation* P_c (also denoted $\succ_c$) for each $c \in C$ is a binary relation subsumed by $O \times O$, in which a tuple $(\mathsf{x}, \mathsf{y}) \in P_c$ (also denoted $\mathsf{x} \succ_c \mathsf{y}$) is interpreted as "$\mathsf{x}$ is better than (preferred over) y with regard to criterion c". Hence, if $(\mathsf{x}, \mathsf{y}) \notin P_c$ (also denoted $\mathsf{x} \not\succ_c \mathsf{y}$), x is not better than y by criterion c. We say x and y are *indifferent* regarding c (denoted $\mathsf{x} \sim_c \mathsf{y}$), if $(\mathsf{x}, \mathsf{y}) \notin P_c \wedge (\mathsf{y}, \mathsf{x}) \notin P_c$. We consider the setting where each P_c is a *strict partial order* as opposed to a bucket order [2] or a total order, i.e., P_c is irreflexive ($\forall \mathsf{x} : (\mathsf{x}, \mathsf{x}) \notin P_c$) and transitive ($\forall \mathsf{x}, \mathsf{y} : (\mathsf{x}, \mathsf{y}) \in P_c \wedge (\mathsf{y}, \mathsf{z}) \in P_c \Rightarrow (\mathsf{x}, \mathsf{z}) \in P_c$), which together imply asymmetry ($\forall \mathsf{x}, \mathsf{y} : (\mathsf{x}, \mathsf{y}) \in P_c \Rightarrow (\mathsf{y}, \mathsf{x}) \notin P_c$). We note that such definition of better-than relation has been widely used in modeling preferences (e.g., [3–5]).

Pareto-optimal object finding lends itself to applications in several areas, including public opinion collection, group decision making, and information exploration, exemplified by the following motivating examples.

Example 1 (Collecting Public Opinion and Group Decision Making)**.** *Consider a set of movies O=$\{$a,b,c,d,e,f$\}$ and a set of criteria C=$\{$*story, music, acting$\}$ (denoted by s, m, a in the ensuing discussion). Fig.1a shows the individual preference relations (i.e., strict partial orders), one per criterion. Each strict*



(a) Preference relations (i.e., strict partial orders) on three criteria.

QUESTION	ANSWER $\succ$	ANSWER $\sim$	ANSWER $\prec$	OUTCOME
$a ?_s b$	1	0	4	$b \succ_s a$
$a ?_s c$	0	0	5	$c \succ_s a$
$a ?_s d$	0	2	3	$d \succ_s a$
$a ?_s e$	4	0	1	$a \succ_s e$
$a ?_s f$	3	1	1	$a \succ_s f$
$b ?_s c$	1	2	2	$b \sim_s c$
$b ?_s d$	1	3	1	$b \sim_s d$
$b ?_s e$	5	0	0	$b \succ_s e$
$b ?_s f$	4	1	0	$b \succ_s f$
$c ?_s d$	3	2	0	$c \succ_s d$
$c ?_s e$	4	0	1	$c \succ_s e$
$c ?_s f$	3	1	1	$c \succ_s f$
$d ?_s e$	3	0	2	$d \succ_s e$
$d ?_s f$	3	2	0	$d \succ_s f$
$e ?_s f$	1	1	3	$f \succ_s e$

(b) Deriving the preference relation for criterion *story* by pairwise comparisons. Each comparison is performed by 5 workers. $\theta = 60\%$.

Figure 1: Finding Pareto-optimal movies by *story*, *music*, *acting*.

partial order is graphically represented as a directed acyclic graph (DAG), more specifically a Hasse diagram. The existence of a simple path from x *to* y *in the DAG means* x *is better than (preferred to)* y *by the corresponding criterion. For example,* $(a, e) \in P_m$ *(*$a \succ_m e$*), i.e.,* a *is better than* e *by* music. $(b, d) \notin P_s$ *and* $(d, b) \notin P_s$*; hence* $b \sim_s d$*. The partial orders define the dominance relation between objects. For instance, movie* c *dominates* d *(*$c \succ d$*), because* c *is preferred than* d *on* story *and* music *and they are indifferent on* acting*, i.e.,* $c \succ_s d$*,* $c \succ_m d$*, and* $c \sim_a d$*;* a *and* b *do not dominate each other (*$a \sim b$*), since* $b \succ_s a$*,* $a \succ_m b$ *and* $b \succ_a a$*. Based on the three partial orders,* b *is the only Pareto-optimal object, since no other objects dominate it and every other object is dominated by some object. Note that tasks such as the above one may be used in both understanding the public's preference*

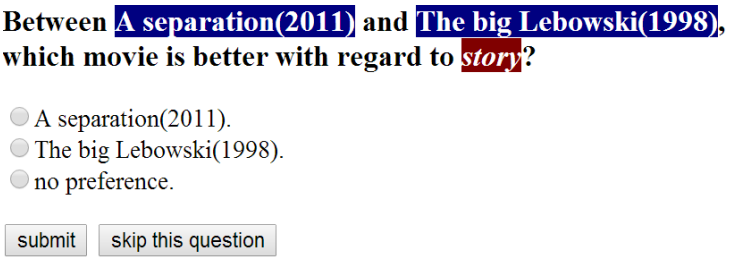


Figure 2: A question that asks to compare two movies by *story*.

(i.e., the preference relations are collected from a large, anonymous crowd) and making decisions for a target group (i.e., the preference relations are from a small group of people). □

Example 2 (Information Exploration)**.** *Consider a photography enthusiast,* Amy*, who is drown in a large number of photos she has taken and wants to select a subset of the better ones. She resorts to crowdsourcing for the task, as it has been exploited by many for similar tasks such as photo tagging, location/face identification, sorting photos by (guessed) date, and so on. Particularly, she would like to choose Pareto-optimal photos with regard to* color*,* sharpness *and* landscape*.* □

By definition, the crux of finding Pareto-optimal objects lies in obtaining the preference relations, i.e., the strict partial orders on individual criteria. Through crowdsourcing, the preference relations are derived by aggregating the crowd's responses to *pairwise comparison* tasks. Each such comparison between objects x and y by criterion c is a question, denoted $\mathsf{x}?_c\mathsf{y}$, which has three possible outcomes—$\mathsf{x}\succ_c\mathsf{y}$, $\mathsf{y}\succ_c\mathsf{x}$, and $\mathsf{x}\sim_c\mathsf{y}$, based on the crowd's answers. An example is as follows.

To the best of our knowledge, this paper is the first work on crowdsourcing Pareto-optimal object finding. The definition of Pareto-optimal objects follows the concept of *Pareto composition* of preference relations in [4]. It also resembles the definition of *skyline* objects on totally-ordered attribute domains (pioneered by [1]) and partially-ordered domains [5–8]. However, except for [9], previous studies on preference and skyline queries do not use the crowd; they focus on query processing on *existing* data. On the contrary, we examine how to ask the crowd as few questions as possible in obtaining sufficient data for determining

Pareto-optimal objects. Furthermore, our work differs from preference and skyline queries (including [9]) in several radical ways:

- The preference relation for a criterion is *not* governed by explicit scores or values on object attributes (e.g., sizes of houses, prices of hotels), while preference and skyline queries on both totally- and partially-ordered domains assumed explicit attribute representation. For many comparison criteria, it is difficult to model objects by explicit attributes, not to mention asking people to provide such values or scores; people's preferences are rather based on complex, subtle perceptions, as demonstrated in Examples 1 and 2.
- Due to the above reason, we request crowdsourcers to perform pairwise comparisons instead of directly providing attribute values or scores. On the contrary, [9] uses the crowd to obtain missing attribute values. Pairwise comparison is extensively studied in social choice and welfare, preferences, and voting. It is known that people are more comfortable and confident with comparing objects than directly scoring them, since it is easier, faster, and less error-prone [10].
- The crowd's preference relations are modeled as strict partial orders, as opposed to bucket orders or full orders. This is not only a direct effect of using pairwise comparisons instead of numeric scores or explicit attribute values, but also a reflection of the psychological nature of human's preferences [3, 4], since it is not always natural to enforce a total or bucket order. Most studies on skyline queries assume total/bucket orders, except for [5–8] which consider partial orders.

Our objective is to find all Pareto-optimal objects with as few questions as possible. A brute-force approach will obtain complete preference relations via pairwise comparisons of all object pairs by every criterion. However, without such exhaustive comparisons, incomplete knowledge collected from a small set of questions may suffice in discerning all Pareto-optimal objects. Toward this end, it may appear that we can take advantage of the transitivity of object dominance—a cost-saving property often exploited in skyline query algorithms (e.g., [1]) to exclude dominated objects from participating in any future comparison once they are detected. But, we shall prove that object dominance in our case is *not*

transitive (Property 1), due to the lack of explicit attribute representation. Hence, the aforementioned cost-saving technique is inapplicable.

Aiming at Pareto-optimal object finding by a short sequence of questions, we introduce a general, iterative algorithm framework (§ 3.1). Each iteration goes through four steps—*question selection*, *outcome derivation*, *contradiction resolution*, and *termination test*. In the i-th iteration, a question q_i=x?$_c$y is selected and its outcome is determined based on crowdsourcers' answers. On unusual occasions, if the outcome presents a contradiction to the obtained outcomes of other questions, it is changed to the closest outcome such that the contradiction is resolved. Based on the transitive closure of the outcomes to the questions so far, the objects O are partitioned into three sets—$O_\surd$ (objects that must be Pareto-optimal), $O_\times$ (objects that must be non-Pareto optimal), and $O_?$ (objects whose Pareto-optimality cannot be fully discerned by the incomplete knowledge so far). When $O_?$ becomes empty, $O_\surd$ contains all Pareto-optimal objects and the algorithm terminates. The question sequence so far is thus a *terminal sequence*.

There are a vast number of terminal sequences. Our goal is to find one that is as short as possible. We observe that, for a non-Pareto optimal object, knowing that it is dominated by at least one object is sufficient, and we do not need to find all its dominating objects. It follows that we do not really care about the dominance relation between non-Pareto optimal objects and we can skip their comparisons. Hence, the overriding principle of our question selection strategy is to identify non-Pareto optimal objects as early as possible. Guided by this principle, the framework only chooses from *candidate questions* which must satisfy three conditions (§ 3.1.1). This design is sufficient, as we prove that an empty candidate question set implies a terminal sequence, and vice versa (Proporty 2). The design is also efficient, as we further prove that, if a question sequence contains non-candidate questions, there exists a shorter or equally long sequence with only candidate questions that produces the same $O_\times$, matching the principle of eagerly finding non-Pareto optimal objects (Theorem 1). Moreover, by the aforementioned principle, the framework selects in every iteration such a candidate question x?$_c$y that x is more likely to be dominated by y. The selection is steered by two ideas—

macro-ordering and *micro-ordering*. By using different micro-ordering heuristics, the framework is instantiated into several algorithms with varying power in pruning questions (§ 3.2). We also derive a lower bound on the number of questions required for finding all Pareto-optimal objects (Theorem 2).

3.1 General Framework

By the definition of Pareto-optimal objects, the key to finding such objects is to obtain the preference relations, i.e., the strict partial orders on individual criteria. Toward this end, the most basic operation is to perform *pairwise comparison*—given a pair of objects x and y and a criterion c, determine whether one is better than the other (i.e., $(\mathsf{x}, \mathsf{y}) \in P_c$ or $(\mathsf{y}, \mathsf{x}) \in P_c$) or they are indifferent (i.e., $(\mathsf{x}, \mathsf{y}) \notin P_c \wedge (\mathsf{y}, \mathsf{x}) \notin P_c$).

The problem of crowdsourcing Pareto-optimal object finding is thus essentially crowdsourcing pairwise comparisons. Each comparison task between x and y by criterion c is presented to the crowd as a question q (denoted $\mathsf{x}?_c\mathsf{y}$). The outcome to the question (denoted $rlt(q)$) is aggregated from the crowd's answers. Given a set of questions, the outcomes thus contain an (incomplete) knowledge of the crowd's preference relations for various criteria. Fig.2 illustrates the screenshot of one such question (comparing two movies by *story*) used in our empirical evaluation. We note that there are other viable designs of question, e.g., only allowing the first two choices ($\mathsf{x} \succ_c \mathsf{y}$ and $\mathsf{y} \succ_c \mathsf{x}$). Our work is agnostic to the specific question design.

Given n objects and r criteria, a brute-force approach will perform pairwise comparisons on all object pairs by every criterion, which leads to $r \cdot n \cdot (n-1)/2$ comparisons. The corresponding question outcomes amount to the complete underlying preference relations. The quadratic nature of the brute-force approach renders it wasteful. The bad news is that, in the worst case, we cannot do better than it. To understand this, consider the scenario where all objects are indifferent by every criterion. If any comparison $\mathsf{x}?_c\mathsf{y}$ is skipped, we cannot determine if x and y are indifferent or if one dominates another.

In practice, though, the outlook is much brighter. Since we look for only Pareto-optimal objects, it is an overkill to obtain complete preference relations.

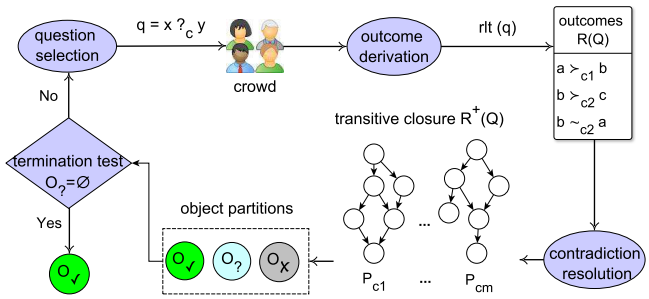


Figure 3: The general framework.

Specifically, for a Pareto-optimal object, knowing it is not dominated by any object is sufficient, and we do not need to find all the objects dominated by it; for a non-Pareto optimal object, knowing it is dominated by at least one object is sufficient, and we do not need to find all its dominating objects. Hence, without exhausting all possible comparisons, incomplete knowledge on preference relations collected from a set of questions may suffice in fully discerning all Pareto-optimal objects.

Our objective is to find all Pareto-optimal objects with as few questions as possible. By pursuing this goal, we are applying a very simple cost model—the cost of a solution only depends on its number of questions. Although the cost of a task in a crowdsourcing environment may depend on monetary cost, latency and other factors, the number of questions is a generic, platform-independent cost measure and arguably proportionally correlates with the real cost. Therefore, we assume a sequential execution model which asks the crowd an ordered sequence of questions $Q = \langle q_1, ..., q_n \rangle$—it only asks q_{i+1} after $rlt(q_i)$ is obtained. Thereby, we do not consider asking multiple questions concurrently. Furthermore, in discussion of our approach, the focus shall be on how to find a short question sequence instead of the algorithms' complexity.

To find a short sequence, we design a general algorithm framework, as displayed in Fig.3. Alg.1 shows the framework's pseudo-code. Its execution is iterative. Each iteration goes through four steps—*question selection*, *outcome derivation*, *contradiction resolution*, and *termination test*. In the i-th iteration, a question q_i=x?$_c$y is selected and presented to the crowd. The question outcome $rlt(q_i)$ is derived from the crowd's aggregated answers. On unusual occasions, if the

Algorithm 1 The general framework

Input. O : the set of objects

Output. $O_{\surd}$: Pareto-optimal objects of O

1: **while** $O_? = \{\}$ **do**
2: $\mathsf{x}?_c\mathsf{y} \leftarrow$ question selection
3: $rlt(\mathsf{x}?_c\mathsf{y}) \leftarrow$ outcome derivation// resolve conflict, if any
4: $R(Q) \leftarrow R(Q) \cup \{rlt(\mathsf{x}?_c\mathsf{y})\}$
5: $(O_{\surd}, O_{\times}, O_?) \leftarrow$ partitioning objects based on $R^+(Q)$// $R^+(Q)$ is the transitive closure of $R(Q)$
6: **end while**

outcome presents a contradiction to the obtained outcomes of other questions so far, it is changed to the closest outcome to resolve contradiction. By computing $R^+(Q_i)$, the *transitive closure* of $R(Q_i)$—the obtained outcomes to questions so far $\langle q_1, \ldots, q_i \rangle$, the outcomes to certain questions are derived and such questions will never be asked. Based on $R^+(Q_i)$, if every object is determined to be either Pareto-optimal or non-Pareto optimal without uncertainty, the algorithm terminates.

Below, we discuss outcome derivation and termination test. § 3.1.1 examines the framework's key step—question selection, and § 3.1.2 discusses contradiction resolution.

Outcome derivation Given a question $\mathsf{x}?_c\mathsf{y}$, its outcome $rlt(\mathsf{x}?_c\mathsf{y})$ must be aggregated from multiple crowdsourcers, in order to reach a reliable result with confidence. Particularly, one of three mutually-exclusive outcomes is determined based on k crowdsourcers' answers to the question:

$$rlt(\mathsf{x}?_c\mathsf{y}) = \begin{cases} \mathsf{x} \succ_c \mathsf{y} & \text{if } \frac{\#\mathsf{x}}{k} \geq \theta \\ \mathsf{y} \succ_c \mathsf{x} & \text{if } \frac{\#\mathsf{y}}{k} \geq \theta \\ \mathsf{x} \sim_c \mathsf{y}\ (\mathsf{x} \not\succ_c \mathsf{y} \wedge \mathsf{y} \not\succ_c \mathsf{x}) & \text{otherwise} \end{cases} \quad (1)$$

where θ is such a predefined threshold that $\theta{>}50\%$, $\#\mathsf{x}$ is the number of crowdsourcers (out of k) preferring x over y on criterion c, and $\#\mathsf{y}$ is the number of crowdsourcers preferring y over x on c. Fig.1b shows the outcomes of all 15 questions according to Equation (1) for comparing movies by *story* using $k{=}5$ and

θ=60%. Other conceivable definitions may be used in determining the outcome of $\mathsf{x}?_c\mathsf{y}$. For example, the outcome may be defined as the choice (out of the three possible choices) that receives the most votes from the crowd. The ensuing discussion is agnostic to the specific definition.

The current framework does not consider different levels of confidence on question outcomes. The confidence on the outcome of a question may be represented as a probability value based on the distribution of crowdsourcers' responses. An interesting direction for future work is to find Pareto-optimal objects in probabilistic sense. The confidence may also reflect the crowdsourcers' quality and credibility [11].

Termination test In each iteration, Alg.1 partitions the objects into three sets by their Pareto-optimality based on the transitive closure of question outcomes so far. If every object's Pareto-optimality has been determined without uncertainty, the algorithm terminates. Details are as follows.

Definition 1 (Transitive Closure of Outcomes)**.** *Given a set of questions* Q=$\langle q_1, ..., q_n\rangle$*, the transitive closure of their outcomes* $R(Q)$= $\{rlt(q_1), ..., rlt(q_n)\}$ *is* $R^+(Q)$=*{*$\mathsf{x}\sim_c\mathsf{y}$ | $\mathsf{x}\sim_c\mathsf{y} \in R(Q)$*}* $\bigcup$ *{*$\mathsf{x}\succ_c\mathsf{y}$ | $(\mathsf{x}\succ_c\mathsf{y} \in R(Q)) \vee (\exists\ \mathsf{w}_1,\mathsf{w}_2,...,\mathsf{w}_m :\ \mathsf{w}_1{=}\mathsf{x},\ \mathsf{w}_m{=}\mathsf{y} \wedge (\forall 0 < i < m :\ \mathsf{w}_i \succ_c \mathsf{w}_{i+1} \in R(Q)))$ *}*. □

In essence, the transitive closure dictates $\mathsf{x}\succ_c\mathsf{z}$ without asking the question $\mathsf{x}?_c\mathsf{z}$, if the existing outcomes $R(Q)$ (and recursively the transitive closure $R^+(Q)$) contains both $\mathsf{x}\succ_c\mathsf{y}$ and $\mathsf{y}\succ_c\mathsf{z}$. Based on $R^+(Q)$, the objects O can be partitioned into three sets:

$O_{\surd} = \{\mathsf{x} \in O \mid \forall \mathsf{y} \in O : (\exists c \in C : \mathsf{x}\succ_c\mathsf{y} \in R^+(Q)) \vee (\forall c \in C : \mathsf{x}\sim_c\mathsf{y} \in R^+(Q))\}$;

$O_{\times} = \{\mathsf{x} \in O \mid \exists \mathsf{y} \in O : (\forall c \in C : \mathsf{y}\succ_c\mathsf{x} \in R^+(Q) \vee \mathsf{x}\sim_c\mathsf{y} \in R^+(Q)) \wedge (\exists c \in C : \mathsf{y}\succ_c\mathsf{x} \in R^+(Q))\}$;

$O_? = O \backslash (O_{\surd} \cup O_{\times})$.

$O_{\surd}$ contains objects that must be Pareto-optimal, $O_{\times}$ contains objects that cannot possibly be Pareto-optimal, and $O_?$ contains objects for which the incomplete

knowledge $R^+(Q)$ is insufficient for discerning their Pareto-optimality. The objects in $O_?$ may turn out to be Pareto-optimal after more comparison questions. If the set $O_?$ for a question sequence Q is empty, $O_\surd$ contains all Pareto-optimal objects and the algorithm terminates. We call such a Q a *terminal sequence*, defined below.

Definition 2 (Terminal Sequence)**.** *A question sequence Q is a terminal sequence if and only if, based on $R^+(Q)$, $O_?$=$\varnothing$.*

3.1.1 Question Selection

Given objects O and criteria C, there can be a huge number of terminal sequences. Our goal is to find a sequence as short as possible. As Fig.3 and Alg.1 show, the framework is an iterative procedure of object partitioning based on question outcomes. It can also be viewed as the process of moving objects from $O_?$ to $O_\surd$ and $O_\times$. Once an object is moved to $O_\surd$ or $O_\times$, it cannot be moved again. With regard to this process, we make two important observations, as follows.

- *In order to declare an object x not Pareto-optimal, it is sufficient to just know x is dominated by another object*. It immediately follows that we do not really care about the dominance relationship between objects in $O_\times$ and thus can skip the comparisons between such objects. Once we know x$\in O_?$ is dominated by another object, it cannot be Pareto-optimal and is immediately moved to $O_\times$. Quickly moving objects into $O_\times$ can allow us skipping many comparisons between objects in $O_\times$.
- *In order to declare an object x Pareto-optimal, it is necessary to know that no object can dominate x.* This means we may need to compare x with all other objects including non Pareto-optimal objects. As an extreme example, it is possible for x to be dominated by only a non-Pareto optimal object y but not by any other object (not even the objects dominating y). This is because object dominance based on preference relations is intransitive, which is formally stated in Property 1.

  Property 1 (Intransitivity of Object Dominance)**.** *Object dominance based on the preference relations over a set of criteria is not transitive. Specifically, if x$\succ$y*

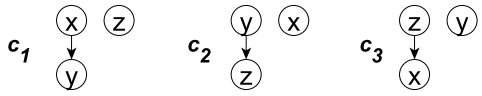


Figure 4: Intransitivity of object dominance: x≻y, y≻z, z≻x.

and y≻z, it is not necessarily true that x≻z. In other words, it is possible that x∼z or even z≻x. □

We show the intransitivity of object dominance by an example. Consider objects $O = \{\mathsf{x},\mathsf{y},\mathsf{z}\}$, criteria $C = \{c_1, c_2, c_3\}$, and the preference relations in Fig.4. Three dominance relationships violate transitivity: (i) $\mathsf{x} \succ \mathsf{y}$ (based on $\mathsf{x} \succ_{c_1} \mathsf{y}$, $\mathsf{x} \sim_{c_2} \mathsf{y}$, $\mathsf{x} \sim_{c_3} \mathsf{y}$), (ii) $\mathsf{y} \succ \mathsf{z}$ (based on $\mathsf{y} \sim_{c_1} \mathsf{z}$, $\mathsf{y} \succ_{c_2} \mathsf{z}$, $\mathsf{y} \sim_{c_3} \mathsf{z}$), and (iii) $\mathsf{z} \succ \mathsf{x}$ (based on $\mathsf{z} \sim_{c_1} \mathsf{x}$, $\mathsf{z} \sim_{c_2} \mathsf{x}$, $\mathsf{z} \succ_{c_3} \mathsf{x}$). As another example, in Fig.1a, $\mathsf{b} \succ \mathsf{c}$ (since $\mathsf{b} \sim_s \mathsf{c}$, $\mathsf{b} \sim_m \mathsf{c}$, $\mathsf{b} \succ_a \mathsf{c}$, where s =*story*, m =*music*, a =*acting*) and $\mathsf{c} \succ \mathsf{a}$ (since $\mathsf{c} \succ_s \mathsf{a}$, $\mathsf{c} \sim_m \mathsf{a}$, $\mathsf{c} \succ_a \mathsf{a}$), but $\mathsf{a} \sim \mathsf{b}$ (since $\mathsf{b} \succ_s \mathsf{a}$, $\mathsf{a} \succ_m \mathsf{b}$, $\mathsf{b} \succ_a \mathsf{a}$) where transitivity does not hold.

analysis [1]. The contradiction is due to the lack of explicit attribute representation—in our case two objects may be considered equally good on a criterion if they are indifferent, while in skyline analysis they are equally good regarding an attribute if they bear identical values. Skyline query algorithms exploit the transitivity of object dominance to reduce execution cost, because an object can be immediately excluded from further comparison once it is found dominated by any other object. However, due to Property 1, we cannot leverage such pruning anymore.

Based on these observations, the overriding principle of our question selection strategy (Alg.2) is to identify non-Pareto optimal objects as early as possible. At every iteration of the framework (Alg.1), we choose to compare x and y by criterion c (i.e., ask question $\mathsf{x}?_c\mathsf{y}$) where $\mathsf{x}?_c\mathsf{y}$ belongs to *candidate questions*. Such candidate questions must satisfy three conditions (Definition 3). There can be many candidate questions. In choosing the next question, by the aforementioned principle, we select such $\mathsf{x}?_c\mathsf{y}$ that x is more likely to be dominated by y. More specifically, we design two ordering heuristics—*macro-ordering* and *micro-ordering*. Given the three object partitions $O_\surd$, $O_\times$ and $O_?$, the macro-ordering idea is simply that we choose x from $O_?$ (required by one of the conditions on candidate questions) and y

Algorithm 2 Question selection
Input. $R^+(Q_i), O_{\surd}(Q_i), O_?(Q_i), O_\times(Q_i)$
Output. Q^1_{can} or Q^2_{can}

1: $Q_{can} \leftarrow \{\mathsf{x}?_c\mathsf{y} \mid rlt(\mathsf{x}?_c\mathsf{y}) \notin R^+(Q_i) \wedge \mathsf{x} \in O_?(Q_i) \wedge (\nexists c' \in C : \mathsf{x} \succ_{c'} \mathsf{y} \in R^+(Q_i))\}$
2: $Q^1_{can} \leftarrow \{\mathsf{x}?_c\mathsf{y} \mid \mathsf{x}?_c\mathsf{y} \in Q_{can}, \mathsf{y} \notin O_\times(Q_i)\}$
3: $Q^2_{can} \leftarrow \{\mathsf{x}?_c\mathsf{y} \mid \mathsf{x}?_c\mathsf{y} \in Q_{can}, \mathsf{y} \in O_\times(Q_i)\}$
4: `// Macro-ordering: consider` Q^1_{can} `before` Q^2_{can}.
5: **if** $Q^1_{can} \neq \varnothing$ **then**
6: **return micro-ordering**(Q^1_{can})
7: **else**
8: **return micro-ordering**(Q^2_{can})
9: **end if**

from $O_{\surd} \cup O_?$ (if possible) or $O_\times$ (otherwise). The reason is that it is less likely for an object in $O_\times$ to dominate x. Micro-ordering further orders all candidate questions satisfying the macro-ordering heuristic. In § 3.2, we instantiate the framework into a variety of solutions with varying power in pruning questions, by using different micro-ordering heuristics.

Definition 3 (Candidate Question)**.** *Given Q, the set of asked questions so far, $\mathsf{x}?_c\mathsf{y}$ is a* candidate question *if and only if it satisfies the following conditions:*

1. *The outcome of $\mathsf{x}?_c\mathsf{y}$ is unknown yet, i.e., $rlt(\mathsf{x}?_c\mathsf{y}) \notin R^+(Q)$;*
2. *x must belong to $O_?$;*
3. *Based on $R^+(Q)$, the possibility of $\mathsf{y} \succ \mathsf{x}$ must not be ruled out yet, i.e., $\nexists c' \in C : \mathsf{x} \succ_{c'} \mathsf{y} \in R^+(Q)$.*

We denote the set of candidate questions by Q_{can}. Thus, $Q_{can} = \{\mathsf{x}?_c\mathsf{y} \mid rlt(\mathsf{x}?_c\mathsf{y}) \notin R^+(Q) \wedge \mathsf{x} \in O_? \wedge (\nexists c' \in C : \mathsf{x} \succ_{c'} \mathsf{y} \in R^+(Q))\}$. □

If no candidate question exists, the question sequence Q is a terminal sequence. The reverse statement is also true, i.e., upon a terminal sequence, there is no candidate question left. This is formalized in the following property.

Property 2. *$Q_{can} = \varnothing$ if and only if $O_? = \varnothing$.*

Proof. The proof is omitted due to space limitations and can be found in the technical report [12]. □

Questions violating the three conditions may also lead to terminal sequences. However, choosing only candidate questions matches our objective of quickly identifying non-Pareto optimal objects. Below we justify the conditions.

Condition (1): This is straightforward. If $R(Q)$ or its transitive closure already contains the outcome of $\mathsf{x}?_c\mathsf{y}$, we do not ask the same question again.

Condition (2): This condition essentially dictates that at least one of the two objects in comparison is from $O_?$. (If only one of them belongs to $O_?$, we make it x.) Given a pair x and y, if neither is from $O_?$, there are three scenarios—(1) $\mathsf{x} \in O_\surd, \mathsf{y} \in O_\surd$, (2) $\mathsf{x} \in O_\surd, \mathsf{y} \in O_\times$ or $\mathsf{x} \in O_\times, \mathsf{y} \in O_\surd$, (3) $\mathsf{x} \in O_\times, \mathsf{y} \in O_\times$. Once we know an object is in $O_\surd$ or $O_\times$, its membership in such a set will never change. Hence, we are not interested in knowing the dominance relationship between objects from $O_\surd$ and $O_\times$ only. In all these three scenarios, comparing x and y is only useful for indirectly determining (by transitive closure) the outcome of comparing other objects. Intuitively speaking, such indirect pruning is not as efficient as direct pruning.

Condition (3): This condition requires that, when $\mathsf{x}?_c\mathsf{y}$ is chosen, we cannot rule out the possibility of y dominating x. Otherwise, if y cannot possibly dominate x, the outcome of $\mathsf{x}?_c\mathsf{y}$ cannot help prune x. Note that, in such a case, comparing x and y by c may help prune y, if y still belongs to $O_?$ and x may dominate y. Such possibility is not neglected and is covered by a different representation of the same question—$\mathsf{y}?_c\mathsf{x}$, i.e., swapping the positions of x and y in checking the three conditions. If it is determined x and y cannot dominate each other, then their further comparison is only useful for indirectly determining the outcome of comparing other objects. Due to the same reason explained for condition (2), such indirect pruning is less efficient.

The following simple Property 3 helps to determine whether $\mathsf{y} \succ \mathsf{x}$ is possible: If x is better than y by any criterion, then we can already rule out the possibility of $\mathsf{y} \succ \mathsf{x}$, without knowing the outcome of their comparison by every criterion. This allows us to skip further comparisons between them. Its correctness is straightforward based on the definition of object dominance.

Property 3 (Non-Dominance Property)**.** *At any given moment, suppose the set*

of asked questions is Q. Consider two objects x and y for which the comparison outcome is not known for every criterion, i.e., $\exists c$ such that $rlt(x?_c y) \notin R^+(Q)$. It can be determined that $y \not\succ x$ if $\exists c \in C$ such that $x \succ_c y \in R^+(Q)$. □

In justifying the three conditions in defining candidate questions, we intuitively explained that indirect pruning is less efficient—if it is known that x does not belong to $O_?$ or y cannot possibly dominate x, we will not ask question $x?_c y$. We now justify this strategy theoretically and precisely. Consider a question sequence $Q=\langle q_1, \ldots, q_n \rangle$. We use $O_\surd(Q)$, $O_?(Q)$, $O_\times(Q)$ to denote object partitions according to $R^+(Q)$. For any question q_i, the subsequence comprised of its preceding questions is denoted $Q_{i-1}=\langle q_1, \ldots, q_{i-1} \rangle$. If q_i was not a candidate question when it was chosen (i.e., after $R(Q_{i-1})$ was obtained), we say it is a *non-candidate*. The following Theorem 1 states that, if a question sequence contains non-candidate questions, we can replace it by a shorter or equally long sequence without non-candidate questions that produces the same set of dominated objects $O_\times$. Recall that the key to our framework is to recognize dominated objects and move them into $O_\times$ as early as possible. Hence, the new sequence will likely lead to less cost when the algorithm terminates. Hence, it is a good idea to only select among candidate questions.

Theorem 1. *If Q contains non-candidate questions, there exists a question sequence Q' without non-candidate questions such that $|Q'| \leq |Q|$ and $O_\times(Q') = O_\times(Q)$.*

Proof. We prove by demonstrating how to transform Q into such a Q'. Given any non-candidate question $q_i = x?_c y$ in Q, we remove it and, when necessary, replace several questions. The decisions and choices are partitioned into three mutually exclusive scenarios, which correspond to violations of the three conditions in Definition 3. The detailed proof is omitted due to space limitations and can be found in the technical report [12]. □

3.1.2 Resolving Unusual Contradictions in Question Outcomes

A preference relation can be more accurately derived, if more input is collected from the crowd. However, under practical constraints on budget and time, the

limited responses from the crowd (k answers per question) may present two types of contradicting preferences.

(i) Suppose $rlt(\mathsf{x}?_c\mathsf{y}) = \mathsf{x} \succ_c \mathsf{y}$ and $rlt(\mathsf{y}?_c\mathsf{z}) = \mathsf{y} \succ_c \mathsf{z}$ have been derived, i.e., they belong to $R(Q)$. They together imply $\mathsf{x} \succ_c \mathsf{z}$, since a preference relation must be transitive. Therefore the question $\mathsf{x}?_c\mathsf{z}$ will not be asked. If the crowd is nevertheless asked to further compare x and z, the result $rlt(\mathsf{x}?_c\mathsf{z})$ might be possibly $\mathsf{z} \succ_c \mathsf{x}$, which presents a contradiction.

(ii) Suppose $rlt(\mathsf{x}?_c\mathsf{y}) = \mathsf{x} \sim_c \mathsf{y}$ and $rlt(\mathsf{y}?_c\mathsf{z}) = \mathsf{y} \succ_c \mathsf{z}$ have been derived from the crowd. If the crowd is asked to further compare x and z, the result $rlt(\mathsf{x}?_c\mathsf{z})$ might be possibly $\mathsf{z} \succ_c \mathsf{x}$. The outcomes $\mathsf{y} \succ_c \mathsf{z}$ and $\mathsf{z} \succ_c \mathsf{x}$ together imply $\mathsf{y} \succ_c \mathsf{x}$, which contradicts with $\mathsf{x} \sim_c \mathsf{y}$. (A symmetric case is $rlt(\mathsf{x}?_c\mathsf{y}) = \mathsf{x} \sim_c \mathsf{y}$, $rlt(\mathsf{y}?_c\mathsf{z}) = \mathsf{z} \succ_c \mathsf{y}$, and the crowd might respond with $rlt(\mathsf{x}?_c\mathsf{z}) = \mathsf{x} \succ_c \mathsf{z}$, which also leads to contradiction with $\mathsf{x} \sim_c \mathsf{y}$. The following discussion applies to this symmetric case, which is thus not mentioned again.)

In practice, such contradictions are uncommon. This is easy to understand intuitively—as long as the underlying preference relation is transitive, collective wisdom of the crowds will reflect it. We can find evidence of it in [13, 14], which confirmed that preference judgments of relevance in document retrieval are transitive.

Nevertheless, contradictions still occur. Type (i) contradictions can be prevented by enforcing the following simple Rule 1 to assume transitivity and thus skip certain questions. They will never get into the derived preference relations. In fact, in calculating transitive closure (Definition 1) and defining candidate questions (§ 3.1.1), we already apply this rule.

Rule 1 (Contradiction Prevention by Skipping Questions)**.** *Given objects x, y, z and a criterion c, if $rlt(x?_c y) = x \succ_c y$ and $rlt(y?_c z) = y \succ_c z$, we assume $rlt(x?_c z) = x \succ_c z$ and thus will not ask the crowd to further compare x and z by criterion c.* □

To resolve type (ii) contradictions, we enforce the following simple Rule 2.

Rule 2 (Contradiction Resolution by Choosing Outcomes)**.** *Consider objects x, y, z and a criterion c. Suppose $rlt(x?_c y) = x \sim_c y$ and $rlt(y?_c z) = y \succ_c z$ are obtained from the crowd. If $rlt(x?_c z) = z \succ_c x$ is obtained from the crowd afterwards, we*

replace the outcome of this question by $x \sim_c z$*. (Note that we do not replace it by* $x \succ_c z$*, since* $z \succ_c x$ *is closer to* $x \sim_c z$*.)* □

3.2 Micro-Ordering in Question Selection

At every iteration of Alg.1, we choose a question $x?_c y$ from the set of candidate questions. By macro-ordering, when available, the question selection strategy (Alg.2) chooses a candidate question in which $y \notin O_\times$, i.e., it chooses from Q^1_{can}. Otherwise, it chooses from Q^2_{can}. The size of Q^1_{can} and Q^2_{can} can be large. Micro-ordering is for choosing from the many candidates. As discussed in § 3.1, in order to find a short question sequence, the overriding principle of our question selection strategy is to identify non-Pareto optimal objects as early as possible. Guided by this principle, this section discusses several micro-ordering strategies. Since the strategies are the same for Q^1_{can} and Q^2_{can}, we will simply use the term "candidate questions" without distinction between Q^1_{can} and Q^2_{can}.

3.2.1 Random Question (RandomQ)

RandomQ, as its name suggests, simply selects a random candidate question. Table 1 shows an execution of the general framework under RandomQ for Example 1. For each iteration i, the table shows the question outcome $rlt(q_i)$. Following the question form $x?_c y$ in Definition 3, the object "x" in a question is underlined when we present the question outcome. The column "derived results" displays derived question outcomes by transitive closure (e.g., $a \succ_m e$ based on $rlt(q_7)$=$d \succ_m e$ and $rlt(q_{10})$=$a \succ_m d$) and derived object dominance (e.g., $b \succ d$ after q_{20}). The table also shows the object partitions ($O_\surd$, $O_?$ and $O_\times$) when the execution starts and when the partitions are changed after an iteration. Multiple iterations may be presented together if other columns are the same for them.

As Table 1 shows, this particular execution under RandomQ requires 30 questions. When the execution terminates, it finds the only Pareto-optimal object b. This simplest micro-ordering strategy already avoids many questions in the brute-force approach. The example clearly demonstrates the benefits of choosing candidate

i	$rlt(q_i)$	Derived Results	$O_\surd$	$O_?$	$O_\times$
1-9	$b\succ_m e, c\sim_a d, a\sim_m c$ $c\succ_s e, b\sim_s d, b\succ_a a$ $d\succ_m e, b\sim_m d, b\succ_s f$		∅	{a,b,c,d,e,f}	∅
10	$a\succ_m d$	$a\succ_m e$			
11	$c\succ_a a$				
12	$b\sim_s c$				
13	$c\succ_m d$	$c\succ_m e$			
14-19	$d\succ_s e, e\sim_a c, d\sim_a f$ $a\sim_a d, f\succ_a a, b\succ_a e$				
20	$b\succ_a d$	$b\succ d$	∅	{a,b,c,e,f}	{d}
21-23	$c\succ_s f, a\succ_s e, f\sim_m b$				
24	$a\succ_s f$	$a\sim f$			
25	$e\succ_a f$	$b\succ_a f, b\succ_a a$ $e\succ_a a, b\succ f$	∅	{a,b,c,e}	{d,f}
26	$b\succ_a c$				
27	$a\succ_m b$				
28	$b\succ_s e$	$b\succ e$	∅	{a,b,c}	{d,e,f}
29	$c\succ_s a$	$c\succ_s e, c\succ a$	∅	{b,c}	{a,d,e,f}
30	$b\sim_m c$	$b\succ c$	{b}	∅	{a,c,d,e,f}

Table 1: RandomQ on Example 1.

questions only and applying macro-strategy.

3.2.2 Random Pair (RandomP)

RandomP randomly selects a pair of objects x and y and keeps asking questions to compare them ($x?_c y$ or $y?_c x$) until no such candidate question remains, upon which it randomly picks another pair of objects. This strategy echoes our principle of eagerly identifying non-Pareto optimal objects. To declare an object x non-Pareto optimal, we must identify another object y such that y dominates x. If we directly compare x and y, it requires comparing them by every criterion in C in order to make sure $y \succ x$. By skipping questions according to transitive closure, we do not need to directly compare them by every criterion. However, Property 4 below states that we still need at least $|C|$ questions involving x—some are direct comparisons with y, others are comparisons with other objects which indirectly lead to outcomes of comparisons with y. When there is a candidate question $x?_c y$, it means y may dominate x. The fewer criteria remain for comparing them, the more likely y will dominate x. Hence, by keeping comparing the same object pair, RandomP aims at finding more non-Pareto objects by less questions.

Property 4. *Given a set of criteria C and an object $x \in O$, at least $|C|$ pairwise*

i	$rlt(q_i)$	Derived Results	$O_{\surd}$	$O_?$	$O_\times$
1	$\mathsf{c} \succ_s \mathsf{f}$		$\varnothing$	{a,b,c,d,e,f}	$\varnothing$
2	$\mathsf{f} \succ_m \mathsf{c}$	$\mathsf{f} \sim \mathsf{c}$			
3 − 4	$\mathsf{a} \succ_s \mathsf{e}, \mathsf{a} \succ_m \mathsf{e}$				
5	$\mathsf{e} \succ_a \mathsf{a}$	$\mathsf{a} \sim \mathsf{e}$			
6 − 7	$\mathsf{c} \succ_s \mathsf{e}, \mathsf{c} \succ_m \mathsf{e}$				
8	$\mathsf{e} \sim_a \mathsf{c}$	$\mathsf{c} \succ \mathsf{e}$	$\varnothing$	{a,b,c,d,f}	{e}
9	$\mathsf{b} \succ_s \mathsf{a}$	$\mathsf{b} \succ_s \mathsf{e}$			
10	$\mathsf{a} \succ_m \mathsf{b}$	$\mathsf{a} \sim \mathsf{b}$			
11	$\mathsf{d} \succ_s \mathsf{f}$				
12	$\mathsf{f} \succ_m \mathsf{d}$	$\mathsf{f} \sim \mathsf{d}$			
13	$\mathsf{d} \succ_s \mathsf{a}$	$\mathsf{d} \succ_s \mathsf{e}$			
14	$\mathsf{a} \succ_m \mathsf{d}$	$\mathsf{a} \sim \mathsf{d}$			
15 − 16	$\mathsf{b} \sim_s \mathsf{c}, \mathsf{b} \sim_m \mathsf{c}$				
17	$\mathsf{b} \succ_a \mathsf{c}$	$\mathsf{b} \succ \mathsf{c}$	$\varnothing$	{a,b,d,f}	{c,e}
18 − 19	$\mathsf{d} \sim_s \mathsf{b}, \mathsf{d} \sim_m \mathsf{b}$				
20	$\mathsf{b} \succ_a \mathsf{d}$	$\mathsf{b} \succ \mathsf{d}$	$\varnothing$	{a,b,f}	{c,d,e}
21	$\mathsf{a} \succ_s \mathsf{f}$	$\mathsf{b} \succ_s \mathsf{f}$			
22	$\mathsf{a} \sim_m \mathsf{f}$				
23	$\mathsf{f} \succ_a \mathsf{a}$	$\mathsf{a} \sim \mathsf{f}$			
24	$\mathsf{b} \sim_m \mathsf{f}$				
25	$\mathsf{b} \succ_a \mathsf{f}$	$\mathsf{b} \succ_a \mathsf{a}, \mathsf{b} \succ \mathsf{f}$	{b}	{a}	{c,d,e,f}
26 − 27	$\mathsf{c} \succ_s \mathsf{a}, \mathsf{a} \sim_m \mathsf{c}$				
28	$\mathsf{c} \succ_a \mathsf{a}$	$\mathsf{c} \succ \mathsf{a}$	{b}	$\varnothing$	{a,c,d,e,f}

Table 2: RandomP on Example 1.

comparison questions involving x are required in order to find another object y such that y≻x.

Proof. By the definition of object dominance, if y≻x, then $\forall c \in C$, either y$\succ_c$x$\in R^+(Q)$ or x$\sim_c$y$\in R^+(Q)$, and $\exists c \in C$ such that y$\succ_c$x $\in R^+(Q)$. Given any particular c, if x$\sim_c$y $\in R^+(Q)$, then x$\sim_c$y $\in R(Q)$, i.e., a question x$?_c$y or y$?_c$x belongs to the sequence Q, because indifference of objects on a criterion cannot be derived by transitive closure. If y$\succ_c$x$\in R^+(Q)$, then y$\succ_c$x$\in R(Q)$ or $\exists$ w$_1, \ldots,$ w$_m \in O$ such that y$\succ_c$w$_1 \in R(Q), \ldots,$ w$_i\succ_c$w$_{i+1} \in R(Q), \ldots,$ w$_m\succ_c$x$\in R(Q)$. Either way, at least one question involving x on each criterion c is required. Thus, it takes at least $|C|$ questions involving x to determine y≻x. □

Table 2 illustrates an execution of RandomP for Example 1. The initial two questions are between c and f. Afterwards, it is concluded that c∼f by Property 3. Therefore, RandomP moves on to ask 3 questions between a and e. In total, the execution requires 28 questions. Although it is shorter than Table 1 by only 2 questions due to the small size of the example, it clearly moves objects into $O_\times$

more quickly. (In Table 1, $O_\times$ is empty until the 20th question. In Table 2, $O_\times$ already has 3 objects after 20 questions.) The experiment results in § 3.3 exhibit significant performance gain of RandomP over RandomQ on larger data.

3.2.3 Pair with Fewest Remaining Questions (FRQ)

Similar to RandomP, once a pair of objects x and y are chosen, FRQ keeps asking questions between x and y until there is no such candidate questions. Different from RandomP, instead of randomly picking a pair of objects, FRQ always chooses a pair with the fewest remaining questions. There may be multiple such pairs. To break ties, FRQ chooses such a pair that x has dominated the fewest other objects and y has dominated the most other objects. Furthermore, in comparing x and y, FRQ orders their remaining questions (and thus criteria) by how likely x is worse than y on the criteria. Below we explain this strategy in more detail.

Selecting Object Pair Consider a question sequence Q_i so far and FRQ is to select the next question Q_{i+1}. We use $C_{\mathsf{x},\mathsf{y}}$ to denote the set of criteria c such that $\mathsf{x}?_c\mathsf{y}$ is a candidate question, i.e., $C_{\mathsf{x},\mathsf{y}} = \{c \in C \mid \mathsf{x}?_c\mathsf{y} \in Q^1_{can}\}$. (We assume Q^1_{can} is not empty. Otherwise, FRQ chooses from Q^2_{can} in the same way; cf. Alg.2.) By Definition 3, the outcomes of these questions are unknown, i.e., $\forall c \in C_{\mathsf{x},\mathsf{y}} : rlt(\mathsf{x}?_c\mathsf{y}) \notin R^+(Q_i)$. Furthermore, if any remaining question (whose outcome is unknown) between x and y is a candidate question, then all remaining questions between them are candidate questions. FRQ chooses a pair with the fewest remaining candidate questions, i.e., a pair belonging to $S_1 = \mathrm{argmin}_{(\mathsf{x},\mathsf{y})} |C_{\mathsf{x},\mathsf{y}}|$.

The reason to choose such a pair is intuitive. It requires at least $|C_{\mathsf{x},\mathsf{y}}|$ candidate questions to determine $\mathsf{y} \succ \mathsf{x}$. (The proof would be similar to that of Property 4.) Therefore, $\min_{(\mathsf{x},\mathsf{y})} |C_{\mathsf{x},\mathsf{y}}|$ is the minimum number of candidate questions to further ask, in order to determine that an object is dominated, i.e., non-Pareto optimal. Thus, a pair in S_1 may lead to a dominated object by the fewest questions, matching our goal of identifying non-Pareto optimal objects as soon as possible.

We further justify this strategy in a probabilistic sense. For $\mathsf{y} \succ \mathsf{x}$ to be realized, it is necessary that none of the remaining questions has an outcome $\mathsf{x} \succ_c \mathsf{y}$, i.e., $\forall c \in C_{\mathsf{x},\mathsf{y}} : rlt(\mathsf{x}?_c\mathsf{y}) \neq \mathsf{x} \succ_c \mathsf{y}$. Make the simplistic assumption that every question

$\mathsf{x}?_c\mathsf{y}$ has an equal probability p of not having outcome $\mathsf{x}\succ_c\mathsf{y}$, i.e., $\forall \mathsf{x}?_c\mathsf{y} \in Q^1_{can}$, $P(rlt(\mathsf{x}?_c\mathsf{y})\neq \mathsf{x}\succ_c\mathsf{y})=p$. Further assuming independence of question outcomes, the probability of satisfying the aforementioned necessary condition is $p^{|C_{\mathsf{x},\mathsf{y}}|}$. By taking a pair belonging to S_1, we have the largest probability of finding a dominated object. We note that, for $\mathsf{y}\succ\mathsf{x}$ to be realized, in addition to the above necessary condition, another condition must be satisfied—if $\nexists c$ such that $\mathsf{y}\succ_c\mathsf{x} \in R^+(Q_i)$, the outcome of at least one remaining question should be $\mathsf{y}\succ_c\mathsf{x}$, i.e., $\exists c \in C_{\mathsf{x},\mathsf{y}} : rlt(\mathsf{x}?_c\mathsf{y}) = \mathsf{y}\succ_c\mathsf{x}$. Our informal probability-based analysis does not consider this extra requirement.

Breaking Ties There can be multiple object pairs with the fewest remaining questions, i.e., $|S_1| > 1$. To break ties, FRQ chooses such an x that has dominated the fewest other objects, since it is more likely to be dominated. If there are still ties, FRQ further chooses such a y that has dominated the most other objects, since it is more likely to dominate x. More formally, FRQ chooses a pair belonging to $S_2 = \{(\mathsf{x},\mathsf{y}) \in S_1 \mid \nexists(\mathsf{x}',\mathsf{y}') \in S_1$ such that $d(\mathsf{x}') > d(\mathsf{x}) \vee (d(\mathsf{x}') = d(\mathsf{x}) \wedge d(\mathsf{y}') > d(\mathsf{y}))\}$, where the function $d(\cdot)$ returns the number of objects so far dominated by an object, i.e., $\forall \mathsf{x}, d(\mathsf{x}) = |\{\mathsf{y}|\mathsf{x}\succ\mathsf{y}$ based on $R^+(Q_i)\}|$. This heuristic follows the principle of detecting non-Pareto optimal objects as early as possible. Note that S_2 may still contain multiple object pairs. In such a case, FRQ chooses an arbitrary pair.

Selecting Comparison Criterion Once a pair (x,y) is chosen, FRQ has to select a criterion for the next question. FRQ orders the remaining criteria $C_{\mathsf{x},\mathsf{y}}$ based on the heuristic that the sooner it understands $\mathsf{y}\succ\mathsf{x}$ will not happen, the lower cost it pays. As discussed before, $|C_{\mathsf{x},\mathsf{y}}|$ questions are required in order to conclude that $\mathsf{y}\succ\mathsf{x}$; on the other hand, only one question (if asked first) can be enough for ruling it out. Consider the case that x is better than y by only one remaining criterion, i.e., $\exists c \in C_{\mathsf{x},\mathsf{y}} : rlt(\mathsf{x}?_c\mathsf{y}) = \mathsf{x}\succ_c\mathsf{y}$ and $\forall c' \in C_{\mathsf{x},\mathsf{y}}, c' \neq c : rlt(\mathsf{x}?_{c'}\mathsf{y}) = \mathsf{x} \not\succ_{c'} \mathsf{y}$. If FRQ asks $\mathsf{x}?_c\mathsf{y}$ after all other remaining questions, it takes $|C_{\mathsf{x},\mathsf{y}}|$ questions to understand y does not dominate x; but if $\mathsf{x}?_c\mathsf{y}$ is asked first, no more questions are necessary, because there will be no more candidate questions in the form of $\mathsf{x}?_c\mathsf{y}$.

i	$rlt(q_i)$	Derived Results	(x,y), $C_{\mathsf{x,y}}$	$O_{\surd}$	$O_?$	$O_\times$
			(a,b), $\{s, m, a\}$	∅	{a,b,c,d,e,f}	∅
1	b$\succ_s$a		(a,b), $\{m, a\}$			
2	a$\succ_m$b	a∼b	(a,c), $\{s, a, m\}$			
3	c$\succ_s$a		(a,c), $\{a, m\}$			
4	c$\sim_a$a		(a,c), $\{m\}$			
5	c$\succ_m$a	c$\succ$a	(b,c), $\{a, s, m\}$	∅	{b,c,d,e,f}	{a}
6	b$\sim_a$c		(b,c), $\{s, m\}$			
7	b$\sim_s$c		(b,c), $\{m\}$			
8	b$\succ_m$c	b$\succ_m$a, b$\succ$c	(d,b), $\{a, s, m\}$	∅	{b,d,e,f}	{a,c}
9	b$\sim_a$d		(d,b), $\{s, m\}$			
10	b$\sim_s$d		(d,b), $\{m\}$			
11	b$\succ_m$d	b$\succ$d	(e,b), $\{a, s, m\}$	∅	{b,e,f}	{a,c,d}
12	b$\succ_a$e		(e,b), $\{s, m\}$			
13	b$\succ_s$e		(e,b), $\{m\}$			
14	b$\succ_m$e	a$\succ_m$e, b$\succ$e	(f,b), $\{a, s, m\}$	∅	{b,f}	{a,c,d,e}
15	b$\succ_a$f		(f,b), $\{s, m\}$			
16	b$\succ_s$f		(f,b), $\{m\}$			
17	b$\succ_m$f	b$\succ$f		{b}	∅	{a,c,d,e,f}

Table 3: FRQ on Example 1.

Therefore, FRQ orders the criteria $C_{\mathsf{x,y}}$ by a scoring function that reflects the likelihood of x's superiority than y by the corresponding criteria. More specifically, for each $c \in C_{\mathsf{x,y}}$, its score is $r_c(\mathsf{x,y}) = r_c(\mathsf{y}) + {r'}_c(\mathsf{y}) - {r''}_c(\mathsf{y}) - (r_c(\mathsf{x}) + {r'}_c(\mathsf{x}) - {r''}_c(\mathsf{x}))$ where $r_c(\mathsf{y}) = |\{\mathsf{z} \mid \mathsf{z} \succ_c \mathsf{y} \in R^+(Q_i)\}|$, ${r'}_c(\mathsf{y}) = |\{\mathsf{z} \mid \mathsf{y} \sim_c \mathsf{z} \in R^+(Q_i)\}|$, and ${r''}_c(\mathsf{y}) = |\{\mathsf{z} \mid \mathsf{y} \succ_c \mathsf{z} \in R^+(Q_i)\}|$. In this scoring function, $r_c(\mathsf{y})$ is the number of objects preferred over y by criterion c, ${r'}_c(\mathsf{y})$ is the number of objects equally good (or bad) as y by c, and ${r''}_c(\mathsf{y})$ is the number of objects to which y is preferred with regard to c. FRQ asks the remaining questions in decreasing order of the corresponding criteria's scores. This way, it may find such a question that $rlt(\mathsf{x}?_c\mathsf{y}) = \mathsf{x} \succ_c \mathsf{y}$ earlier than later.

Table 3 presents the framework's execution for Example 1, by applying the FRQ policy. In addition to the same columns in Tables 1 and 2, Table 3 also includes an extra column to show, at each iteration, the chosen object pair for the next question (x,y) and the set of remaining comparison criteria between them ($C_{\mathsf{x,y}}$). The criteria in $C_{\mathsf{x,y}}$ are ordered by the aforementioned ranking function $r(\cdot)$. At the beginning of the execution, the object pair is arbitrarily chosen and the criteria are arbitrarily ordered. In the example, we assume $\mathsf{a}?_s\mathsf{b}$ is chosen as the first question. After q_2, FRQ can derive that a∼b. Hence, there is no more candidate question between them and FRQ chooses the next pair (a,c). Three questions are asked for comparing

them. At the end of q_5, multiple object pairs have the fewest remaining questions. By breaking ties, (b,c) is chosen as the next pair, since only c has dominated any object so far. The remaining criteria $C_{\mathsf{b,c}}$ are ordered as $\{a, s, m\}$, because $r_a(\mathsf{b,c}) > r_s(\mathsf{b,c})$ and $r_a(\mathsf{b,c}) > r_m(\mathsf{b,c})$. The execution sequence terminates after 17 questions, much shorter than the 30 and 28 questions by RandomQ and RandomP, respectively.

To conclude the discussion on micro-ordering, we derive a lower bound on the number of questions required for finding all Pareto-optimal objects (Theorem 2). The experiment results in § 3.3 reveal that FRQ is nearly optimal and the lower bound is practically tight, since the number of questions used by FRQ is very close to the lower bound.

Theorem 2. *Given objects O and criteria C, to find all Pareto-optimal objects in O, at least $(|O| - k) \times |C| + (k - 1) \times 2$ pairwise comparison questions are necessary, where k is the number of Pareto-optimal objects in O.*

Proof. Suppose the non-Pareto optimal objects are O_1 and the Pareto-optimal objects are O_2 ($O_1 \cup O_2 = O$ and $O_1 \cap O_2 = \varnothing$). We first separately consider n_1 (the minimum number of questions involving objects in O_1) and n_2 (the minimum number of questions comparing objects within O_2 only).

(1) By Property 4 (and its proof), for every non-Pareto optimal object $\mathsf{x} \in O_1$, at least $|C|$ questions involving x are required. There exists at least an object y such that $\mathsf{y} \succ \mathsf{x}$. The required $|C|$ questions lead to outcome either $\mathsf{y} \sim_c \mathsf{x}$ or $\mathsf{z} \succ_c \mathsf{x}$ such that $\mathsf{y} \succ_c \ldots \succ_c \mathsf{z} \succ_c \mathsf{x}$ (z can be y) for each $c \in C$. For different x, the $|C|$ questions cannot overlap—for a question with outcome $\mathsf{z} \succ_c \mathsf{x}$, the x is different; for a question with outcome $\mathsf{y} \sim_c \mathsf{x}$, the same question cannot be part of both the $|C|$ questions for x and the $|C|$ questions for y to detect both as non-Pareto optimal, because it is impossible that $\mathsf{x} \succ \mathsf{y}$ and $\mathsf{y} \succ \mathsf{x}$. Hence, $n_1 = (|O| - k) \times |C|$.

(2) Given any Pareto-optimal object $\mathsf{x} \in O_2$, for any other $\mathsf{y} \in O_2$, either (a) $\mathsf{x} \sim_c \mathsf{y}$ for all criteria $c \in C$ or (b) there exist at least two criteria c_1 and c_2 such that $\mathsf{x} \succ_{c_1} \mathsf{y}$ and $\mathsf{y} \succ_{c_2} \mathsf{x}$. Among the $k - 1$ other objects in O_2, suppose k_a and k_b of them belong to cases (a) and (b), respectively ($k_a + k_b = k - 1$). Under case (a), each of the k_a objects requires $|C|$ questions. Under case (b), there must be a question

leading to outcome $\mathsf{z} \succ_{c_1} \mathsf{y}$, where $\mathsf{z}=\mathsf{x}$ or $\mathsf{x} \succ_{c_1} \ldots \succ_{c_1} \mathsf{z} \succ_{c_1} \mathsf{y}$. Similarly, there must be a question with outcome $\mathsf{y} \succ_{c_2} \mathsf{z}$ such that $\mathsf{z}=\mathsf{x}$ or $\mathsf{y} \succ_{c_2} \mathsf{z} \succ_{c_2} \ldots \succ_{c_2} \mathsf{x}$. Therefore, each of the k_b objects requires at least 2 questions. Clearly, such required questions for comparing x with the $k-1$ other objects in O_2 are all distinct. They are also all different from the questions involving non-Pareto optimal objects (case (1)). Hence, $n_2 = k_a \times |C| + k_b \times 2 \geq (k-1) \times 2$.

Summing up n_1 and n_2, a lower bound on the number of required questions is thus $(|O| - k) \times |C| + (k-1) \times 2$. Note that, when $k = 0$, a trivial, tighter lower bound is $|O| \times |C|$. (One example in which $k = 0$ is Fig.4.) □

3.3 Experiments

We designed and conducted experiments to compare the efficiency of different instantiations of the general framework under varying problem sizes. Our experiments used both a real crowdsourcing marketplace and simulations based on a real dataset.

3.3.1 Efficiency and Scalability

We studied the efficiency and scalability of various instantiations of the general framework. Given the large number of questions required for such a study, we cannot afford using a real crowdsourcing marketplace. Hence, we performed the following simulation. Each object is an NBA player in a particular year. The objects are compared by 10 criteria, i.e., performance categories such as *points*, *rebounds*, *assists*, etc. We simulated the corresponding 10 preference relations based on the players' real performance in individual years, as follows. Consider a performance category c and two objects x=(player1, year1) and y=(player2, year2). $\mathsf{x}.c$ is player1's per-game performance on category c in year1 (similarly for $\mathsf{y}.c$). Values in each category c are normalized into the range $[0, 1]$, where 0 and 1 correspond to the minimal and maximal values in c, respectively. Suppose $\mathsf{x}.c > \mathsf{y}.c$. We generated a uniform random number v in $[0, 1]$. If $v < 1 - e^{-(x.c-y.c)}$, we set $\mathsf{x} \succ_c \mathsf{y}$, otherwise we set $\mathsf{x} \sim_c \mathsf{y}$. This way, we introduced a perturbation into the

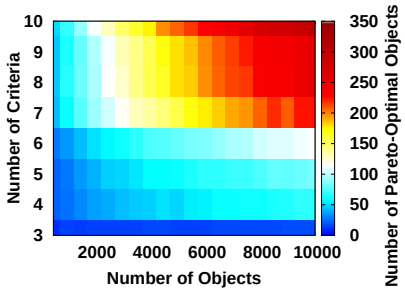
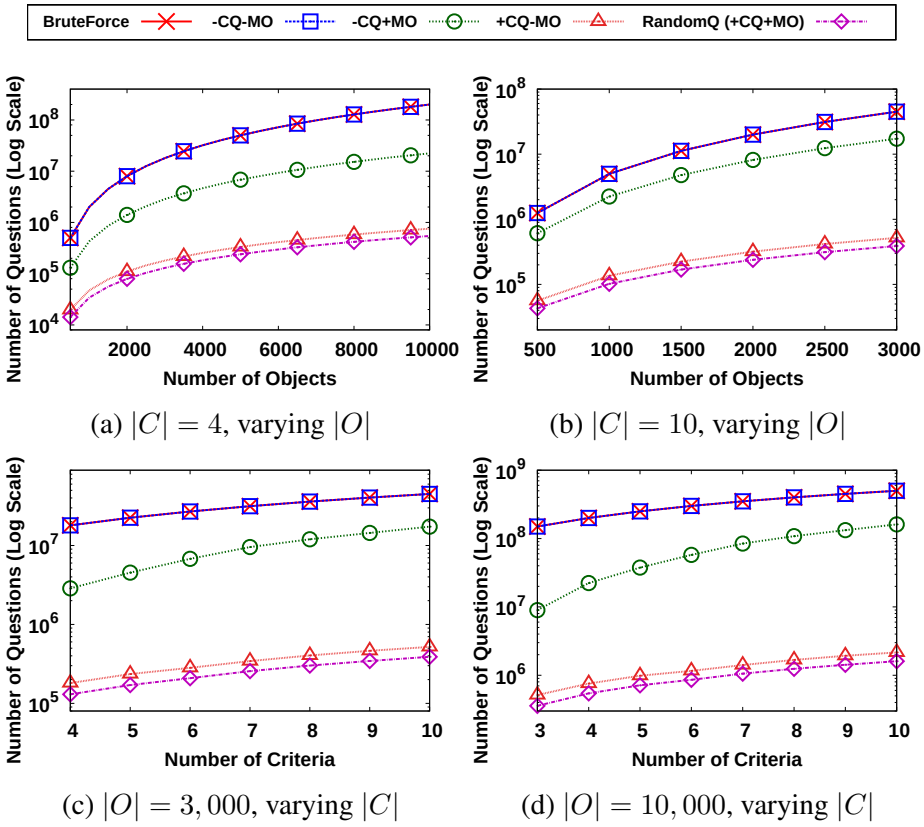


Figure 5: No. of Pareto-optimal objects. Varying $|O|$ and $|C|$.

preference relations in order to make sure they are partial orders, as opposed to directly using real performance statistics (which would imply bucket orders). Fig.5 shows that the number of Pareto-optimal objects increases by the sizes of both object set O (objects are randomly selected) and criteria set C (the first $|C|$ criteria of the aforementioned 10 criteria).

Effectiveness of candidate questions and macro-ordering

To verify the effectiveness of candidate questions and macro-ordering, we compared five methods—BruteForce, –CQ–MO, –CQ+MO, +CQ–MO, and +CQ+MO. The notation +/– before CQ and MO indicates whether a method only selects candidate questions (CQ) and whether it applies the macro-ordering strategy (MO), respectively. In all these five methods, qualifying questions are randomly selected, i.e., no particular micro-ordering heuristics are applied. For instance, +CQ+MO selects only candidate questions and applies macro-ordering. Hence, it is equivalent to RandomQ. Fig.6 shows the numbers of required pairwise comparisons (in logarithmic scale) for each method, varying by object set size ($|O|$ from 500 to $10,000$ for $|C|$=4 and $|C|$=10) and criterion set size ($|C|$ from 3 to 10 for $|O|$=3,000 and $|O|$=10,000). The figure clearly demonstrates the effectiveness of both CQ and MO, as taking out either feature leads to significantly worse performance than RandomQ. Particularly, the gap between +CQ–MO and –CQ+MO suggests that choosing only candidate questions has more fundamental impact than macro-ordering. If neither is applied (i.e., –CQ–MO), the performance is equally poor as that of BruteForce. (–CQ–MO uses slightly less questions than BruteForce, since it can terminate before exhausting all questions. However, the difference is negligible for practical purpose, as their curves overlap under logarithmic scale.)



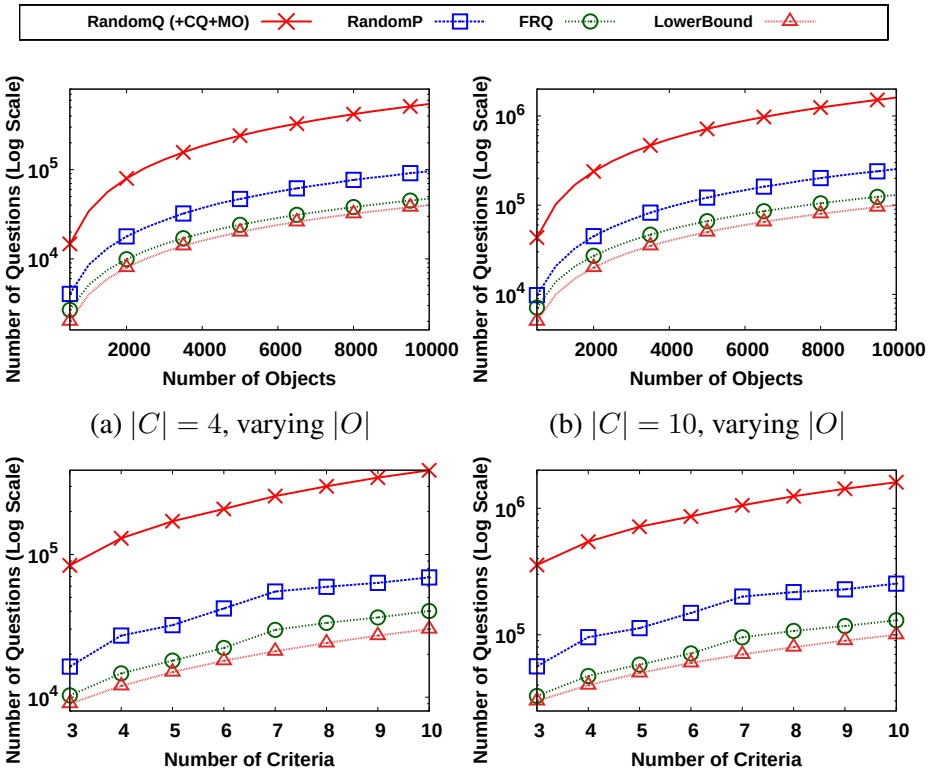
(a) $|C| = 4$, varying $|O|$ (b) $|C| = 10$, varying $|O|$

(c) $|O| = 3,000$, varying $|C|$ (d) $|O| = 10,000$, varying $|C|$

Figure 6: No. of questions by BruteForce and basic methods.

Effectiveness of micro-ordering

Fig.7 presents the numbers of pairwise comparisons required by different micro-ordering heuristics (RandomQ, i.e., +CQ+MO, RandomP, FRQ) and LowerBound (cf. Theorem 2) under varying sizes of the object set ($|O|$ from 500 to $10,000$ for $|C| = 4$ and $|C| = 10$) and the criteria set ($|C|$ from 3 to 10 for $|O| = 3,000$ and $|O| = 10,000$). In all these instantiations of the general framework, CQ and MO are applied. The results are averaged across 30 executions. All these methods outperformed BruteForce by orders of magnitude. (BruteForce is not shown in Fig.7 since it is off scale, but its number can be calculated by equation $|C| \times |O| \times (|O| - 1)/2$.) For instance, for $5,000$ objects and 4 criteria, the ratio of pairwise comparisons required by even the naive RandomQ to that used by



(a) $|C| = 4$, varying $|O|$

(b) $|C| = 10$, varying $|O|$

(c) $|O| = 3,000$, varying $|C|$

(d) $|O| = 10,000$, varying $|C|$

Figure 7: No. of questions by different micro-ordering heuristics.

BruteForce is already as low as 0.0048. This clearly shows the effectiveness of CQ and MO, as discussed for Fig.6. The ratios for RandomP and FRQ are further several times smaller (0.00094 and 0.00048, respectively). The big gain by FRQ justifies the strategy of choosing object pairs with the fewest remaining questions. Especially, FRQ has nearly optimal performance, because it gets very close to LowerBound in Fig.7. The small gap between FRQ and LowerBound also indicates that the lower bound is practically tight. The figure further suggests excellent scalability of FRQ as its number of questions grows almost linearly by both $|C|$ and $|O|$.

3.3.2 Experiments Using a Real Crowdsourcing Marketplace

We also studied the performance of the proposed algorithms using the popular crowdsourcing marketplace Amazon Mechanical Turk (AMT). The task is to compare 100 photos of our institution with regard to *color*, *sharpness* and *landscape*. To obtain the ground-truth data, all $14,850$ possible pairwise questions were partitioned into $1,650$ tasks, each containing 9 questions on a criterion. An AMT crowdsourcer is allowed to perform a task only if they have responded to at least 100 HITs (Human Intelligence Tasks) before with at least 90% approval rate. Furthermore, we implemented basic quality control by including 2 additional validation questions in each task that expect certain answers. For instance, one such question asks the crowd to compare a colorful photo and a dull photo by criterion *color*. A crowdsourcer's responses in a task are discarded if their response to a validation question deviates from our expectation. (236 crowdsourcers failed on this.) The parameters in Equation (1) were set to be $k = 5$ and $\theta = 0.6$. Hence, in total $(1,650 \times 5 + 236) \times (9 + 2) = 93,346$ pairwise comparisons were performed by AMT crowdsourcers. We paid 1 cent for each comparison and therefore spent close to $1,000 in total.

The responses to all possible questions provide the ground-truth data. An algorithm execution only needs the responses to a subset of the questions. We randomly selected a subset of photos O ($|O|$ from 10 to 100) and applied various algorithms to find Pareto-optimal photos. Figure 8 shows, for varying $|O|$, the number of questions (in logarithmic scale) required by each micro-ordering strategy. To account for the randomness in RandomP and RandomQ, we repeated these two algorithms, respectively, 30 times, and we reported the average numbers of questions. Confirming the results in Figure 7, FRQ was close to the theoretical lower bound, performing better than the other two methods, and RandomP outperformed RandomQ.

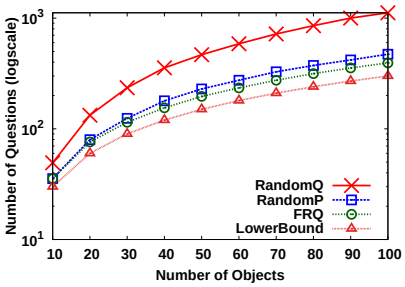


Figure 8: No. of questions by different micro-ordering heuristics. $|C| = 3$, varying $|O|$.

	Task	Question type	Multiple attributes	Order among objects (on each attribute)	Explicit attribute representation
[15]	full ranking	pairwise comparison	no	bucket/total order	no
[16]	top-k ranking	rank subsets of objects	no	bucket/total order	no
[17]	top-k ranking and grouping	pairwise comparison	no	bucket/total order	no
[9]	skyline queries	missing value inquiry	yes	bucket/total order	yes
This work	Pareto-optimal object finding	pairwise comparison	yes	strict partial order	no

Table 4: Related work comparison.

3.4 Related Work

This is the first work on crowdsourcing Pareto-optimal object finding. There are several recent studies on using crowdsourcing to rank objects and answer group-by, top-k and skyline queries. Crowd-BT [15] ranks objects by crowdsourcing pairwise object comparisons. Polychronopoulos et al. [16] find top-k items in an itemset by asking human workers to rank small subsets of items. Davidson et al. [17] evaluate top-k and group-by queries by asking the crowd to answer *type* questions (whether two objects belong to the same group) and *value* questions (ordering two objects). Lofi et al. [9] answer skyline queries over incomplete data by asking the crowd to provide missing attribute values. Table 4 summarizes the similarities and differences between these studies and our work. The studies on full and top-k ranking [15–17] do not consider multiple attributes in modeling objects. On the contrary, the concepts of skyline [9] and Pareto-optimal objects (this paper) are defined in a space of multiple attributes. [9] assumes explicit attribute representation. Therefore, they resort to the crowd for completing missing values, while other studies including our work request the crowd to compare

objects. Our work considers strict partial orders among objects on individual attributes. Differently, other studies assume a bucket/total order [15–17] or multiple bucket/total orders on individual attributes [9].

Besides [15], there were multiple studies on ranking objects by pairwise comparisons, which date back to decades ago as aggregating the preferences of multiple agents has always been a fundamental problem in social choice and welfare [18]. The more recent studies can be categorized into three types: **1)** Approaches such as [19–21] predict users' object ranking by completing a user-object scoring matrix. Their predications take into account users' similarities in pairwise comparisons, resembling *collaborative filtering* [22]. They thus do not consider explicit attribute representation for objects. **2)** Approaches such as [23–25] infer query-specific (instead of user-specific) ranked results to web search queries. Following the paradigm of *learning-to-rank* [26], they rank a query's result documents according to pairwise result comparisons of other queries. The documents are modeled by explicit ranking features. **3)** Approaches such as [27–30] are similar to [15] as they use pairwise comparisons to infer a single ranked list that is neither user-specific nor query-specific. Among them, [28] is special in that it also applies learning-to-rank and requires explicit feature representation. Different from our work, none of these studies is about Pareto-optimal objects, since they all assume a bucket/total order among objects; those using learning-to-rank require explicit feature representation, while the rest do not consider multiple attributes. Moreover, except [28, 29], they all assume comparison results are already obtained before their algorithms kick in. In contrast, we aim at minimizing the pairwise comparison questions to ask in finding Pareto-optimal objects.

3.5 Final Remarks

In this chapter we studied crowdsourcing Pareto-optimal finding when objects do not have explicit attributes and preference relations are strict partial orders. The partial orders are obtained by pairwise comparison questions to the crowd. It introduces an iterative question-selection framework that is instantiated into

different methods by exploiting the ideas of candidate questions, macro-ordering and micro-ordering. Experiment were conducted by simulations on large object sets and by using a real crowdsourcing marketplace. The results exhibited not only orders of magnitude reductions in questions against a brute-force approach, but also close-to-optimal performance from the most efficient method.

4 Discovering the Skyline of Web Databases

Motivation: Skyline for structured databases has been extensively studied in recent years. Consider a database with n tuples over m numerical/ordinal attributes, each featuring a domain that has a preferential order for certain applications, e.g., price (smaller the better), model year (newer the better), etc. A tuple t is said to *dominate* a tuple u if for every attribute A_i, the value of $t[A_i]$ is preferred over $u[A_i]$. The skyline is the set of all tuples t_i such that t_i is not dominated by any other tuple in the database.

Skyline is important for multi-criteria decision making, and is further related to well-known problems such as convex hulls, top-k queries and nearest neighbor search. For example, a precomputed skyline can serve as an index for efficiently answering any top-1 query with a monotonic ranking function over attributes. The extension of a skyline to a K-*sky band* (containing all tuples not dominated by more than $K-1$ others) enables efficient answering of top-k queries when $k \leq K$. For a summary of research on skyline computation and their applications, please refer to Section 4.8.

Much of the prior work assumes a traditional database with full SQL support [1, 31–33] or databases that expose a ranked list of all tuples according to a pre-known ranking function [34, 35]. In this paper, we consider a novel problem of how to compute the skyline over a *deep web, "hidden", database* that only exposes a top-k query interface. Unlike the traditional assumptions, real-world web databases place severe limits on how external users can perform searches. Typically, a user can only specify conjunctive queries with range or (single-valued) point conditions, depending on which one(s) the web interface supports, and receive at most k matching tuples, selected and sorted according to a ranking function that is often proprietary and unknown to the external user.

Discovering skyline tuples from a hidden web database enables a wide variety of third-party applications, ranging from understanding the "performance envelope" of tuples in the database to enabling uniform ranking functions over multiple web databases. For example, consider the construction of a diamond search service that

taps into web databases of several jewelry stores such as Blue Nile (by collecting data through their web search interfaces). While there are well-known preferential orders on all critical attributes of a diamond such as clarity, carat, color, cut and price, each jewelry store may design its own ranking function as a unique weighting of these attributes. On the other hand, the third-party service needs to rank all tuples from all stores consistently, and ideally support user-specified ranking functions (e.g., different weightings of the attributes) according to his/her own need. An efficient and effective way to enable this is to first discover the skyline tuples from the hidden web database of each jewelry store, and then apply a user-specified ranking function on all the retrieved data to obtain tuples most preferred by the user. One can see that, similarly, this approach can be used to enable third-party services such as flight search with user-defined ranking functions on price, duration, number of stops, etc.

Challenges: The technical challenges we face are fundamentally different from traditional skyline computation techniques, mainly because the data access model is completely different. In traditional skyline research, there is no top-k constraint on data access, so the algorithms can take advantage of either full SQL power or certain pre-existing data indices such as sequence access according to a known ranking function [34, 35]. On the other hand, as mentioned earlier, in hidden databases the data access is severely restricted. In principle, one can apply prior techniques developed to crawl the entire hidden database (e.g., using algorithms such as [36]), and then compute the skyline over a local copy of the database. However, as we shall show in the experimental results, such an approach is often impractical as crawling the entire database (as opposed to just the skyline) requires an inordinate number of search queries (i.e., web accesses). Note that many real-world web databases limit the number of web accesses one can issue through per-IP-address or per-API-key limits. In many cases, this limit is too small to sustain the execution of a complete crawl. Thus, it is necessary to develop skyline discovery algorithms that execute as few search queries via the restrictive web interface as possible.

Technical Highlights: We distinguish between several important categories of

web search interfaces: whether range predicates are supported for the attributes (either one-ended, e.g. Price < 300, or two-ended, e.g., $200 <$ Price ≤ 300), or only single-value/point predicates (e.g., Number of Stops = 0) are allowed. We also consider hidden databases where a mix of range and point attributes exist. Computing skylines over each type of interface offers its own unique challenges.

For the case of one-ended range queries, we develop **SQ-DB-SKY**, an iterative divide-and-conquer skyline discovery algorithm that starts by issuing broad queries (i.e., queries with few predicates), determines which queries to issue next based on the tuples received so far, and then gradually narrows them to more specific ones. For the case of two-ended range queries, we develop algorithm **RQ-DB-SKY**, which is similar to the previous algorithm, except that instead of being forced to issue overlapping queries, the algorithm is able to take advantage of the more powerful search interface and issue mutually exclusive queries to cover the search space and be able to terminate earlier.

In the worst case, the maximum number of queries issued by SQ-DB-SKY may be $O(m \cdot |S|^{m+1})$ where m is the number of attributes and $|S|$ is the size of the skyline set. Note that this running time is independent[1] of the database size n. In contrast, the worst case query cost for RQ-DB-SKY is $O(m \cdot \min(|S|^{m+1}, n))$. More interestingly, while the worst case behavior appears to grow fast with $|S|$ when m is large, we show through theoretical analysis and real-world experiments that this is the artifact of some extremely-ill-behaving ranking function (which has to be considered in worst-case analysis). In practice, the algorithms perform extremely well.

As additional highlights of our contributions, we provide an interesting theoretical result on the average-case behavior of the above algorithms by proving that, for *any arbitrary database*, the expected query cost taken over the randomness of the ranking function is always bounded from above by $(e + e \cdot |S|/m)^m$. Note that the growth speed of this bound with $|S|$ is orders of magnitude slower than that of the worst-case bound. Furthermore, we also show why the real-world performance of SQ- and RQ-DB-SKY is likely even better than the average-case performance

[1] at least conditionally given m and $|S|$

for any "reasonable" ranking function used by the hidden database.

For the case of point queries, the significantly weaker search interface introduces novel challenges in designing an efficient skyline discovery algorithm. For the special case of 2-dimensional data, we design algorithm **PQ-2D-SKY** that is *instance-optimal*, although the worst-case complexity is a complex function that depends not only on parameters such as n and S, but also on the domain sizes of the attributes. Unfortunately, the generalization to higher dimensions proves much more complicated, as shown by a negative result that no instance-optimal algorithm can exist for higher dimensions.

As such, our eventual algorithm for higher dimensions, **PQ-DB-SKY**, uses as a subroutine a revised version of the 2D algorithm that is able to discover all skyline from a "pruned"' 2D subspace in an instance-optimal manner (though the overall algorithm for higher dimensions is not instance-optimal). Given the exponential nature of dividing a higher-dimensional space into 2D subspaces, the worst-case query cost of the algorithm can be quite large. However, as we shall show through real-world online experiments, the nature of these PQ attributes used in real-world hidden databases (e.g., they usually have small domains with all domain values occupied by real tuples) makes the actual performance of PQ-DB-SKY often fairly efficient in practice.

When the hidden database features a mixture of range and point attributes, we show that the straightforward idea of only applying RQ-DB-SKY directly over the range-predicate attributes and not using the point-predicate attributes at all does not work because some skyline tuples may be missed. These remaining tuples need to be identified by a modified version of PQ-DB-SKY. These ideas are combined into our eventual algorithm **MQ-DB-SKY** that can discover the skyline of a database containing a mixture of one-ended, two-ended, and/or point attributes.

The above algorithms are all about computing the skyline of a hidden database. We have also extended these algorithms to compute the top-K sky band of the database. We conducted comprehensive experiments over multiple real-world datasets to demonstrate the effectiveness of these algorithms and their superiority over the baseline, crawling-based, solution. In addition, we also tested our algo-

rithms live online over multiple real-world web databases such as Yahoo! Autos, Google Flights, and Blue Nile (an online diamond retailer). For all these real-world databases tested, our algorithms can discover all skyline tuples in a highly efficient manner.

Summary of Contributions:

- We introduce the novel problems of computing the skyline/band of hidden web databases with top-k constraints, motivate them with third-party applications,and show why traditional skyline computation approaches are inappropriate for these problems.
- We distinguish between different search interfaces that hidden databases typically provide: one-ended and two-ended ranges and point predicates, and show that each brings different challenges in designing algorithms for skyline discovery.
- For the case of one-ended (resp. two-ended) range predicates, we develop SQ-DB-SKY (resp. RQ-DB-SKY). For the case of point predicates, we develop PQ-2D-SKY for two-dimensional data, and a more general PQ-DB-SKY for higher-dimensional data. For databases with a mixture of range and point predicates, we develop MQ-DB-SKY.
- We provide rigorous theoretical analysis including worst/average-case analysis and instance-optimality in certain cases. We also conducted comprehensive experiments over real datasets and live web databases to demonstrate the effectiveness of our algorithms.

4.1 Preliminaries

4.1.1 Model of Hidden Database

Database: Consider a hidden web database D with n tuples over m attributes $A_1, \ldots, A_m$. Let the domain of A_i be $Dom(A_i)$ and the value of A_i for tuple t be $t[A_i] \in Dom(A_i) \cup \{\text{NULL}\}$.

Skyline: The m attributes of a web database can be divided into two categories: *ranking attributes* with an inherent preferential order (either numeric or ordinal); and *filtering attributes* whose values are not ordered. The skyline definition only

concerns the ranking attributes. For a ranking attribute A_i, we denote the total order by $<$, i.e., v_i ranks *higher* than v_j if $v_i < v_j$. With this notation, a tuple $t \in D$ is a skyline tuple if and only if there does not exist any other tuple $t' \in D$ with $t' \neq t$ such that t' dominates t, i.e. $t'[A_i] \leq t[A_i]$ for each and every ranking attribute A_i. In other words, no other tuple t' in the database outranks t on every ranking attribute.

Note that the skyline definition can be easily extended to *sky band* - i.e., a tuple is in the K-skyband if and only if it is not dominated by more than $K-1$ tuples. One can see that the skyline is indeed a special case of (top-1) sky band. In most parts of the paper, we focus on the problem of skyline discovery. The extension to discovering the K-skyband ($K > 1$) is discussed in Section 4.6.

Query Interface: The web interface of a hidden database takes as input a user-specified query (supported by the interface) and produces as output at most k tuples matching the query. At the input side, the interface generally supports conjunctive queries on one or more attributes. The *predicate* supported for each attribute, however, is a subtle issue that depends on the type of the attribute and the interface design. While filtering attributes with categorical values generally support equality (=) only, a ranking attribute may support any subset of $<$, $=$, $>$, $\leq$, and $\geq$ predicates. Since the supported predicate types turn out to be critical for our algorithm design, we leave it for detailed discussions in the next subsection.

Output-wise, the query answer is subject to the top-k constraint, i.e., when more than k tuples match the input query, instead of returning all matching tuples, the hidden database preferentially selects k of them according to a *ranking function* and returns only these top-k tuples through the interface. In this case, we say that query q *overflows* and triggers the top-k limitation.

The design of this ranking function has been extensively studied in the database literature, leading to numerous variations. In this paper, we support a very broad set of ranking functions with only one requirement: *domination-consistent*, i.e., if a tuple t dominates t' and both match a query q, then t should be ranked higher than t' in the answer. All results in the paper hold on any arbitrary ranking function so long as it satisfies this requirement.

Filtering Attributes: While a web database may contain order-less filtering attributes, they have no bearing on the definition of skyline tuples. We further note that filtering attributes have no implication on skyline discovery *unless* there are skyline tuples with the exact same value combination on all ranking attributes. Even in this case, what one needs to do is to simply issue, for each discovered skyline tuple, a conjunctive query with equality conditions on all ranking attributes. If the query overflows, one can then crawl all tuples matching the query using the techniques in [36] .

Since such a case (i.e., multiple skyline tuples having the exact same value combination on all ranking attributes) is unlikely to happen when we have a meaningful skyline definition, in most parts of the paper we make the general positioning assumption, i.e., all skyline tuples have unique value combinations on ranking attributes, as assumed in most prior work [1, 31–33]. Our experiments in § 4.7, however, do involve filtering attributes and confirm that they have no implication on skyline discovery.

Finally, for the purpose of this paper, we consider the problem of discovering skyline tuples over the *entire database*. If the goal is to discover skyline tuples for a subset of the database subject to certain filtering conditions, all results in the paper still readily apply. The only change required is to simply append the filtering conditions as conjunctive predicates to *all* queries issued.

4.1.2 Taxonomy of Attribute Search Interface

We now discuss in detail what types of predicates may be supported for an attribute - an issue that, somewhat surprisingly, turns out crucial for the efficiency of skyline discovery. Specifically, we partition the support into three categories depending on two factors: (1) whether range predicates are supported for the attribute, or only equality (i.e., point) predicates are allowed, and (2) when range predicates are supported, whether the range is one-ended (i.e., "better than" a user-specified value), or two-ended.

- SQ, i.e., *Single-ended range Query predicate*, means that predicate on A_i can be $A_i < v$, $A_i \leq v$ or $A_i = v$, where $v \in Dom(A_i)$. Note that we do not

further distinguish whether $<$ or $\leq$ (or both) is supported, because they are easily reducible to each other - e.g., one can combine the answers to $A_i < v$ and $A_i = v$ to produce that for $A_i \leq v$. On the other hand, if $A_i \leq v$ is supported but not $A_i < v$, one can take the next smaller value (than v) in $Dom(A_i)$, say v', and then query $A_i \leq v'$ instead[2].

- RQ, i.e., *Range Query predicate*, means that predicate on A_i can be $A_i <$ (or $\leq$) v, $A_i = v$ or $A_i >$ (or $\geq$) v.
- PQ, i.e., *Point Query predicate*, means that predicate on A_i can only be of the form $A_i = v$.

Having defined the three types of predicates, SQ, RQ and PQ, we now discuss the comparisons between them, starting with SQ vs RQ within range predicates, and then range vs point predicates.

SQ vs RQ: One might wonder why both single-ended SQ and two-ended RQ exist in a web interface. To understand why, consider two examples: the *memory size* and *price* of a laptop, respectively. Both have an inherent order: the larger the memory size or the lower the price, the better. Nonetheless, their presentations in the search interface are often different:

Memory size is often presented as SQ, because there is little motivation for a user to specify an upper bound on the memory size. Price, on the other hand, is quite different. Specifically, it is usually set as an RQ attribute with two-ended range support because, even though almost all users prefer a lower price (for the same product), many users indeed specify both ends of a price range to *filter* the search results to the items they desire. The underlying reason here is that price is often correlated (or perceived to be correlated) with the quality or performance of a laptop. For the lack of understanding of the more "technical" attributes, or for the simplicity of considering only one factor, many users set a lower bound on price to filter out low-performance laptops that do not meet their needs.

[2]Of course, in the case where $Dom(A_i)$ is an infinite set, e.g., when A_i is continuous, a tacit assumption here is that we know a small value ϵ such that no tuple can have $A_i \in (v - \epsilon, v)$. Given that the values represented in a database are anyway discrete in nature, this assumption can be easily satisfied by assuming a fixed precision level for the skyline definition.

SQ/RQ vs PQ: Note that range-predicate support (SQ or RQ) is strictly "stronger" than PQ: While it is easy to specify a range predicate that is equivalent to a point one, to "simulate" a range query, one might have to issue numerous point queries, especially when the domain sizes and the number of attributes are large.

Fortunately though, real-world hidden databases often only represent an ordinal ranking attribute as PQ when it has (or is discretized to) a very small domain size. For example, flight search websites set the number of stops as PQ because it usually takes only 3 values: 0, 1, or 2+. On the other hand, price is rarely PQ given the wide range of values it can take. As we shall elaborate later, the small domain sizes of PQ attributes help with keeping the query cost small, even though PQ still generally requires a much higher query cost for skyline discovery than SQ/RQ.

4.1.3 Problem Definition

Performance Measure: In most parts of the paper, we consider the objective of discovering *all* skyline tuples from the hidden web database. Interestingly, our solutions also feature the *anytime* property [37] which enables them to quickly discover a large portion of the skyline.

When our goal is complete skyline discovery, what we need to optimize is a single target: efficiency. We note the most important efficiency measure here is *not* the computational time, but *the number of queries we must issue* to the underlying web database. The rationale here is the query rate limitation enforced by almost all web databases - in terms of the number of queries allowed from an IP address or a user account per day. For example, Google Flight Search API allows only 50 free queries per user per day.

SKYLINE DISCOVERY PROBLEM: Given a hidden database D with query interface supporting a mixture of SQ, RQ or PQ for ranking attributes, without knowledge of the ranking function (except that it is domination-consistent as defined above), retrieve all skyline tuples while minimizing the number of queries issued through the interface.

4.2 Skyline Discovery for SQ-DB

We start by considering the problem of skyline discovery for interfaces that support single-ended range queries. Recall from Section 5.3 that a single-ended range supports $<$ (along with $=$ and $\leq$) only, but not $>$. In this section, we first prove the problem of skyline discovery single-ended range queries is exponential, then develop the main ideas behind our SQ-DB-SKY algorithm, and discuss its query cost analysis.

Theorem 3. *Considering the SQ interface, there exists a data- base D such that discovering its skyline requires at least $O(|S|^m)$ queries.*

Proof. We construct the proof for the case where $|S|$ is larger than m. Let the domain of each attribute A_i ($i \in [1, m]$) be $[0, h+1]$, with smaller values preferred over larger ones. We first insert into the database D the following m tuples $t_1^0, \ldots, t_m^0$, such that

$$t_i^0[A_j] = \begin{cases} 0, & \text{if } i \neq j, \\ h+1, & \text{if } i = j. \end{cases} \tag{2}$$

There are two key observations here. First is that, knowing the insertion of these m tuples, any optimal skyline discovery algorithm for SQ-DB must issue solely fully-specified queries (i.e., those with one conjunctive predicate on each attribute A_i). The reason is that any query with fewer than m predicates will always return one of $t_1^0, \ldots, t_m^0$, rendering the query answer useless.

Second is that the insertion of these tuples do not affect the skyline nature of any skyline tuples in D, so long as we keep the domain of A_i for any tuple in D within $[1, h]$. The reason is that any tuple with attribute values solely in $[1, h]$ cannot be dominated by a tuple in $t_1^0, \ldots, t_m^0$, which always has one attribute equal to $h+1$.

Having established the fact that the query sequence issued for skyline discovery consists solely of fully-specified SQ queries, we can safely represent each query by a point in the m-dimensional space, specifically the lowest-ranked point covered by the query. For example, given an SQ query

q: SELECT * FROM D WHERE $A_1 < v_1$ AND $\cdots$ AND $A_m < v_m$,

we can represent q as the point $v(q) : \langle v_1, \ldots, v_m \rangle$. We are now ready to introduce the following key proposition:

Proposition: If a point v ($v \notin D$) satisfies two conditions: (1) v is a skyline tuple over $D \cup \{v\}$, and (2) any tuple dominated by v must also be dominated by at least one tuple in D, then any skyline discovery algorithm over D must issue the query corresponding to v.

The proof to the proposition is simple. Since, as proved above, skyline discovery algorithms can only issue fully specified queries, the only such queries that return v are corresponding to points that are equal to or dominated by v. Since any point v' dominated by v is also dominated by a tuple in D, it means that issuing v' may reveal the tuple in D instead of v. In other words, without issuing v, there is no way for a skyline discovery algorithm to distinguish between D and $D \cup \{v\}$, meaning that the algorithm cannot safely conclude that it has crawled all skyline tuples over D. Thus, any skyline discovery algorithm over D must issue the query corresponding to v.

Given the proposition, one can see that the lower bound proof is essentially reduced to a count of points that satisfy the two conditions in the proposition. Consider a database with $|S|$ skyline tuples $t_1, \ldots, t_{|S|}$, each having a unique permutation of $1, 2, \ldots, m$ as the values for $A_1, \ldots, A_m$, respectively. To better illustrate the proof, we add a unique, arbitrarily small, noise ϵ_{ij} to the value of A_j for skyline tuple t_i ($i \in [1, |S|], j \in [1, m]$), such that ϵ_{ij} is unique for each combination of i and j.

We now show that every unique combination of $m - 1$ tuples in $t_1, \ldots, t_{|S|}$ yields a unique point v that satisfies the two above-described conditions. Without loss of generality, consider $m - 1$ tuples $t_1, \ldots, t_{m-1}$. Consider the following construction:

We start with A_1 and assign $v[A_1] = \max(t_1[A_1], \ldots, t_{m-1}[A_1])$. Again without loss of generality, let t_1 be the tuple featuring this "worst" value on A_1. Next, we exclude t_1 from consideration and find the worst value on A_2, i.e.,

$v[A_2] = \max(t_2[A_2], \ldots, t_{m-1}[A_2])$, and continue this process. One can see that by the time we reach A_{m-1}, there is only one tuple left, say t_{m-1}, and we assign $v[A_{m-1}] = t_{m-1}$.

To determine the value for $v[A_m]$, we issue the following query

$$q\text{: SELECT MIN}(A_m)\text{ FROM D WHERE } A_1 \leq v[A_1] \text{ AND } \cdots \text{ AND } A_{m-1} \leq v[A_{m-1}],$$

and assign to $v[A_m]$ the result minus an arbitrarily small noise, i.e., $v[A_m] = q - \epsilon$ where ϵ is arbitrarily close to 0. Note that this query will never return empty because the above construction guarantees that A_{m-1} satisfies the selection conditions in the query.

There are two key observations from this construction. First, this constructed v is guaranteed to satisfy both conditions described above. The proof is straightforward, given that $v[A_m]$ is equal to (sans an arbitrarily small noise) the MIN(A_m) among all tuples dominating v on the other $m - 1$ attributes.

Second, each different combination of $m - 1$ tuples in $t_1, \ldots, t_{|S|}$ yields a different point v. The reason for this is also simple: each of the first $m - 1$ attributes of v comes from a different tuple. Since each attribute of each tuple features a unique value (thanks to the inserted noise ϵ_{ij}), each unique combination of $m - 1$ tuples yields a unique v.

One can see that, given these two observations, there are at least $\binom{|S|}{m}$ points v that satisfy both of the above conditions. Thus, the query cost for a skyline discovery algorithm is $O(|S|^m)$. □

4.2.1 Key Idea: Algorithm SQ-DB-SKY

Our SQ-DB-SKY algorithm is an iterative divide-and-conquer one that starts by issuing broad queries, determines which queries to issue next based on the tuples received so far, and then gradually narrowing them to more specific ones. For the ease of understanding, consider the example of a 3-dimensional database. Suppose the tuple returned by q_1 : SELECT * FROM D is t_1. Algorithm SQ-DB-SKY first issues the following three queries:

q_2: SELECT * FROM D WHERE $A_1 < t_1[A_1]$
q_3: SELECT * FROM D WHERE $A_2 < t_1[A_2]$
q_4: SELECT * FROM D WHERE $A_3 < t_1[A_3]$

A key observation here is that the *comprehensiveness* of skyline discovery is maintained when we divide the problem to the subspaces defined by q_2, q_3, q_4. Specifically, every skyline tuple (besides t_1) must satisfy at least one of q_2, q_3, q_4 because otherwise it would be dominated by t_1. Now suppose q_2 returns t_2 as top-1 (which must be on the skyline because no tuple with $A_i \geq v$ can dominate one with $A_i < v$). We continue with further "dividing" (the subspace defined by) q_2 into three queries according to t_2:

q_5: WHERE $A_1 < t_2[A_1]$
q_6: WHERE $A_1 < t_1[A_1]$ AND $A_2 < t_2[A_2]$
q_7: WHERE $A_1 < t_1[A_1]$ AND $A_3 < t_2[A_3]$

Again, any skyline tuple that satisfies q_2 (i.e., with $A_1 < t_1[A_1]$) must match at least one of the three queries. One can see that this process can be repeated recursively from here: Every time a query q_j returns a tuple t, we generate m queries by appending $A_1 < t[A_1], \ldots, A_m < t[A_m]$ to q_j, respectively. A critical observation here is that any skyline tuple matching q_j must match at least one of the m generated queries, because it has to surpass t on at least one attribute in order to be on the skyline. As such, so long as we follow the process to traverse a "query tree" as shown in Figure 9, we are guaranteed to discover all skyline tuples.

Theorem 4. *Algorithm SQ-DB-SKY is guaranteed to discover all skyline tuples.*

Proof. Consider any skyline tuple t. To prove that t will always be discovered by SQ-DB-SKY, we construct the proof by contradiction. Suppose that t is not discovered, i.e., it is not returned by any node in the tree. We start by considering the m branches of the root node. Since t is a skyline tuple, it must satisfy at least one of these branches, as otherwise it would be dominated by the tuple returned by the root node (contradicting the assumption that t is a skyline tuple). When there are multiple branches matching t, choose one branch arbitrarily. Consider the node corresponding to the branch, say $q_i : A_i < t_1[A_i]$. Since q_i matches t yet does not

return it (because otherwise t would have been discovered), it must overflow and therefore have m branches of its own.

Once again, t has to satisfy at least one of these m branches (of q_i), as otherwise t would have been dominated by the tuple returned by q_i (contradicting the skyline assumption). Repeat this process recursively; and one can see that there must exist a path from the root to a leaf node in the tree, such that t satisfies each and every node on the path. Since every leaf node of the tree is a valid or underflowing query, this means that the leaf node must return t, contradicting the assumption that t is not discovered. This proves the completeness of skyline discovery by SQ-DB-SKY. □

In order to better understand the correctness of the algorithm, consider the dummy example provided in Figure 10, and its corresponding SQ-DB-SKY tree in Figure 11. One can see that each skyline tuple appears in at least one of the branches, as otherwise it would have been dominated by another (skyline) tuple.

Algorithm 3 depicts the pseudo code for SQ-DB-SKY. Note from the algorithm that a larger k (as in top-k returned by the database) reduces query cost for two reasons: First, every returned tuple that is not dominated by another in the top-k is guaranteed to be a skyline tuple. Second, a larger k also makes the tree shallower because a node becomes leaf if it returns fewer than k tuples. This phenomenon is verified in our experimental studies.

We would like to clarify that, it is *not* needed to find *the largest domain value* of A_i smaller than v. Instead, so long as we find $v' < v$ such that replacing the predicate $A_i \leq v$ with $A_i \leq v'$ still leads to an non-empty query answer, the algorithms will work. The only case where we may have trouble with a $\leq$ interface is when $A_i \leq v$ overflows, yet it takes a larger number of queries to perform binary search to find $v' < v$ with nonempty $A_i \leq v'$. This means that there is a tuple with value $v - \epsilon$ on A_i, with ϵ extremely close to 0. While it is true that this situation may lead to a high query cost for our algorithm, we have not seen this behavior in any real-world database for the simple reason that it will make it extremely difficult for a normal user of the hidden database to specify a query that unveils the tuple with $A_i = v - \epsilon$.

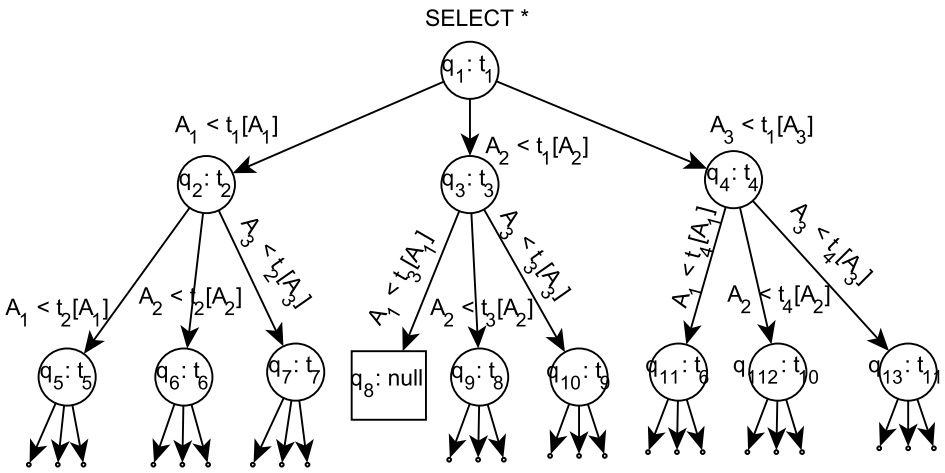


Figure 9: Tree illustration

	A_1	A_2	A_3
t_1	5	1	9
t_2	4	4	8
t_3	1	3	7
t_4	3	2	3

Figure 10: Illustration of example

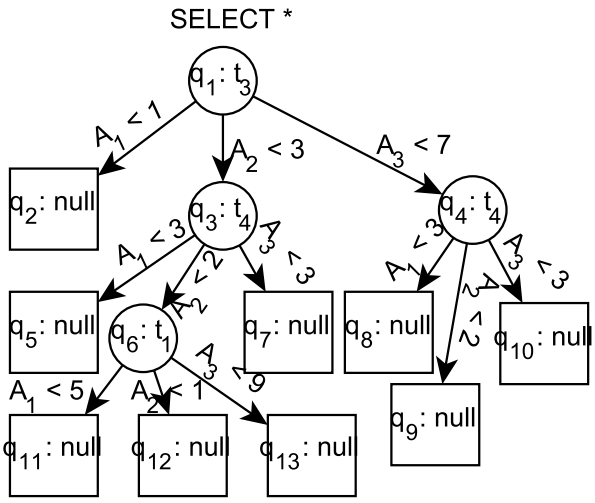


Figure 11: SQ-DB-SKY example tree

Algorithm 3 SQ-DB-SKY

QueryQ = {SELECT * FROM D}
$S = \varnothing$
while QueryQ is not empty **do**
  q = QueryQ.deque(); T = Top-$k(q)$
  if T is not empty **then**
    Append the none-dominated tuples in T to S
    if T contains k tuples **then**
      Construct m queries $q_1, \ldots, q_m$ where query q_i appends predicate "$A_i < T_0[A_i]$" to q
      Append $q_1, \ldots q_m$ to QueryQ
    end if
  end if
end while

4.2.2 Query-Cost Analysis

Algorithm SQ-DB-SKY has one nice property and one problem in terms of query cost: The nice property is that the top-1 tuple returned by every node (i.e., query) must be on the skyline (because it cannot be dominated by a tuple not matching

the query). The problem, however, is that a skyline tuple t might be returned as No. 1 by *multiple* nodes, potentially leading to a large tree size and thus a high query cost. For example, if t has $t[A_1] < t_1[A_1]$ and $t[A_2] < t_2[A_2]$, then it might be returned by both q_2 and q_3.

Worst-Case Analysis: Given the overlap between tuples returned by different nodes, the key for analyzing the query cost of SQ-DB-SKY is to count how many nodes in the tree return a tuple. Because we are analyzing the worst-case scenario, we have to consider $k = 1$ and any arbitrary, ill-behaved, system ranking functions. In other words, so long as a tuple matches a node, it may be returned by it. To this end, there is almost no limit on how many times a tuple can be returned, except the following *prefix-free* rule:

Note that each node in the tree can be (uniquely) represented by a sequence of 2-tuples $\langle t_i, A_j \rangle$, where t_i is a skyline tuple returned by a node, and A_j is an attribute corresponding to the branch taken from the node. For example, the nodes corresponding to q_2 and q_5 are represented as $\langle t_1, A_1 \rangle$ and $\langle t_1, A_1 \rangle$, $\langle t_2, A_1 \rangle$, respectively. The one property that all nodes returning the same tuple t must satisfy is that the sequence representing one node, say q, *cannot be a prefix* of the sequence representing another, say q'. The reason is simple: if the sequence of q is a prefix of q', then q' must be in the subtree of q. However, according to the design of SQ-DB-SKY, since q returns t, none of the nodes in the subtree of q matches t. This contradicts the assumption that both q and q' return t.

Given the prefix-free rule, a crude upper bound for the number of nodes returning a tuple is $w \leq |S|^m$, where $|S|$ is the number of skyline tuples. This is because a query can have at most m predicates, each with a different attribute and a value (i.e., v as in $A_i < v$) equal to that of one of the skyline tuples (i.e., $v = t[A_i]$ where t is a skyline tuple). Since no query of concern can be the prefix of another, the maximum number of such queries is $O(|S|^m)$. Given this bound, the maximum number of nodes in the tree is $O(|S| \cdot (|S|^m) \cdot (m + 1)) = O(m \cdot |S|^{m+1})$.

One can make two observations from this worst-case bound: First, the query cost of SQ-DB-SKY depends on the number of *skyline* tuples, not the total number of tuples. This is good news because, as prior research on skyline sizes [38] shows,

the number of skyline tuples is likely orders of magnitude smaller than the number of tuples. Another observation, however, is seemingly bad news: the worst-case cost grows exponentially with the number of attributes m. Fortunately, this is mostly the artifact of an arbitrary system ranking function we must assume in the worst-case analysis, rather than an indication of what happens in practice. To understand why, consider what really happens when the worst-case result strikes, i.e., a tuple t is returned by queries with $\Omega(m)$ predicates.

Consider a Level-m node returning t. Let its 2-tuple sequence be $\langle t_1, A_1 \rangle$, ..., $\langle t_m, A_m \rangle$. What this means is not only that t outperforms t_i on A_i for all $i \in [1, m]$, but also that t_m does the same (i.e., outperforms t_i on A_i) for all $i \in [1, m-1]$, t_{m-1} for all $i \in [1, m-2]$, etc. In other words, this tuple t is likely ranked highly on many attributes - yet its overall rank is too low to be returned by any of the m predecessor queries. While this could occur for an ill-behaved system ranking function, it is difficult to imagine a reasonable ranking function doing the same. As we show as follows, so long as we assume a "reasonable" ranking function, the worst-case query cost can indeed be reduced by orders of magnitude, no matter what the underlying data distribution is.

Average-Case Analysis: By "average-case" analysis, we mean an analysis done based on a single assumption: the system ranking function is *random among skyline tuples* - i.e., for any query q, the ranking function returns a tuple chosen uniformly at random from $S(q)$, i.e., the set of skyline tuples matching q. One can see that this represents the "average" case as a randomly chosen skyline tuple from $S(q)$ can be considered an average of the top-1 selections of all legitimate ranking functions given q and the database D. As we shall discuss after this analysis, this is likely still "worse" than what happens in practice. Yet even this conservation assumption is enough to significantly reduce the worst-case query cost.

The most important observation for our average-case analysis can be stated as follows: The expected query cost (taken over the aforementioned randomness of the system ranking function) of SQ-DB-SKY is a *deterministic function* of the number of skyline points $|S|$, regardless of how the tuple are actually distributed.

To understand why, we start from the simplest case of $|S| = 1$. In this case,

the SELECT * query returns the single skyline tuple, while the m branches of it all return empty, finishing the algorithm execution. In other words, the query cost is always $C_1 = m + 1$ (where the subscript 1 stands for $|S| = 1$). Now consider $|S| = 2$. Here, depending on which tuple is returned by SELECT *, some of its m branches may be empty; while some others may return the other skyline tuple. Let m_0 be the number of empty branches. For the $(m - m_0)$ non-empty branches, we essentially need C_1 queries to examine each and its m sub-branches (all of which will return empty). One can see that the overall query cost will be

$$C_2 = 1 + m_0 + (m - m_0) \cdot C_1. \tag{3}$$

Interestingly, regardless of how tuples are distributed, the above-described random ranking always yields $E(m_0) = m/2$ and thus

$$E(C_2) = 1 + m/2 + C_1 \cdot m/2, \tag{4}$$

where the expected value $E(\cdot)$ is taken over the randomness of the ranking function. To see why, note that m_0 is indeed the number of attributes on which the tuple returned by SELECT * outperforms the other tuple in the database. Since the ranking function chooses the returned tuple uniformly at random, the expected value of m_0 is always $m/2$ regardless of what the actual values are.Similarly when $|S| > 2$, $C_s = 1 + m_0 + m_1 \cdot C_1 + \ldots + m_{s-1} \cdot C_{s-1}$, where m_i is the number of attributes on which i skyline tuples outrank the tuple returned by SELECT * (t_0). Since the probability that t_0 is outranked by i skyline tuples on a given attribute is $1/s$, the expected number of such attributes is m/s. Consequently, the expected query cost of SQ-DB-SKY is

$$E(C_s) = 1 + \frac{m}{s} \cdot \sum_{i=0}^{s-1} E(C_i) \tag{5}$$

where $C_0 = 1$. With Z-transform and differential equations,

$$E(C_s) = \frac{m((m+s-1)! - (m-1)!s!)}{(m-1)(m-1)!s!}. \tag{6}$$

For example, when $m = 2$, we have $E(C_s) = 2s$.

We now show why this average-case query cost is orders of magnitude smaller than the worst-case result. First, since $E(C_i) \geq m + 1$ for all $i \geq 1$, we can derive from (5) that

$$E(C_s) \leq \frac{m+1}{m} \cdot \frac{m}{s} \cdot \sum_{i=0}^{s-1} E(C_i) = \frac{m+1}{s} \cdot \sum_{i=0}^{s-1} E(C_i) \tag{7}$$

Clearly, if we set F_i such that $F_0 = 1$ and $F_s = ((m+1)/s) \cdot \sum_{i=0}^{s-1} F_i$, then we have $E(C_i) \leq F_i$ for all $i \geq 0$. Consider the ratio between F_s and F_{s-1} when $s \gg m$. Note that $F_{s-1} = (m+1)/(s-1) \cdot \sum_{i=1}^{s-2} F_i$ - i.e.,

$$\sum_{i=1}^{s-1} F_i = \frac{s+m}{m+1} \cdot F_{s-1}. \tag{8}$$

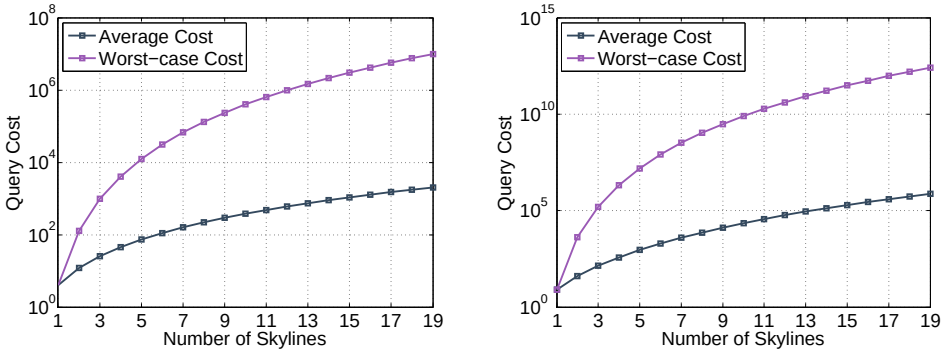
In other words,

$$E(C_s) \leq F_s = \frac{m+1}{s} \cdot \frac{s+m}{m+1} \cdot F_{s-1} = \frac{s+m}{s} F_{s-1} \tag{9}$$

$$= \frac{(s+m)!}{s! \cdot m!} = \binom{s+m}{m} \tag{10}$$

$$\leq \left(\frac{(s+m) \cdot e}{m}\right)^m = \left(e + \frac{e \cdot s}{m}\right)^m \tag{11}$$

One can see that the growth rate of F_s (with $|S|$) is much slower than what is indicated by the worst-case analysis - specifically, the base of exponentiation is approximately $(e/m) \cdot |S|$ instead of $|S|$. Figure 12 confirms this finding by showing the average and worst-case cost of SQ-DB-SKY for the cases where $m = 4$ and $m = 8$. One can observe from the figures the significantly smaller query cost indicated by the average-case analysis.



(a) m=4 (b) m=8

Figure 12: Comparing worst and average cost of SQ-DB-SKY

Before concluding the average case analysis, we would like to point out that even this analysis is likely an overly conservative one. To understand why, note from (3) that the smaller m_0 is, i.e., the more branches return empty, the smaller the query cost will be. In the average-case analysis, since we assume a random order of skyline tuples, $E(m_0) = m/|S|$, i.e., the top-ranked tuple returned by SELECT * features the top-ranked value on an average of $m/|S|$ attributes. Clearly, with a real-world ranking function, this number is likely to be much higher, simply because the more "top" attributes values a tuple has, the more likely a reasonable ranking function would rank the tuple at the top. As a result, the query cost in practice is usually even lower than what the average-case analysis suggests, as we show in the experimental results.

4.3 Skyline Discovery for RQ-DB

We now consider the RQ-DB case where range queries support two-ended ranges, rather than one-ended as in the SQ-DB case. Since RQ-DB has a more powerful interface, a straightforward solution here is to directly use Algorithm SQ-DB-SKY. One can see that the algorithm still guarantees complete skyline discovery.

The problem with this solution, however, lies in cases where $|S|$, the number of skyline tuples, is large. Specifically, when $|S|$ approaches the database size n, the worst-case query cost may actually be larger than the baseline query cost

of $O(m \cdot n)$ for crawling the entire database over a RQ-DB interface [36]. This indicates what SQ-DB-SKY fails to (or cannot, as it was designed for SQ-DB) leverage - i.e., the availability of both ends on range queries - may reduce the query cost significantly when $|S|$ is large. We consider how to leverage this opportunity in this section.

4.3.1 Key Idea: Algorithm RQ-DB-SKY

A Simple Revision and Its Problem: Our first idea for reducing the query cost stems from a simple observation on the design of q_2 to q_4 described above: Instead of having them as three *overlapping* queries, we can revise them to be *mutually exclusive*:

q_2: WHERE $A_1 < t_1[A_1]$
q_3: WHERE $A_1 \geq t_1[A_1]$ & $A_2 < t_1[A_2]$
q_4: WHERE $A_1 \geq t_1[A_1]$ & $A_2 \geq t_1[A_2]$ & $A_3 < t_1[A_3]$

With this new design, all m branches from a node in the tree (Figure 9) represent mutually exclusive queries. Interestingly, the completeness of skyline discovery is not affected! For example, any skyline tuple other than t_1 still belongs to at least one of q_2 to q_4.

The effectiveness of this revision is evident from one key observation - because of the mutual exclusiveness and the (still valid) completeness of skyline discovery, now every skyline tuple is returned by *exactly* one node in the tree. While this seemingly solves all the problems in the query-cost analysis for SQ-DB-SKY, it unfortunately introduces another challenge:

Unlike in SQ-DB-SKY where the top-1 tuple returned by every node is a skyline tuple, with this revised tree, a node might return a tuple *not* on the skyline as the No. 1. This can be readily observed from the design of q_2 and q_3: it is now possible for a tuple returned by q_3 to be dominated by q_2 - as the space covered by q_3 now excludes the space of q_2. Because of this new problem, the worst-case query cost for this revised algorithm becomes $O(n \cdot m)$, as it is now possible for each of the n tuples in the database (even those not on the skyline) to be returned by a interior node in the tree. While this bound may still be smaller than that of

SQ-DB-SKY when $|S|$ approaches n, it may also be much worse when $|S|$ is small. Since we do not have any prior knowledge of $|S|$ before running the algorithm, we need a solution that adapts to the different $|S|$ and offers a consistently small query cost in all cases.

Algorithm RQ-DB-SKY: To achieve this, our key idea is to combine SQ-DB-SKY with the above-described revision to be the more efficient of the two. To understand the idea, note a 1-1 correspondence between the tree constructed in SQ-DB-SKY and the revised tree: In the revised tree, we map every query q in the tree of SQ-DB-SKY to a query $R(q)$ covering all value combinations matching q but not any q' in SQ-DB-SKY which appears before q in the (depth-first) post-order traversal of the tree. Based on this 1-1 mapping, RQ-DB-SKY works as follows.

We traverse the tree in SQ-DB-SKY and issue queries in depth-first preorder. A key additional step here is that, for each query q in the tree, before issuing it, we first check all tuples returned by previously issued queries and check if any of these tuples match q. If none of them does, then we proceed with issuing q and continuing on with the traversal process.

Otherwise, if at least one previously retrieved tuple matches q, then instead of issuing q, we issue its *counterpart* $R(q)$. If $R(q)$ is empty, no new skyline tuple can be discovered from the subtree of q. Thus, we should abandon this subtree and move on. If $R(q)$ returns as No. 1 a tuple t, then either t is dominated by a previously retrieved (skyline) tuple, or it must be a (new) skyline tuple itself. Either way, we must have never seen t before in the answers to the issued queries. If t is dominated by a previously retrieved tuple, say t', then we generate the children of q according to t'. Otherwise, we generate them according to t. In either case, we continue on with exploring the subtree of q in depth-first preorder. Algorithm 4 depicts the pseudocode of RQ-DB-SKY.

The correctness of RQ-DB-SKY follows directly from that of SQ-DB-SKY, because these two algorithms essentially follow the exact same query sequence with only one exception: In RQ-DB-SKY, when we are certain from the answer to $R(q)$ that no skyline tuple could possibly be discovered from the subtree of q, we forgo the exploration of this subtree and move on. Instead, SQ-DB-SKY

Algorithm 4 RQ-DB-SKY

```
1: S = ∅
2: seen = ∅
3: while traversing the SQ-DB-SKY tree in depth first preorder and at each q in
   the tree do
4:    if ∄ t ∈Seen that matches q then
5:       T = Top-k(q)
6:       if T contains k tuples then
7:          generate the children of q based on T_0
8:       end if
9:    else
10:      T = Top-k(R(q))
11:      if T contains k tuples then
12:         if ∃t′ ∈ S that dominates T_0 then
13:            generate the children of q based on t′
14:         else
15:            generate the children of q based on T_0
16:         end if
17:      end if
18:   end if
19:   Update S by T;
20:   seen = seen ∪ T
21: end while
```

does not have this early-termination detection (because the SQ-DB interface does not support $R(q)$), and therefore has to complete the useless subtree exploration process. As we shall show in the next subsection, this early-termination detection can lead to a significant saving of query cost, especially when the number of skyline tuples $|S|$ is large.

Theorem 5. *Algorithm RQ-DB-SKY is guaranteed to discover all skyline tuples.*

Proof. The proof can be constructed in analogy to that of Theorem 4. The only difference is that, unlike in the proof for SQ-DB-SKY where t might match more than one of the m branches of a node, here t must match *exactly* one of the m branches, simply because these m branches are mutually exclusive by design in RQ-DB-SKY. Despite of this difference, the logic of the proof stays exactly the

same: there must be exactly one branch of the root satisfying t because otherwise t would be dominated by the tuple returned by the root. Recursively, we can construct a path from the root to a leaf node in the tree, such that t satisfies each and every node on the path. Since every leaf node of the tree is a valid or underflowing query, this means that the leaf node must return t, contradicting the assumption that t is not discovered. □

Once again, let us consider the dummy example provided in Figure10, and its corresponding RQ-DB-SKY tree in Figure 13. One can see that applying $R(q_4)$= WHERE $A_2 \geq 3$ AND $A_3 < 7$, instead of q_4, causes that each skyline tuple appears in exactly one of the branches.

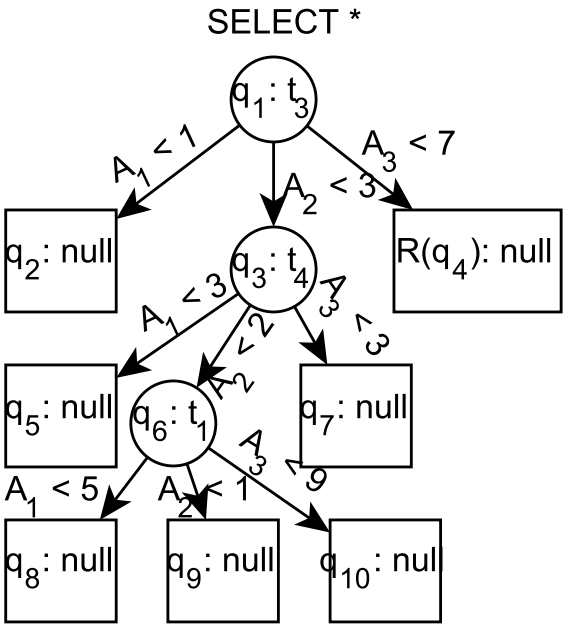
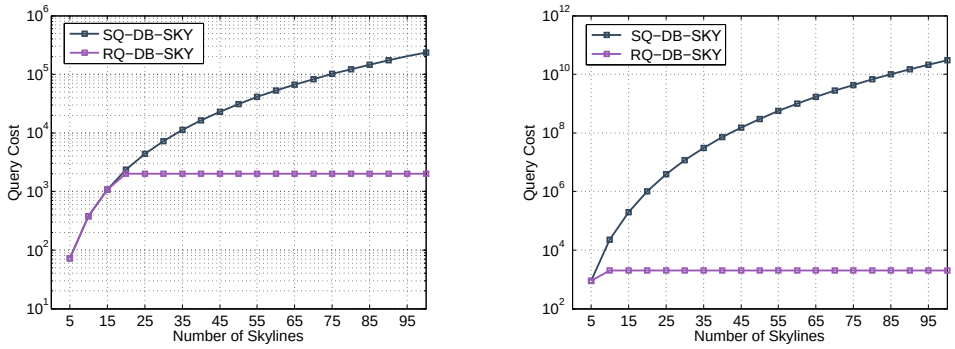


Figure 13: RQ-DB-SKY example tree



(a) 4D (b) 8D

Figure 14: simulation results for RQ-DB-SKY, in comparison with SQ-DB-SKY

4.3.2 Query-Cost Analysis

The key to the query-cost analysis of RQ-DB-SKY is to count the number of internal, i.e., interior, nodes of the tree. There are two important observations: First, the SQ-query q of a interior node must match at least one skyline tuple, as otherwise it would have to return empty which makes the node a leaf. Second, if a interior node is not the first (according to preorder) which returns the skyline tuple, then the node's RQ-query (i.e., $R(q)$) must return a unique tuple in the database that does not match any node accessed before it, because otherwise the node would return empty and become a leaf. With these two observations, an upper bound on the number of internal nodes is $\min(|S|^{m+1}, n)$. As a result, the total query cost of RQ-DB-SKY is $O(m \cdot \min(|S|^{m+1}, n))$.

One might wonder if, for RQ-DB-SKY, we can derive a similar result to the average-case analysis of SQ-DB-SKY which is oblivious to the data distribution. Unfortunately, the query cost of RQ-DB-SKY is data-dependent. The reason is simple: the query cost of RQ-DB-SKY is essentially determined by how many non-skyline tuples match and are returned by the RQ-queries $R(q)$. This number, however, depends on the data distribution: e.g., if all non-skyline tuples are dominated by the skyline tuple returned by SELECT *, then the query cost of RQ-DB-SKY can be extremely small ($\leq m \cdot |S|$). Meanwhile, if very few non-skyline tuples are dominated by skyline tuples returned from nodes at the top of the tree,

then RQ-DB-SKY requires many more queries.

Because of the data-dependent nature of RQ-DB-SKY's query cost, to demonstrate the power of its early-termination idea, we resort to the numeric simulations conducted in Section 4.2. Figure 14 depicts how the query costs of SQ- and RQ-DB-SKY change with the percentage of tuples on the skyline (when the database contains 2000 tuples each with 2 Boolean i.i.d. uniform-distribution attributes). Note that we control the percentage of skyline tuples by adjusting the correlation between the two attributes, where positive correlation leads to fewer skyline tuples. Interestingly, one can observe from the figure that while the performance of RQ- and SQ- do not differ much when $|S|$ is small, RQ- has a much smaller query cost when $|S|$ is large - consistent with the theoretical analysis.

4.4 Skyline Discovery for PQ-DB

We now turn our attention point-query PQ-predicates. We first discuss the 2D case (i.e., a database with two attributes) and present an instance-optimal solution PQ-2D-SKY. Then, after pointing out the key differences between 2D and higher dimensional cases, we present Algorithm PQ-DB-SKY, which discovers all skyline tuples from a higher dimensional database by calling (a variation of) PQ-2D-SKY as a subroutine.

4.4.1 2D Case

Design of Algorithm PQ-2D-SKY: We start with SELECT * which is guaranteed to return a skyline tuple, say (x_1, y_1). As shown in Figure 15, we can now prune the 2D search space (for skyline tuples) into two disconnected subspaces, both rectangles. One has diagonals $(0, y_{\max})$ and (x_1, y_1), while the other has (x_1, y_1) and $(x_{\max}, 0)$, where $x_{\max}$ and $y_{\max}$ are the maximum values for x and y, respectively. We do not need to explore the rectangle with diagonals $(0, 0)$ and (x_1, y_1) because there is no tuple in it (as otherwise it would dominate (x_1, y_1)). We do not need to explore the rectangle with diagonals (x_1, y_1) and $(x_{\max}, y_{\max})$ either because all tuples in it must be dominated by (x_1, y_1).

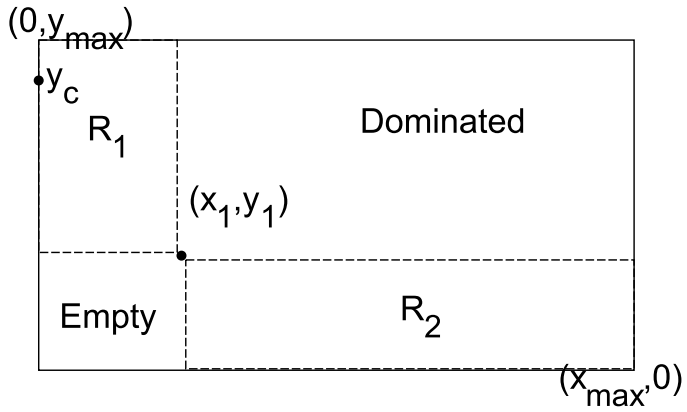


Figure 15: Pruning, R_1, R_2, & demo of algorithm execution

From this point forward, our goal becomes to discover skyline tuples by issuing 1D queries - i.e., queries of the form of either $x = x_0$ or $y = y_0$. An important observation here is that any 1D query we issue will "affect" (precise definition to follow) exactly *one* of the two above-described subspaces. For example, if $x_0 > x_1$, query $x = x_0$ affects only R_2 in Figure 15: It either proves part of the rectangle to be empty (when the query returns empty or a tuple with $y > y_1$), or returns a tuple in the second rectangle that dominates all other tuples with $x = x_0$. In either case, Rectangle R_1 remains the same and still needs to be explored. As another example, if $y_0 > y_1$, then query $y = y_0$ affects only R_1.

This observation actually leads to a simple algorithm that is guaranteed to be optimal in terms of query cost: at any time, pick one of the remaining (rectangle) subspaces to explore. Let the diagonal points of the subspace be $(x_{\rm L}, y_{\rm T})$ and $(x_{\rm R}, y_{\rm B})$, where $x_{\rm L} \leq x_{\rm R}$ and $y_{\rm T} \geq y_{\rm B}$. If $x_{\rm R} - x_{\rm L} < y_{\rm T} - y_{\rm B}$, then we issue query $x = x_{\rm L}$. Otherwise, we issue $y = y_{\rm B}$. For example, in Figure 15, if $x_{\max} - x_1 > y_1$, we issue $y = 0$.

Note the implications of the query answer on the remaining subspace to search: Consider query q: $x = x_{\rm L}$ as an example. If q returns empty, then the subspace is shrunk to between $(x_{\rm L} + 1, y_{\rm T})$ and $(x_{\rm R}, y_{\rm B})$. Otherwise, if q returns $(x_{\rm L}, y_2)$, then the subspace is shrunk to between $(x_{\rm L} + 1, y_2)$ and $(x_{\rm R}, y_{\rm B})$. Either way, the

subspace becomes smaller and remains disjoint from other remaining subspace(s). For example, in Figure 15, if $y = 0$ is empty, R_2 is shrunk to between (x_1, y_1) and $(x_{max}, 1)$. Otherwise, if it returns $(x_2, 0)$, then the subspace is now between (x_1, y_1) and $(x_2, 1)$.

What we do next is to simply repeat the above process, i.e., pick a subspace, determine whether the width or height is larger, and issue the corresponding query. This continues until no subspace remains. Algorithm 5 depicts the pseudo code for PQ-2D-SKY.

Algorithm 5 PQ-2D-SKY

```
1: T = Top-k(SELECT * FROM D);
2: S = {T_0}
3: Partition search space into rectangles R_1 and R_2 based on T_0
4: while search space is not fully explored do
5:    Pick a rectangle and identify point query q to issue
6:    T = Top-k(q);
7:    S = S ∪ T_0
8:    if T contains k tuples then
9:       prune search space based on T_0
10:   end if
11: end while
```

Instance Optimality Proof: We now prove the instance optimality of PQ-2D-SKY, i.e., for any given database, there is no other algorithm that can use fewer queries to discover all skyline tuples *and prove* that all skyline tuples have been discovered. Note that the latter requirement (i.e., proof of completeness) is important. To see why, consider an algorithm that issues SELECT * and then stops. For a specific database that contains only one skyline tuple, this algorithm indeed finds all skyline tuples extremely efficiently. But it is not a valid solution because it cannot guarantee the completeness of skyline discovery.

We prove the instance optimality of PQ-2D-SKY by contradiction: Suppose there exists an algorithm $\mathcal{A}$ that requires fewer queries. Consider the (rectangle) subspace between $(x_{\mathrm{L}}, y_{\mathrm{T}})$ and $(x_{\mathrm{R}}, y_{\mathrm{B}})$. If $x_{\mathrm{R}} - x_{\mathrm{L}} < y_{\mathrm{T}} - y_{\mathrm{B}}$ yet $\mathcal{A}$ does *not* issue $x = x_{\mathrm{L}}$, then the only alternative is to issue queries $y = y_{\mathrm{B}}$, $y_{\mathrm{B}} + 1$, ...,

y_c, where y_c is the y-coordinate value of the tuple returned by $x = x_{\mathrm{L}}$ or, in the case where $x = x_{\mathrm{L}}$ returns empty, $y_c = y_{\mathrm{T}}$. An example of this is illustrated in Figure 15: Suppose $y_{max} - y_1 > x_1$. If $\mathcal{A}$ does not issue $x = 0$, then it must issue $y = y_1$, y_2, ..., y_c. This is because, in order to guarantee the completeness of skyline discovery, one must "prove" the emptiness of points $(x_{\mathrm{L}}, y_{\mathrm{B}})$, $(x_{\mathrm{L}}, y_{\mathrm{B}} + 1)$, ..., $(x_{\mathrm{L}}, y_c - 1)$, (resp. $(x_0, y_1), \ldots, (x_0, y_c - 1)$ in Figure 15) while retrieving tuple (x_{L}, y_c) (resp. (x_0, y_c) in Figure 15). Given that $x = x_{\mathrm{L}}$ is not issued, the only feasible solution is to issue the above-described $y = y_i$ queries.

Yet this contradicts the optimality of Algorithm $\mathcal{A}$. To understand why, consider two cases respectively: First is when $x = x_{\mathrm{L}}$ returns empty. In this case, $\mathcal{A}$ calls for $y_{\mathrm{T}} - y_{\mathrm{B}} + 1$ queries to be issued, while PQ-2D-SKY issues at most $x_{\mathrm{R}} - x_{\mathrm{L}}$ queries. Since $x_{\mathrm{R}} - x_{\mathrm{L}} < y_{\mathrm{T}} - y_{\mathrm{B}}$, $\mathcal{A}$ is actually worse. Now consider the second case, where $x = x_{\mathrm{L}}$ does return a tuple (x_{L}, y_c). In this case, $\mathcal{A}$ calls for c queries to be issued. We also require at most c queries, as $y = y_c$ is no longer needed given the answer to $x = x_{\mathrm{L}}$. This again contradicts the superiority of $\mathcal{A}$.

Query Cost Analysis: Having established the instance optimality of PQ-2D-SKY, we now analyze exactly how many queries it needs to issue. Let A_1 and A_2 be the two attributes and $t_1, \ldots, t_{|S|}$ be the skyline tuples in the database. Without loss of generality, suppose t_i is sorted in the increasing order of A_1, i.e., $t_i[A_1] \leq t_{i+1}[A_1]$. Note that, since t_i are all skyline tuples, correspondingly there must be $t_i[A_2] \geq t_{i+1}[A_2]$. Denote as t_0 and $t_{|S|+1}$ the two diagonal points of the domain, i.e., $t_0 = \langle 0, \max(Dom(A_2))\rangle$ and $t_{|S|+1} = \langle \max(Dom(A_1)), 0\rangle$. One can see from the design of PQ-2D-SKY that its query cost is simply

$$C = \sum_{i=0}^{|S|} \min(t_{i+1}[A_1] - t_i[A_1], t_i[A_2] - t_{i+1}[A_2]). \tag{12}$$

Immediately, following Equation 12, a few upper bounds on C are: e.g., $C \leq t_1[A_2]$, $C \leq t_{|S|}[A_1]$, and $C \leq \min_{i\in[1,|S|]} (t_i[A_1] + t_i[A_2])$. These upper bounds indicate a likely small query cost in practice. To understand why, recall that most web interfaces only present a ranking attribute as PQ when it has a small domain. In addition, it is highly unlikely for such an attribute to have empty

domain values - i.e., $v \in Dom(A_i)$ that is not taken by any tuple in the database - because otherwise users of the PQ interface would be frustrated by the empty result returned after selecting $A_i = v$. When every value in $Dom(A_1)$ and $Dom(A_2)$ is occupied, unless the number of skyline tuples is very large, $t_i[A_j]$ is likely small for t_i to be on the skyline, leading to a small query cost in practice. We verify this finding through experimental results in Section 4.7.

4.4.2 Higher-D Case: Negative Results

Unfortunately, the optimal 2D skyline discovery algorithm cannot be directly extended to solve the higher-dimensional cases. This subsection describes two main obstacles which explains why: The first proves that there does not exist any deterministic algorithm that can be instance optimal for higher-D databases like what PQ-2D-SKY achieves for 2D. The second obstacle shows that even if we are willing to abandon optimality and consider a greedy algorithm that deals with each 2D subspace at a time for higher-D databases, PQ-2D-SKY still cannot be directly used.

Fortunately, the second negative result also sheds positive lights towards solving the higher-D problem. As we shall show in the next subsection, it is indeed possible to revise PQ-2D-SKY and retain instance optimality for any 2D subspace of a higher-D database - a result that eventually leads to our design of PQ-DB-SKY for higher-D databases.

Non-existence of Optimal Higher Dimensional Skyline Discovery Algorithms: The first obstacle brought by higher-D skyline discovery that makes it impossible for any deterministic algorithm to achieve instance optimality as in the 2D case discussed above. Here we shall first describe the obstacle, and then discuss why it eliminates the possibility of having an optimal (deterministic) skyline discovery algorithm.

The obstacle here is the loss of a property which we refer to as "guaranteed single skyline return" - i.e., every 1D query (which is the focus of consideration in 2D skyline discovery) is guaranteed to return the (at most one) skyline tuple covered by the query. 2D and higher dimensional queries, on the other hand, may

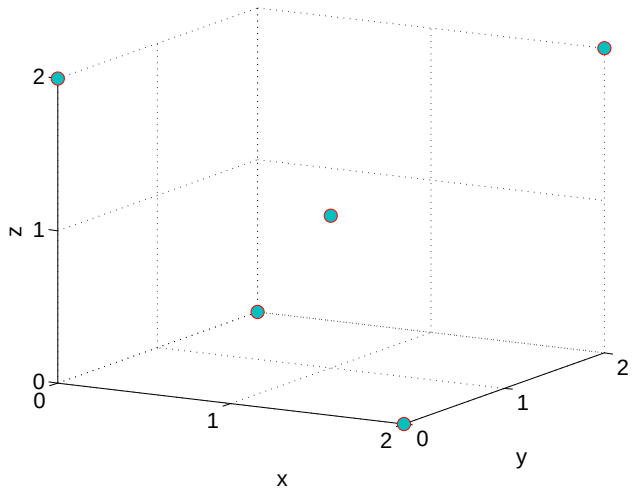


Figure 16: Illustration of negative-proof construction

not reveal all skyline tuples. Specifically, even when $k > 1$, some of the returned tuples may not be skyline tuples even when there are skyline tuples matching the query that are not returned.

This property makes it no longer possible to guarantee the optimality of skyline discovery without knowledge of the actual data distribution. To understand why, consider a simple example depicted in Figure 16, where the database features a top-2 interface (i.e., $k = 2$) and contains the following five tuples (in addition to potentially many others): (1, 1, 1), (2, 2, 2), (2, 0, 0), (0, 2, 0), (0, 0, 2). Suppose that the SELECT * query returns (1, 1, 1) and (2, 2, 2); while SELECT * FROM D WHERE $z = 0$ returns (2, 0, 0) and (0, 2, 0). Further assume that neither query SELECT * FROM D WHERE $x = 0$ nor WHERE $y = 0$ returns more than one skyline tuples - e.g., say the first one returns (0, 2, 0) and (0, 3, 0) and the latter returns (2, 0, 0) and (3, 0, 0).

The first observation we make here is that the optimal query plan consists of only 3 queries (no matter what the other tuples are):

SELECT * FROM D

SELECT * FROM D WHERE $z = 0$

SELECT * FROM D WHERE $x = 0$ AND $y = 0$

One can see that, given the above setup, these three queries are guaranteed to return all five aforementioned tuples, which by themselves prove that there are only four skyline tuples in the database: (1, 1, 1), (2, 0, 0), (0, 2, 0), and (0, 0, 2), because any other possible value combination must fall into one of the two categories: Either it is dominated by at least one of the four tuples, or it must be one of (0, 0, 0), (1, 0, 0), (0, 1, 0), (0, 0, 1). Nonetheless, these four value combinations have been proven non-existent in the database as otherwise it must be returned by the 3 issued queries.

The next critical observation is that any optimal query plan *must* contain SELECT * FROM D WHERE $z = 0$. The reason here is simple: given the four skyline tuples and the above assumptions, the only query that returns more than one skyline tuple is SELECT * FROM D WHERE $z = 0$. In other words, if this query is not included in a query plan, then the query plan must contain at least four queries - i.e., it is *not* an optimal plan.

It is exactly this observation which eliminates the existence of a deterministic yet instance-optimal skyline discovery algorithm. To understand why, consider a slight change to the query-answer setup when all tuples in the database remain exactly the same: now query WHERE $z = 0$ returns $(0, 2, 0)$ and $(0, 3, 0)$, while WHERE $x = 0$ returns $(0, 2, 0)$ and $(0, 0, 2)$. In analogy to the above analysis, now the optimal query plan *must* contain SELECT * FROM D WHERE $x = 0$ (along with, say, SELECT * and SELECT * FROM D WHERE $y = 0$ AND $z = 0$, to make a 3-query optimal plan) and must *not* contain SELECT * FROM D WHERE $z = 0$.

The problem with this new setup, however, is that no deterministic algorithm can achieve optimality in both this setup and the original one, because it simply cannot distinguish between the two cases without committing to a query that is part of the optimal plan for one but *cannot* be part of the optimal plan for the other. For example, the SELECT * query returns the exact same answer in the two cases - making it impossible to make the distinction. On the other hand, while SELECT * FROM D WHERE $z = 0$ does distinguish between the two cases, the very issuance of this query already means the loss of instance optimality, as it cannot be part of

an optimal query plan for the second setup. More formally, the only queries that enable the distinction are those that have different query answers between the two cases - i.e., query SELECT * FROM D WHERE $z = 0$ and query WHERE $x = 0$ - yet neither can appear in both optimal query plans.

One can observe from the above discussions what makes instance-optimal skyline discovery impossible over a higher-D database: Unlike in the 2D case where every 1D query always returns the one and only skyline tuple matching the query (or returns empty) no matter what the ranking function actually is, in higher-D cases whether and how many skyline tuples are "hidden" from the answer to a matching 2D query depends on the ranking function. Since the algorithm has no prior knowledge of the ranking function (which might even differ for different queries), it has to rely on the returned query answers to determine which queries to issue next. This eliminates the possibility of an instance-optimal algorithm because, by the time the algorithm can learn enough information about the ranking function and the underlying database, it may have already issued unnecessary queries, making the algorithm suboptimal.

No Direct Extension to Optimal 2D Subspace Discovery: Since the above obstacle eliminates the possibility of an instance-optimal h-D skyline discovery algorithm, we now turn our attention to a simpler, greedy, version of the solution: how about we partition the higher-dimensional space into mutually exclusive 2D subspaces, and then run the instance-optimal 2D skyline discovery algorithm over each subspace?

Unfortunately, even in this case, the 2D algorithm cannot be directly applied without losing its optimality. To understand why, recall that in the 2D case, there is a clean "separation" of effect for a query answer: no matter what the query answer is, it shrinks one and exactly one rectangle subspace (to another rectangle). This enables us to devise a divide-and-conquer approach which focuses on one (rectangle) subspace at a time.

This clean property, however, is lost once the dimensionality increases to 3 (and above). For example, consider the search space in the 3D case after pruning based on the answer of SELECT * FROM D, say point (x, y, z). One can see that the

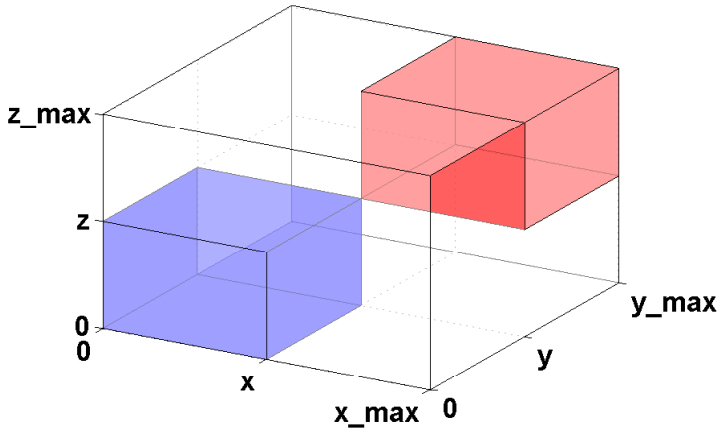


Figure 17: Illustration of example

pruning carves out two (small) cubes from the original space - one with diagonal points being $(0, 0, 0)$ and (x, y, z); while the other with diagonal points being (x, y, z) and $(x_{\max}, y_{\max}, z_{\max})$. The pruned result, however, is still one connected space with a complex shape as depicted in Figure 17, which may become more and more complex once pruning is done with additional query answers.

After projecting the pruned subspace to each 2D subspace, one can see that the pruned space is now of the shape of a rectangle "minus" a number of smaller rectangles. For example, consider a simple 3D case where the domains of x, y and z are [0, 6], [0, 9], and [0, 1], respectively. Suppose that the SELECT * query returns tuple (4, 6, 1), while SELECT * FROM D WHERE $z = 0$ returns tuple (0, 9, 0). We now consider the problem of skyline discovery the 2D subspace defined by query WHERE $z = 0$.

Figure 18 depicts the shape of this subspace after pruning based on the SELECT * query. One can see that three rectangles are excluded from the original space of $[0, 6] \times [0, 9]$. One is $x = 0$ - it is removed because the return of (0, 9) guarantees no other tuple with $z = 0$ could have $x = 0$. Another is $y = 9$ - it is excluded because any tuple within must be dominated by (0, 9). The final excluded rectangle

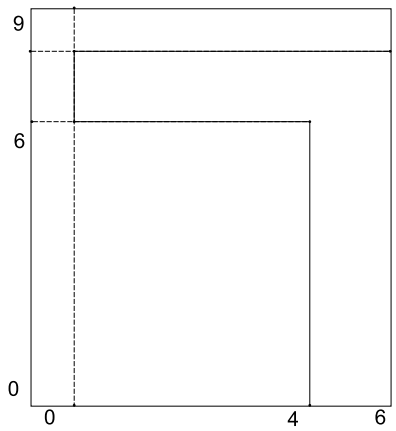


Figure 18: Illustration of example

has diagonals (0, 0) and (4, 6). It is excluded because, according to the SELECT * query answer, we are assured that no tuple exists with $x \leq 4$, $y \leq 6$, and $z = 0$, as otherwise this tuple must have been returned ahead of (4, 6, 1).

Note, however, a significant difference with the original 2D case: the rectangle with diagonals (4, 6) and (6, 9) is *not* removed. Unlike in the original case where the other diagonal rectangle can also be removed because all points in it are dominated by the returned tuple, in this new case the pruning of rectangle (0,0)-(4,6) is not based on a tuple with $z = 0$. As such, it is still possible for a tuple in rectangle (4,6)-(6,9) to be a skyline tuple. In the following discussions, one can see that it is exactly this change which introduces additional complexity to the design of 2D-subspace skyline discovery in higher-dimensional databases.

We now show that the 2D algorithm loses its optimality when being applied to this pruned 2D space. Note that, according to the algorithm, we shall start with issuing $x = 1, 2, \ldots$ because the domain size of x (i.e., 6 after pruning) is smaller than that of y (i.e., 9 after pruning). Consider the case when there is only one more tuple (in addition to (0, 9) returned by the SELECT * query) in this subspace: (5, 0). One can see that this algorithm will issue 5 queries - i.e., $x = 1, \ldots, 5$, after which it stops execution because all the subspace can then be pruned by (5, 0). Nonetheless, the optimal query plan for this subspace consists of only 3 queries

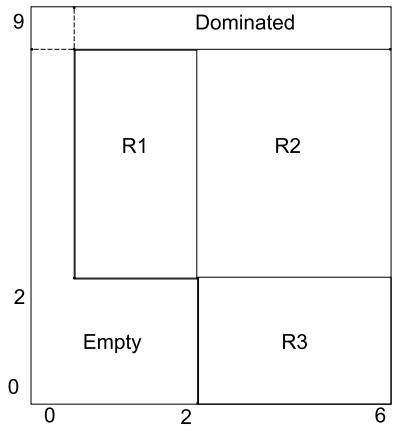


Figure 19: Illustration of example

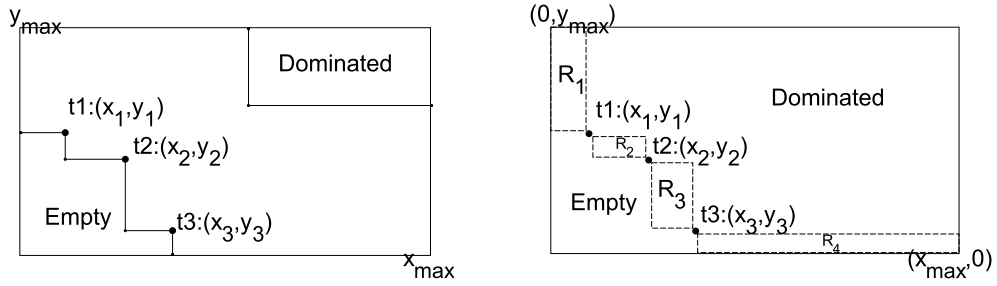
- e.g., WHERE $x = 5$, WHERE $y = 7$, and WHERE $y = 8$. This shows that the original 2D algorithm is no longer optimal for this subspace skyline discovery task.

Note that this problem cannot be simply solved by partitioning the subspace into rectangles before applying the 2D algorithm over each. This can be seen from another simple construction: Consider a change to the above database which makes (2, 2, 1) the tuple returned by the SELECT * FROM D query. The pruned subspace in this case is depicted in Figure 19. One can see that, if we partition the subspace into the three rectangles marked in the figure, then we would have issued queries WHERE $y = 0$ and WHERE $y = 1$ for the bottom rectangle - yet these two queries cannot be in the optimal plan when there is no tuple other than (0, 9, 0) on the plane.

4.4.3 Algorithm PQ-DB-SKY

Optimal 2D Subspace Skyline Discovery

To develop an instance-optimal algorithm for discovering all skyline in a 2D subspace, we start by considering the possible shape of such a subspace. As discussed above, the subspace may be pruned by answers to queries that contain the subspace. Without loss of generality, consider a 2D subspace $\mathcal{S}$ "spanning" on attributes A_1 and A_2. Let $\mathcal{S}[A_i]$ ($i > 2$) be the value of the subspace on any



(a) Example of empty and dominated area (b) "Block-diagonal" rectangles

Figure 20: Illustration of idea for PQ-2DSUB-SKY

other attribute. If a query containing the subspace returns a tuple t such as $\forall i > 2$, $t[A_i] \geq \mathcal{S}[A_i]$, then we can prune from $\mathcal{S}$ the rectangle with diagonals (0, 0) and $(t[A_1], t[A_2])$, because any tuple in this rectangle would dominate t, contradicting the fact that t is returned by a query containing $\mathcal{S}$. Figure 20a depicts such a scenario.

Besides such pruning, another possible way to prune $\mathcal{S}$ is to exclude from it rectangles that we are no longer interested in. For example, if we have retrieved a tuple t such that $\forall i > 2$, $t[A_i] \leq \mathcal{S}[A_i]$, then we are no longer interested in the rectangle corresponding to $A_1 \geq t[A_1]$ and $A_2 \geq t[A_2]$, because any other in the rectangle would be dominated by t and therefore cannot be a skyline tuple. One can see that the end result of pruning is a shape like what is depicted in Figure 20a.

Given the pruned shape, the key idea of our PQ-2DSUB-SKY algorithm is depicted in Figure 20 and can be stated as follows. First, we remove all rows and columns that have already been completely pruned. Then, we consider a series of "block-diagonal" rectangles as depicted in Figure 20b. Formally, if $t_1 : (x_1, y_1)$ and $t_2 : (x_2, y_2)$, as shown in the figure, are adjacent "lower-bound" skyline points in the subspace, then we add to the series a rectangle with diagonals (x_1, y_1) and (x_2, y_2).

There are two critical observations here that lead to the instance optimality of this idea: First, no matter which tuples there are in the database, these rectangles must be covered for a complete discovery of all skyline tuples. For example,

consider a point outside these rectangles, say $(x_2 + 1, y_1 + 1)$. If (x_2, y_1) turns out to be occupied by a tuple, then $(x_2 + 1, y_1 + 1)$ can be pruned without any query covering the rectangle containing it (say the one with diagonals (x_2+1, y_1+1) and $(x_{\max}, y_{\max})$). On the other hand, any point inside one of the series of rectangles cannot be pruned unless there has been at least one query "hitting" the rectangle containing the point.

The second critical observation is that *at least* one of the series of rectangle must "agree" with the overall subspace (i.e., the one after removing all completely pruned rows and columns) on which dimension to follow for discovery (as dictated by the skyline discovery rule in Algorithm PQ-2D-SKY). The underlying reason is straightforward: let the width and height of each rectange in the series, say R_i, be w_1 and h_1, respectively. Note that the overall width and height of the subspace satisfy

$$w = w_1 + \cdots + w_s, \tag{13}$$

$$h = h_1 + \cdots + h_s, \tag{14}$$

where s is the number of rectangles in the series. One can see that, clearly, if $w < h$, there must be at least one rectangle in the series with $w_i < h_i$ - i.e., the rectangle "agrees" with the overall subspace to discover skylines along the y-dimension (by issuing queries of the form SELECT * FROM D WHERE $x = v$).

Based on these two observations, we can now develop Algorithm PQ-2DSUB-SKY. It starts with the above two steps, and then selects any arbitrary rectangle in the series so long as it agrees with the overall pruned subspace on which dimension to follow. Based on the second observation, there must exist at least one such rectangle. We crawl this rectangle using the previously developed 2D skyline discovery algorithm, and then repeat the entire process - i.e., starting once again by removing rows and columns that have been completely discovered. Algorithm 6 depicts the pseudo code of PQ-2DSUB-SKY. The proof of instance optimality for this algorithm is straightforward: As proved in Section 4.4.1, the skyline discovery in each rectangle in the series is instance optimal. As discussed above, any complete discovery of skyline tuples in the subspace must cover all rectangles. Thus, the

Algorithm 6 PQ-2DSUB-SKY

Assuming that A_1 and A_2 create the current subspace $\mathcal{S}$
for queries q that contains $\mathcal{S}$ and tuple t discovered by q **do**
  if $\forall i > 2$, $t[A_i] \geq \mathcal{S}[A_i]$ **then**
    Remove the rectangle (0,0) and $(t[A_1], t[A_2])$ from $\mathcal{S}$
  end if
end for
for discovered tuples t that $\forall i > 2$, $t[A_i] \leq \mathcal{S}[A_i]$ **do**
  Remove the rectangle corresponding to $A_1 \geq t[A_1]$ and $A_2 \geq t[A_2]$ from $\mathcal{S}$
end for
while $\mathcal{S}$ is not completely pruned **do**
  Remove the pruned rows and columns
  Construct the "block-diagonal" rectangles ($\mathcal{R}$) between adjacent "lower-bound" skyline points in the subspace
  Apply PQ-2D-SKY on a rectangle r in $\mathcal{R}$ that agrees with the overall pruned subspace on the dimension to follow
end while

only remaining issue to ensure that the issued queries indeed cover the entire subspace (i.e., containing not only the series of rectangles but the other unpruned part as well). Since the rectangle we choose at each step always has the same discovery direction as the entire subspace, one can see that either the discovery of all rectangles are along the same dimension - i.e., a complete discovery, or the direction changes when a returned tuple triggers pruning of not only the rectangle being processed but also the part of the subspace dominated by the rectangle - i.e., the skyline discovery will still be complete.

Design and Analysis of PQ-DB-SKY: Our proposed technique for higher-dimensional skyline discovery has a key step of applying the application of this algorithm over each 2D subspace of a higher-dimensional space.

As discussed above, instance optimality is lost once the dimensionality reaches 3. A key reason for this is because one does not know which dimension to "crawl first", i.e., how to partition a higher-D space into 2D subspaces (e.g., along x, y or z?). Fortunately, heuristics for dimension selection are easy to identify. The most important factor here is the domain size. To understand why, note that the domain

Algorithm 7 PQ-DB-SKY

1: T = Top-k(SELECT * FROM D); $S = \{T_0\}$
2: Prune search space based on T_0
3: **while** search space is not fully explored **do**
4: Pick the 2D subspace spanning 2 attributes with largest domain sizes
5: Identify skyline tuples on subspace using PQ-2DSUB-SKY
6: **end while**

sizes for the two dimensions selected into the 2D subspace have an *additive* effect on query cost, while the others have a *multiplicative* effect. Thus, generally, we should choose the two attributes with the largest domain sizes as the 2D subspace.

Based on the heuristics, the pseudo code of PQ-DB-SKY is depicted in Algorithm 7. Given the exponential nature of dividing a higher-D space into 2D subspaces, the worst-case query cost grows exponentially with the number of attributes. Nonetheless, as argued in the 2D case, the small domain sizes and the value-occupancy property usually lead to a much smaller query cost in practice. Such an effect is likely amplified even further in higher-D cases, as we shall show in the experimental results in Section 4.7, because of the aforementioned heuristics which places the largest domain-sized attributes in the 2D subspace, leaving the other (multiplicative) attributes with even smaller domains.

Suppose V_{m_1} and V_{m_2} are the attributes with the largest domain size, and $V\prime = V\backslash\{V_{m_1}, V_{m_2}\}$. PQ-DB-SKY processes the 2D plane for each value combination in $V\prime$. Assume for each combination of values v_c for $V\prime$ there exists a sorted list, L_{v_c}, of its skyline tuples with regard to their values on V_{m_1}, extended by the top-left and bottom-right corner points. Using Equation 15, the following is an upper-bound for PQ-DB-SKY query cost, which is in the order of $O((|V_{m_1}| + |V_{m_2}|) \times \prod_{\forall v\prime \in V\prime} |v\prime|)$:

$$C = \sum_{\forall v_c \, for \, V\prime} \cdots \sum \sum_{i=0}^{|L_{v_c}|} \min(L_{v_c}[i].V_{m_2} - L_{v_c}[i+1].V_{m_2}\,, \; L_{v_c}[i+1].V_{m_1} - L_{v_c}[i].V_{m_1}) \tag{15}$$

Nonetheless, it is also important to note that when the number of attributes

is relatively small and the attribute selection is straightforward (e.g., when two attributes have significantly larger domains than the others), Algorithm PQ-DB-SKY can approach the provable lower bound of query cost for skyline discovery. To illustrate this, in the following special-case example, we show that Algorithm PQ-DB-SKY achieves a query cost with constant difference from the proved (instance-optimal) lower bound.

Case Study for PQ-DB-SKY: Let there be a 3D database with attributes x, y, and z. The database ranking function follows a simple rule: If there is a tuple with $z < z_0$ that satisfies a query, the query will never return any tuple with $z \geq z_0$ (i.e., z is the first-priority ordering attribute). Furthermore, for any possible value of x, say v_x, there is at least one tuple in the database with $x = v_x$ and $z = 0$. Similarly, for any possible value v_y of y, there is at least one tuple in the database with $y = v_y$ and $z = 0$.

An interesting property for this construction is that it excludes most higher-dimensional (i.e., 2D or 3D) queries from consideration in building the optimal query plan. The reason for doing so is as follows. First, note that the only 3D query possible will return the same result as SELECT * FROM D WHERE $z = 0$. Second, every 2D query of the form $x = v_x$ or $y = v_y$ is guaranteed to return a tuple with $z = 0$ - i.e., they become equivalent with queries ($x = v_x$ AND $z = 0$) and ($y = v_y$ AND $z = 0$), respectively.

According to these two observations, one can see that there is always an optimal query plan which only includes a subset of the following queries: (a) 2D queries of the form $z = v_z$; (b) 1D queries because any other query is equivalent with a query of these two types. We now consider the queries issued by the above-described, optimal, 2D skyline crawling algorithm on the plane with $z = 0$. An important observation here is that any query with (conjunctive) predicate $z = v_z$ ($v_z \neq 0$) cannot reveal any information about tuples (or even the data space) with $z = 0$. As such, we consider next the following question: can the queries in optimal 3D skyline crawling query plan with predicate $z = 0$ significantly differ from the optimal 2D plan?

To answer this question, we need to consider the alternative queries that can be

included in the 3D optimal plan - i.e., those queries that contribute to the skyline crawling of the plane $z = 0$ yet are not part of the 2D optimal plan. One can see from the above discussions that these queries must be 1D queries of the form $x = v_x$ AND $y = v_y$ - which reveals whether a tuple occupies the point $(v_x, v_y, 0)$ on the plane $z = 0$. We refer to such 1D queries as xy queries.

Now consider how many 1D queries one must issue to "replace" a query in the optimal 2D plan. An important observation here is the on number of unique points "covered" by a query q in the optimal 2D plan, which we refer to as the *unique coverage count* of q. By "unique points" we mean points covered by exactly one query q in the optimal plan. In other words, one cannot determine if a skyline tuple resides on the point if q is removed from the query plan. The interesting observation about the unique coverage count is that, for any $h \geq 0$, there must be *at most* h queries with a unique coverage count of h or less. This easily follows from the 2D optimality proof discussed in Section 4.4.1.

Given this observation, one can derive the optimization ratio of simply running the 2D optimal algorithm over $z = 0, 1, \ldots, |V_z| - 1$, respectively. Suppose that the query cost of doing so on $z = i$ is c_i, leading to an overall query cost of $C_{2D} = \sum_{i=0}^{|V_z|-1} c_i$. One can see that any 3D skyline crawling algorithm must have a query cost of at least

$$C \geq \min_{h \geq 0} \left(\sum_{i=0}^{|V_z|-1} (c_i - h) \right) + \frac{h(h+1)}{2} \tag{16}$$

$$= \min_{h \geq 0} \left(C_{2D} - |V_z|h + \frac{h(h+1)}{2} \right) \geq C_{2D} - \frac{|V_z|^2}{2} \tag{17}$$

4.5 Skyline Discovery for Mixed-DB

We now combine our ideas for SQ, RQ and PQ to produce MQ-DB-SKY, our final algorithm for a mixture of all attributes.

4.5.1 Overview

When the hidden database features a mixture of range- and point-predicates, a straightforward idea appears to be applying RQ-DB-SKY directly over the range-predicate attributes and not using the point ones at all (by setting them to *), because RQ-DB-SKY is significantly more efficient than PQ-DB-SKY. The problem, however, is that doing so misses skyline tuples, as shown below.

First, note that by setting $A_i = *$ on all point-predicate attributes, the skyline tuples discovered by applying RQ-DB-SKY must indeed be skyline tuples. The problem here, however, is that the completeness proof no longer holds because a skyline tuple might be dominated by another tuple on all range-predicate attributes. Such a tuple will be missed by RQ-DB-SKY. Fortunately, the missing tuples must share a common property which we refer to as the *range-domination property*: every tuple t missed here must be dominated by an already-discovered skyline tuple, say $D(t)$, on all range attributes. Meanwhile, t must surpass $D(t)$ on at least one of the point attributes.

Range-domination is an interesting property because it significantly shrinks the search space for finding the remaining skyline tuples. Consider a simple example where the execution of RQ-DB-SKY returns only one tuple t_0. In this case, we can define our new search space (for all missing skyline tuples) by simply constructing a conjunctive query with predicates $A_i \geq t_0[A_i]$ for every range-predicate attribute A_i. Depending on the value of t_0 and the data distribution, these conjunctive predicates may significantly reduce the space we must search through with PQ-DB-SKY.

When the range attributes only support one-ended ranges, the above search-space-pruning idea does not work because predicates like $A_i \geq t_0[A_i]$ are not supported. Nonetheless, it is still possible to prune the search space because, in order for a missing tuple to be on the skyline, it must dominate an already discovered tuple on at least one point-predicate attribute. In other words, in the execution of PQ-DB-SKY, we no longer need to consider value combinations of point-predicate attributes that are dominated by all discovered tuples. While this idea has a much weaker pruning power than the above one, it works for the case of

two-ended ranges as well, and can be readily integrated with the above idea.

In the following discussions, we shall first describe our key idea for leveraging the pruning power afforded by two-ended ranges. Then, we develop our most generic Algorithm MQ-DB-SKY which supports a mixture of two-ended range, one-ended range, and point-predicate attributes.

4.5.2 Details for Leveraging Two-Ended Ranges

Before presenting our final MQ-DB-SKY algorithm, an important issue remains on how exactly to leverage the above-described RQ-based search-space pruning. A straightforward method is to construct for each discovered skyline tuple t_i the above-described subspace defined by conjunctive predicates $A_i \geq t_i[A_i]$, and then run PQ-DB-SKY over the space. The problem, however, is that PQ-DB-SKY cannot be directly used in this case because its 2D-subspace-discovery subroutine relies on an important property: if a tuple matches but is not returned by a 1D query q_0 as the No. 1 tuple, then it cannot be on the skyline. Unfortunately, this property no longer holds in the mixed case.

To address this problem, we devise a new subroutine MIXED-DB-SKY as follows. For each skyline tuple t_0 discovered by the range-query algorithm, let predicate $P(t_0)$ be $(t[A_1] \geq t_0[A_1])$ & $\cdots$ & $(t[A_h] \geq t_0[A_h])$ for all range attributes $A_1, \ldots, A_h$. For each point attribute $B_i (i \in [1, g])$ and each value $v < t_0[B_i]$, we construct a query q: WHERE $P(t_0)$ & $(t[B_i] = v)$.

If this query returned empty, we move on to the next query. The premise (of the efficient execution of this algorithm) is that, in practice, most such queries q will return empty, quickly pruning the remaining search space. If q returns at least one tuple, we need to start crawling the subspace defined by q. Now recall our PQ-DB-SKY algorithm for point-query skyline discovery. Our first step over there is to "partition" the space into 2-dimensional subspaces (i.e., by enumerating all possible value combinations for the other $g - 2$ attributes, where g is the number of point attributes) and deal with them one after another. This step remains the same. Specifically, at any point we have an empty answer, we can stop further partitioning the current subspace. When we go all the way to a 2-dimensional

subspace (without being stopped by an empty answer) then we'll have to crawl the entire 2D plane to find all tuples in it, instead of using the "2D skyline discovery" approach in PQ-DB-SKY. This is the only difference with MIXED-DB-SKY.

A concern with this design is the large number of times MIXED-DB-SKY may have to be called to completely discover the skyline. Note that a *single* call of MIXED-DB-SKY without any appended predicates is sufficient to unveil all skyline tuples. Yet when we append the range predicates to prune the search space, the repeated executions of MIXED-DB-SKY, especially many skyline tuples are discovered by RQ-DB-SKY, may lead to an even higher query cost.

To address this problem, we consider a slightly different solution of maintaining a single execution of MIXED-DB-SKY. This time, instead of designing m_{TE} conjunctive predicates for each of the discovered skyline tuples, we do so only once for the *union* of (dominated) data spaces corresponding to all of them. Specifically, for each two-ended range attribute A_j, its corresponding (appended) predicate is now

$$A_j \geq \min(t_1[A_j], \ldots, t_h[A_j]), \tag{18}$$

where $t_1, \ldots, t_h$ are the initially-discovered skyline tuples. One can see that these predicates ensure comprehensiveness of skyline discovery, as any tuple that fails to satisfy (18) must not be dominated by any discovered tuple on the range-predicate attributes - in other words, this tuple must have already been discovered by RQ-DB-SKY. On the other hand, given the (relatively) small number of skyline tuples, $\min(t_1[A_j], \ldots, t_h[A_j])$ may still have substantial pruning power, yet reducing the number of executions of MIXED-DB-SKY to exactly 1.

4.5.3 Algorithm MQ-DB-SKY

We now combine all the above ideas to produce our ultimate (most generic) algorithm, MQ-DB-SKY, which supports any arbitrary combination of two-ended range, one-ended range, and point predicate attributes. Note that when there are two-ended range attributes in the database, we use the pruning idea discussed in

Algorithm 8 MQ-DB-SKY

```
1: S = apply RQ-DB-SKY() on Range predicates; P =""
2: for range attributeS r ∈ R do
3:    append P by "AND t[r] ≥ min_{∀t_j∈S}(t_j[r])"
4: end for
5: for point attributeS B_i and each value v < max_{∀t_j∈S}(t_j[B_i]) do
6:    q: WHERE P AND (t[B_i] = v)
7:    T = Top-k(q); update S by T
8:    if T contains k tuples then
9:       partition the space defined q in 2D planes
10:      for all planes ρ do
11:         crawl the tuples in ρ and update S
12:      end for
13:   end if
14: end for
```

the above subsection. When there are only one-ended range attributes besides point ones, our algorithm is limited to using the weaker pruning idea discussed in Section 4.2. If there are only one-ended range, two-ended range, or point-predicate attributes in the database, MQ-DB-SKY is reduced to SQ-, RQ-, and PQ-DB-SKY, respectively. Finally, if there are a mixture of one-ended and two-ended range-predicate attributes but no point-predicate attribute in the database, MQ-DB-SKY is reduced to a simple revision of RQ-DB-SKY which leverages the availability of "$>$" predicates on only attributes that support two-ended ranges.

4.6 Extensions

4.6.1 Anytime Property

An desirable feature shared by all algorithms developed in the paper is their *anytime property* - i.e., one can stop the algorithm execution at any time to return a subset of all skyline tuples. One can see that this property can be very useful for discovering the skyline over real-world web databases, as many of them enforce a (many times secret and dynamic) limit on the number of queries that can be issued from an IP address (or an API account) per day. Without knowing such a limit ahead of

time, it becomes extremely important to ensure that the algorithm returns as many skyline tuples as possible (instead of simply returning a failure message) when the query limit is triggered.

In SQ-DB-SKY, note that any tuple returned by an issued query is a skyline tuple. Thus, the property always holds. In RQ-DB-SKY, note if we traverse the tree in a depth-first fashion, then a tuple returned is either on the skyline or dominated by one of the already discovered skyline tuple. Thus, the anytime property holds here as well. In PQ-2D-SKY, just like SQ-DB-SKY, any tuple returned by an issued query is a skyline tuple - leading to the anytime property. Since PQ-DB-SKY explores one 2D subspace at a time, so long as we process values of the other attributes (i.e., those not selected into the 2D subspace) in their preferential order, all tuples discovered by the algorithm at any time are on the eventual skyline - i.e., the anytime property holds. Finally, in the mixed case, the initial call of RQ- or SQ-DB-SKY satisfies the anytime property, as shown above. The subsequent call of (a small variation of) PQ-DB-SKY satisfies the property as well, leading to the anytime property of MQ-DB-SKY

4.6.2 Sky Band

We now consider an extension of the objective from discovering the *skyline* tuples to *top-h sky band* tuples - i.e., those tuples that are dominated by fewer than h other tuples in the database. One can see that the top-1 sky band is exactly the traditional skyline. Quite surprisingly, the simplest case discussed above - i.e., SQ-DB - becomes the most difficult case for sky band discovery. In the following discussions, we shall first illustrate how to extend Algorithms RQ- and PQ-DB-SKY (and thereby MQ-DB-SKY) to discover the top-h sky band, and then discuss SQ-DB.

Extending RQ-DB-SKY: The extension is enabled by the following simple yet important observation: for any tuple t_2 on the top-2 sky band but not on the skyline, there must exist a skyline tuple t_1 such that when we consider the subspace dominated by t_1 (and the subset of the database in it), henceforth referred to as t_1's *domination subspace*, t_2 becomes a skyline tuple. Given this observation,

discovering the top-2 sky band becomes straightforward: for each skyline tuple t discovered by RQ-DB-SKY, we run RQ-DB-SKY again, just this time on the domination subspace of t. It is possible to specify such a subspace through conjunctive queries because RQ-DB supports two-ended ranges.

We now consider the discovery of top-h sky band. While this seemingly requires us to consider any size-$(h-1)$ subset of tuples on the top-$(h-1)$ sky band, fortunately this is not the case in reality. To understand why, consider a tuple t_3 which is on the top-3 sky band but not top-2 sky band. Interesting, t_3 must be a skyline tuple on the domination subspace of either a skyline tuple of the entire database or a tuple on its top-2 sky band. For example, suppose that t_3 is dominated by two skyline tuples t and t'. Note that this means t_3 must be a *skyline* tuple in the domination subspace of t (and that of t' as well), simply because t' is excluded from this subspace. As such, the total number of times we have to run RQ-DB-SKY to discover the top-h sky band is simply the number of tuples on the top-$(h-1)$ sky band plus one (i.e., the original execution for discovering the skyline).

Extending PQ-DB-SKY: For PQ-DB-SKY, the extension is indeed straightforward - since the algorithm is eventually reduced to each 2D subspace (and the 1D queries issued within), the only difference here for sky band discovery is the pruning rule: after issuing a 1D query q which returns t, instead of eliminating the subspace with $x > t[x]$ and $y > t[y]$ from consideration as in the skyline case, we have to find the top-h sky band tuples matching q instead. This is simple when the system returns top-k tuples where $k \geq h$ - as we can simply take the top-h returned tuples of q and determine based on the previously retrieved tuples which of the h tuples are indeed on the top-h sky band, and then perform the pruning accordingly. If $k < h$, however, we may have to issue the 0D (base) queries one by one until finding all possible tuples matching q that are on the top-h sky band. Once the pruning process is updated, the remaining design remains unchanged.

Extending SQ-DB-SKY: The most difficult case, unfortunately, happens for SQ-DB which only supports one-ended ranges. Indeed, it might not be possible to discover even the top-2 sky band without crawling the entire database. To

understand why, consider a simple case where the system features a top-1 interface (i.e., $k = 1$). Here we note a simple fact: any query consisting solely of $<$ or $\leq$ predicates will *never* return a non-skyline tuple t - because this query will always match the skyline tuple dominating t and return it over t according to the system ranking function. This essentially requires us to resort to "=" predicates (this is even assuming we know all the domain values) in order to discover the top-h ($h > 1$) sky band. One can see that this easily reduces to crawling the database in the worst-case scenario.

Having stated the negative result, in practice, it is still possible to efficiently discover the top-h sky band for SQ-DB, especially when k (as in the top-k interface offered by the hidden database) is large. To understand why, note the following critical observation: if among the (up to k) results returned by a query q, say SELECT *, we can find a tuple t dominated by $h - 1$ other tuples, then we can safely conclude that any tuple on the top-h sky band must not be dominated by t. In other words, we can branch out from the query according to t like what we did in SQ-DB-SKY for the top-1 tuple returned by q. We know that for any top-h sky band tuple matching q, it must belong to at least one of the m branches, as otherwise it must be dominated by t and therefore out of the top-h sky band. Of course, as we drill further down into the tree, there is a decreasing chance for a query to return a tuple dominated by $h - 1$ others, simply because the appended $<$ predicates narrow the field to "highly ranked" tuples. Nonetheless, with a large k, many of these deep queries may already return valid answers, allowing us to safely stop exploring it further. In the unfortunate case where a query still overflows - and we do not have any way of further branching it out without losing the comprehensiveness of top-h sky band discovery - then we have two choices: either to stop exploring this query and accept partial discovery; or to crawl the entire subspace corresponding to this query.

4.7 Experimental Evaluation

4.7.1 Experimental Setup

In this section, we present the results of our experiments, all of which were run on real-world data. Specifically, we started by testing a real-world dataset we have already collected. We constructed a top-k web search interface for it and then ran our algorithms through the interface. Since we have full knowledge of the dataset and control over factors such as database size, etc., this dataset enables us to verify the correctness of our algorithms and test their performance over varying characteristics of the database. Then, we tested our algorithms *live online* over three real-world websites, including the largest online diamond and flight search services in the world, echoing the motivating examples discussed in Section 4.

Offline Dataset: The offline dataset we used is the flight on-time database published by the US Department of Transportation (DOT). It records, for all flights conducted by the 14 US carriers in January 2015,[3], attributes such as scheduled and actual departure time, taxiing time and other detailed delay metrics. The dataset has been widely used by third-party websites to identify the on-time performance of flights, routes, airports, airlines, etc.

The dataset consists of 457,013 tuples over 28 attributes, from which 9 ordinal attributes were used as *ranking attributes*[4]: *Dep-Delay, Taxi-out, Taxi-in, Actual-elapsed-time, Air-time, Distance, Delay-group-normal, Distance-group, ArrivalDelay*. The domain of the 9 ranking attributes range from 11 to 4,983. Two of the 9 attributes, *Delay-group-normal* and *Distance-group*, were already discretized by DOT (i.e., "grouped", according to the dataset description). Thus, we used them as PQ (point-query-predicate) attributes by default. For a few tests which call for more PQ attributes, we also consider four other derived attributes, *Taxi-out group, Taxi-in group, ArrivalDelay group, Air-Time group* as potential PQ. The other attributes were used as range-predicate attributes - whether it is SQ or

[3]from `http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time`

[4]The others, such as Flight Number, are considered filtering attributes and not used in the experiments.

RQ depends on the specific test setup.

For all attributes, we defined the preferential order so that shorter delay/duration ranks higher than longer values. For non-time attributes, i.e., *Distance* and *Distance-group*, we assigned a higher rank to longer distances than shorter ones, given that the same amount of delay likely impacts short-distance flights more than longer ones. We also tested the case where shorter distances are ranked higher, and found little difference in the performance. To construct the top-k interface, we also need to define a ranking function it uses. Here we simply used the SUM of attributes for which smaller values are preferred MINUS the SUM of attributes for which larger values are preferred.

Online Experiments: We conducted live experiments over three real-world websites: Blue Nile (BN) diamonds, Google Flights (GF), and Yahoo! Autos (YA).

Blue Nile (BN)[5] is the largest online retailer of diamonds. At the time of our tests, its database contained 209,666 tuples (diamonds) over 6 attributes: *Shape*, *Price*, *Carat*, *Cut*, *Color*, *Clarity*, the last 5 of which have universally accepted preferential (global) orders, i.e., lower *Price*, higher *Carat*, more precise *Cut*, low trace of *Color* and high *Clarity*. We used these 5 attributes to define skyline tuples. BN offers two-ended range predicates (RQ) on all five attributes, with the default ranking function being *Price* (low to high).

Google Flights (GF) is one of the largest flight search services and offers an interface called QPX API[6]. We consider the scenario of a traveler looking to get away with a one-way flight after a full day of work. We used three filtering attributes, *DepartureCity*, *ArrivalCity* and *DepartureDate*, and four supported ranking attributes: *Stops*, *Price*, *ConnectionDuration*, and *DepartureTime*. Here the traveler likely prefers fewer *Stops*, lower *Price*, shorter *ConnectionDuration*, and later *DepartureTime*. QPX API supports SQ (i.e., single-ended ranges) on *Stops*, *Price*, *ConnectionDuration*, and RQ (i.e., two-ended) on *DepartureTime*. The default ranking function used by GF is price (low to high).

Yahoo! Autos (YA)[7] offers a popular search service for used cars. In our

[5] `http://www.bluenile.com/diamond-search`
[6] `https://developers.google.com/qpx-express/`
[7] `https://autos.yahoo.com/used-cars/`

experiments, we considered those listed for sale within 30 miles of New York City, totaling 125,149 cars. We considered three ranking attributes *Price* (lower preferred), *Mileage* (lower preferred), *Year* (higher preferred), all of which are supported as two-ended ranges (RQ) by YA, and the ranking function of *Price* (low to high).

Algorithms Evaluated: We tested the four main algorithms described in the paper, SQ-, RQ-, PQ-, and MQ-DB-SKY. We also compared their performance with a baseline technique of first crawling all tuples from the hidden web database using the state-of-the-art crawling algorithm in [36], and then extracting the skyline tuples locally. We refer to this technique as BASELINE.

Performance Measures: As we proved theoretically in the paper, all algorithms guarantee complete skyline discovery. We confirmed this in all experiments we ran offline (and have the ground truth for verification). Since precision is not an issue, the key performance measure becomes efficiency which, as we discussed earlier, is the number of queries issued to the web database.

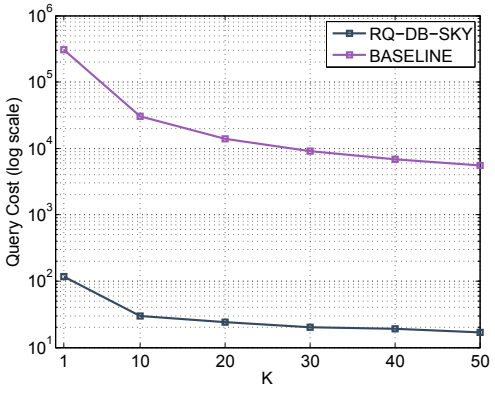


Figure 21: Range Predicates: Impact of k

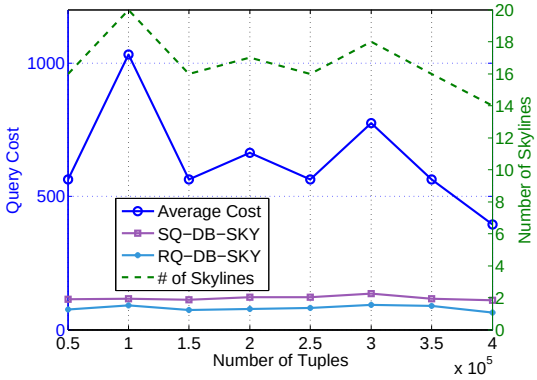


Figure 22: Range Predicates: Impact of n

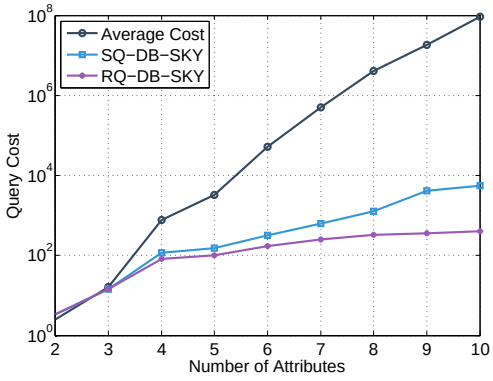


Figure 23: Range Predicates: Impact of m

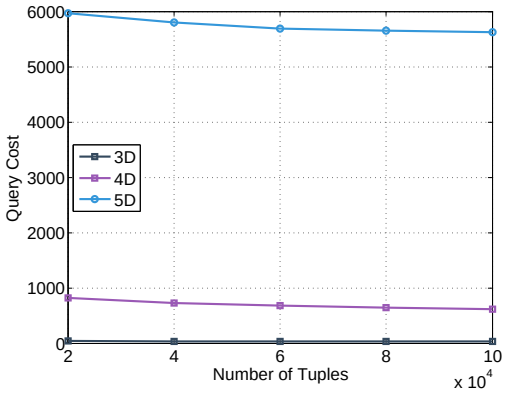


Figure 24: Point Predicates: Impact of n

4.7.2 Experiments over Real-World Dataset

Interfaces with Range Predicates: We started with testing skyline discovery through range-query interfaces, i.e., SQ and RQ, over the DOT dataset. Figure 21 compares the query cost required for complete skyline discovery by RQ-DB-SKY and BASELINE when k (as in top-k offered by the web database) varies from 1 to 50. Note that SQ-DB-SKY is not depicted here because the range-query-based crawling in BASELINE requires two-ended range support. One can observe from the figure that, while both algorithms benefit from a larger k as we predicted, our RQ- algorithm outperforms the baseline by orders of magnitude for all k values. Given the significant performance gap between BASELINE and our solutions, we skip the BASELINE figure for most of the offline results, before showing it again in the online live experiments.

Figure 22 depicts how the query cost of SQ- and RQ-DB-SKY change when the database size n ranges from 50K to 400K. To test databases with varying sizes, we drew uniform random samples from the DOT dataset. The figure also shows the change of $|S|$, the number of skyline tuples. One can see from the figure that RQ-DB-SKY is more efficient than SQ- because it uses the more powerful, two-ended, search interface. Perhaps more interestingly, neither algorithm's query cost depend much on n. Instead, they appear more dependent on the number of

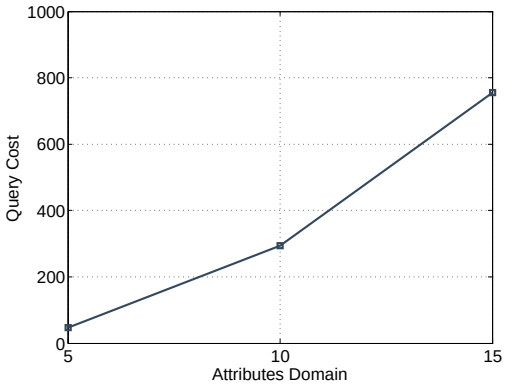


Figure 25: Point Predicates: Impact of Domain Size

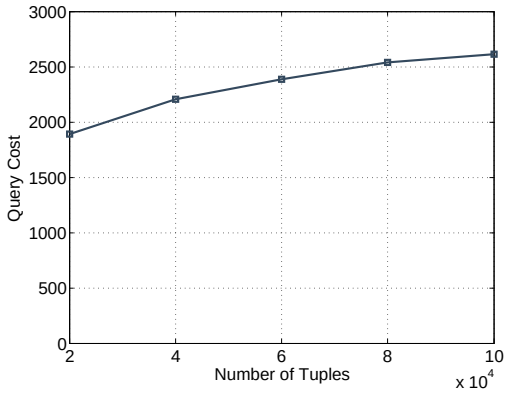


Figure 26: Mixed Predicates: Impact of n

skyline tuples $|S|$ - consistent with our theoretical analysis.

Figure 23 varies the number of attributes m. While both RQ- and SQ- require more queries when there are more attributes, RQ- again consistently outperforms SQ-DB-SKY. Note that the increase on query cost is partially because of the rapid increase of the number of skyline tuples with dimensionality [38]. In any case, the query cost for RQ- and SQ-DB-SKY remain small, compared to the theoretical bounds, even when the dimensionality reaches 10.

Interfaces with Point Predicates: In the next set of experiments, we tested PQ-DB-SKY. Figure 24 shows how its query cost varies with n and m. Interestingly, while the query cost barely changes with n varying from 20,000 to 100,000, it increases significantly when m changes from 3 to 5, just as predicted by our theoretical analysis. In Figure 25, we further tested how the query cost changes with varying domain sizes. To enable this test, for each given domain size (from $v = 5$ to 15), we first select all PQ attributes with domain larger than v, and then remove from the domain of each attribute all but v values (along with their associated tuples). Then, we randomly selected 100,000 tuples from the remaining tuples as our testing database. One can see from the result that, while larger attribute domains do lead to a higher query cost, the increase on query cost is not nearly as fast as the data space (which grows with v^m) - indicating the scalability

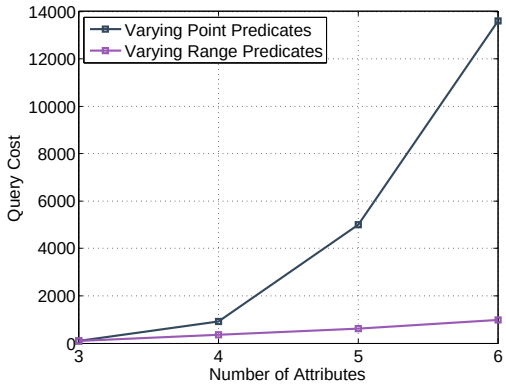


Figure 27: Mixed Predicates: Varying Range and Point Predicates

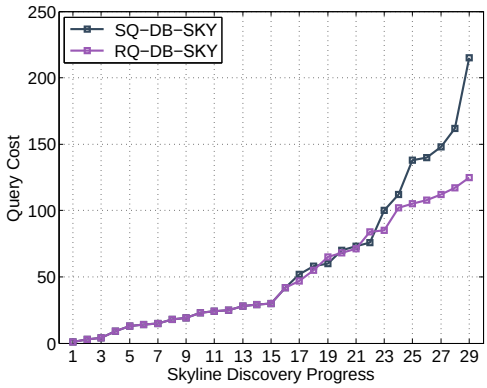


Figure 28: Anytime Property of SQ and RQ-DB-SKY

of PQ-DB-SKY to larger domains.

Interfaces with Mixed Predicates: We next tested a more realistic search interface that contains a mixture of range and point predicates. We started with 3 RQ and 2 PQ predicates and evaluated how the query cost varies with database size. Figure 26 shows that, as expected, the number of tuples only have minimal impact on query cost. We then tested how varying number of RQ and PQ attributes affect our performance. The two lines in Figure 27 represent, respectively, (1) 1 PQ attribute with the number of RQ attributes varying from 2 to 5, and (2) 1 RQ attribute with the number of PQ ones from 2 to 5. One can observe from the figure that the impact on query cost is much more pronounced on an increase of the number of PQ attributes - consistent with earlier discussions in the paper.

Anytime Property of Skyline Discovery: Recall from §1 that all algorithms in the paper feature the anytime property, i.e., one can stop the algorithm execution at any time to return a subset of skyline tuples (over the entire database). Note that BASELINE does not have this feature, as there is no way for it to determine if a tuple is truly on the skyline before the entire database is crawled. Figures 28 and 29 trace the progress of SQ-, RQ- and PQ-DB-SKY over 100,000 tuples (5 predicates in RQ-DB and 4 in PQ-DB case) and demonstrate how the number of discovered skyline tuples changes with query cost.

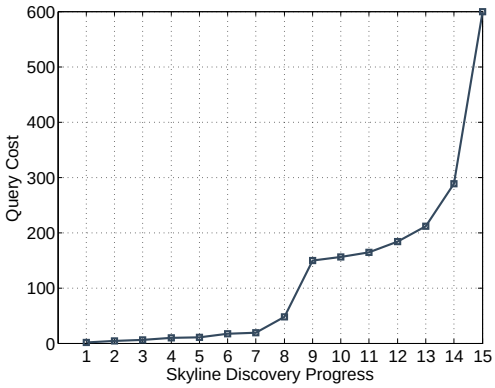


Figure 29: Anytime Property of PQ-DB-SKY

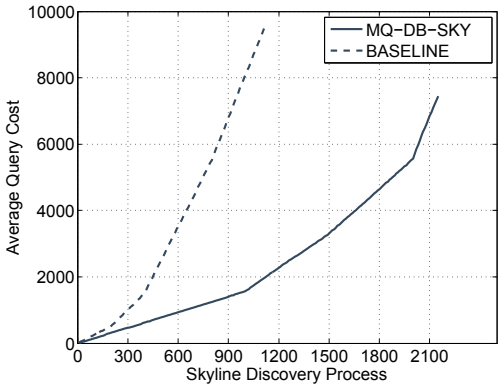


Figure 30: Online Experiments: Blue Nile Diamonds

There are some interesting observations from the two figures. In Figure 28, note that SQ-DB-SKY could find the first 16 skylines without facing a skyline twice, leading to identical performance with RQ- up to that point. Afterwards, however, it started getting the same skyline tuple multiple times, leading to poorer performance than RQ-DB-SKY when the number of discovered skyline tuples reaches 23. In Figure 29, note that despite the limitations of PQ, our algorithm managed to discover all skyline tuples with fewer than 600 queries. The peak between the 8^{th} and 9^{th} tuples is caused by queries "wasted" for crawling an area that did not contain any skyline tuple.

4.7.3 Online Demonstration

As discussed earlier, we conducted live online experiments by applying our final algorithm, MQ-DB-SKY, over three real-world web databases, Blue Nile diamond search (BN), Google Flights (GF), and Yahoo! Autos (YA), respectively.

Skyline Discovery over Blue Nile (BN): For BN, we discovered a total of 2,149 tuples on the skyline. We compared the performance of MQ-DB-SKY with BASELINE ($k = 50$), with the results depicted in Figure 30. Note that we stopped the execution of BASELINE when its query cost reached 10,000 queries,

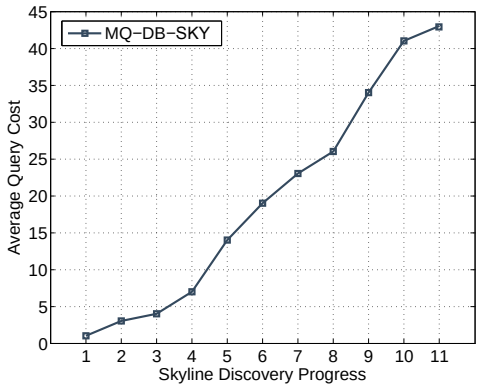


Figure 31: Online Experiments: Google Flights

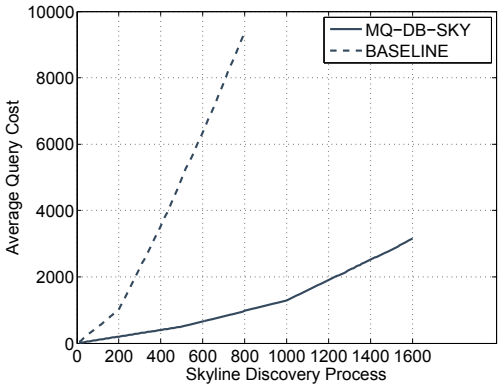


Figure 32: Online Experiments: Yahoo! Autos

at which time it only managed to discover 1113 skyline tuples[8]. On the other hand, our MQ- algorithm discovers the entire skyline with an average query cost of only 3.5 per skyline tuple.

Skyline Discovery over Google Flights (GF): Our experiment setup was as follows. We randomly chose a pair of airports from the top-25 busiest airports in USA and a date between November 1 and 30, 2015, and sought to find all skyline flights on that day. We repeated this process for 50 different pairs and report the average query cost. The number of skyline flights varied between 4 to 11. Figure 31 shows the results. Note that we did not compare against BASELINE here because GF offers SQ only for attributes such as *Stops*, *Price*, and *ConnectionDuration*, while BASELINE requires two-ended range support for crawling. We verified the correctness of the results by crawling all the flights for the same date and comparing the results. One can observe that our algorithm is highly efficient even when $k = 1$. Specifically, it was able to discover all skyline tuples with query cost below 50, which is the (free) rate limit imposed per user account per day by GF (QPX API).

Skyline Discovery over Yahoo! Autos (YA): For YA, we discovered a total of

[8]Note that, as discussed earlier, BASELINE would not be able to output these skyline tuples despite of having discovered them because BASELINE lacks the anytime property.

1,601 skyline tuples. Figure 32 shows the performance of our MQ- algorithm and the comparison with BASELINE. Here $k = 50$. Once again, we had to discontinue BASELINE at 10,000 queries before it were able to complete crawling. On the other hand, our MQ-DB-SKY algorithm managed to discover the entire skyline with an average query cost below 2 per skyline tuple.

4.8 Related Work

Crawling and Data Analytics over Hidden Databases: While there has been a number of prior works on crawling, sampling, and aggregate estimation over hidden web databases, there has not been any study on the discovery of skyline tuples over hidden databases. Crawling structured hidden web databases have been studied in [36, 39, 40]. [41–43] describe efficient techniques to obtain random samples from hidden web databases that can then be utilized to perform aggregate estimation. Recent works such as [44, 45] propose more sophisticated sampling techniques that reduce variance of aggregate estimation.

Skyline Computation: Skyline operator was first described in [1] and number of subsequent work have studied it from diverse contexts. [31] and [32] proposed efficient algorithms with the help of indices and pre-sorting respectively. Online and progressive algorithms were described in [33, 46]. The problem of skyline over streams [47], partial orders [48], uncertain data [49] and groups [50] have also been studied. [34, 35] study the problem of retrieving the skyline from multiple web databases that expose a ranked list of all tuples according to a pre-known ranking function. Such special access might not always be available for a third party operator. Our work is the first to study the problem of skyline computation over structured hidden databases by using only the publicly available access channels.

Applications of Skyline Tuples: Skyline tuples have a number of applications in diverse contexts. A skyline tuple is not dominated by another tuple while a K-Skyband tuple is dominated by at most $K - 1$ tuples in the database. The top-k tuples of any monotone aggregate function must belong to K-Skyband where $k \leq K$ [51]. The numerous applications of top-k queries can be found in [52]. Other applications of Skyline include nearest neighbor search, answering

the preference queries and finding the convex-hull. Recently, the notion of reverse skyline [53], K-Dominating and K-Dominant [54], and top-K representative skylines [55] have been investigated with a number of applications including query re-ranking and product design.

4.9 Final Remarks

In this paper, we studied an important yet novel problem of skyline discovery over web databases with a top-k interface. We introduced a taxonomy of the search interfaces offered by such a database, according to whether single-ended range, two-ended range, or point predicates are supported. We developed efficient skyline discovery algorithms for each type and combine them to produce a solution that works over a combination of such interfaces. We developed rigorous theoretical analysis for the query cost, and also conducted a comprehensive set of experiments on real-world datasets, including a live online experiment on Google Flights, which demonstrate the effectiveness of our proposed techniques.

5 Regret-ratio Minimizing Set: A Compact Maxima Representative

5.1 Motivation

A maxima query returns a tuple from a large database of n tuples, preferentially selected and returned according to a ranking/utility function that is used to model user preferences. Such queries are very useful in application domains where end-users are interested in multi-criteria decision making, and would like to see the most important tuples from the potentially huge answer space. Thus, much recent studies, including online, view-based and index-based techniques such as [56–60] have focused on this direction.

In many applications, especially in databases containing numeric attributes, the ranking function used to model user preferences is expressed in the form of a linear combination of query attributes – i.e. $\sum w_i A_i$. Finding the top houses in a real estate database based on a linear combination of some criteria such as price and floor area [61], or finding the best NBA player based on a linear combination of his performance criteria such as points and assists, are a few examples of this class of ranking function.

A critical observation is that if the tuples are viewed as points in a high-dimensional space, the convex hull is the subset of points that can be used to find the maxima for any linear ranking function [60]. However, in some real world applications such a convex hull can be overwhelmingly large, and therefore its performance is greatly reduced because many tuples have to be examined during query processing [58]. The size of such a set is highly correlated with the number of attributes, i.e., the number of convex hull tuples radically increases with the number of attributes/dimensions. Even in the case of two-dimensional database, the number of convex hull tuples might also be large. As has been studied in [62], the "curvature" of the shape of a region within which the database tuples are distributed greatly affects the number of convex hull tuples. As the curvature increases the number of convex hull tuples increases. For example, when n points

are uniformly distributed inside a convex polygon with k sides, the expected number of the convex hull points is $O(k \log n)$, while this value is $O(n^{\frac{1}{3}})$ when points are uniformly distributed inside a circle.

Consequently, it is of interest to develop a set limited to $r << n$ tuples (where r is an input parameter). Given such a reduced set, for a given ranking function, we can identify the maximum of the reduced set and return it as the query answer. One can observe a tradeoff between the size of the set and the accuracy of query answers (i.e., how the result might differ from the real maximum over the entire table). The task then is to design the most accurate set, i.e., the subset of r tuples for which the "user dissatisfaction" over all possible ranking functions is minimized.

Prior works on convex hull discovery in high-dimensions, such as [63, 64], focus on designing efficient approximate algorithms with small approximate ratios. Thus, their goal is to discover a set that is as similar to the real convex hull as possible, rather than resolving the problem of a large convex hull, and usually find a *super-set* of the convex hull. There has also been work, such as [65,66], on reducing the *skyline* [67] (the maxima representative that applies to more general *monotonic* ranking functions rather than just linear functions) size. However, their objective in ranking the skyline tuples is not minimizing the user dissatisfaction on maxima queries. For example, [65] relaxes the notion of domination to "$k-$domination" in order to increase the chance domination and reduce the skyline.

The problem investigated in this section, which we call the *Regret-ratio Minimizing Set (RRMS)* problem, has been studied in prior papers. Nanongkai et. al. [68] introduced the notion of *regret ratio* in order to measure the user dissatisfaction with the top result returned by the representative set. Given a set of r tuples and a specific ranking/utility function, they define regret ratio as the ratio of the difference between the scores of the top tuple in the set and the top tuple in the entire database, divided by score of the top tuple in the entire database. Given a set (or space) of ranking functions, the maximum regret ratio is the regret ratio with the largest value. The RRMS problem seeks to find the subset of r tuples that minimizes the maximum regret ratio. It is known that for arbitrary dimensions, the problem is NP-hard [69]. The two state of art algorithms for arbitrary dimensions are (a) a

greedy heuristic with unproven theoretical guarantees, which is based on executing $O(nr)$ linear programs in total, and (b) a simple space discretization approach that produces an approximate regret ratio that is within a fixed distance from the optimal and has the time complexity of $O(nd + r)$ [68]. Further investigation on this problem has also been done by [69] for the special case of two dimensions (where the problem is not NP-hard), and a quadratic ($O(n^2r)$) algorithm to find the optimal set has been developed, that leverages the notions of geometric duality and line arrangements.

5.2 Technical Highlights

In this section, we make several fundamental theoretical as well as practical advances for the RRMS problem, in both two-dimensional and high-dimensional databases.

In the case of two-dimensional databases, we develop an innovative dynamic programming algorithm to find the optimal set by leveraging the total order property of the tuples that occur in the skyline of the database. Our two-dimensional exact polynomial time algorithm (**2D-RRMS**) runs in $O(rs \log s \log c)$ time, where s and c are the number of skyline and convex hull tuples in the database respectively. This is a huge improvement over the $O(n^2r)$ algorithm proposed by [69], which is based on the notions of geometric duality and line arrangements.

Next, as perhaps one of the major results of this section, we develop an approximation algorithm (**HD-RRMS**) that guarantees a regret ratio that is within a small *user-controllable* distance from the optimal regret ratio. This algorithm is based on several innovative ideas. First, we model the problem conceptually as an infinitely large matrix *min-max* problem [70], where the rows are the tuples and the (infinitely many) columns correspond to each possible ranking function. Given an user controlled discretization parameter, we discretize the ranking functions space into a bounded number of functions (based on the control parameter) in the polar system. We then take the advantage of the linear-size discretized problem space in order to find the optimal value for the discretized matrix min-max problem in $O(n \log n)$ time, which is an approximate solution for the original problem.

To do so, we convert the problem into linear number of fixed-size instances of the *set-cover* problem [71]. Thus, our eventual algorithm is able to guarantee a regret ratio that is within a user-controllable distance from the optimal regret ratio. The HD-RRMS algorithm is a theoretical algorithm, because although it runs in linearithmic time (assuming that the dimensions of the space and the user controlled parameter are both constants), the proportionality constant in the running time is large (exponentially dependent on the number of dimensions), mainly due to the large size of the set cover instances. Therefore, we make an important practical adaptation to HD-RRMS, by replacing subroutine calls to an exact set-cover algorithm with calls to the well-known greedy approximate set-cover algorithm [72].

Beside the theoretical analysis, we also provide extensive experimental results over three publicly available real-world datasets, i.e., an Airline dataset, Department of Transportation (DOT), and Basketball dataset (NBA), with sizes up to several million records. We also used synthetic datasets to evaluate the performance of the proposed algorithms in the presence of different correlation models, i.e., correlated, independent, and anti-correlated. All experimental results confirm that our algorithms not only are more efficient than the existing solutions, and are scalable, but also produce representative sets with smaller regret ratios.

5.2.1 Summary of Contributions

In summary, we make the following main contributions:

- For the two-dimensional scenario we propose a *linearithmic* time dynamic programming algorithm **2D-RRMS** which is much faster than the existing quadratic time algorithm.
- We develop **HD-RRMS**, an algorithm for higher dimensions that can approximate the regret ratio to within a user controlled parameter. This algorithm discretizes the ranking function space and models the problem as a discrete matrix min-max problem. Although the HD-RRMS algorithm is linearithmic in theory, it may become inefficient in practice. We propose how to make the algorithm practical.

- We perform a comprehensive set of experiments on synthetic (with different correlation models) and real datasets of size up to several million records that demonstrate the efficiency, scalability, and effectiveness of our algorithms.

5.3 Preliminaries

Database Model: Consider a database, D with m numeric attributes $\mathcal{A} = \{A_1, \ldots, A_m\}$. Let $Dom(A_i) \geq 0$ be the domain of attribute A_i. The database may also have non-numeric attributes, but since they are usually not part of any ranking function, they are not considered in our context. We assume there are n distinct tuples in the database. For a tuple $t \in D$, we use $t[A_i] \in Dom(A_i)$ to denote the non-NULL value of attribute A_i in t.

Consider a ranking function $F(\cdot) : D \rightarrow \mathbb{R}$ that assigns a score to each tuple t in the database. The ranking function sorts the tuples based on the scores assigned to them. Given such a user specified function, the database determines the tuple with maximum score that should be returned. In this chapter, we assume a tuple $t \in D$ outranks a tuple $t' \in D$ based on F, if $F(t) > F(t')$. Furthermore, we follow the existing work [68, 69] and focus on the popular-in-practice [60, 61] linear ranking functions, defined by Equation 19.

$$F(t) = \sum_{i=1}^{m} w_i \cdot t[A_i] \tag{19}$$

$w_i \in [0, 1]$, in Equation 19, is the weight of the attribute A_i. Please note that since we are only restricted to positive weights, everything is confined to the first quadrant.
We also define the *contour* of F as the sets of m-attribute-value combinations that have the same score based on F.

Convex hull: The convex hull is the boundary of the smallest convex region containing all the tuples in D and is formed by the subset of tuples on the boundary [60]. In this chapter, we denote the convex hull by $\mathcal{C}$. As proved

in [73], $\mathcal{C}$ is the minimal subset that can be used to find the maxima for any linear ranking function[9]. Formally (considering $\mathcal{F}$ as the set of all possible linear ranking functions):

$$\begin{aligned}(i)\,&\forall F \in \mathcal{F}, \exists t \in \mathcal{C} \text{ s.t. } \forall t' \in D, F(t) \geq F(t') \text{ and} \\ (ii)\,&\forall t \in \mathcal{C}, \exists F \in \mathcal{F} \text{ s.t. } \forall t' \in \mathcal{C} \backslash \{t\}, F(t) > F(t')\end{aligned} \tag{20}$$

Depending on the number of tuples in D, and their distributions, size of the convex hull, $|\mathcal{C}| = c$, may be large. As discussed in § 2, even in a two-dimensional database a large number of tuples, $O(n^{\frac{1}{3}})$, may place in the convex hull [62]. If the convex hull is small, it can be pre-computed, and can then be extremely useful in efficiently finding the maximum scoring tuple of any user-specified ranking function, since a linear scan that examines c tuples will suffice rather than having to examine all n database tuples. The problem with a large $\mathcal{C}$ is that the scan becomes slow. In high dimensional databases, the problem is even worse because when m increases, more and more tuples belong to $\mathcal{C}$, which results in an exponential growth in the size of the convex hull [74]. Obviously, when $\mathcal{C}$ is a large portion of D, it loses its power as a representative. Figure 33 shows how c grows with number of attributes, m, when tuples are uniformly distributed.

Performance Measure: Based on Equation 20, every tuple t in $\mathcal{C}$ is the maximum of at least one linear ranking function. Let $\mathcal{F}_t$ be the set of linear ranking functions for which t is the maximum. If a tuple t is removed from $\mathcal{C}$, the next "best" alternative (the tuple that outranks all the tuples in $\mathcal{C} \backslash t$) is returned as the maximum of a ranking function $F \in \mathcal{F}_t$.

Consider the example provided in Figure 34, where $\mathcal{C} = \{t_1, t_2, t_3, t_4\}$ is the convex hull. The lines between l_1 clockwise to l_3 represent $\mathcal{F}_{t_3}$. If we remove t_3 then t_2 will be returned as the maximum for the ranking functions from l_1 clockwise to l_2, while t_4 will be returned for the ones from l_3 anti-clockwise to l_2. One may

[9]The maxima representative of more general monotonic ranking functions is *skyline* [67]. Skyline is the set of non-dominated tuples in the database, where a tuple t dominates a tuple t' ($t \succ t'$), *iff* (i)$\forall A \in \mathcal{A}$, $t[A] \geq t'[A]$ and (ii) $\exists A \in \mathcal{A}$ such that $t[A] > t'[A]$.

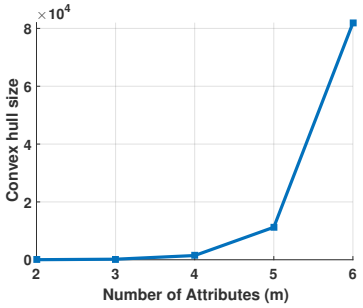


Figure 33: Impact of the number of attributes in $\mathcal{C}$

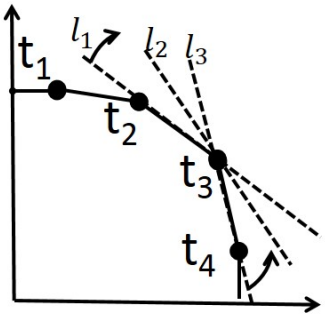


Figure 34: Contour of the ranking functions in a 2D example.

notice from the figure that as the function gets closer to l_2 (from either sides), the difference between $\mathcal{F}_{t_3}$ and the best alternative increases, i.e. the maximum score difference happens exactly for the ranking function represented by l_2.

In this chapter, we use the *maximum regret-ratio* defined by [68] in order to measure the error of removing a tuple t from a database D, formally defined by Equation 21[10] Intuitively, the maximum regret-ratio is the worst-case score (ratio) difference between the true maximum tuple and the one in the compact set.

$$\forall t \in D,\ E(t, D) = \sup_{\forall F \in \mathcal{F}_t} \frac{\min_{\forall t' \in D \setminus \{t\}}(F(t) - F(t'))}{F(t)} \tag{21}$$

The regret-ratio can be extended to measure the error of removing a set of tuples $T \subset D$. Specifically, this error is the maximum score-difference ratio of each tuple in T and its best alternative in the remaining convex hull, formally defined by Equation 22.

$$\forall T \subset D,\ E(T, D) = \max_{\forall t \in T}(\sup_{\forall F \in \mathcal{F}_t} \frac{\min_{\forall t' \in D \setminus \{T\}}(F(t) - F(t'))}{F(t)}) \tag{22}$$

In the rest of the chapter, for simplicity we use E_T as $E(T, D)$.

[10]We use the notation $E(T, D)$ to show $rr_D(D \setminus T, \mathcal{L})$ in [68].

5.3.1 Problem Definition

In this chapter, we consider the problem of Regret-ratio Minimizing Set, i.e., given a database D and an integer $r \geq 1$, our objective is to find a set of at most r tuples such that the regret-ratio is minimum. This problem is formally defined as follows:

> REGRET-RATIO MINIMIZING SET PROBLEM (RRMS): Given a database D and an integer $r \geq 1$, find a subset $\hat{\mathcal{C}} \subseteq D$ such that (1) $|\hat{\mathcal{C}}| \leq r$ and (2) $E(D \backslash \hat{\mathcal{C}}, D)$ is minimum.

In the following, we propose our efficient algorithms to solve the Regret-ratio Minimizing Set problem in both two-dimensional and high-dimensional databases.

5.4 2D Regret-ratio Minimizing Set

We start by considering the two-dimensional scenario which, as discussed in the introduction, not only has significant theoretical implications but also represents popular use cases in practice.

We show in Theorem 6 that the search space can be reduced to the skyline tuples rather than in all n tuples in the database.

Theorem 6. *Let T be the set of tuples which are removed from D. The maximum regret-ratio of the optimal solution for the regret-ratio minimizing set problem on $\mathcal{S}$ is the same as the maximum regret-ratio of the optimal solution for the regret-ratio minimizing set problem on D, i.e., $E(T, D) = E(T, \mathcal{S})$.*

Proof. Suppose, $\exists t \in \hat{\mathcal{C}}$ where $t \notin \mathcal{S}$.

$$\Rightarrow \exists t' \in D, \text{ s.t. } t' \succ t$$

Since $\forall A \in \mathcal{A}$, $t[A] \geq t'[A]$ and $\exists A \in \mathcal{A}$, $t[A] > t'[A]$, for any ranking function F:

$$F(t) = \sum_{\forall A_i \in \mathcal{A}} w_i \cdot t[A_i] < F(t') = \sum_{\forall A_i \in \mathcal{A}} w_i \cdot t'[A_i]$$

It means by replacing t with t' the size of the set does not change, and the maximum regret-ratio after the replacement is less than or equal to the maximum regret-ratio

before the replacement. Therefore, the maximum regret-ratio of the optimal solution for the skyline tuples, $\mathcal{S}$, is the same as the maximum regret-ratio of the optimal solution for all tuples in D. □

Considering Theorem 6, we first order all skyline tuples in a two-dimensional table from top left to the bottom right, i.e. $\mathcal{S} = \{t_1, t_2, \cdots, t_s\}$. We add two dummy tuples t_0 and t_{s+1} to the left of the top left skyline tuple t_1 and to the right of the bottom right skyline tuple t_s respectively. In other words, $t_0 = (0, \max t_i[A_2])$, and $t_0 = (\max t_i[A_1], 0)$, where $\max t_i[A_j]$ is the maximum value of the A_j in all skyline tuples. Figure 35 shows a dataset with 5 skyline tuples $\{t_1, t_2, \cdots, t_6\}$ and two dummy tuples t_0 and t_7. Next, we propose a graph model for the two-dimensional Regret-ratio Minimizing Set problem and we propose a polynomial time algorithm using dynamic programming in order to find the r tuples such that the maximum regret-ratio is minimum.

5.4.1 Graph Modeling

Reduction to Path Search in Graph We model the two-dimensional Regret-ratio Minimizing Set problem as a weighted complete graph $G = (V, E)$, where V is the set of skyline tuples, $\{t_1, t_2, \cdots, t_s\}$, and two dummy tuples, t_0 and t_{s+1}. Edge weight $w(t_i, t_j)$ represents the regret-ratio of removing all skyline tuples between t_i and t_j. Note that as we proved in the Theorem 6, the optimal solution is a subset of skyline tuples. Using this graph model, our goal is to find a path from t_0 to t_{s+1} with at most r intermediate tuples such that tuples follow an increasing order of the subscript in the path and the maximum of the edge weights are minimized. Next, we discuss how to efficiently compute the edge weight $w(t_i, t_j)$. Using the graph model we first discuss the baseline solution for the problem, and then we describe the detail of the dynamic programming approach.

Edge weight computation Each convex hull tuple is the maximum for a set of ranking functions. For example in Figure 36, t_1 is the maximum for all linear ranking functions from $F \in [0, \theta_1]$, while t_3 is the representative of all linear ranking functions from $F \in [\theta_1, \theta_2]$. Similarly, t_4, and t_5 are the representatives

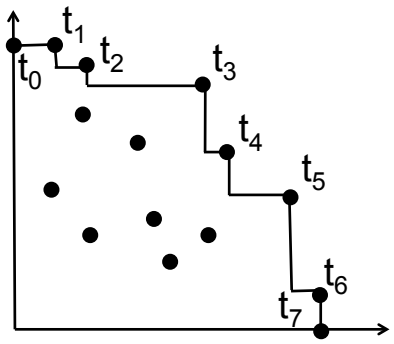


Figure 35: Dataset with skyline tuples $\{t_1, t_2, \cdots t_6\}$ and two dummy tuples t_0 and t_7.

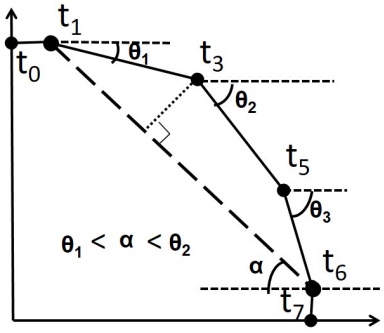


Figure 36: Error of removing $\{t_2, \cdots, t_5\}$

of all linear ranking functions of $F \in [\theta_2, \theta_3]$ and $F \in [\theta_3, \pi/2]$ respectively. One may note that moving from the top-left to the bottom-right, the contours of the all ranking functions where the convex hull tuples are maxima form a sorted range of the angles from 0 to $\pi/2$. For example, in Figure 36, the sorted list $\ell = [0, \theta_1, \theta_2, \theta_3, \pi/2]$ shows all possible linear ranking functions where the convex hull tuples are the maxima. In other words the ith element of l shows all ranking functions $F \in [\theta_{i-1}, \theta_i]$ where the ith convex hull tuple is the maximum.

In Theorem 7, we prove that for two-dimensional databases, the function F that causes the maximum regret-ratio for removing the tuples between two skyline tuples t_i and t_j is specified by the line between tuples t_i and t_j. Intuitively, when we keep t_i and t_j while removing all skyline tuples in between, and start tallying the loss from removing t_{i+1}, ..., t_{j-1} in order, then the loss (max over all ranking functions) must first increase and then decrease. Having identified the max-loss function, we now take the advantage of the sorted angle list (ℓ) and apply *binary search* on ℓ to find the convex hull tuple between t_i and t_j which is the maximum for the ranking function represented by the line passing through t_i and t_j. Algorithm 9 shows the pseudocode of the function **ComputeEdgeWeight** for the two tuples t_i and t_j. Note that if it can not find a convex hull tuple between t_i and t_j the edge weight is zero (line 4-5). Clearly since we use binary search on the sorted list l, we are able to find weight of each edge in $O(\log c)$, where c is the size of the convex hull.

Algorithm 9 ComputeEdgeWeight

1: **Input:** Tuples t_i and t_j, Sorted list $l = [0, \theta_1, \cdots, \pi/2]$
2: **Output:** Edge weight $w(t_i, t_j)$
3: **if** i = 0 **then return** $t_1[A_2] - t_j[A_2]$
4: **if** j = s+1 **then return** $t_s[A_1] - t_i[A_1]$
5: compute α, where t_i and t_j are the maxima of all linear ranking functions $F \in [0, \alpha]$.
6: k = Use binary search on l to find the location of α
7: **if** $i \leq k \leq j$ **then** $w(t_i, t_j) = \frac{\min(F_\alpha(t_k) - F_\alpha(t_i), F_\alpha(t_k) - F_\alpha(t_j))}{F_\alpha(t_k)}$
8: **else** $w(t_i, t_j)$ = 0
9: **return** $w(t_i, t_j)$

Theorem 7. *In 2D, after removing the tuples between two skyline tuples t_i and t_j, the maximum regret-ratio occurs for the function, F, corresponding to the line between tuples t_i and t_j.*

Proof. Let F be the ranking function specified by the line between tuples t_i and t_j. For a tuple t between t_i and t_j and a ranking function $F' \in \bigcup_{\forall k \in [i,j]} \mathcal{F}_k$ either t_i or t_j is the maxima after removing the tuples $\{t_k | i < k < j\}$. More specifically, if (the angle of) $F' < F$, t_i (and if $F' > F$, t_j) will be the maxima. Let us name the best alternative for F' (either t_i or t_j) as t'. For example in Figure 34, l_2 shows the function F and for any ranking function between l_1 and l_2, t_2 is the best alternative,

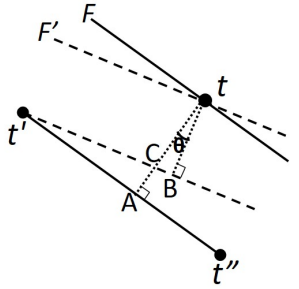


Figure 37: Illustration of the distance of function scores

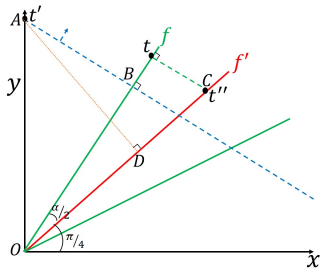


Figure 38: Illustration of maximum growth in 2D.

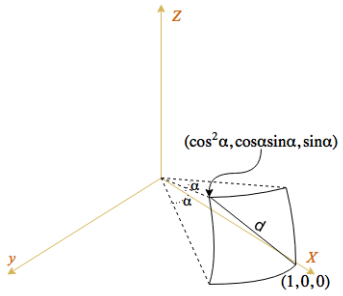


Figure 39: Illustration of the cell diameter in 3D.

while the best alternative for the functions between l_2 and l_3 is t_4.

Suppose the lines F in Figure 37 is parallel with the line passing though the points t' and t'' (representing the function for which both t' and t'' have equal scores) and let F' be a function in $\mathcal{F}_t$ for which t' is the best alternative (the same analysis is valid for a F'' for which t'' is the best alternative). Since the line tA is perpendicular to the line $t't''$, $F(t) - F(t')$ is equal to the distance between t and A (shown as $|tA|$). On the other hand, since the lines tB and F' are perpendicular, $F'(t) - F'(t')$ is equal to the distance between t and B (shown as $|tB|$). Looking at the figure, $|tC| < |tA|$. Moreover:

$$|tC| = \frac{tB}{\cos\theta}$$

since $\cos\theta < 1$

$$\Rightarrow F'(t) - F'(t') = |tB| < |tC| < |tA| = F(t) - F(t')$$

Now given that $F(t) - F(t') > F'(t) - F'(t')$, our goal is to prove that:

$$\frac{F(t) - F(t')}{F(t)} > \frac{F'(t) - F'(t')}{F'(t)} \tag{23}$$

If $F(t) \leq F'(t)$, since $F(t) - F(t') > F'(t) - F'(t')$, then Equation 23 holds.
If $F(t) > F'(t)$:

$$F(t) - F(t') > F'(t) - F'(t') \Rightarrow F(t) - F'(t) > F(t') - F'(t')$$

Let $\sigma = F(t') - F'(t')$ and $\delta = F(t) - F'(t) - \sigma$.

$$\begin{aligned}
&\Rightarrow F(t) = F'(t) + \sigma + \delta \text{ and } F(t') = \sigma + F'(t') \\
&\Rightarrow \frac{F(t) - F(t')}{F(t)} = \frac{F'(t) + \sigma + \delta - \sigma - F'(t')}{F'(t) + \sigma + \delta} \\
&\quad = \frac{F'(t) - F'(t') + \delta}{F'(t) + \sigma + \delta}
\end{aligned}$$

Since $\sigma \geq 0$ and $\delta \geq 0$,

$$\Rightarrow \frac{F'(t) - F'(t') + \delta}{F'(t) + \sigma + \delta} > \frac{F'(t) - F'(t')}{F'(t)}$$

□

For example, let us consider the edge between t_1 and t_5 (representing the removal of $\{t_2, \cdots, t_4\}$). As shown in Figure 36, t_1 and t_5 will be the representative of all linear ranking functions $F \in [0, \alpha]$. Using binary search on l, it turns out that $\alpha \in (\theta_1, \theta_2)$, i.e., t_3 is the tuple which is removed and has the maximum loss. Therefore, $w(t_1, t_5) = \frac{\min(F_\alpha(t_3) - F_\alpha(t_1), F_\alpha(t_3) - F_\alpha(t_5))}{F_\alpha(t_3)} = \frac{F_\alpha(t_3) - F_\alpha(t_1)}{F_\alpha(t_3)}$ (line 4 in Algorithm 9). As another example let us consider $w(t_1, t_2)$. Since there are no convex hull tuples between these two tuples, $w(t_1, t_2) = 0$. Nevertheless, t_3 is the maximum for the function represented by the line passing through t_1 and t_2, which is not removed by considering the edge t_1 to t_2.

Next, we discuss a baseline solution for RRMS problem in 2D, based on the proposed graph modeling.

5.4.2 Baseline Solution

Given the graph model and weight definition, a baseline solution is to compute all weights of the graph and then enumerate all paths from t_0 to t_{s+1} with at most r intermediate tuples. Among those paths the one whose maximum edge weight is the minimum is the solution. Clearly this is inefficient because it takes time quadratic in the number of skyline tuples ($O(r^2)$) to calculate all edge weights in the graph. Moreover, it has to enumerate all $\sum_{l=0}^{r} \binom{n-2}{l}$ paths from t_0 to t_{s+1} with at most r intermediate tuples, which can take exponential time. Next, we leverage the locality property of the skyline tuples in order to propose a polynomial time algorithm **2D-RRMS** using a dynamic programming approach.

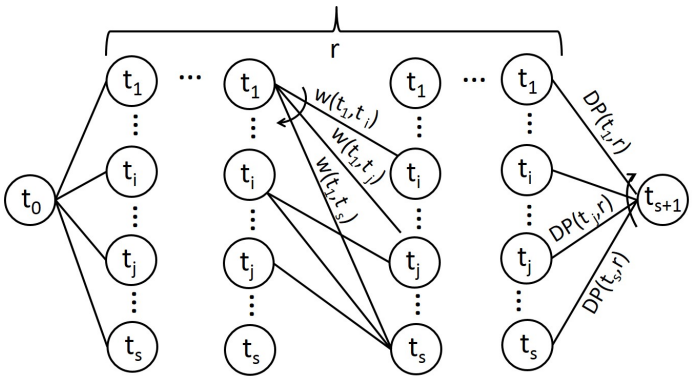


Figure 40: Dynamic programming approach of 2D-RRMS Algorithm.

5.4.3 Dynamic Programming Algorithm

Let $DP(t_i, r')$ be the optimal solution, i.e., a path from t_i to t_{s+1} with at most $r' \leq r$ intermediate nodes which minimizes the error. Thus, $DP(t_0, r)$ would be the solution to our problem. The recursive formula for the dynamic programming is given by Equation 24:

$$\begin{aligned} DP(t_i, 0) &= \max(w(t_0, t_i), w(t_i, t_{s+1})) \\ DP(t_i, r') &= \min_{\forall j > i} (max\,(w(t_i, t_j), DP(t_j, r' - 1))) \end{aligned} \tag{24}$$

Since skyline tuples are ordered, they provide two important properties which are helpful to efficiently solve the recursive Equation 24:

1. $w(t_i, t_j) \leq w(t_i, t_{j+1})$
2. $DP(t_j, r') \leq DP(t_{j-1}, r')$.

Figure 40 shows the construction of the dynamic programming algorithm for DP. It contains t_0 and t_{s+1} at its first and last tuples. Every cell (i, j) in the middle matrix represents $DP(t_i, r - j)$. As shown in the figure, the weights increase from top to bottom, while DP increases from bottom to top.

In order to find the $\min$ value in Equation 24, Algorithm 10 divides the space between t_i and t_{s+1} into two halves and picks the one in the middle as t_m. Note that the weight computation will be done online as needed. If the edge weight $w(t_i, t_m)$ is not previously calculated it will call the function **ComputeEdgeWeight** in

Algorithm 9 (lines 6-16). Then if $w(t_i, t_m) \geq DP(t_m, r' - 1)$, we can ignore the nodes between t_m and t_{s+1} because, based on property (1), $w(t_i, t_m)$ is the smallest among them and those branches will not find a better result (line 14). In this case, Algorithm 10 continues by dividing the tuples between t_{i+1} and t_{m-1}. On the other hand, if $w(t_i, t_m) < DP(t_m, r' - 1)$ (line 15), we can ignore the nodes between t_i and t_m because, based on property (2), $DP(t_m, r' - 1)$ is the smallest among them. In this case, Algorithm 10 continues to divide the tuples between t_{m+1} and t_s.

Algorithm 10 2D-RRMS

1: **Input:** Integer $r \geq 1$, Skyline tuples $\mathcal{S} = \{t_0, t_2, \cdots, t_{s+1}\}$, Sorted list $l = [0, \theta_1, \cdots, \pi/2]$
2: **Output:** The optimal regret-ratio $DP(t_0, r)$
3: **for** i from 1 to s **do** $DP(t_i, 0) = \max(w(t_0, t_i), w(t_i, t_{s+1}))$
4: **for** r' from 1 to r **do**
5: **for** i from 1 to s **do**
6: $low = i + 1$, $high = s$
7: **while** True **do**
8: **if** $low = high$ **then**
9: $DP(t_i, r') = \max(w(t_i, t_{low}), DP(t_{low}, r' - 1))$
10: break
11: **end if**
12: Let t_m be the middle tuple in $\{t_{low}, \cdots, t_{high}\}$
13: **if** $w(t_i, t_m)$ is unknown **then**
 $w(t_i, t_m) =$ **ComputeEdgeWeight**(t_i, t_m, l)
14: **if** $w(t_i, t_m) \geq DP(t_m, r' - 1)$ **then** $high = m$
15: **else** $low = m + 1$
16: **end while**
17: **end for**
18: **end for**
19: **return** $DP(t_0, r)$

Theorem 8. *Time complexity of the algorithm 10 (2D-RRMS) is in* $O(rs \log s \log c)$.

Proof. As shown in Figure 40, DP can get constructed considering a back-track matrix completion approach from level (column) r to level 0, while the values at

level i can get computed from the level $i+1$. The matrix has s rows and r columns. For each cell of the matrix, a binary search with order of $O(\log s)$ is applied for finding the $\min$ value in Equation 24, and at each step of the binary search, if $w(t_i, t_j)$ is unknown, it will call the Algorithm 9 to compute the edge weight which takes $O(c)$. Thus, the overall complexity of 2DEP is $O(rs \log s \log c)$. □

5.5 HD Regret-ratio Minimizing Set

It is well known that the size of convex hull grows exponentially with the data dimensionality (i.e., the number of attributes) [74] (also shown in Figure 33). The need for designing a compact representation of high dimensional databases thus becomes even more pronounced. We address the discovery of regret-ratio minimizing sets over high-dimensional databases in this section.

Specifically, we start with discussing the deficiency of the existing heuristic solution. Then, we introduce a conceptual model of the problem as an infinitely large matrix *min-max* problem [70]. We "operationalize" such a conceptual model with a matrix discretization approach that provides a user-controllable discretization parameter. After discretizing the problem space to a manageable size, we then construct a reduction to *set-cover* [71] which solves the min-max problem deterministically in (theoretically) $O(n \log n)$ time, while guaranteeing a regret ratio within any arbitrarily small user-controllable distance from the optimal regret ratio. We make this algorithm more practical by incorporating an existing approximation algorithm for set-cover. Our final algorithm outperforms existing solutions by an order of magnitude over real-world datasets, as we shall show in the experimental evaluations.

5.5.1 Problem with Existing Heuristic Solution

Finding r tuples that minimize the regret-ratio over a high-D database has been proven to be NP-hard [69]. The existing solutions limit to a greedy heuristic (named as GREEDY) and a simple space discretization approach that produces a regret ratio within *a fixed distance from the optimal*, both proposed by Nanongkai

et. al. [68]. The basic idea of GREEDY is to start by selecting the point that has the highest value on the first attribute, and then iteratively selecting the point with the maximum error from the selected points and adding it to the set. This algorithm is not designed to provide any approximate-ratio guarantee. In addition, as we shall show below, it performs quite badly in some cases. Specifically, for any *arbitrarily large* value v, we can always find a case in which the regret-ratio of the solution provided by GREEDY is no better than v times the optimal solution.

Given $v > 0$, let $\epsilon = 1/(2+v)$. Consider a 3-dimensional database D which contains four tuples $t_0(1,0,0)$, $t_1(0,1,0)$, $t_2(0,0,1)$, and $t_3(1-\epsilon, 1-\epsilon, 1-\epsilon)$, along with an arbitrary number of other tuples that are distributed in the region $[0, 1-\epsilon) \times [0, 1-\epsilon) \times [0, 1-\epsilon)$. One can see that, when we run GREEDY with output-size requirement $r = 3$, GREEDY will pick t_0, t_1, and t_2, while the optimal solution is t_3 together with two of t_0, t_1, and t_2.

To see how bad the regret ratio for GREEDY's solution is, note that its regret ratio is equal to the distance between point t_2 and the line passing through points t_0 and t_1, which is equal to $1 - 2\epsilon$. Meanwhile, the regret ratio for the optimal solution is ϵ. Thus, the approximate-ratio is $(1-2\epsilon)/\epsilon \geq v$.

5.5.2 Conceptual Model

An intuitive illustration of our conceptual model is shown in Figure 41. Specifically, consider a matrix M that has the n tuples as its rows, while its columns consist of all possible linear ranking functions ($\mathcal{F}$). Each cell $M[t_i, f]$ of the matrix is the regret-ratio of t_i with regard to the ranking function f - i.e., the regret ratio for f if we only have t_i in the minimizing set. Thus for each tuple $t_i \in \mathcal{C}$ and the set of ranking functions $f \in \mathcal{F}_{t_i}$ (the set for which t_i is the maximum), $M[t_i, f]$ is zero, while this value is greater than zero for other tuples. If we keep r rows of the matrix, the regret-ratio for each function, is the minimum value (among the selected rows) on its corresponding column, and the regret-ratio of these r tuples is the maximum assigned regret-ratio of all columns. We can see the problem cleanly transforms to a *min-max* problem over the matrix.

This conceptual model, unfortunately, has an important issue which makes it

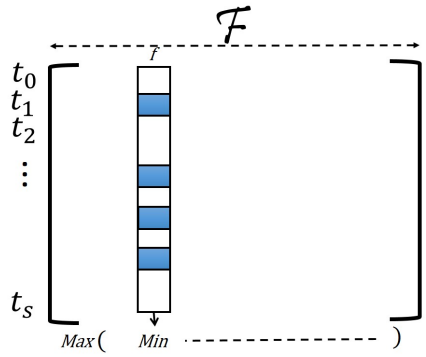


Figure 41: Illustration of Matrix M.

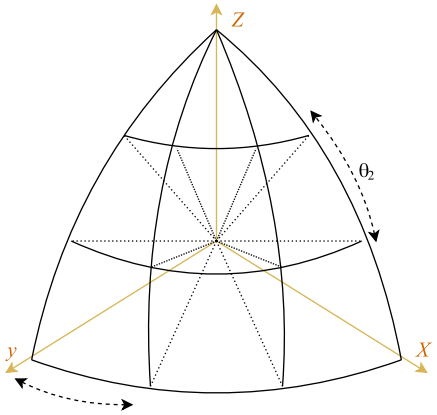


Figure 42: Illustration of space partitioning example: $m = 3, \gamma = 3$.

impractical: $\mathcal{F}$ is continuous and therefore requires an infinite number of columns to capture! This issue inspires us to develop a *matrix discretization* approach which eventually leads to a linearithmic approximate solution that offers a guaranteed approximation ratio adjustable by a user-controlled parameter.

5.5.3 Matrix Discretization

In order to resolve the aforementioned issues with the conceptual model, we first discretize $\mathcal{F}$ by only considering a subset of all possible linear ranking functions $F \subset \mathcal{F}$ as the columns of the matrix M. We take the help of Polar coordinate system for selecting F. Note that, with help of the polar system, each point is denoted by one magnitude and $m - 1$ angles. For example, tuple $t(1, 1)$ is transformed to $t\langle\sqrt{2}, \pi/4\rangle$ in polar and $t'(1, 0, 1)$ to $t'\langle\sqrt{2}, 0, \pi/4\rangle$.

Specifically, we introduce a user-controllable parameter γ which determines the size of F, by dividing each angle into γ equal-size partitions. Thus, for a given value γ each angle partition is:

$$\alpha = \frac{\pi}{2\gamma} \tag{25}$$

Applying this discretization policy, the total number of selected functions ($|F|$) is:

$$|F| = (\gamma + 1)^{m-1} \tag{26}$$

Algorithm 11 shows the pseudo-code of the DISCRETIZE algorithm, that partitions the ranking function space based on α and selects F. For example, when $m = 3$ and $\gamma = 3$, we have $\alpha = \pi/6$ according to (25). Figure 42 shows the three-dimensional function space discretization based on α.

Algorithm 11 DISCRETIZE

Input: Control Parameter γ, Number of attributes m
Output: Discretized functions F
$F = \{\}$, $\alpha = \frac{\pi}{2\gamma}$
for i from 1 to $m - 1$ **do** $\theta[i] = 0$
for i from 1 to γ^{m-1} **do**
    $r = 1$
    {*transforming the function from polar system to scalar*}
    for j from m downto 2 **do**
        $v[j] = r \cos(\theta[j-1]\alpha)$
        $r = r \sin(\theta[j-1]\alpha)$
    end for
    $v[1] = r$
    $F = F \cup \{v\}$
    {*finding the next function*}
    for j from 1 until $\theta[j] < \gamma$ **do** $\theta[j] = 0$
    $\theta[j] = \theta[j] + 1$
end for
return F

Theorem 9. *If a set T of tuples guarantee a regret-ratio threshold of ϵ for all the ranking functions in $f \in F \subseteq \mathcal{F}$, constructed based on the angle partitioning (α) in Equation 25, the maximum regret-ratio of those points for any ranking function $f' \in \mathcal{F}$ is:*

$$\epsilon' \leq c\epsilon + (1 - c) \tag{27}$$

where $c = \frac{\cos(\alpha'/2)\cos(\pi/4)}{\cos(\pi/4-\alpha'/2)}$ *and* $\alpha' = 2\arcsin(\sqrt{\frac{1-cos^{m-1}\alpha}{2}})$.

Proof. We want to see if the regret-ratio of each representative tuple for all ranking function $f \in F$ that are picked is within the ϵ regret-ratio bound, how much is the maximum regret-ratio for a missing ranking function. For the simplicity, let us start with the 2D case where there are two attributes x and y (Figure 38). Since each missing ranking function is bounded between two selected functions with angle α, its maximum angle with its closest selected ranking function is at most $\frac{\alpha}{2}$ (the worst case is when the ranking function is in the middle of two consequent selected functions). Consider a representative tuple for a ranking function $f \in F$ which is in the ϵ range of the top representative of f. Again the worst case happens when the regret-ratio of such tuple is exactly ϵ for the selected ranking function. Suppose the top green line and the red line in Figure 38 shows f and the missing ranking function with angle distance $\frac{\alpha}{2}$ from it respectively. Any tuple above the blue perpendicular dashed line with f is within the ϵ threshold bound for it. As specified in the figure, the intersection of the dashed blue line and the y-axis (tuple t' in the figure) maximizes the distance of the representative of f for f'. Moreover, in order to maximize the distance we put the Top-1 for f to be exactly on it, i.e. tuple t. Thus, the maximum regret-ratio of t' for f' is:

$$\begin{aligned}
\epsilon' =& \frac{f'(t'') - f'(t')}{f'(t'')} = \frac{OC - OD}{OC} \\
OC =& \frac{f(t)}{\cos(\alpha/2)} \\
OD =& OA\cos(\pi/4), OA = \frac{f(t')}{\cos(\pi/4 - \alpha/2)} \\
\Rightarrow OD =& \frac{f(t')}{\cos(\pi/4 - \alpha/2)}\cos(\pi/4)
\end{aligned}$$

$$\epsilon' = \frac{\frac{f(t)}{\cos(\alpha/2)} - \frac{f(t')}{\cos(\pi/4-\alpha/2)}\cos(\pi/4)}{\frac{f(t)}{\cos(\alpha/2)}}$$
$$= \frac{f(t) - f(t')\frac{\cos(\alpha/2)\cos(\pi/4)}{\cos(\pi/4-\alpha/2)}}{f(t)}$$

$$\text{Let } c = \frac{\cos(\alpha/2)\cos(\pi/4)}{\cos(\pi/4-\alpha/2)}$$
$$\Rightarrow \epsilon' = \frac{f(t) - cf(t')}{f(t)} = c\frac{f(t) - f(t')}{f(t)} + (1-c)$$

Since $0 < c \leq q$ and $\frac{f(t)-f(t')}{f(t)} \leq \epsilon$, $\Rightarrow \epsilon' \leq c\epsilon + (1-c) \leq \epsilon + (1-c)$.

Now let us extend the computation to the high-dimensional case. As shown in Figure 42, the space partitioning can be seen as a set of (hyper-)cones originated at point $(0, \ldots, 0)$. Looking from the surface of the cone, each missing function is covered by a set of selected ranking functions in F which together form a (hyper-)trapezius around it. Consider the hyper-plane constructed between the origin and the two points in the diameter of hyper-trapezius, in which the distance between the two selected points is maximum. One can see that the maximum growth in the regret-ratio of a point and a missing function happens for the function in the middle of this hyper-plane (that has the maximum angle distance with the selected ranking functions). In the following, looking at Figure 39, we compute the distance d between the two diagonal points, and use it to compute the maximum angle (α') between two adjacent ranking functions.

In order to do so, we consider the corners (1,0,. . . ,0), i.e. the point on X-axis and the point $(\cos^{m-1}\alpha,\ \cos^{m-2}\alpha\sin\alpha,\ \cos^{m-3}$
$\alpha\sin\alpha,\ \ldots,\ \sin\alpha)$. (computed by transforming the coordinates from the polar to scalar system).

$$d = \sqrt{(1-\cos^{m-1}\alpha)^2 + (\cos^{m-2}\alpha\sin\alpha)^2 + \cdots + \sin^2\alpha}$$

$$= \sqrt{(1 - \cos^{m-1}\alpha)^2 + \sin^2\alpha \sum_{i=1}^{m-2} \cos^{2i}\alpha}$$

Following the geometric series (while replacing $\sin^2\alpha$ with $1 - \cos^2\alpha$),

$$\begin{aligned} d &= \sqrt{(1 - \cos^{m-1}\alpha)^2 + (1 - \cos^{2(m-1)}\alpha)} \\ &= \sqrt{2(1 - cos^{m-1}\alpha)} \end{aligned} \tag{28}$$

Considering the value of $\frac{d}{2}$, while knowing that the radius of the hyper-sphere is 1,

$$\alpha' = 2\arcsin(\sqrt{\frac{1 - cos^{m-1}\alpha}{2}}) \tag{29}$$

Now using the same analysis we did for the two-dimensional case:

$$\epsilon' \leq c\epsilon + (1 - c) \leq \epsilon + (1 - c)$$

where

$$c = \frac{\cos(\alpha'/2)\cos(\pi/4)}{\cos(\pi/4 - \alpha'/2)}$$

□

Considering the guarantee provided in Theorem 9, we discuss our approximate algorithm over the discretized function space in the next section.

5.5.4 HD-RRMS Algorithm

In this section, we first model the problem as the discretized min-max problem. We then take the advantage of the linear-size discretized problem space in order to find the optimal value for the discretized matrix min-max problem.

DMM: Discretized min-max Problem Given the (above described) discretized matrix M, and the value r, find a set of r rows that minimizes the maximum of the

minimum values of all columns among the selected r rows. Formally, find:

$$\min_{\forall R \subseteq \mathcal{S} \text{ s.t. } |R|=r} \max_{\forall f \in F} \min_{\forall i \in R} (M[i, f]) \tag{30}$$

If we assume that the number attributes, m, is bounded by a constant, and the user controlled parameter γ is a constant, then the total number of discrete functions $|F|$ becomes bounded by a (albeit large) constant. Recall that M is a matrix with n rows and $|F|$ columns, where each cell shows the regret-ratio. Given any set of r rows, *the solution to the discretized min-max problem is one of the cell values*. The total number of such values is, at most, $n.|F|$, which since $|F|$ is bounded by a constant, is in $O(n)$. We consider each distinct value in cells of the matrix M as a possible ϵ threshold of the following problem.

MRST: Minimum Rows Satisfying the given Threshold problem: Given the discretized matrix M and the threshold value ϵ, find the minimum number of rows in matrix M such that for each column of matrix M, the minimum value of each column among the selected rows is less than or equal to ϵ. Formally:

$$\text{minimize } |R|, \forall R \subseteq \mathcal{S} \text{ where } \max_{\forall f \in F} \min_{\forall i \in R} (M[i, f]) \leq \epsilon \tag{31}$$

Consider an oracle that solves the MRST problem. HD-RRMS sorts all the distinct values in matrix M, and applying the binary search strategy, passes the cell-values to the oracle to find the set of rows that satisfy the threshold. Based on the fact that the size of returned set is either less than r or not, it continues the search in lower/upper half. Algorithm 12 shows the pseudo-code of the HD-RRMS algorithm in the presence of the MRST oracle.

Suppose T_O is the optimal set with the minimum regret-ratio. Note that since the output of the HD-RRMS algorithm is the optimal solution for a subset of ranking functions, ϵ_{min} is less than or equal to the regret-ratio of T_O:

$$\epsilon_{min} \leq E_{D \setminus T_O} \tag{32}$$

That is because, if there is no subset of tuples with size of at most r that satisfy

Algorithm 12 HD-RRMS

Input: The discretized matrix M and the value r
Output: selected tuples (T_A)
v = sorted list of distinct values in M
$T_A = \{\}$, $\epsilon_{min} = \infty$
$low = 0$, $high = |v|$
while $low < high$ **do**
  $mid = \frac{low+high}{2}$
  R = **MRST**(M, $v[mid]$)
  if $|R| \leq r$ **then**
    $T_A = R$, $\epsilon_{min} = v[mid]$
    $high = mid - 1$
  else
    $low = mid + 1$
  end if
end while
return T_A

the regret-ratio of less than ϵ_{min} for F (which is a subset of $\mathcal{F}$), no set will have the regret-ratio of less than ϵ_{min} for its super-set ($\mathcal{F}$). Moreover, based on Theorem 9 we know $E_{D\setminus T_A} \leq c\epsilon_{min} + (1-c)$. From Equation 32 $c\epsilon_{min} + (1-c) \leq cE_{D\setminus T_O} + (1-c)$. Thus:

$$E_{D\setminus T_A} \leq cE_{D\setminus T_O} + (1-c) \tag{33}$$

Since $c \leq 1$, based on Equation 33, the regret-ratio of the set discovered by HD-RRMS is within $(1-c)$ distance from the regret-ratio of the optimal solution.

MRST Oracle: The only missing part of the algorithm is the MRST oracle. We model the MRST problem with the set-cover problem [71] by constructing the matrix M' as following:

$$\forall i, f: \; M'[i,f] = \begin{cases} 0, & \text{if } M[i,f] > \epsilon \\ 1, & otherwise \end{cases} \tag{34}$$

M' contains constant number of columns and s rows. Since the number of columns

is bounded by a constant, there exists at most a constant number (more precisely, the power-set of $|F|$) of distinct combinations for the row values of M'. In the next step, the algorithm (giving a distinct id to each value-combination of columns) parses M' and removes the duplicate rows. As a result, M' becomes a matrix whose number of rows and columns are bound by constants! Next we transform the MRST problem to the *set-cover* problem as follows:

- Each column f in M' is an item in the set-cover problem.
- Each row i in M' is a set $\mathbb{S}_i$ in the set-cover problem such that $\forall$ column f, $f \in \mathbb{S}_i$ *iff* $M'[i, f] = 1$.

Now the problem is converted to a set-cover instance with a constant number of items and constant number of sets. Thus, even though the set-cover problem is NP-complete in general, an optimal solution for this instance is, *theoretically*, possible in constant-time order. MRST uses the set-cover solver to find the minimum number of sets that cover all the columns and returns its corresponding rows as the optimal solution for the MRST problem. Algorithm 13 shows the pseudo-code of MRST oracle.

Time Complexity of the HD-RRMS Algorithm Since the size of the discretized matrix M is $n|F|$, the binary search over the possible values of ϵ takes $\log(n|F|)$ steps. At each step, a set-cover with $|F|$ items and $\min(2^{|F|}, n)$ sets is constructed. Converting M to the set-cover instance takes $n|F|$ time, while solving it takes, at most, $2^{\min(2^{|F|},n)}|F|$ time.

Since, $|F| = \gamma^m$, the total time to solve HD-RRMS is:

$$\log(n\gamma^m).(n\gamma^m + (2^{\min(2^{\gamma^m},n)}.\gamma^m)) \tag{35}$$

Assuming that γ and m are constants, the running-time of HD-RRMS is, *theoretically*, in $O(n \log(n))$.

5.5.5 Practical HD-RRMS Algorithm

Although HD-RRMS is linearithmic in theory, it can be inefficient in practice. Even with the existence of an efficient set-cover solver (such as [75]), depending

Algorithm 13 MRST Oracle

Input: The discretized matrix M and the threshold ϵ
Output: The set of tuples satisfying the threshold on M
{*constructing the matrix* M'}
for i from 1 to s **do**
  for f in columns of M **do**
    $M'[i, f] = 1$ **if** $(M[i, f] \leq \epsilon)$ **else** 0
  end for
end for
{*removing the duplicate rows*}
seen = $\{\}$
for i in rows of M' **do**
  if id($M'[i]$) $\in$ seen) **then**
    M'.remove(i)
  else
    seen = seen $\cup\{i\}$
  end if
end for
{*constructing the set-cover problem*}
items=$\{\}$, sets=$\{\}$
for f in columns of M' **do** items = items $\cup\{f\}$
for i in rows of M' **do**
  set=$\{\}$
  for f in columns of M' **do**
    if $M'[i, f] == 1$ **then** set = set $\cup\{f\}$
  end for
  sets = sets $\cup\{$ set $\}$
end for
return set-cover(items, sets)

on the values of γ and m, solving this problem may become infeasible.

To make the HD-RRMS Algorithm practical, we solve the set-cover instances approximately using the well-known greedy approximate algorithm for set-cover [72]. Since this algorithm guarantees a $\log$ factor in the approximate-ratio, applying it adds another level of approximation and increases the set size to up to $r \log(|F|) = rm \log(\gamma)$ (rather than r) tuples, while maintaining the distance from the optimal

regret ratio in the bound provided in Theorem 9. Alternatively, one can find a new value (r') for set-cover size such that $r' = \frac{r}{m \log(\gamma)}$. In this way, we can make sure that the set size will not be more than r. However, the distance from the optimal solution may become larger than what provided in Theorem 9. Applying the greedy approximate set-cover reduces the running time to:

$$\begin{aligned} &\log(n\gamma^m).(n\gamma^m + (\min(2^{\gamma^m}, n).\gamma^m)) \\ \leq &\, 2n\gamma^m \log(n\gamma^m) \end{aligned} \tag{36}$$

Given the importance of the choice of γ in Equation 36, we evaluate the impact of the discretization control parameter (γ) in § 5.7.3 (Figures 59, 60, and 61). On examining the experiment results, it turns out even though increasing the value γ increases the running time significantly, the improvement on the quality of results drops quickly. In our experiments, choosing γ between 4 and 6 seemed to be appropriate.

5.6 Discussion

5.6.1 Top-k Extension

For enabling efficient computation of Top-k queries (instead of only Top-1 queries) requires us to extend the compact set for higher values of $k > 1$, while minimizing the dissatisfaction of the k^{th} top tuple of the compact set versus the k^{th} tuple of the actual database, An easy way of adopting the existing algorithms is an iterative approach with k iterations. At each iteration we discover the maxima set over the remaining tuples. Then we remove the tuples in the maxima set, as well as the tuples falling outside of the convex shape formed by the set, and start the next iteration over the remaining tuples in the database. One can see that this method will construct k compact layers around the data and can serve for discovering the Top-k. However, studying the theoretical guarantees of such adaptation requires further investigation in a future work.

5.6.2 Alternative Matrix Discretization

As an alternative for matrix discretization proposed in § 5.5.3, we can change the algorithm to let the user specify the size of function space as the control parameter (rather than the value γ) which makes $|F|$ independent from the value of m. Now the discretization problem is reduced to the problem of evenly distributing $|F|$ points on the surface of a quarter hyper-sphere (the vector from the origin to a point on surface represents a function $f \in F$).

One way for doing so, is to adopt the force-directed drawing algorithms [76], such as the Barycentric algorithm [77], for evenly distributing the points. Suppose each point (bound to exist on the surface of the hyper-sphere) is a particle with a fixed positive charge. The idea is that if we throw $|F|$ particles on the hyper-sphere, they start moving (based on the average of forces between them) until they form an even distributed in which the superposition of forces is zero. The algorithm thus is as following: until the superposition of forces on all the points has not converged to zero, compute the force on each point and move it on the hyper-sphere based on the direction and the magnitude of it. Another alternative is relaxing the notion of an even distribution of the points to the randomly at uniform distribution. Thus, we can uniformly (at random) select each functions, in the polar space, by specifying the value of each angle uniformly at random between 0 and $\Pi/2$. Now, we can compute the expected angle distance between the two neighboring functions which specifies a expected bound based on Theorem 9.

5.7 Experiments

5.7.1 Experimental Setup

Hardware and Platform: All our experiments were performed on a Core-I7 machine running Ubuntu 14.04 with 8 GB of RAM. The algorithms were implemented in Python.

Real-world Datasets: We used three publicly available real-world datasets, i.e., Airline dataset, Department of Transportation (DOT), and Basketball dataset

(NBA).

- *Airline dataset*[11]*:* The 2008 Airline dataset has $5,810,463$ records with 13 attributes, out of which two of them (namely *Actual Elapsed Time* and *Distance*) are ordinal. We used this dataset to test the performance of the two-dimensional algorithms over a large dataset.
- *DOT dataset*[12]*:* The flight on-time dataset is published by the US Department of Transportation. It records, for all flights conducted by the 14 US carriers in January 2015, attributes such as scheduled and actual departure time, taxiing time and other detailed delay metrics. The dataset consists of 457,013 tuples over 28 attributes with 7 ordinal attributes *Dep-Delay, Taxi-out, Taxi-in, Actual-elapsed-time, Air-time, Distance, ArrivalDelay*.
- *NBA dataset*[13]*:* This basketball dataset contains the points for the combination of player/team/season up to 2009. It contains $21,961$ tuples and 17 ordinal attributes: *gp, minutes, pts, oreb, dreb, reb, asts, stl, blk, turnover, pf, fga, fgm, fta, ftm, tpa, tpm*.

Synthetic Data: We also used synthetic data to evaluate the performance of our algorithms in the presence of different kind of correlations between attributes. Note that such correlations affect the performance of baseline solutions, e.g., the size of the skyline, as the more correlated all attributes are, the smaller the skyline becomes. Moreover, the correlations could affect the regret ratio of our outputs too. Specifically, we used the method proposed in [67] to generate three datasets with correlated, independent, and anti-correlated attributes. Each dataset has $10M$ tuples with 10 attributes.

Algorithms Evaluated: For the two-dimensional case, we evaluated the performance of our two-dimensional algorithm, 2D-RRMS, and compared its performance with the two-dimensional Sweeping-Line algorithm proposed in [69]. Sweeping-Line is a quadratic algorithm that considers the points in the dual space

[11] `http://kt.ijs.si/elena_ikonomovska/datasets/airline/2008_14col.data.bz2`

[12] `http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time`

[13] `http://www.databasebasketball.com/`

and covers the function space using a sweeping line while updating the regret-ratio of the points, as their corresponding lines intersect. We used the Nested Block Loop algorithm [67] in order to compute the skylines for our 2D-RRMS algorithm.

We also evaluated the performance of the high-dimensional algorithm discussed in § 5.5, namely GREEDY [68] and HD-RRMS (Algorithm 12). In the practical implementation of HD-RRMS algorithm, we applied the greedy approximate solution to solve the set-cover problem [71]. Nonetheless, to be fair in comparing the algorithms' performance, we only accept the set-cover result if its size is at most r. Note that our HD-RRMS algorithm features two main ideas: One is the conceptual model (of matrix min-max problem) along with its practical discretization, and the other is the reduction to set-cover and the corresponding approximation algorithm. To test the effectiveness of these two ideas separately, we devised another greedy algorithm called HD-GREEDY, which simply applies an iterative greedy approach over the discretized matrix generated by the first idea (i.e., as explained in § 5.5.3). Specifically, HD-GREEDY iteratively picks a tuple that minimizes the max of the min value of the columns for the selected set of tuples. The complexity of HD-GREEDY is $O(rn)$, because each iteration requires passing through the matrix once, while computing the reduction in the matrix max column value only takes $O(1)$. As one can see, HD-GREEDY uses only the (discretized version of the) conceptual model, but not our second idea of reduction to set-cover. As we shall show latter in the section, the performance of HD-GREEDY almost always falls in between GREEDY and HD-RRMS, thus demonstrating the effectiveness of both of our ideas.

The previous studies on approximating high-dimensional convex hulls [63] and relaxing skyline definitions [65] differ in objective from our paper (which aims to minimize the user dissatisfaction on maxima queries). Nonetheless, in order to provide a broader context for the efficacy of our algorithms, we still implemented both [63] and [65], and studied the possibility of applying them for regret minimization.

Performance Measures: For the two-dimensional case, since our 2D-RRMS algorithm always guarantees optimality in terms of regret ratio, we focus our

evaluations on the *execution time*. Specifically, to be fair to the sweeping-line algorithm, we considered the execution time of our 2D-RRMS algorithm to be the SUM of the execution time of both the skyline computation process (we used the skyline computation algorithm in [67]) and the actual execution of 2D-RRMS. For the high-dimensional case, we used both the regret-ratio and the overall execution time of an algorithm measure its performance - naturally, the shorter the execution time and the smaller the regret ratio, the better.

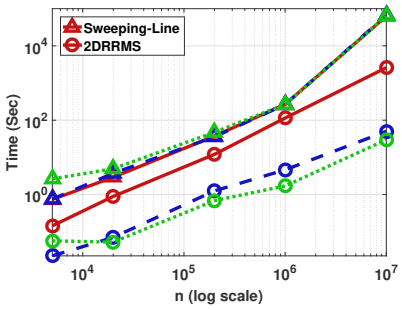


Figure 43: 2D, Impact of dataset size (n) on correlated, independent, and anti-correlated datasets

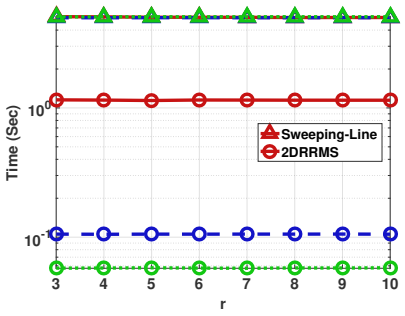


Figure 44: 2D, Impact of output size (r) on correlated, independent, and anti-correlated datasets

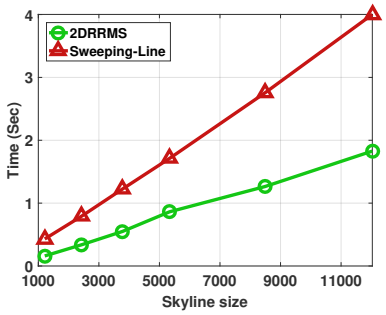


Figure 45: 2D, Impact of skyline size (n) on skyline-only datasets

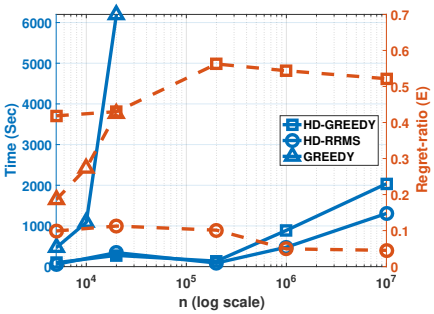


Figure 46: HD, Impact of dataset size (n) on Anti-correlated dataset

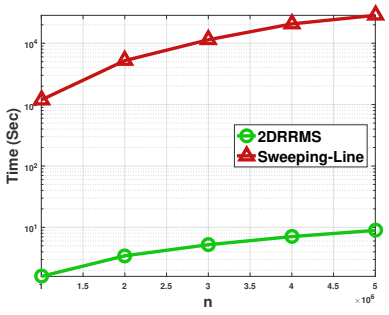


Figure 47: 2D, Airline dataset, varied the dataset size (n)

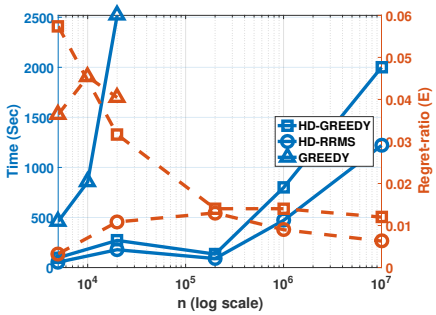


Figure 48: HD, Impact of dataset size (n) on correlated dataset

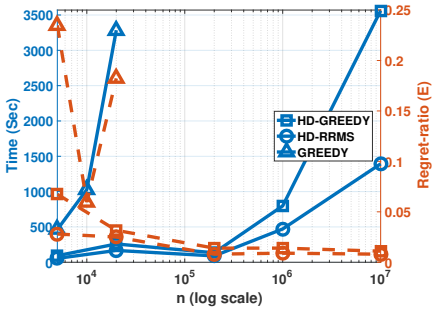


Figure 49: HD, Impact of dataset size (n) on independent dataset

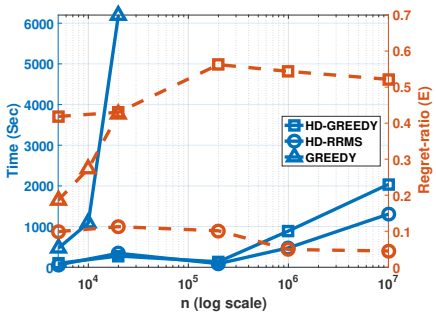


Figure 50: HD, Impact of dataset size (n) on Anti-correlated dataset

5.7.2 Two-dimensional Experimental Result

Figures 43 and 44 show the performance of our 2D-RRMS algorithm (circle marker) and the Sweeping-Line algorithm proposed in [69] (triangle marker). We tested both algorithms over the correlated, independent, and anti-correlated synthetic datasets. In these figures, green dotted line, blue dashed line, and red solid line are used for the correlated, independent, and anti-correlated synthetic datasets respectively.

Impact of the dataset size (n): In these set of experiments, we varied the dataset size from 5K to 10M for each of correlated, independent, and anti-correlated synthetic datasets. Figure 43 shows the execution time of each algorithm. From the figure, one can see that 2D-RRMS algorithm outperforms the Sweeping-Line

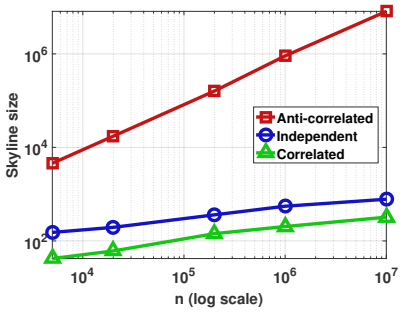


Figure 51: HD, skyline size when varying the dataset size (n)

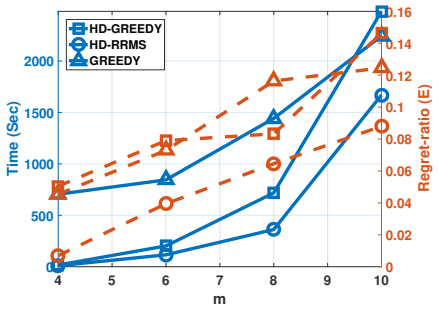


Figure 52: HD, Impact of number of attributes (m) on correlated dataset

algorithm by orders of magnitude. As expected, the performance of the Sweeping-Line algorithm does not depend on the correlation of the attributes because it considers all the joins between points in the dual environment. The 2D-RRMS algorithm performs better for the correlated and independent datasets because the anti-correlated case generates a larger skyline which affects the performance of the Nested Block Loop algorithm [67] used for skyline discovery. Note that, while utilizing more efficient skyline algorithms will reduce the measured execution algorithm of 2D-RRMS, we did not further pursue this direction as it is orthogonal to our research. In addition, 2D-RRMS already outperforms the sweeping-line algorithm by an order of magnitude even in the anti-correlated case.

Impact of the output size (r): Next, we fixed the dataset size to 40K, and varied the output size (r) from 3 to 10 (Figure 44). Again in all experiments, 2D-RRMS algorithm significantly outperforms the Sweeping-Line algorithm. Since the time complexity of the Sweeping-Line algorithm, $O(rn^2)$, is quadratic in dataset size and linear in output size, varying r does not affect the performance of the algorithm. The execution time of our 2D-RRMS algorithm also does not change by varying r. The reason is actually *not* because the execution time of our algorithm has no dependency on r, but because the pre-processing step (skyline discovery) actually dominates the overall running time, and this pre-processing step is independent to r.

Experiment on the skyline-only datasets: To test the skyline-size effect on the

performance of our algorithm, we generated synthetic, "skyline-only" datasets (i.e., in which every tuple is on the skyline), with varying sizes. We did so by drawing uniformly at random from all points inside the 2D unit circle, and then iteratively removing a point if it is dominated by others. We generated 6 such skyline-only datasets of sizes 1212, 2431, 3782, 5335, 8488, and 12032 (skyline) tuples, respectively. Figure 45 shows the performance of the algorithms. We can see that in all cases 2D-RRMS outperformed the sweeping line algorithm significantly. Indeed, the improvement is even more pronounced when the skyline size becomes larger.

Real datasets: Figure 47 show the total execution time of the two-dimensional algorithms over Airline dataset (by only considering Air-time and ArrivalDelay attributes). We used the Airline dataset to evaluate the performance of the algorithms over a large real dataset. Figure 47 shows their performance when we varied the dataset size (n) from 1M to 5M. In both experiments, 2D-RRMS algorithm outperforms the Sweeping-Line algorithm by the orders of magnitude. For example, 2D-RRMS algorithm executed in less than 10 seconds in the Airline dataset for n =5M, while Sweeping-Line algorithm took tens of thousand seconds!

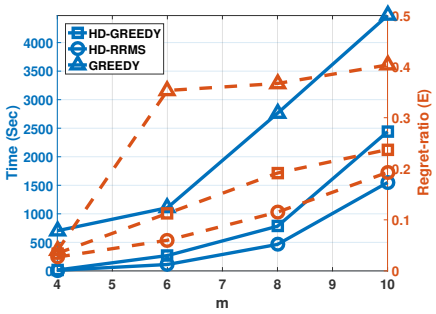


Figure 53: HD, Independent, Impact of number of attributes (m) on independent dataset

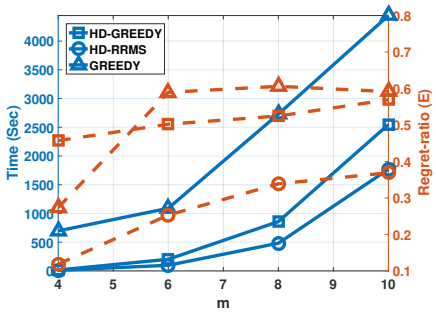


Figure 54: HD, Impact of number of attributes (m) on Anti-correlated dataset

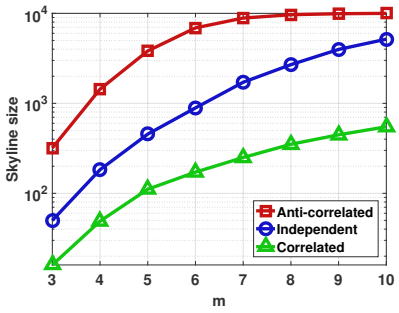


Figure 55: HD, skyline size when varying the number of attributes (m)

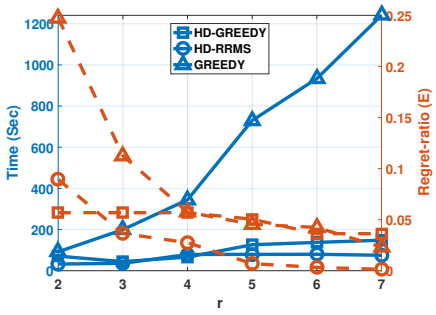


Figure 56: HD, Correlated, Impact of output size (r) on correlated dataset

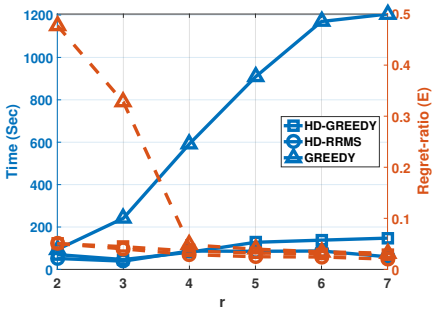


Figure 57: HD, Independent, Impact of output size (r) on independent dataset

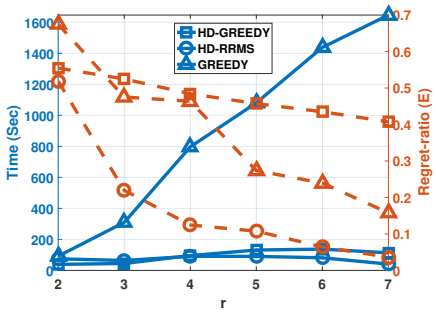


Figure 58: HD, Impact of output size (r) on Anti-correlated dataset

5.7.3 High-dimensional Experimental Result

The performance of the HD algorithms (§ 5.5) is studied under three correlation models, i.e., correlated, independent, and anti-correlated, on synthetic datasets created based on [67]. The default values for the dataset size, number of attributes, output size, and the control parameter to $n = 10$K, $m = 4$, $r = 5$, and $\gamma = 4$ respectively. We studied the impact of each parameter individually as well. Note that in all HD Figures 13 to 28 the left (blue) y-axis shows the execution time of the algorithms while the right (orange) y-axis shows the regret-ratio (E). The triangle, rectangle, and circle line markers represent GREEDY, HD-GREEDY, and HD-RRMS algorithms respectively, while blue solid lines shows the execution

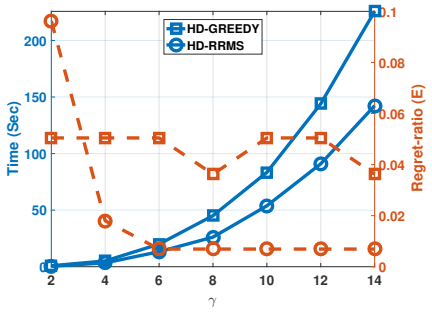


Figure 59: HD, Impact of number of partitions (γ) on correlated dataset

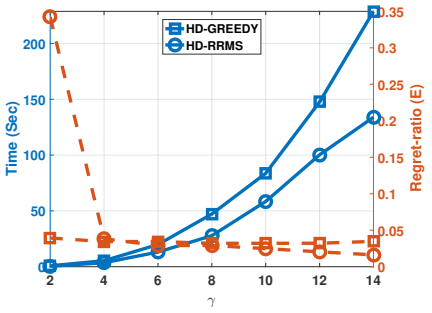


Figure 60: HD, Impact of number of partitions (γ) on independent dataset

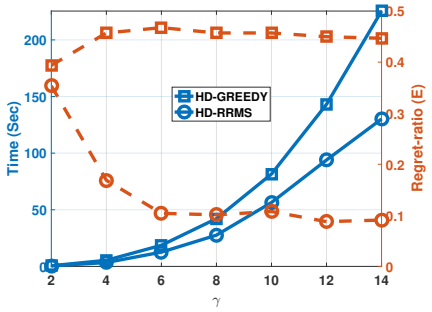


Figure 61: HD, Impact of number of partitions (γ) on Anti-correlated dataset

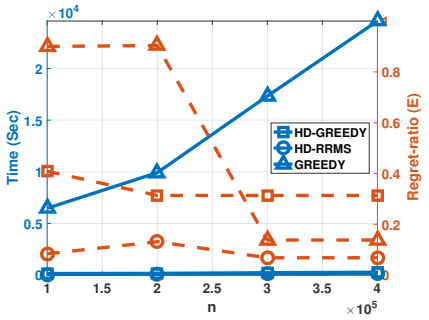


Figure 62: HD, DOT dataset, varied the dataset size (n)

time and dashed orange lines represent the regret-ratio.

Impact of the dataset size (n): In these set of experiments, we varied the size of the dataset (n) from 5K to 10M and evaluated the performance of the three aforementioned algorithms on the synthetic datasets. Figures 48, 49, and 50 show the results for correlated, independent, and anti-correlated datasets respectively. We have compared both the execution time and the regret-ratio of the algorithms to evaluate their performance.

As explained in § 5.5, the GREEDY algorithm suffers from running n LP optimizations before picking a single tuple, which lead to high execution time. As shown in these figures (48, 49, and 50) the GREEDY algorithm did not scale in any

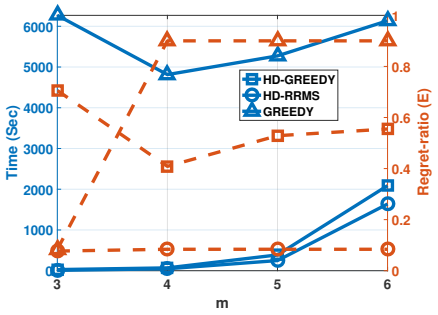


Figure 63: HD, DOT dataset, varied the number of attributes (m)

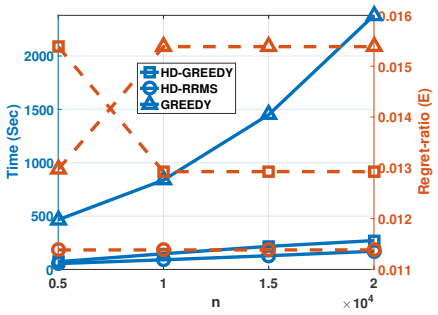


Figure 64: HD, NBA dataset, varied the dataset size (n)

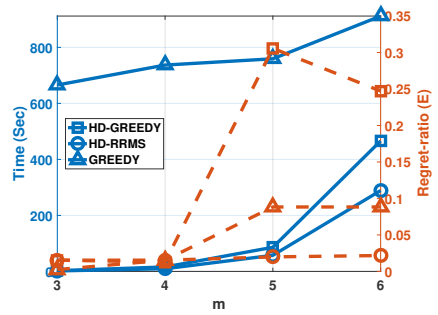


Figure 65: HD, NBA dataset, varied the number of attributes (m)

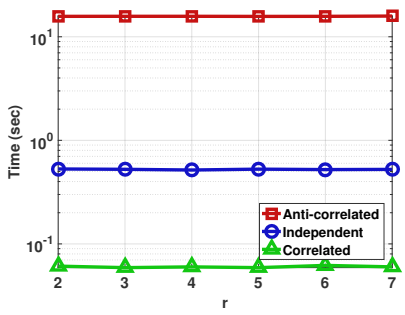


Figure 66: Adopting the k-dominant skyline

of the experiments, e.g., it required several thousands of seconds for $n = 20$K. On the other hand, the other two algorithms, namely HD-GREEDY and HD-RRMS scaled perfectly in all the experiments as their performances were less dependent on the dataset size. Yet among the two algorithms, HD-RRMS algorithm performs better than HD-GREEDY algorithm, both in time and regret-ratio.

The fluctuation in the regret-ratio of the output of the GREEDY algorithm in Figure 49 confirms the fact that it cannot guarantee the output quality. Since HD-GREEDY algorithm deals with the discretized subset of function $F \in \mathcal{F}$ and runs the greedy approach on top of it (while GREEDY applies the greedy manner on the whole set of function, $\mathcal{F}$) one expect that the output quality of the GREEDY algorithm should be better than the HD-GREEDY algorithm. However,

in Figure 49 and part of the Figure 48, the regret-ratio of the output of the HD-GREEDY algorithm is better than that of the GREEDY algorithm. This is due to the starting point selected by the GREEDY algorithm, where it does not select the first point greedily and simply picks the maximum on the first dimension. Since the latter points are selected based on the initial point, a bad selection of it may highly propagate to the final quality of the output. On the other hand, this problem cannot easily get fixed in the GREEDY algorithm; because in order to construct the LPs, the GREEDY algorithm needs a *non-empty* set of so far selected points – i.e. it cannot start the greedy optimization without pre-selecting at least one point! One way to resolve this, is to run the algorithm with all possible choices of initial point, which multiplies the algorithm complexity with n. Applying the set-cover idea, the output of the HD-RRMS algorithm had less regret-ratio than the outputs of the both GREEDY and HD-GREEDY algorithms.

Additionally, the skyline size for each of these experiments is reported in Figure 51. As expected, in anti-correlated dataset, most of the tuples are skyline, while the size is less in the two other datasets. Despite of such differences, however, our HD-RRMS algorithm significantly outperforms the competitors in all these cases.

Impact of the number of attributes (m): In these set of experiments, we varied the number of the attributes (m) from 4 to 10 for all the three synthetic datasets, setting the dataset size, and output size to $n = 10$K and $r = 5$ respectively. Figures 52, 53, and 54 show the results for the correlated, independent, and anti-correlated datasets respectively. As expected for all three datasets the HD-RRMS algorithm outperforms the GREEDY and HD-GREEDY algorithms both in execution time and regret-ratio. While the execution time of the HD-GREEDY algorithm is almost similar to the HD-RRMS algorithm, it outperforms the Greedy algorithm. The regret-ratio of the output of the HD-GREEDY algorithm is better in Figure 53 and part of Figure 54, which is, as explained, due to the bad initial point selected by the GREEDY algorithm.

The skyline sizes are reported in Figure 55. After 6 dimensions, almost all the tuples in anti-correlated database and most of the tuples in the independent

dataset are skyline. Once again, HD-RRMS algorithm significantly outperforms the competitors even when almost all tuples are on the skyline.

Impact of the output size (r): Figures 56, 57, and 58 show the results for correlated, independent, and anti-correlated datasets, where the output size (r) is varied from 2 to 7, and dataset size and number of attributes are set to $n = 10$K and $m = 4$ respectively. As expected, in these set of experiments, also, HD-RRMS algorithm outperforms the HD-GREEDY and GREEDY algorithms in both time and regret-ratio, while the execution time of the HD-GREEDY algorithm is almost similar. The regret-ratio of the output of the HD-GREEDY algorithm is better than the GREEDY algorithm in Figures 56, 57, and part of 58, due to the bad initial point selection in GREEDY algorithm. One interesting fact in these figures is the bad performance of the GREEDY algorithm for the small output sizes (especially 2 and 3). This is due to the fact that the first point was not selected greedily and the other point(s) was selected with maximum (regret-ratio) distance to it, which (together with the initial point) may not be a good selection. However, as the algorithm picks more points, the effect of the initial point reduces.

Impact of the discretization control parameter (γ): In these set of experiments, we studies the effect of the value of γ in the performance of the HD-RRMS and HD-GREEDY algorithms. Figures 59, 60, and 61 show the performance of the algorithms for correlated, independent, and anti-correlated datasets respectively. While increasing the value of γ does not highly affect the performance of the HD-GREEDY algorithm, it highly affects on the performance of the HD-RRMS algorithm. On the other hand, increasing the value of γ increases the number of columns of the discretized matrix by γ^m and directly affects on the running time of the algorithms. Looking at the figures, at least in these experiments, selecting γ between 4 and 6 seems appropriate, as it seems to reach a point of saturation where increasing γ benefits little in terms of execution time.

Real datasets: We also evaluated the performances of the high-dimensional algorithms over the DOT and NBA real datasets. We set the default values of the number of attributes, number of outputs and discretization control parameter to $m = 4$, $r = 5$, and $\gamma = 6$ respectively. We set the default dataset size of NBA and

DOT to $n = 10$K and $n = 100$K respectively.

In the first set of the experiments we varied the dataset size (n) from 100K to 400K for DOT and from 5K to 20K for NBA dataset. Figures 62 and 64 show the performance of the algorithms for the DOT and NBA respectively. Similar to the synthetic experiments, in both experiments the HD-RRMS and HD-GREEDY algorithms run faster in the order of magnitude, while the the GREEDY algorithm does not perform well as the input size (n) increased, e.g., it requires more than $20,000$ seconds for $n = 400$K in DOT. From the regret-ratio point of view the HD-RRMS algorithm outperforms the other two algorithms. The HD-GREEDY algorithm has a better output quality than the GREEDY algorithm for some of the cases.

In the second set of experiments, we varied the number of attributes (m) from 3 to 6. In both experiments, the HD-RRMS algorithm has the best running time, while the HD-GREEDY algorithm performs similarly. While the HD-RRMS algorithm has the best output quality in both experiments, the GREEDY algorithm provides better output quality for NBA and the output of the HD-GREEDY algorithm is better for the DOT dataset.

Adopting the state-of-the-art: In order to test whether one could achieve a reasonable regret-ratio by selecting a subset according to existing techniques that were not designed to optimize the regret-ratio, we implemented a number of existing algorithms for related problems. First, we tested the partitioning-based approximate convex hull method proposed in [63]. As expected, since the goal of this algorithm is efficiently discovering a set which is as similar to the real convex hull as possible, the method is not adoptable here. For example, in these experiments, the size of the discovered set was always larger than the original convex hull, defeating the purpose of generating a more compact representation of the original data.

In another set of experiments, we considered the existing technique of finding the k-dominant skyline [65]. Once again, minimizing the dissatisfaction on maxima query is not its objective - testing it here is solely to demonstrate that it should not be used for the purpose of minimizing regret-ratio. k-dominant skyline is a subset

of the skyline when we relax the definition of dominance to k-dominance (i.e., a tuple t k-dominates a tuple t' if there are $k \leq m$ attributes in which t is better than or equal to t' and is better in at least one of these k attributes). k-dominant skyline is the set of tuples are not k-dominated by any other points in the database.

In order to adopt the k-dominant skyline for finding a set of at most r tuples, we do a binary search over the value of k and at each iteration, if the size of the discovered set is larger than r, we increase the value of k and reduce it otherwise. Somewhat surprisingly, we found a major problem with this adaptation - the chance of returning the empty set is high! To understand it, let us consider the case where $m = 2$ and the tuples are distributes inside a circle. The expected number of convex hull points (a subset of skyline) is $O(n^{\frac{1}{3}})$ [62], meaning that having $k = 2$ would often lead to too large an output. However, if we reduce k from 2 to 1, the output size will most likely become zero because, for every skyline point, there is very likely a point that is better on either x or y! Indeed, we ran an experiment on $n = 10000$ $m = 4$ over the three synthetic datasets - in all experiments, the returned set was empty. Figure 66 shows the running time of the algorithm over each of the datasets.

5.8 Related Work

Top-k discovery algorithms: These algorithms can be divided into on-demand query processing and index construction. On-demand Top-k algorithms focus on the data access methods. For example, *NRA* [56] considers the existence of one sorted list of tuples for each attribute, and finds the Top-k only by exploring the lists, while *TA* [57] applies both random and sorted access. *CA* [57], *Upper/Pick* [78], and [79] are the more advanced algorithms in this category. Besides, methods like *PREFER* [58] and *LPTA* [59], employ the materialized views to increase the efficiency of Top-k discovery process. While the first class of Top-k algorithms focuses on efficiently answering the queries on demand, the other set of works aims toward indexing the data beforehand, such that they can answer future queries fast. For example, *ONION* [60] constructs k layers of convex hull that can serve as the representative for linear ranking functions. [61] adds the notion of robustness

as the set that performs the best in the worst case scenario.

Approximate convex hull and skyline reduction algorithms:
Given the complexity of convex hull discovery algorithms, especially in high-dimensions, designing effective approximate algorithms, with tight approximate-ratios, for finding the convex hull has attracted many researchers. For example, J. L. Bently et. al. [63] propose a FPTAS ϵ-approximate convex hull algorithm for the two-dimensional algorithms and extend it to high-dimensions. Similarly, [64] partitions the space of skyline tuples into several subregions and finds the convex hull in each sub-region, which will result in finding a super-set of convex hull. The objective here is to approximately find a set which is as similar to the original convex hull, not reducing the size of it. On the other hand, a set of work, such as [65, 66], aim toward reducing the skyline size. For example, Chan et. al. [65] relax the notion of domination to "$k-$domination" in order to increase the chance domination and reduce the skyline size. However, their objective in ranking the skyline tuples is not minimizing the user dissatisfaction on maxima queries.

Regret-ratio minimizing problem: The authors in [68, 69] focused on regret minimization of a database to support multi-criteria decision making. Regret-ratio and Regret-ratio minimizing problem were first introduced in [68] in order to minimize the maximum user dissatisfaction of a Top-k query. They proposed the so called CUBE algorithm to provide an upper-bound guarantee for the regret-ratio of the optimal solution and more importantly they showed it is independent of the input size (n). They also provided the GREEDY heuristic for the problem that runs $O(nr)$ linear programs and picks the points greedily. Chester et.al. [69] extended the regret-ratio notion to k-regret ratio which measures how far from a kth "best" tuple is the "best" tuple in a subset. They propose k-regret minimizing sets problem and proved that it is NP-hard in a high-dimensional database. They also proposed the quadratic Sweeping-Line algorithm for the two-dimensional scenario. Kessler et. al. [80] extended the k-regret minimizing notion to nonlinear functions. The focus of this paper is designing efficient algorithms for the regret-ratio minimizing problem proposed in [68]. In two-dimensional case, compared to quadratic existing Sweeping-Line, we propose the linearithmic dynamic programming algorithm

2D-RRMS, while for the high-dimensional case we provide the linearithmic HD-RRMS that guarantees a user controllable distance from the optimal solution.

5.9 Final Remarks

In this chapter, we made several fundamental theoretical as well as practical advances in developing a compact maxima representative. We studied both two-dimensional as well as high-dimensional databases to find a set limited to only r tuples that minimizes the maximum regret-ratio. In the case of two-dimensional databases, we have developed an innovative dynamic programming algorithm to build the optimal index that runs in linearithmic time. We developed an innovative linearithmic algorithm that guarantees a regret ratio that is within a user-controllable distance from the optimal regret ratio. Our comprehensive set of experiments on synthetic (with different correlation models) and real datasets of size up to several million records confirm the efficiency, scalability, and effectiveness of our algorithms.

6 Query Reranking As A Service

Motivation: The ranked retrieval model has rapidly replaced the traditional Boolean retrieval model as the de facto way for query processing in client-server (e.g., web) databases. Unlike the Boolean retrieval model which returns all tuples matching the search query selection condition, the ranked retrieval model orders the matching tuples according to an often proprietary ranking function, and returns the top-k tuples matching the selection condition (with possible page-turn support for retrieving additional tuples).

The ranked retrieval model naturally fits the usage patterns of client-server databases. For example, the short attention span of clients such as web users demands the most desirable tuples to be returned first. In addition, to achieve a short response time (e.g., for web databases), it is essential to limit the length of returned results to a small value such as k. Nonetheless, the ranked retrieval model also places more responsibilities on the web database designer, as the *ranking function* design now becomes a critical feature that must properly capture the need of database users.

In an ideal scenario, the database users would have fairly homogeneous preferences on the returned tuples (e.g., newer over older product models, cheaper over more expensive goods), so that the database owner can provide a small number of ranking functions from which the database users can choose to fulfill their individual needs. Indeed, the database community has developed many ranking function designs and techniques for the efficient retrieval of top-k query answers according to a given ranking function.

The practical situation, however, is often much more complex. Different users often have diverse and sometimes contradicting preferences on numerous factors. Even more importantly, many database owners simply lack the expertise, resources, or even motivation (e.g., in the case of government web databases created for policy or legal compliance purposes) to properly study the requirements of their users and design the most effective ranking functions. For example, many flight-search websites, including Kyak, Google Flights, Sky Scanner, Expedia, and Priceline offer limited ranking options on a subset of the attributes, that, for example, does

not help ranking based on cost per mileage. Similar limitations apply to the websites such as Yahoo! Autos (resp. Blue Nile), if we want to rank the results, for example, based on mileage per year (resp. summation of depth and table percent). As a result, there is often a significant gap, in terms of both design and diversity, between the ranking function(s) supported by the client-server database and the true preferences of the database users. The objective of this paper is to define and study the *query re-ranking* problem which bridges this gap for real-world client-server databases.

Query Re-Ranking: Given the challenge for a real-world database owner to provide a comprehensive coverage of user-preferred ranking functions, in this paper we develop a *third-party query re-ranking service* which uses nothing but the public search interface of a client-server database to enable the on-the-fly processing of queries with user-specified ranking functions (with or without selection conditions), no matter if the ranking function is supported by the database or not.

This query re-ranking service can enable a wide range of interesting applications. For example, one may build a personalized ranking application using this service, offering users with the ability to remember their preferences across multiple web databases (e.g., multiple car dealers) and apply the same personalized ranking over all of them despite the lack of such support by these web databases. As another example, one may use the re-ranking service to build a dedicated application for users with disabilities, special needs, etc., to enjoy appropriate ranking over databases that do not specifically tailor to their needs.

There are two critical requirements for a solution to the query re-ranking service: First, the output query answer must precisely follow the user-specified ranking function, i.e., there is no loss of accuracy and the query re-ranking service is transparent to the end user as far as query answers are concerned. Second, the query re-ranking service must minimize the number of queries it issues to the client-server database in order to answer a user-specified query. This requirement is crucial for two reasons: First is to ensure a fast response time to the user query, given that queries to the client-server database must be issued on the fly. Second is to reduce the burden on the client-server database, as many real-world ones,

especially web databases, enforce stringent rate limits on queries from the same IP address or API user (e.g., Google Flight Search API allows only 50 free queries per user per day).

Problem Novelty: While extensive studies have focused on translating an unsupported query to multiple search queries supported by a database, there has not been research on the *translation of ranking* requirements of queries. Related to our problem here includes the existing studies on crawling client-server databases [36], as a baseline solution for query re-ranking is to first crawl all tuples from the client-server database, and then process the user query and ranking function locally. The problem, however, is the high query cost. As proved in [36], the number of queries that have to be issued to the client-server database for crawling ranges from at least linear to the database size in the best-case scenario to quadratic and higher in worse cases. As such, it is often prohibitively expensive to apply this baseline to real-world client-server databases, especially those large-scale web databases that constantly change over time.

Another seemingly simple solution is for the third-party service to retrieve more than k tuples matching the user query, say $h \cdot k$ tuples by using the "page-down" feature provided by a client-server database (or [81] when such a feature is unavailable), and then locally re-rank the $h \cdot k$ tuples according to the user-specified ranking function and return the top-k ones. There are two problems with this solution. First, since many client-server databases choose not to publish the design of their proprietary ranking functions (e.g., simply naming it "rank by popularity" in web databases), results returned by this approach will have unknown error unless all tuples satisfying the user query are crawled. Second, when the database ranking function differs significantly from the user-specified one, this approach may have to issue many page-downs (i.e., a large h) in order to retrieve the real top-k answers according to the user-specified ranking function.

Finally, note that our problem stands in sharp contrast with existing studies on processing top-k queries over traditional databases using pre-built indices and/or materialized views (e.g., [58, 60]). The key difference here is the underlying *data access model*: Unlike prior work which assume complete access to data, we are

facing a restricted, top-k, search interface provided by the database.

Outline of Technical Results: We start by considering a simple instance of the problem, where the user-desired ranking function is on a single attribute, and developing Algorithm 1D-RERANK to solve it. Note that this special, 1D, case not only helps with explaining the key technical challenges of query reranking, but also can be surprisingly useful for real-world web databases. For example, a need often arising in flight search is to maximize or minimize the layover time, so as to either add a free stopover for a sightseeing day trip or to minimize the amount of agonizing time spent at an airport. Unfortunately, while flight search websites like Kayak offer the ability to specify a range query on layover time, it does not support ranking according to the attribute. The 1D-RERANK algorithm handily addresses this need by enabling a "Get-Next" primitive - i.e., upon given a user query q, an attribute A_i, and the top-h tuples satisfying q according to A_i, it finds the "next", i.e., $(h+1)$-th ranked, tuple.

In the development of 1D-RERANK, we rigidly prove that, in the worst-case scenario, retrieving even just the top-1 tuple requires crawling of the entire database. Nonetheless, we also show that the practical query cost tends to be much smaller. Specifically, we found a key factor (negatively) affecting query cost to be what we refer to as "dense regions" - i.e., a large number of tuples clustering together within a small interval (on the attribute under consideration). The fact that a dense region may be queried again and again (by the third-party query reranker) for the processing of different user queries prompts us to propose an *on-the-fly indexing* idea that detects such dense regions and proactively crawls top-ranked tuples in it to avoid the waste on processing future user queries. We demonstrate theoretically and experimentally the effectiveness of such an index on reducing the overall query cost.

To solve the general problem of query reranking for any arbitrary user-desired ranking function (rather than just 1D), a seemingly simple solution is to directly apply a classic top-k query processing algorithm that leverages sorted access to each attribute, e.g., Fagin's or TA algorithm [56], by calling the "Get-Next" primitive provided by 1D-RERANK as a subroutine. The problem with this simple

solution, however, is that it incurs a significant waste of queries when applied to client-server databases, mainly because it fails to leverage the multi-predicate (conjunctive) queries supported by the underlying database. We demonstrate in the paper that this problem is particularly significant when a large number of tuples satisfying a user query feature extreme values on one or more attributes.

To address the issue, we develop MD-RERANK (i.e., Multi-Dimensional Rerank), a query re-ranking algorithm that identifies a small number of multi-predicate queries to directly retrieve the top-k tuples according to a user query. We note a key difference between the 1D and MD cases: In the 1D case, a single query is enough to cover the subspace outranking a given tuple, while the MD case requires a much larger number of queries due to the more complex shape of the subspace. We develop two main ideas, namely *direct domination detection* and *virtual tuple pruning*, to significantly reduce the query cost for MD-RERANK. In addition, like in the 1D case, we observe the high query cost incurred by "dense regions", and include in MD-RERANK our on-the-fly indexing idea to reduce the amortized cost of query re-ranking.

Our contributions also include a comprehensive set of experiments on real-world web databases, both in an offline setting (for having the freedom to control the database settings) and through online *live* experiments over real-world web databases. Specifically, we constructed a Top-k web search interface in the offline experiment, and evaluated the performance of the algorithms in different situations, by varying the parameters such as database size, system-k, and system ranking function. In addition we also tested our algorithms live online over two popular websites, namely Yahoo! Autos and and Blue Nile, the largest diamond online retailer. The experiment results verify the effectiveness of our proposed techniques and their superiority over the baseline competitors.

6.1 Preliminaries

6.1.1 Database Model

Database: Consider a client-server database D with n tuples over m ordinal attributes $A_1, \ldots, A_m$. Let the value domain of A_i be $V(A_i) = \{v_{i1}, \ldots, v_{i|V(A_i)|}\}$. The database may also have other categorical attributes $B_1, \ldots, B_{m'}$. But since they are usually not part of any ranking function, they are not the focus of our attention for the purpose of this paper. We assume each tuple t to have a none-NULL value on each (ordinal) attribute A_i, which we refer to as $t[A_i]$ ($t[A_i] \in V(A_i)$). Note that if NULL values do exist in the database, the ranking function usually substitutes it with another default value (e.g., the mean or extreme value of an attribute). In that case, we simply consider the occurrence of NULL as the substituted value. In most part of the paper, we make the general positioning assumption [82], before introducing a simple post-processing step that removes this assumption in § 6.4.

Query Interface: Most client-server database allow users to issue certain "simplistic" search queries. Often these queries are limited to conjunctive ones with predicates on one or a few attributes. Examples here include web databases, which usually allows such conjunctive queries to be specified through a form-like web search interface. Formally, we consider search queries of the form

$$q\text{: SELECT * FROM D WHERE } A_{i_1} \in (v_{i_1}, v'_{i_1}) \text{ AND } \cdots \text{ AND } A_{i_p} \in (v_{i_p}, v'_{i_p}) \text{ AND } \textit{conjunctive predicates on } B_1, \ldots, B_{m'},$$

where $\{A_{i_1}, \ldots, A_{i_p}\} \subseteq \{A_1, \ldots, A_m\}$ is a subset of ordinal attributes, and $(v_{i_j}, v'_{i_j}) \subseteq V(A_{i_j})$ is a range within the value domain of A_{i_j}.

A subtle issue here is that our definition of q only includes open ranges (x, y), i.e., $x < A_i < y$, while real-world client-server databases may offer close ranges $[x, y]$, i.e., $x \leq A_i \leq y$, or a combination of both (e.g., $(x, y]$). We note that these minor variations do not affect the studies in this paper, because it is easy to derive the answer to q even when only close ranges are allowed by database: One simple needs to find a value arbitrarily close to the limits, say $x + \epsilon$ and $y - \epsilon$ with an arbitrarily small $\epsilon > 0$, and substitute (x, y) with $[x + \epsilon, y - \epsilon]$. In the case where

the value domains are discrete, substitutions can be made to the closest discrete value in the domain.

As discussed in § **??**, once a client-server database receives query q from a user, it often limits the number of returned tuples to a small value k. Without causing ambiguity, we use q to refer to the set of tuples *actually returned* by q, $R(q)$ to refer to the the set of tuples matching q (which can be a proper superset of the returned tuples q when there are more than k returning tuples, and $|R(q)|$ to refer to the number of tuples matching q. When $|R(q)| > k$, we say that q *overflows* because only k tuples can be returned. Otherwise, if $|R(q)| \in [1, k]$, we say that q returns a *valid* answer. At the other extreme, we say that q *underflows* when it returns empty, i.e., $|R(q)| = 0$.

System Ranking Function: In most parts of the paper, we make a conservative assumption that, when $|R(q)| > k$, the database selects the k returned tuples from $R(q)$ according to a proprietary *system ranking function* unbeknown to the query reranking service. That is, we make *no* assumption about the system ranking function whatsoever. In § 6.4, we also consider cases where the database offers more ranking options, e.g., ORDER BY according to a subset of ordinal attributes.

6.1.2 Problem Definition

The objective of this paper is to enable a *third-party query reranking service* which enables a user-specified ranking function for a user-specified query q, when the query q is supported by the underlying client-server database but the ranking function is *not*.

User-Specified Ranking Functions: We allow a user of the query reranking service to specify a *user-specified ranking function* $\mathcal{S}(q, t)$ which takes as input the user query q and one or more ordinal attributes (i.e., $A_1, \ldots, A_m$) of a tuple t, and outputs the ranking score for t in processing q. The *smaller* the score $\mathcal{S}(q, t)$ is, the *higher ranked* t will be in the query answer, i.e., the more likely t is included in the query answer when $R(q) > k$. Without causing ambiguity, we also represent $\mathcal{S}(q, t)$ as $\mathcal{S}(t)$ when the context (i.e., the user query being processed) is clear.

We support a wide variety of user-specified ranking functions with only one requirement: *monotonicity*. Given a user query q, a ranking function $\mathcal{S}(t)$ is monotonic if and only if there exists an order of values for each attribute domain, which we represent as $\prec$ with $v_1 \prec v_2$ indicating v_1 being higher-ranked than v_2, such that there does not exist two possible tuple values t_1 and t_2 with $\mathcal{S}(t_1) < \mathcal{S}(t_2)$ yet $t_2[A_i] \prec t_1[A_i]$ for all $i \in [1, m]$.

Intuitively, the definition states that if t_1 outranks t_2 according to $\mathcal{S}(\cdot)$, then t_1 has to outrank t_2 on at least one attribute according to the order $\prec$. In other words, t_1 cannot outrank t_2 if it is dominated [4] by t_2. Another interesting note here is that we do *not* require all user-specified ranking functions to follow the same attribute-value order $\prec$. For example, one ranking function may prefer higher prices while the other prefers lower prices. We support both ranking functions so long as each is monotonic according to its own order of attribute values.

Performance Measure: To enable query reranking, we have to issue a number of queries to the underlying client-server database. It is important to understand that the most important efficiency factor here is *the total number of queries* issued to the database, not the computational time. The rational behind it is that almost many client-server databases, e.g., almost all client-server databases, enforce certain *query-rate limit* by allowing only a limited number of queries per day from each IP address, API account, etc.

Problem Definition: In this paper, we consider the problem of query reranking in a "Get-Next", i.e., incremental processing, fashion. That is, for a given user query q, a user-specified ranking function $\mathcal{S}$, and the top-h tuples satisfying q according to $\mathcal{S}$, we aim to find the No. $(h + 1)$ tuple. When $h = 0$, this means finding the top-1 for given q and $\mathcal{S}$. One can see that finding the top-h tuples for q and $\mathcal{S}$ can be easily solved by repeatedly calling the Get-Next function. The reason why we define the problem in this fashion is to address the real-world scenario where a user first retrieves the top-h answers and, if still unsatisfied with the returned tuples, proceeds to ask for the No. $(h + 1)$. By supporting incremental processing, we can progressively return top answers while paying only the incremental cost.

QUERY RERANKING PROBLEM: Consider a client-server database D with a top-k interface and an arbitrary, unknown, system ranking function. Given a user query q, a user-specified monotonic ranking function $\mathcal{S}$, and the top-h ($h \geq 0$ can be greater than, equal to, or smaller than k) tuples satisfying q according to $\mathcal{S}$, discover the No. $(h+1)$ tuple for q while minimizing the number of queries issued to the client-server database D.

6.2 1D-RERANK

We start by considering the simple 1D version of the query reranking problem which, as discussed in the introduction, can also be surprisingly useful in practice. Specifically, for a given attribute A_i, a user query q, and the h tuples having the minimum values of A_i among $R(q)$ (i.e., tuples satisfying q), our goal here is to find tuple $t(q, A_i, h+1)$, which satisfies q and has the $(h+1)$-th smallest value on A_i among $R(q)$, while minimizing the number of queries issued to the underlying database.

6.2.1 Baseline Solution and Its Problem

1D-BASELINE

Baseline Design: Since our focus here is to discover $t(q, A_i, h+1)$ given q, A_i and h, without causing ambiguity, we use t_{h+1} as a short-hand representation of $t(q, A_i, h+1)$. A baseline solution for finding t_{h+1} is to start with issuing to the underlying database query q_1: SELECT * FROM D WHERE $A_i > t_h[A_i]$ AND $Sel(q)$, where $Sel(q)$ represents all selection conditions specified in q. If $h = 0$, this query simply becomes SELECT * FROM D WHERE $Sel(q)$.

Note that the answer to q_1 must return non-empty, because otherwise it means there are only h tuples matching q. Let a_1 be the one having minimum A_i among all returned tuples. Given a_1, the next query we issue is q_2: WHERE $A_i \in (t_h[A_i], a_1[A_i])$ AND $Sel(q)$. In other words, we narrow the search region on A_i to "push the envelop" and discover any tuple with even "better" A_i than what we have seen so far.

If q_2 returns empty, then $t_{h+1} = a_1$. Otherwise, we can construct and issue q_3, q_4, . . ., in a similar fashion. More generally, given a_j being the tuple with minimum A_i returned by q_j, the next query we issue is q_{j+1}: WHERE $A_i \in (t_h[A_i], a_j[A_i])$ AND $Sel(q)$. We stop when q_{j+1} returns empty, at which time we conclude $t_{h+1} = a_j$. Algorithm 14, 1D-BASELINE, depicts the pseudo-code of this baseline solution.

Leveraging History: An implementation issue worth noting for 1D-BASELINE is how to leverage the historic query answers we have already received from the underlying client-server database. This applies not only during the processing of a user query, but also across the processing of different user queries.

During the process of user query q, for example, we do not have to start with the range of $A_i \in (t_h[A_i], \infty)$ as stated in the basic algorithm design. Instead, if we have already "seen" tuples in $R(q)$ that have $A_i > t_h[A_i]$ in the historic query answers, then we can first identify such a tuple with the minimum A_i, denoted by t', and then start the searching process with $A_i \in (t_h[A_i], t')$, a much smaller region that can yield significant query savings, as shown in the query cost analysis below.

More generally, this exact idea applies across the processing of different user queries. What we can do is to inspect every tuple we have observed in historic query answers, identify those that match the user query being processed, and order these matching tuples according to the attribute A_i under consideration. By doing so, the more queries we have processed, the more likely we can prune the search space for t_{h+1} based on historic query answers, and thereby reduce the query cost for re-ranking.

Negative Result: Lower Bound on Worst-Case Query Cost

While simple, 1D-BASELINE has a major problem on query cost, as it depends on the correlation between A_i and the system ranking function which we know nothing about and has no control over. For example, if the system ranking function is exactly according to A_i, then the query cost of finding t_{h+1} is 2: q_1 returns t_{h+1} and q_2 returns empty to confirm that t_{h+1} is indeed the "next" tuple. On the other hand, if the system ranking function is the exact opposite to A_i (i.e., returning

Algorithm 14 1D-BASELINE

1: t_{h+1} = argmin$_{t[A_i]}\{t \in$ history $\mid t[A_i] > t_h[A_i]\}$
2: T = Top-k(WHERE $t_{h+1}[A_i] > A_i > t_h[A_i]$ AND $Sel(q)$)
3: **while** T is overflow **do**
4: $\quad t_{h+1}$ = argmin$_{t[A_i]}\{t \in T\}$
5: $\quad T$ = Top-k(WHERE $t_{h+1}[A_i] > A_i > t_h[A_i]$ AND $Sel(q)$)
6: **end while**
7: **return** t_{h+1}

tuples with maximal A_i first), then the query cost for the baseline solution is exactly $|R(q)| + 1$ in the worst-case scenario (when $k = 1$), because every tuple satisfying q will be returned before t_{h+1} is revealed at the end. Granted, this cost can be "amortized" thanks to the leveraging-history idea discussed above, because the $|R(q)| + 1$ queries indeed reveal not just the top-$(h + 1)$ but the complete ranking of all tuples matching q. Nonetheless, the query cost is still prohibitively high when q matches a large number of tuples.

While it might be tempting to try to "adapt to" such ill-conditioned system ranking functions, the following theorem actually shows that the problem is not fixable in the worst-case sense. Specifically, there is a lower bound of n/k on the query cost required for query reranking given the worst-case data distribution and worst-case system ranking function.

Theorem 10. $\forall n > 1$, *there exists a database of* n *tuples such that finding the top-ranked tuple on an attribute through a top-*k *search interface requires at least* n/k *queries that retrieve all the* n *tuples.*

Proof. Without loss of generality, consider a database with only one attribute A and an unknown ranking function. Let (v_0, v_∞) be the domain of A. Note that this means (1) the query re-ranking algorithm can only issue queries of the form SELECT * FROM D WHERE $A \in (v_1, v_2)$, where $v_0 \leq v_1 < v_2 \leq v_\infty$, (2) the returned tuples will be ranked in an arbitrary order, and (3) the objective of the query re-ranking algorithm is to find the tuple with the smallest A.

For any given query re-ranking algorithm $\mathcal{R}$, consider the following query processing mechanism $\mathcal{Q}$ for the database: During the processing of all queries,

we maintain a min-query-threshold v_q with initial value v_∞. If a query q issued by $\mathcal{R}$ has lower bound *not* equal to v_0, i.e., q: WHERE $A \in (v_1, v_2)$ with $v_1 > v_0$, $\mathcal{Q}$ returns whatever tuples already returned in historic query answers that fall into range (v_1, v_2). It also sets $v_q = \min(v_q, v_1)$.

Otherwise, if q is of the form WHERE $A \in (v_0, v_2)$ with $v_2 > v_0$, then $\mathcal{Q}$ returns an overflowing answer with k tuples. These k tuples include those in the historic query answers that fall into (v_0, v_2). If more than k such tuples exist in the history, we choose an arbitrary size-k subset. If fewer than k such tuples exist, we fill up the remaining slots with arbitrary values in range $((v_0 + v_q)/2, v_q)$[14]. We also set v_q to be $(v_0 + v_q)/2$.

There are two critical observations here. First is that for any query sequence $q_1, \ldots, q_h$ with $h \leq n/k$, we can always construct a database D of at most n tuples, such that the query answers generated by $\mathcal{Q}$ are consistent with what D produces. Specifically, D would simply be the union of all tuples returned. Note that our maintenance of v_q ensures the consistency.

The second critical observation is that no query re-ranking algorithm $\mathcal{R}$ can find the tuple with the smallest A without issuing at least n/k queries. The reason is simple: since $n/k - 1$ queries cannot reveal all n tuples, we can add a tuple t with $A = (v_0 + v_q)/2$ to the database, where v_q is its value after processing all $n/k - 1$ queries. One can see that the answers to all $n/k - 1$ queries can remain the same. As such, for any $n > 1$, there exists a database containing n tuples such that finding the top-ranked one for an attribute requires at least n/k queries, which according to [36] is sufficient for crawling the entire database in a 1D space. □

6.2.2 1D-RERANK

Given the above result, we have to shift our attention to reducing the cost of finding t_{h+1} in an average-case scenario, e.g., when the tuples are more or less uniformly distributed on A_i (instead of forming a highly skewed distribution as constructed in the proof of Theorem 10). To this end, we start this subsection by considering

[14]Note that any factor here (besides 2) works too. So in general the range can be $((v_0 + v_q) \cdot \alpha, v_q)$ so long as $\alpha > 0$.

a binary-search algorithm. After pointing out the deficiency of this algorithm when facing certain system ranking functions, we introduce our idea of on-the-fly indexing for the design of 1D-RERANK, our final algorithm for query reranking with a single-attribute user-specified ranking function.

1D-BINARY and its Problem

The binary search algorithm departs from 1D-BASELINE on the construction of q_2: Given a_1, instead of issuing q_2: WHERE $A_i \in (t_h[A_i], a_1[A_i])$ AND $Sel(q)$, we issue here

$$q_2' : \text{ WHERE } A_i \in (t_h[A_i], (a_1[A_i] + t_h[A_i])/2) \text{ AND } Sel(q).$$

This query has two possible outcomes: If it returns non-empty, we consider the returned tuple with minimum A_i, say a_2, and construct q_3' according to a_2. The other possible outcome is for q_2' to return empty. In this case, we issue q_2'': WHERE $A_i \in [(a_1[A_i] + t_h[A_i])/2, a_1[A_i])$ AND $Sel(q)$, which has to return non-empty as otherwise $t_{h+1} = a_1$. In either case, the *search space* (i.e., the range in which t_{h+1} must reside) is reduced by at least half. Algorithm 15, 1D-BINARY, depicts the pseudocode.

Algorithm 15 1D-BINARY

1: t_{h+1} = argmin$_{t[A_i]}\{t \in$History $\mid t[A_i] > t_h[A_i]\}$
2: **repeat**
3: $q\prime =$ WHERE $A_i \in (t_h[A_i], (t_{h+1}[A_i] + t_h[A_i])/2)$ AND $Sel(q)$
4: T = Top-k($q\prime$)
5: **if** T is underflow **then**
6: $q\prime$ = WHERE $A_i \in [(t_{h+1}[A_i] + t_h[A_i])/2, t_{h+1}[A_i])$ AND $Sel(q)$
7: T = Top-k($q\prime$)
8: **end if**
9: **if** T is not underflow **then**
10: t_{h+1} = argmin$_{t[A_i]}\{t \in T\}$
11: **end if**
12: **until** T is not overflow
13: **return** t_{h+1}

Query Cost Analysis: While the design of 1D-BINARY is simple, the query-cost analysis of it yields an interesting observation which motivates the indexing-based design of our final 1D-RERANK algorithm. Let

$$\epsilon_k = t_{h+k+1}[A_i] - t_{h+1}[A_i]. \tag{37}$$

An important observation here is that the execution of 1D-BINARY must conclude when the search space is reduced to width smaller than ϵ_k, because no such range can cover $t_{h+1}[A_i]$ while matching more than k tuples. Thus, the worst-case query cost of 1D-BINARY is

$$O(\min(\log_2(|V(q, A_i)|/\epsilon_k), |R(q)|/k)), \tag{38}$$

where $|V(q, A_i)|$ is the range of A_i among tuples satisfying q - i.e., $\max_{t \in R(q)} t[A_i] - \min_{t \in R(q)} t[A_i]$. Note that the second input to the $\min$ function in (38) is because every pair of queries issued by 1D-BINARY, i.e., q'_j and q''_j, must return at least k tuples never seen before that satisfies q.

The query-cost bound in (38) illustrates both the effectiveness and the potential problem of Algorithm 1D-BINARY. On one hand, one can see that 1D-BINARY performs well when the tuples matching q are uniformly distributed on A_i, because in this case the expected value of ϵ_k becomes $k \cdot |V(q, A_i)|/|R(q)|$, leading to a query cost of $O(\log_2(|R(q)|/k))$.

On the other hand, 1D-BINARY still incurs a high query cost (as bad as $\Omega(|R(q)|/k)$, just as indicated by Theorem 10) when two conditions are satisfied: (1) the system ranking function is ill-conditioned, i.e., negatively correlated with A_i, and (2) Within $R(q)$ there are *densely clustered* tuples with extremely close values on A_i, leading to a small ϵ_k. Unfortunately, once the two conditions are met, the high query cost 1D-BINARY is likely to be incurred again and again for different user queries q, leading to an expensive reranking service. It is this observation which motivates our index-based reranking idea discussed next.

Algorithm 1D-RERANK: On-The-Fly Indexing

Oracle-based Design: According to the above observation, densely clustered

tuples cause a high query cost of 1D-BINARY. To address the issue, we start by considering an ideal scenario where there exists an oracle which identifies these "dense regions" and reveals the tuple with minimum A_i in these regions without costing us any query. Of course, no such oracle exists in practice. Nevertheless, what we shall do here is to analyze the query cost of 1D-BINARY given such an oracle, and then show how this oracle can be "simulated" with a low-cost on-the-fly indexing technique.

Specifically, for any given region $[x, y] \in V(A_i)$, we call it a *dense region* if and only if it covers at least s tuples *and* $y - x < |V(A_i)| \cdot (s/n)/c$, where c and s are parameters. In other words, the density of tuples in $[x, y]$ is more than c times higher than the uniform distribution (which yields an expected value of $E(y - x) = |V(A_i)| \cdot (s/n)$). The setting of c and s is a subtle issue which we specifically address at the end of this subsection. Given the definition of dense region, the oracle functions as follows: Upon given a user query q, an attribute A_i, and a range $[x, y] \subseteq V(A_i)$ as input, the oracle either returns empty if $[x, y]$ is not dense, or a tuple t which (1) satisfies q, (2) has $A_i \in [x, y]$, and (3) features the smallest A_i among all tuples satisfying (1) and (2).

With the existence of this oracle, we introduce a small yet critical revision to 1D-BINARY, by *terminating* binary search whenever the width of the search space becomes narrower than the threshold for dense region, i.e., $\epsilon_k < |V(A_i)| \cdot (s/n)/c$. Then, we call the oracle with the remaining search space as input. Note that doing so may lead to two possible returns from the oracle:

One is when the region is indeed dense. In this case, the oracle will directly return us t_{h+1} with zero cost. The other possible outcome is an empty return, indicating that the region is not really dense, instead containing more than k (otherwise 1D-BINARY would have already terminated) but fewer than s tuples. Note that this is not a bad outcome either, because it means that by following the baseline technique (1D-BASELINE) on the remaining search space, we can always find t_{h+1} within $O(s/k)$ queries.

Algorithm 16 depicts the pseudocode of 1D-RERANK, the revised algorithm. The following theorem shows its query cost, which follows directly from the above

discussions.

Algorithm 16 1D-RERANK

1: t_{h+1} = argmin$_{t[A_i]}\{t \in$History $\mid t[A_i] > t_h[A_i]\}$
2: **while** $(t_{h+1}[A_i] - t_h[A_i]) < |V(A_i)| \cdot (s/n)/c$ **do**
3: $q\prime$ = WHERE $A_i \in (t_h[A_i], (t_{h+1}[A_i] + t_h[A_i])/2)$ AND $Sel(q)$
4: T = Top-k($q\prime$)
5: **if** T is underflow **then**
6: $q\prime$ = WHERE $A_i \in [(t_{h+1}[A_i] + t_h[A_i])/2, t_{h+1}[A_i])$ AND $Sel(q)$
7: T = Top-k($q\prime$)
8: **end if**
9: **if** T is not underflow **then**
10: t_{h+1} = argmin$_{t[A_i]}\{t \in T\}$
11: **end if**
12: **if** T is valid **then break**
13: **end while**
14: **if** T is valid **then**
15: look up t_{h+1} at ORACLE(A_i,$(t_h[A_i], t_{h+1}[A_i])$,q)
16: **end if**
17: **return** t_{h+1}

Theorem 11. *The query cost of 1D-RERANK, with the presence of the oracle, is* $O(\log(c \cdot n/s) + s/k)$.

Proof. The query cost of of 1D-RERANK, with the presence of the oracle, is the summation of the following costs:

- c_1: the query cost of following 1D-BINARY, until the search space becomes narrower than the dense region threshold,
- c_2: the query cost of discovering t_{h+1} in the remaining region, using the oracle.

Following 1D-BINARY takes $O(\log_2(|V(q, A_i)|/\epsilon_k))$ queries. Because $\epsilon_k < |V(A_i)| \cdot (s/n)/c$, c_1 is in the order of $O(\log(c \cdot n/s))$. As discussed previously, if the oracle does not include the remaining region, the region is not dense and contains fewer than s tuples. Then, following 1D-BASELNE, at most s/k queries are requires to discover t_{h+1}, i.e. c_2 is $O(s/k)$. Consequently, the query cost of 1D-RERANK, with the presence of the oracle, is $O(\log(c \cdot n/s) + s/k)$. □

On-The-Fly Indexing: Our idea for simulating the oracle is simple: once 1D-RERANK decides to call the oracle with a range (x, y), we invoke the 1D-BASELINE algorithm on SELECT * FROM D WHERE $A_i \in (x, y)$ to find the tuple t with smallest A_i in the range. If t satisfies the user query q being processed, then we can stop and output t. Otherwise, we call 1D-BASELINE on WHERE $A_i \in (t[A_i], y)$ to find the No. 2 tuple, and repeat this process until finding one that satisfies q. All tuples discovered during the process are then added into the "dense index" that is maintained throughout the processing of all user queries.

Algorithm 17 depicts the on-the-fly index building process. Note that the index we maintain is essentially a set of 3-tuples

$$\langle A_i, (x, y), D(A_i, x, y))\rangle, \tag{39}$$

where A_i is an attribute, (x, y) is a range in $V(A_i)$ (non-overlapping with other indexed ranges of A_i), and $D(A_i, x, y)$ contains all (top-ranked) tuples we have discovered that have $A_i \in (x, y)$.

Algorithm 17 ORACLE

if ORACLE(A_i,x, y) exists **then**
  return argmin$_{t[A_i]}\{t \in D(A_i, x, y))|$ t matches $Sel(q)\}$
end if
t=1D-BASELINE(WHERE $A_i \in (x, y)$)
add t to $D(A_i, x, y)$
while t does not satisfy $Sel(q)$ **do**
  t=1D-BASELINE(WHERE $A_i \in (t[A_i], y)$)
  add t to $D(A_i, x, y)$
end while
return t

Note that this simulation does differ a bit from the ideal oracle. Specifically, it does not really determine if the region is dense or not. Even if the region is not dense, this simulated oracle still outputs the correct tuple. What we would like to note, however, is that this difference has no implication whatsoever on the query cost of 1D-RERANK. Specifically, what happens here is simply that the on-the-fly

indexing process pre-issues the queries 1D-RERANK is supposed to issue when the oracle returns empty. The overall query cost remains exactly the same.

Another noteworthy design in on-the-fly indexing is the call of 1D-BASELINE on SELECT * FROM D WHERE $A_i \in (x, y)$, a query that does not "inherit" the selection conditions in the user query q being processed. This might appear like a waste as 1D-BASELINE could issue fewer queries with a narrower input query. Nonetheless, we note that rationale here is that a dense region might be covered by multiple user queries repeatedly. By keeping the index construction generic to all user queries, we reduce the amortized cost of indexing as the dense index can make future reranking processes more efficient.

Parameter Settings: To properly set the two parameters for dense index, c and s, we need to consider not only the query cost derived in Theorem 11, but also the cost for building the index, which is considered in the following theorem:

Theorem 12. *The total query cost incurred by on-the-fly indexing (for processing all user queries) is at most*

$$\sum_{h=1}^{n-s-1} c(h) \tag{40}$$

where $c(h) = 1$ *if there exists* $j \in [h - s, h]$*, such that*

$$t(*, A_i, j + s + 1)[A_i] - t(*, A_i, j)[A_i] < \frac{s \cdot |V(A_i)|}{c \cdot n}, \tag{41}$$

and 0 otherwise. Here $t(*, A_i, j)$ *refers to the* j*-th ranked tuple according to* A_i *in the entire database.*

Proof. The discovery of every tuple in the dense region takes at most the amortized cost of one query. That is because 1D-BASELINE assures the discovery on k unseen tuples by every non-underflowing query, i.e. every tuple in the dense region is discovered by one and only one query. Thus the query cost is at most equal to the number of tuples in the dense regions. Each tuple t is in the dense region with regard to the dimension A_i, if, sorting the tuples on A_i, we can construct a window

containing t, with size less than the dense region threshold, that has at least s tuples. Suppose t is ranked h-th based on A_i. Equation 41 checks the existence of such a window around it. The total cost thus, is at most the number of the tuples for which this equation is true. This is reflected in Equation 40. □

One can see from the above theorem and Theorem 11 how c and s impacts the query cost: the larger c is, the fewer dense regions there will be, leading to a lower indexing cost. On the other hand, the per-query reranking cost increases at the log scale with c. Similarly, the larger s is, the fewer dense regions there will be (because a larger s reduces the variance of tuple density), while the per-query reranking cost increases linearly with s. Given the different rate of increase for the per-query reranking cost with c and s, we should set c to be a larger value to leverage its log-scale effect, while keep s small to maintain an efficient reranking process.

Specifically, we set $c = n$ and $s = k \cdot \log n$. One can see that the per-query reranking cost of 1D-RERANK in this case is $O(\log n)$. While the indexing cost depends on the specific data distribution (after all, we are bounded by Theorem 10 in terms of worst-case performance), the large value of $c = n$ makes it extremely unlikely for the indexing cost to be high. In particular, note that even if the density surrounding each tuple follows a heavy-tailed scale-free distribution, the setting of $c = n$ still makes the number of dense regions, therefore the query cost for indexing, a constant.We shall verify this intuition and perform a comprehensive test of different parameter settings in the experimental evaluations.

6.3 MD-RERANK

In this section, we consider the generic query reranking problem, i.e., over any monotonic user-specified ranking function. We start by pointing out the problem of a seemingly simple solution: implementing a classic top-k query processing algorithm such as TA [56] by calling 1D-RERANK as a subroutine. The problem illustrates the necessity of properly leveraging the conjunctive queries supported by the search interface of the underlying database. To do so, we start with the design

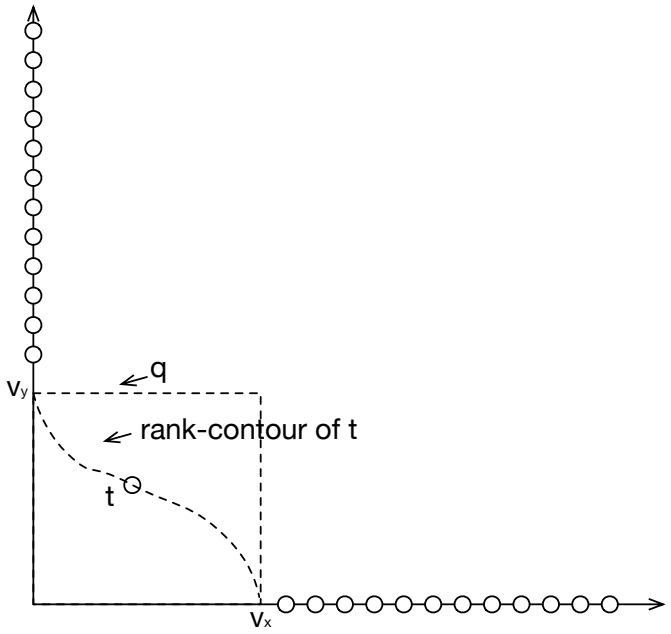


Figure 67: Illustration of problem with TA over 1D-RERANK

of MD-BASELINE, a baseline technique similar to 1D-BASELINE. Despite of the similarity, we shall point out a key difference between two cases: MD-BASELINE requires many more queries because of the more complex shape of what we refer to as a tuple's "rank-contour" - i.e., the subspace (e.g., a line in 2D space) containing all possible tuples that have the same user-defined ranking score as a given tuple t. To reduce this high query cost, we propose Algorithm MD-BINARY which features two main ideas, *direct domination detection* and *virtual tuple pruning*. Finally, we integrate the dense-region indexing idea with MD-BINARY to produce our final MD-RERANK algorithm.

6.3.1 Problem with TA over 1D-RERANK

A seemingly simple solution to solve the generic query reranking problem is to directly apply a classic top-k query processing algorithm, e.g., the threshold (TA) algorithm [56], over the Get-Next primitive offered by 1D-RERANK. While we

refer readers to [56] for the detailed design of TA, it is easy to see that 1D-RERANK offers all the data structure required by TA, i.e., a sorted access to each attribute. Note that the random access requirement does not apply here because, as discussed in the preliminary section, the search interface returns all attribute values of a tuple without the need for accessing each attribute separately. Since TA supports all monotonic ranking functions, this simple combination solves the generic query reranking problem defined in § 6.1.

While simple, this solution suffers from a major efficiency problem, mainly because it does not leverage the full power provided by client-server databases. Note that, by exclusively calling 1D-RERANK as a subroutine, this solution focuses on just one attribute at a time and does not issue any multi-predicate (conjunctive) queries supported by the underlying database (unless such predicates are copied from the user query). The example in Figure 67 illustrates the problem: In the example, there is a large number of tuples with extreme values on both attributes (i.e., tuples on the x- and y-axis). Since this TA-based solution focuses on one attribute at a time, these extreme-value tuples have to be enumerated first *even when* the system ranking function completely aligns (e.g., equals) the user-desired ranking function. In other words, no matter what the system/user ranking function is, discovering the top-1 reranked tuple requires sifting through at least half of the database in this example.

On the other hand, one can observe from the figure the power bestowed by the ability to issue multi-predicate conjunctive queries. As an example, consider the case where the system ranking function is well-conditioned and returns t as the result for SELECT * FROM D. Given t, we can compute its rank-contour, i.e., the line/curve that passes through all 2D points with user-defined score equal to $\mathcal{S}(t)$, i.e., the score of t. The curve in the figure depicts an example. Given the rank-contour, we can issue the smallest 2D query encompassing the contour, e.g., q in Figure 67, and immediately conclude that t is the No. 1 tuple when q returns t and nothing else (assuming $k > 1$). This represents a significant saving from the query cost of implementing TA over 1D-RERANK.

6.3.2 MD-Baseline

Discovery of Top-1

To leverage the power of multi-predicate queries, we start with developing a baseline algorithm similar to 1D-BASELINE. The algorithm starts with discovering the top-1 tuple t according to an arbitrary attribute, say A_1. Then, we compute the rank-contour of t (according to the user ranking function, of course), specifically the values where t's rank-contour intersects with each dimension, i.e.,

$$\ell(A_i) = \max\{v \in V(A_i) | \mathcal{S}(t) \leq \mathcal{S}(0, \ldots, 0, v, 0, \ldots, 0)\}. \tag{42}$$

Figure 68 depicts an example of $\ell(A_i)$ for the two dimensions, computed according to t.

We now issue m queries of the form

$$\begin{aligned}
q_1: &\ A_1 < t[A_1] \ \&\ A_2 < \ell(A_2) \ \&\ \cdots \ \&\ A_m < \ell(A_m) \\
q_2: &\ A_1 \in [t[A_1], \ell(A_1)) \ \&\ A_2 < t[A_2] \ \&\ A_3 < \ell(A_3) \ \&\ \cdots \\
&\ \&\ A_m < \ell(A_m) \\
q_m: &\ A_1 \in [t[A_1], \ell(A_1)) \ \&\ \cdots \ \&\ A_{m-1} \in [t[A_{m-1}, \\
&\ \ell(A_{m-1})) \ \&\ A_m < t[A_m]
\end{aligned} \tag{43}$$

Again, Figure 68 shows an example of q_1 and q_2 for the 2D space.

One can see that the union of these m (mutually exclusive) queries covers in its entirety the region "underneath" the rank-contour of t. Thus, if none of them overflows, we can safely conclude that the No. 1 tuple must be either t or one of the tuples returned by one of the m queries. If at least one query overflows and returns t' with score $\mathcal{S}(t') < \mathcal{S}(t)$, i.e., t' that ranks higher than t, we restart the entire process with $t = t'$.

Otherwise, for each query q_i that overflows, we "partition" it further into $m + 1$ queries. Let t_i be the tuple returned by q_i. We compute for each attribute A_j a

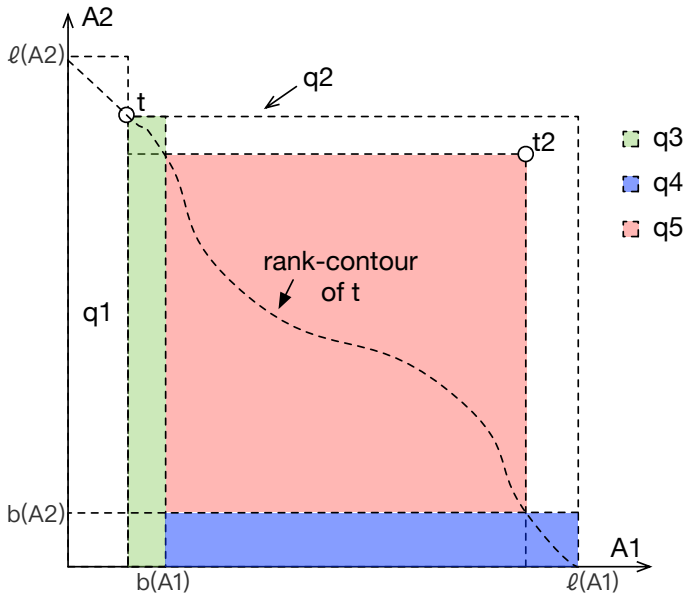


Figure 68: Example of MD-BASELINE

value $b(A_j)$ such that

$$b(A_j) = \min\{v \in V(A_j) | \mathcal{S}(t) \leq \mathcal{S}(t_i[A_1], \ldots, t_i[A_{j-1}], b(A_j), t_i[A_{j+1}], \ldots, t_i[A_m])\}. \quad (44)$$

Intuitively, $b(A_j)$ can be understood as follows: In order for a tuple t' to outrank t, the highest-ranked tuple discovered so far, it must either "outperform" $b(A_j)$ on at least one attribute, i.e., $\exists A_j$ with $t'[A_j] < b(A_j)$, or it must dominate t_i. Examples of $b(A_1)$ and $b(A_2)$ are shown in Figure 68.

Note that, while any monotonic (user-defined) ranking function yields a unique solution for $b(A_j)$, the complexity of computing it can vary significantly depending upon the design of the ranking function. Nonetheless, recall from § 6.1 that our main efficiency concern is on the *query cost* of the reranking process rather than the computational cost for solving $b(A_j)$ locally (which does not incur any additional query to the underlying database). Furthermore, the most extensively studied ranking function in the literature, a linear combination of multiple attributes,

features a constant-time solver for $b(A_j)$.

Given $b(A_j)$, we are now ready to construct the $m+1$ queries we issue. The first m queries $q_{i1}, \ldots, q_{im}$ cover those tuples outperforming $b(A_1), \ldots, b(A_m)$ on $A_1, \ldots, A_m$, respectively; while the last one covers those tuples dominating t_i. Specifically, q_{ij} ($j \in [1, m]$) is the AND of q_i and

$$(A_1 \geq b(A_1)) \text{ AND } \cdots \text{ AND } (A_{j-1} \geq b(A_{j-1})) \text{ AND } (A_j < b(A_j)) \quad (45)$$

The last query is the AND of q_i and $A_1 \leq t_i[A_1]$ AND $\cdots$ AND $A_m \leq t_i[A_m]$, i.e., covering the space dominating t_i.

Once again, at anytime during the process if a query returns t' with $\mathcal{S}(t') < \mathcal{S}(t)$, we restart the entire process with $t = t'$. Otherwise, for each query that overflows, we "partition" it into $m+1$ queries as described above.

In terms of query cost, recall from § 6.1 our idea of leveraging the query history by checking if any previously discovered tuples match the query we are about to issue. Given the idea, each tuple will be retrieved at most once by MD-BASELINE. Since each tuple we discover triggers at most $m+1$ queries which are mutually exclusive with each other, one can see that the worst-case query cost of MD-BASELINE for discovering the top-1 tuple is $O(m \cdot n)$.

Discovery of Top-k

We now discuss how to discover the top-k ($k > 1$) tuples satisfying a given query. To start, consider the discovery of No. 2 tuple after finding the top-1 tuple t_1. What we can do is to pick an arbitrary attribute, say A_1, and partition the search space into two parts: $A_1 < t_1[A_1]$ and $A_1 > t_1[A_1]$. Then, we launch the top-1 discovery algorithm on each subspace. Note that during the discovery, we can reuse the historic query answers - e.g., by starting from the tuple(s) we have also retrieved in each subspace that have the smallest $\mathcal{S}(\cdot)$. One can see that one of the two discovered top-1s must be the actual No. 2 tuple t_2 of the entire space.

Once t_2 is discovered, in order to discover the No. 3 tuple, we only need to further split the subspace from which we just discovered t_2 (into two parts). For

example, if we discovered t_2 from $A_1 > t_1[A_1]$, then we can split it again into $A_1 \in (t_1[A_1], t_2[A_1])$ and $A_2 > t_2[A_1]$. One can see that the No. 3 tuple must be either the top-1 of one of the two parts or the top-1 of $A_1 < t_1[A_1]$, which we have already discovered. As such, the discovery of each tuple in top-k, say No. h, requires launching the top-1 discovery algorithm *exactly twice*, over the two newly split subspaces of the subspace from which the No. $h-1$ tuple was discovered. Thus, the worst-case query cost for MD-BASELINE to discover all top-k tuples is $O(m \cdot n \cdot k)$.

6.3.3 MD-Binary

Problem of MD-Baseline

A main problem of MD-Baseline is its poor performance when the system ranking function is *negatively correlated* to the user-desired ranking function. To understand why, consider how MD-Baseline compared with the 1D-Baseline algorithm discussed in § 6.2. Both algorithms are iterative in nature; and the objectives for each iteration are almost identical in both algorithms: once a tuple t is discovered, find another tuple t' that outranks it according to the input ranking function. The difference, however, is that while it is easy to construct in 1D-Baseline a query that covers only those tuples which outranks t (for the attribute under consideration), doing so in the MD case is impossible.

The reason for this difference is straightforward: observe from Figure 69 that, when there are more than one, say two, attributes, the subspace of tuples outranking t is roughly "triangular" in shape. On the other hand, only "rectangular" queries are supported by the database. This forces us to issue at least m queries to "cover" the subspace outranking t (without covering, and returning, t itself).

The problem for this "coverage" strategy in MD-Baseline, however, is that the rectangular queries it issues may match many tuples that indeed rank lower (i.e., have larger $\mathcal{S}(\cdot)$)) than t according to the desired ranking function. For example, half of the space covered by q_2 in Figure 69 is occupied by tuples that rank lower than t. This means that, when the system ranking function is negatively correlated with our desired one, queries like q_2 in Figure 69 are most likely going to return

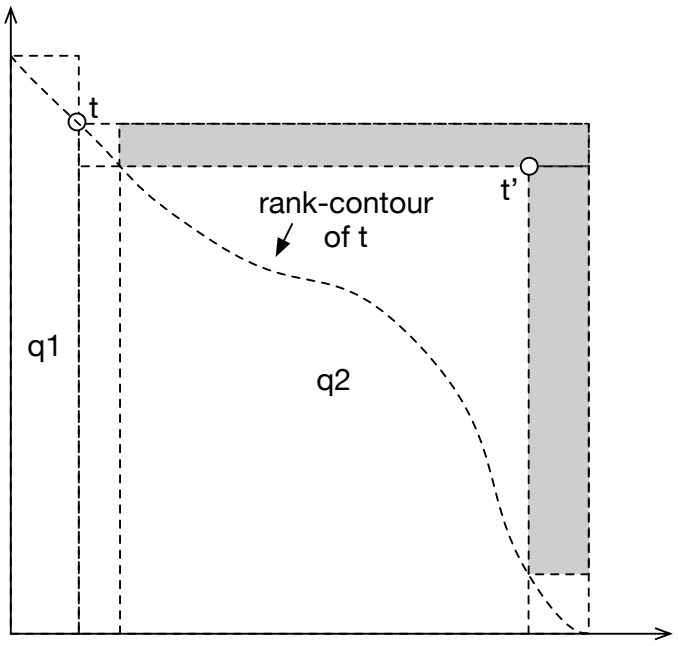


Figure 69: Illustration of problem with MD-Baseline

tuples that rank lower than t. This outcome has two important ramifications on the efficiency of MD-Baseline: First, it significantly slows down the process of iteratively finding a tuple that outranks the previous one. Second, within each iteration, it slows down the pruning of the search space. For example, observe from Figure 69 that, after q_2 returns t', the pruning effect on the space covered by q_2 is minimal, i.e., only the dark subspace on the top-right corner of q_2.

Design of MD-Binary

We propose two ideas in MD-Binary to address the two ramifications of MD-Baseline, respectively:

Direct Domination Detection: The intuition of this idea can be stated as follows: When a query such as q_2 returns a tuple t' that ranks lower than t, we attempt to "test" whether this is indeed caused by the absence of higher-ranked tuples in q_2, or by the ill-conditioned nature of the system ranking function. As discussed above, there is no way to efficiently cover the subspace of tuples outranking t. Thus, what

we do here is to find the single query which (1) is a subquery of q_2, (2) only covers the subspace outranking t, and (3) has the maximum volume among all queries that satisfy (1) and (2).

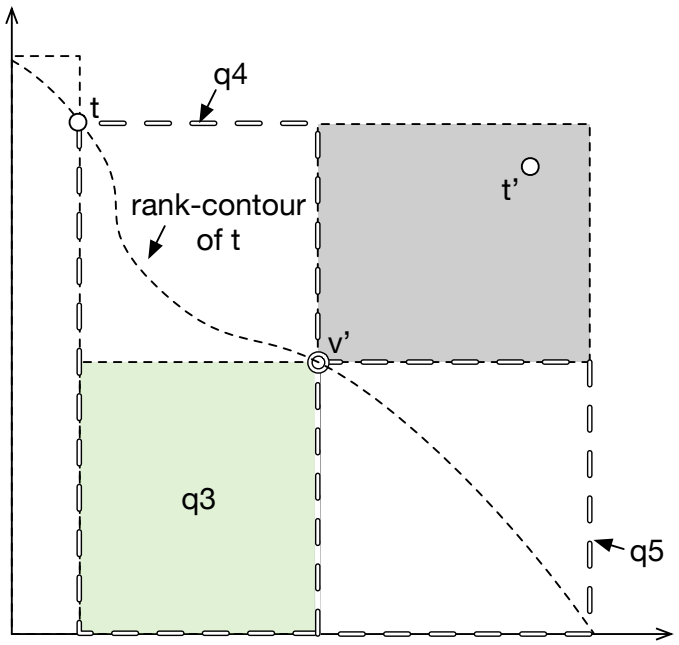


Figure 70: Design of MD-Binary: Example 1

For example, when q_2 in Figure 69 returns t', we issue q_3 (marked in green) in Figure 70 which covers roughly half of the "triangular" subspace underneath the rank-contour of t in q_2. As another example, if q_1 in Figure 68 returns a tuple with lower rank than t, then we the max-volume tuple would be q_7 in Figure 71, which covers almost all of the subspace outranking t in q_1. One can see from these examples that, if the returning of t' is caused by the ill-conditioned system ranking function while there are abundant tuples outranking t, then q_3 and/or q_7 are likely to return such a tuple and successfully push MD-Binary to the next iteration. If, on the other hand, q_3 returns empty, we use the next idea to further partition q_2, in order to determine whether there is any tuple in it that outranks t.

Virtual Tuple Pruning: We now address the second problem of MD-Baseline, i.e.,

the lack of pruning power when the system ranking function is negatively correlated with the desired one. To this end, our idea is to prune the search space according to *not* the returned tuple, but a *virtual tuple* created for the purpose of minimizing the pruned subspace. Figure 70 illustrates an example: Instead of partitioning q_2 with t' like in Figure 69 which results in minimal pruning, we "create" a virtual tuple v' which maximizes the reduction of search space as marked in gray in Figure 70.

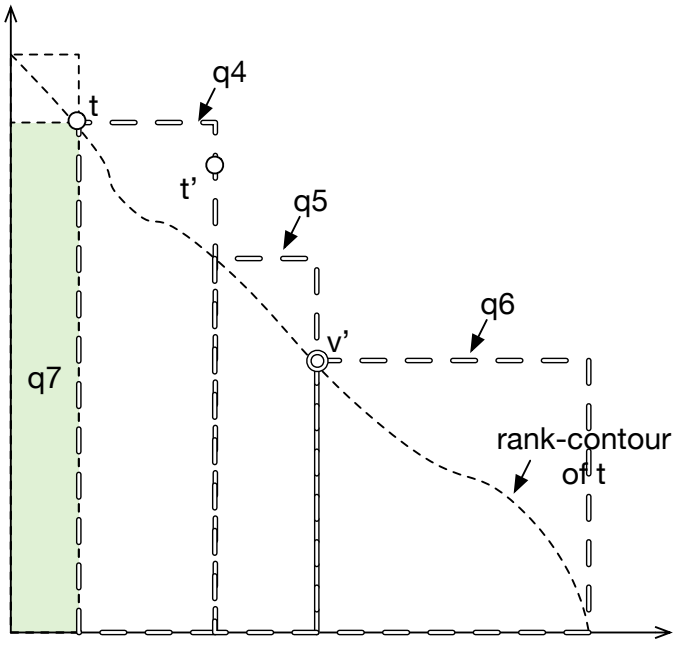


Figure 71: Design of MD-Binary: Example 2

Figure 70 represents one possible outcome of virtual tuple pruning, when v' happens to dominate the tuple t' returned by q_2. The other possible outcome is depicted in Figure 71, where v' does not dominate t'. In this case, if we still split q_2 as in Figure 70, then one of the subspace (i.e., $x \in (t[x], v'[x])$ AND $y < t[y]$) would return t', making the query answer useless. As such, we split q_2 into three pieces in this scenario, as shown in Figure 71.

The more general design of virtual tuple pruning for an m-D database is shown in Algorithm 18. The algorithm also depicts the direct domination detection idea.

Note from the algorithm that, depending on the values of t' and v' on the m attributes, the number of split subspaces can range from m, when v' dominates t', to $2m-1$, when t' dominates v' on all but one attribute.

Algorithm 18 MD-BINARY

1: apply 1D-RERANK on A_1 to t and set threshold=$s(t)$
2: add the queries in Equation 43 to the empty queue
3: **while** queue is not empty **do**
4: $q\prime$=queue.delete
5: T = Top-k($q\prime$);t = argmin$_{s(t)}\{t \in T\}$
6: **if** $s(t)$ <threshold **then**
7: threshold=$s(t)$;
8: **goto** Line 2
9: **end if**
10: **if** T is valid **then continue**
11: $v' = \text{argmax}_{vol(v)}\{v \in \text{contour}(t)\}$
12: T = Top-k($\forall A \in \mathcal{A}$, $A \leq v'[A]$)
13: **if** T is not underflow **then**
14: t=argmin$_{s(t)}\{t \in T\}$
15: threshold=$s(t)$
16: **goto** Line 2
17: **end if**
18: **for** $\forall A_i \in \mathcal{A}$ **do**
19: **if** $t[A_i] \geq v'[A_i]$ add the following query to the queue

$$q_1: \ q\prime \text{ AND } A_i < v'[A_i] \text{ AND } \{\forall_{j=1}^{i-1} A_j >= v'[A_j]\}$$

20: **else** add the following queries to the queue

$$q_1 : q\prime \text{ AND } A_i < t[A_i] \text{ AND } \{\forall_{j=1}^{i-1} A_j >= v'[A_j]\}$$
$$q_2 : q\prime \text{ AND } A_i < v'[A_i] \text{ AND } A_{i+1} < b_t(A_{i+1}) \text{ AND } \{\forall_{j=1}^{i-1} A_j >= v'[A_j]\}$$

21: **end for**
22: **end while**
23: **return** t

One can see from the design that virtual tuple pruning does not affect the

correctness of the algorithm: so long as $\mathcal{S}(v') \geq \mathcal{S}(t)$, the union of the split subspaces still cover q_2. On the other hand, the benefit of the idea can be readily observed from Figure 70: instead of having only a small reduction of the search space like in Figure 69, now we can prune half of the space in q_2 that rank below t (in this 2D case, of course). The experimental results in § 6.5 demonstrate the effectiveness of virtual tuple pruning.

6.3.4 MD-RERANK

Just like the 1D case, the query cost of MD-Binary may increase significantly when there is a dense cluster of tuples right above the rank-contour of the top-1 tuple. In this case, the split in MD-Binary may have to continue for a large number of times before all tuples in the cluster are excluded from the search space. Once again, our solution to this problem is index-based reranking. Like in the 1D case, we proactively *record as an index* densely located tuples once we encounter them, so that we do not need to incur a high query cost every time a query q triggers visits to the same dense region.

More specifically, MD-RERANK follows MD-Binary until a remaining search space (1) is covered by an already crawled region in the index; or (2) has volume smaller than $|V| \cdot (s/n)/c$, where $|V|$ is the volume of the entire data space, and s and c are the same as in 1D. In the earlier case, since the search space has been crawled already, we can directly reuse the crawled tuples. In the latter case, we follow the same procedure as in 1D-RERANK, i.e., we crawl the space and, if it indeed turns out to be dense (by containing at least s tuples), we include the crawled tuples into the index. Algorithm 19 depicts the pseudocode of MD-RERANK.

6.4 discussions

General Positioning Assumption: In previous discussions, we made the general positioning assumption, i.e., each tuple has a unique value on each attribute, for the simplicity of discussions. We now consider the removal of this assumption. Note that the removal of this assumption for MD-RERANK is extremely simple: the only

Algorithm 19 MD-RERANK

follow MD-BINARY
while processing query $q\prime$ **do**
  if $V(q\prime) < |V| \cdot (s/n)/c$ **then**
    $q\prime$ = remove $Sel(q)\}$ from $q\prime$
    if ORACLE($q\prime$) exists **then**
      return argmin$_{s(t)}\{t \in D(q\prime))|$ t matches $Sel(q)\}$
    end if
    t=MD-BASELINE($q\prime$)
    add t to temp
    while t does not satisfy $Sel(q)$ **do**
      t_1 = MD-BASELINE($q\prime$ AND $A_1 < t[A_1]$)
      t_2 = MD-BASELINE($q\prime$ AND $A_1 > t[A_1]$)
      t=min(t_1 , t_2)
      add t_1 and t_2 to temp
    end while
    add temp to $D(q\prime)$
  end if
end while

tuple(s) that can be missed by MD-RERANK are those that have the exact same value on *every single attribute*. Thus, the only post-processing step required for removing the assumption is to form a fully specified query according to No. h tuple just discovered. If more than one, say i, tuples are returned, they become the No. h to No. $(h + i - 1)$ top-ranked tuples. Removing the assumption for 1D-RERANK is slightly more complex. For example, if we are running it over attribute A_1, the removal of the general positioning assumption means query SELECT * FROM D WHERE $A_1 = t[A_1]$ might overflow. In this case, our solution is to call the crawling algorithm [36] to discover, one at a time, tuples satisfying $A_1 = t[A_1]$, as all of these tuples have the same rank for the purpose of 1D-RERANK.

Multiple/Known System Ranking Functions: Another interesting issue arising in practice is when the client-server database offers more than one ranking functions, often times allowing ranking over a specific attribute. For example, Amazon.com offers not only a proprietary rank by "popularity", the design of which is unknown, but also ranking by price, which is an attribute usually involved

in the user-specified ranking function. An interesting implication of such a "public" ranking function is that it might boost the performance of the TA-1D algorithm discussed in the beginning of § 6.3. Specifically, since now TA can simply use the public ranking function on the attribute instead of calling 1D-RERANK, it may have a even lower query cost than MD-RERANK when the user-desired ranking function aligns well with the system one.

Point Predicates: In this paper, we focused on cases where attributes involved in the ranking function are numeric attributes that support range queries. While this is often the case in practice (as evidenced in real-world websites such as the aforementioned Blue Nile where all attributes such as price, carat, clarity, etc., are available as range predicates), there are also cases where a ranking attribute with only a small number of domain values can only be specified as a point predicate (i.e., of the form $A_i = v$) in the database search interface. For 1D-RERANK, this is often a blessing because it simplifies the task to querying the attribute values in the preference order (plus the crawling-based provision as in the discussion for the general positioning assumption). On the other hand, it makes MD-RERANK much more costly, because now a conjunctive query covers a much smaller space than the range case. Thus, an intuition here is to prefer the TA-1D algorithm over MD-RERANK when a large number of attributes are searchable as point predicates only. Due to space limitations, we leave a comprehensive study of this issue to future work.

6.5 Experimental Evaluation

6.5.1 Experimental Setup

In this section, we present our experimental results over a number of several real-world datasets, offline and online. We started with the offline case by testing over a real-world dataset we have already collected. Specifically, we constructed a top-k web search interface over it, and then executed our algorithms through the interface. This offline setting enabled us to not only verify the correctness of our algorithms, but also investigate how the performance of query reranking

changes with various factors such as the database size, the system ranking function, settings of the system search interface, etc. We followed the offline tests with online, live, experiments over two real-world web databases, including the largest online diamond retailer and a popular auto search website. In all these experiments, we applied the extensions described in § 6.4 to resolve the general positioning assumption which may not hold in practice.

Offline Dataset: We used the flight on-time dataset published by the US Department of Transportation (DOT)[15]. A wide range of third-party websites use this dataset to identify on-time performance of flights, routes, airports, airlines, etc. It consists of 457,013 flight records of 14 US carriers during the month of May 2015. It has 28 attributes, out of which we selected the following 8 attributes for ranking: *Dep-Delay, Taxi-Out, Taxi-In, Arr-Delay-New, CRS-Elapsed-Time, Actual-Elapsed-Time, Air-Time,* and *Distance*. The domain sizes are 1988, 180, 180, 1971, 718, 724, 676, and 5000, respectively. For the purpose of the experiments, we considered two system ranking functions: *0.3 AIR-TIME + TAXI-IN* (SR1) and *-0.1 DISTANCE - DEP-DELAY* (SR2). In general, SR1 has a positive correlation with the user-specified ranking functions we tested, while SR2 has a negative one. We set SR1 as the default ranking function in the experiments. The value of k offered by the database is set to 10 by default.

Online Experiments: We conducted live experiments over two real-world websites: *Blue Nile* (BN) and *Yahoo! Autos* (YA).

Blue Nile[16] is the largest diamonds online retailer in the world. At the time of our experiments, its catalog had 117,641 diamonds. We considered *Carat, Depth, LengthWidthRatio, Price,* and *Table* as the ranking attributes, and *Clarity, Color, Cut, Fluorescence, Polish, Shape,* and *Symmetry* for filtering. The domains for the ranking attributes are [0.23,22.74], [0.45,0.86], [0.49,0.89], [$220,$4506938] and [0.75,2.75], respectively. BN allows multiple ranking functions - ordering based on each attributes individually as well as by the derived attribute *price-per-carat*.

[15]downloaded from `http://www.transtats.bts.gov/DL_SelectFields.asp?Table_ID=236&DB_Short_Name=On-Time`

[16]`http://www.bluenile.com/diamond-search`

Yahoo! Autos is a popular website for buying used cars[17]. We considered the 13,169 cars listed for sale within 30 miles of New York city. We treated *Price, Milage* and *Year* as the ranking attributes, and *BodyStyle, DriveType, Transmission, Name* and *Model* as the filtering attributes. The cars had a price range between $0 and $50,000, mileage between 0 and 300,000, and were manufactured between 1993 and 2016. The default ranking function is "distance from a predefined location" (which is not monotonic). Additionally, it supports ranking by each of the numerical attributes individually.

Performance Measures: As explained in § 6.1, our algorithms always return the precise query answer. After verifying the correctness in all offline experiments, we turn our attention to the key performance measure, efficiency, which is measured by the number of queries issued to the web database.

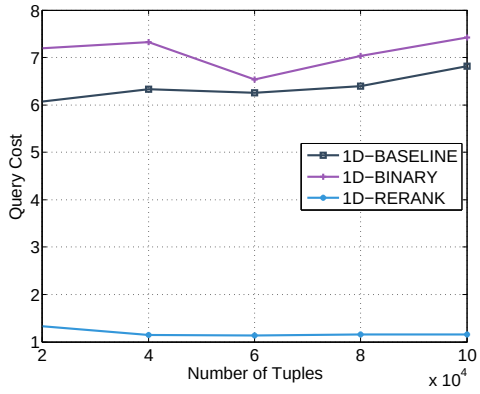


Figure 72: 1D: Impact of n (SR1)

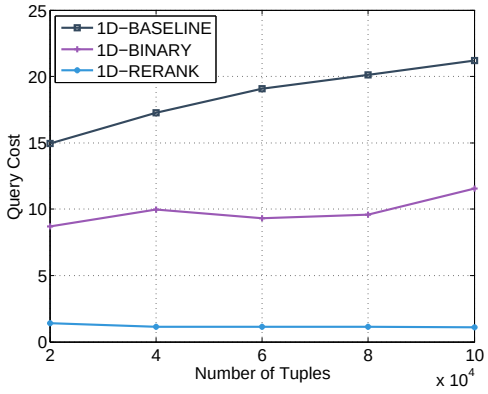


Figure 73: 1D: Impact of n (SR2)

6.5.2 1D Experiments

Constructing Workload of User Preference Queries: We tested a diverse set of user-specified queries of the form SELECT * FROM D WHERE $Sel(q)$ ORDER BY A_i. Specifically, we randomly selected different subsets of filtering attributes for the WHERE clause, while choosing the (1D) ranking attribute uniformly at

[17] `https://autos.yahoo.com/used-cars/`

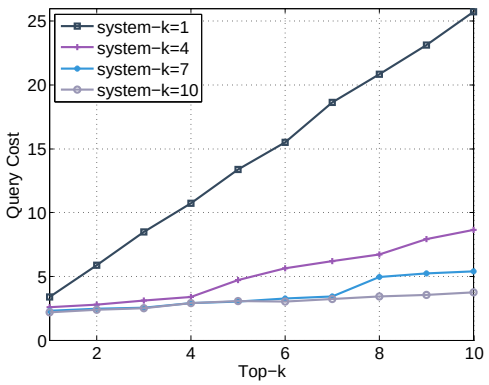


Figure 74: 1D: Impact of System-k

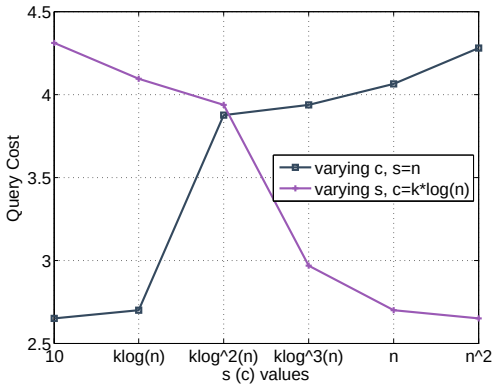


Figure 75: 1D: Impact of s and c

random. This approach has a number of appealing properties. First, it covers diverse cases that include ideal, worst-case and typical scenarios. Second, since 1D-RERANK uses on-the-fly indexing to amortize the cost between different user-issued queries, our diverse query workload simulates a real-world scenario where the service is used by multiple users. For each experimental configuration, we execute each of the queries and report the average query cost. Specifically, for the DOT dataset, we constructed 32 queries of which 25% do not have any filtering condition. For BN, we constructed a set of 20 queries, of which 4 have no filtering conditions, while these values are 15 and 2 for YA, respectively.

Experiments over the Real-world Dataset

Impact of Database Size and System Ranking Function: We started by testing the impact of database size on our algorithms for the two system ranking functions SR1 and SR2. To test databases of varying sizes, we drew 10 simple random samples of a given size from the DOT dataset, and measured the average query cost for the entire workload over these 10 small databases. Figures 72 and 73 show the average query cost for retrieving the top-1 tuple over SR1 and SR2, respectively. As expected, the database size has negligible impact on the query cost. Also note from the figures that, consistent with our theoretical analysis, Algorithm 1D-RERANK outperformed both 1D-BASELINE and 1D-BINARY significantly. One can also note that the change in system ranking function has a major impact

on the performance comparison between 1D-BASELINE and 1D-BINARY, yet has a negligible impact on that of 1D-RERANK, again consistent with our theoretical discussions.

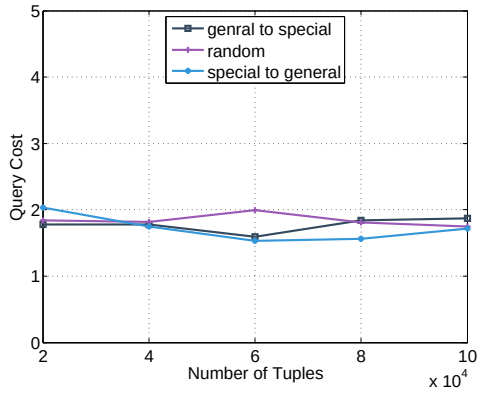


Figure 76: 1D: Impact of Query order in 1D-RERANK

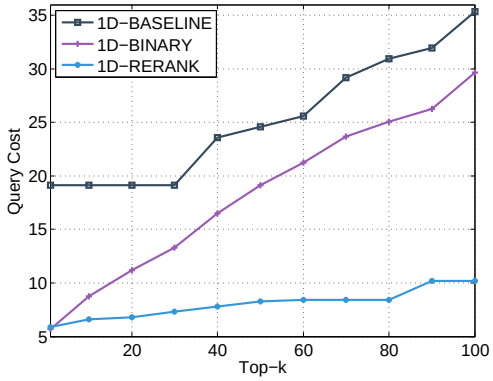


Figure 77: 1D: Top$k$ Query Cost (BN)

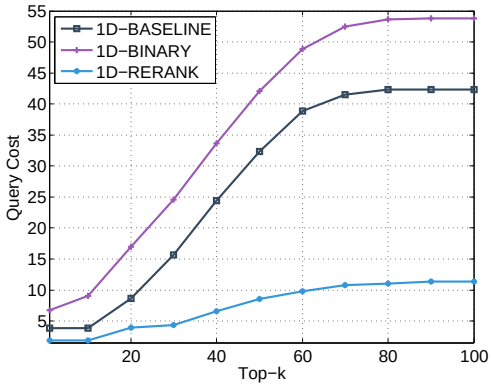


Figure 78: 1D: Top$k$ Query Cost (YA)

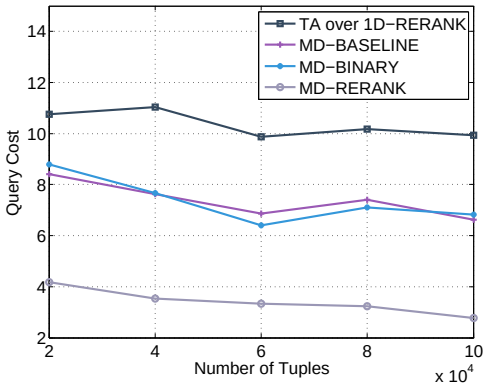


Figure 79: MD: Impact of n (SR1)

Impact of Value of k: Figure 74 shows the average (accumulative) query cost for retrieving top-1 to top-10 tuples when the system k varies from 1 to 10. There are two key observations from the figure: First, our query cost increases (about) linearly with the number of desired top answers, demonstrating its scalability to a

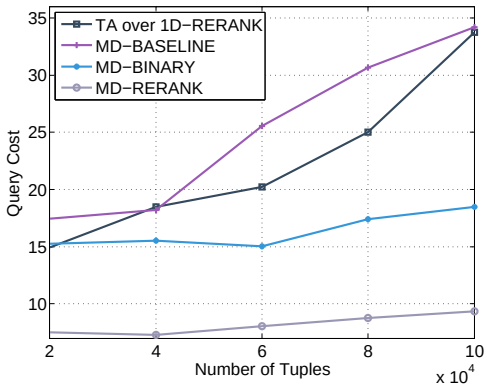


Figure 80: MD: Impact of n (SR2)

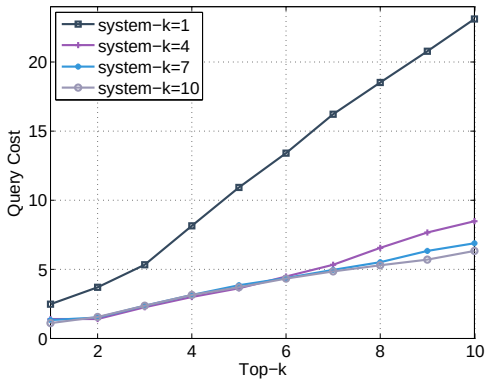


Figure 81: MD: Impact of System-k

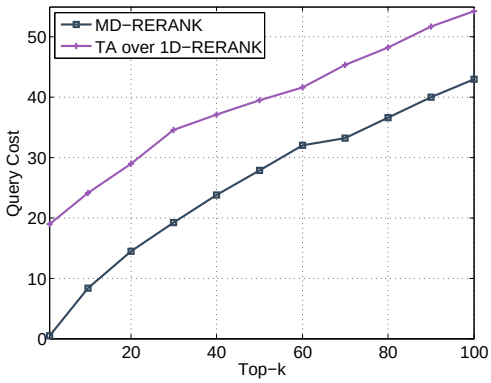


Figure 82: MD: Top$k$ Query Cost (BN)

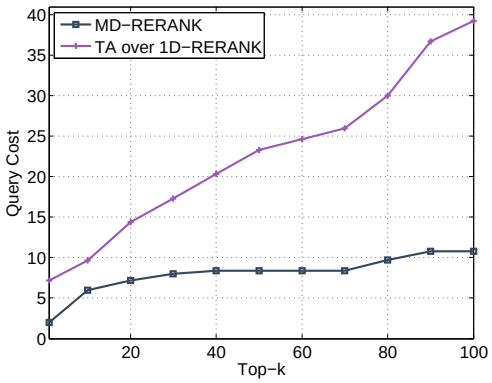


Figure 83: MD: Top$k$ Query Cost (YA)

large desired answer size. Second, the query cost ,as expected, decreases when the system offers a larger k.

Impact of 1D-RERANK parameters s and c: Recall from § 6.1 that the performance of 1D-RERANK can be parameterized by s and c. We conducted two experiments to empirically verify the impact. In the first experiment, we fixed the value of s to n and varied c between 10 and n^2. In the other one, we fixed the value of c to $k \log_2 n$ and varied the value of s from 10 to n^2. Figure 75 shows the average query cost for both settings. As our theoretical results suggest, setting $c = k \log_2 n$ and $s = n$ resulted in the (almost) optimal performance. One can see that further reducing c

or increasing s does not have much affect on query cost, yet significantly increases the index size.

Impact of Query Order on 1D-RERANK: Recall that 1D-RERANK constructs the index on the fly. As such, when queries are issued in different order, the index being maintained may differ. To test whether the order of user queries have a major effect on the performance of 1D-RERANK, we ran an experiment using SR1 with three query-issuing orders: (1) from low to high selectivity (i.e., from more general to narrower queries), (2) from high to low selectivity, and (3) in a random order. Figure 76 shows that the query issuance order has a negligible effect on the query cost of 1D-RERANK.

Online Experiments

We also conducted two live experiments over Blue Nile and Yahoo! Autos, aiming to retrieve the top-100 tuples for each of the user query in the workload. The default system-k for BN and YA are 30 and 15, respectively, with the system ranking function being the default for each website, i.e., descending value of *price per carat* for BN and *distance from the pre-defined location* for YA. Figures 77 and 78 show the average query cost for retrieving top-h tuples. As expected, 1D-RERANK significantly outperforms the other algorithms for both websites. For BN, while 1D-BINARY performed well in the beginning, it required higher query cost for large values of h. That is because the binary search approach keeps dividing the search area in half until the issued query underflows, thus it is likely to end up with an underflowing query that contains fewer tuples, leading to less saving in the query cost. For YA, note that 1D-BINARY does not benefit much from the savings and is hence outperformed by 1D-BASELINE.

6.5.3 MD Experiments

In this subsection, we compare the performance of MD-RERANK against three baseline methods: the aforementioned "TA over 1D-RERANK", as well as MD-BASELINE and MD-BINARY. Once against, we tested both offline and online settings.

Constructing Workload of User Preference Queries: The workload is con-

structed using a process similar to one described in § 6.5.2. However, the ranking functions are constructed by selecting a subset from the set of all ranking attributes and choosing different weights between 0 and 1 for each of them. The workload consists of 32, 12 and 10 queries for DOT, BN and YA, respectively, of which 8, 3 and 2 do not have any filtering conditions.

Experiments over the Real-world Dataset

Impact of Database Size and System Ranking Function: The experimental setup was similar to the 1D experiments in § 6.5.2. We evaluated our algorithms for different database sizes and system ranking functions SR1 and SR2. Figures 79 and 80 shows the results for SR1 and SR2 respectively. In both cases, the algorithm MD-RERANK significantly outperformed all three competing baselines. One may notice an increase in the query cost of the algorithms when n increases in Figures 80, and a decrease in Figures 79. That is because when system and user-specified ranking function are anti-correlated, the more tuples database has, the more queries are required to find top tuples for the user-specified ranking function (since more tuples are ranked higher than them based on SR2). The case is vice-verse for SR1.

Impact of System-k: We then varied k, the number of tuples returned by the web database and measured the average query cost to retrieve top-10 tuples for the query workload. Figure 81 shows the results. As expected, higher values of system-k required lesser query cost to obtain the top-10 tuples. When $k = 1$, our algorithms were not able to use the savings by valid queries resulting in a substantial query cost.

Online Experiments

We applied MD-RERANK, as well as TA over 1D-RERANK, to retrieve the top-100 tuples for each query in the workload. Figure 82 shows the average query cost for the BN experiment. As shown in the figure, MD-RERANK outperformed TA significantly. The results for YA experiment is reflected in Figure 83. The substantial difference in query cost of the algorithms can be explained by the observation by the negative correlation between the ranking tuples in YA queries (for example the cars with higher mileage are probably cheaper). Hence TA

algorithm had to issue many GetNext operations before it finds the top tuples.

6.6 Related Work

Top-k discovery methods can get divided in three main categories: *(sorted/random) access-based* methods, *layering-based* approaches, and *view-based* techniques. The first series of algorithms take the advantage of the data access methods. For example, *NRA* [56] assumes the existence of one sorted list of tuples for each attribute, and finds the Top-k only by exploring the lists, while *TA* [56] applies both random and sorted access. The more advanced algorithms in this category are *CA* [56], *Upper/Pick* [78], and [79]. The next category is the set of algorithms, such as *ONION* [60] and [61], that pre-process the data and index the layers of extremum tuples that gaurantee including the Top-k. View-based methods such as *PREFER* [58] and *LPTA* [59], employ the materialized views to increase the efficiency of Top-k discovery process. While prior work focused on minimizing the storage overhead of indices/materialized views and the computational overhead of processing top-k queries, we have to focus on minimizing the number of queries issued to the underlying database. This fundamentally different data access model also leads to a different cost model. For example, many prior work, such as [56] and [83], assume a separate cost for accessing each attribute and/or evaluating each predicate in the top-k query, while in our problem *all* attributes of a tuple are returned at once.

Hidden Databases Most of the prior works on the hidden databases relate to sampling, crawling the database, and aggregate estimation. Prior works such as [42,45] propose efficient algorithms for collecting unbiased low-variance random samples of a given hidden databasee and [84, 85] provide unbiased aggregate estimators. While [36, 39, 40] aim toward crawling the whole hidden database, [86] only crawls the maxima index.

Top-k queries over Hidden Databases As the best of our knowledge, this is the first paper on reranking the query results of a hidden database. The only prior work about Top-k in hidden databases is [87]. Assuming the full knowledge of the system ranking function and attribute domains, its goal is to go beyond the Top-k

limitation of the database interface, by partitioning the query space.

6.7 Final Remarks

In this paper, we introduced a novel problem of query reranking, a third-party service that takes a client-server database with a proprietary ranking function and enables query processing according to any user-specified ranking function. To enable query reranking while minimizing the number of queries issued to the underlying database, we develop 1D-RERANK and MD-RERANK for user-specified ranking functions that involve only one attribute and any arbitrary set of attributes, respectively. Theoretic analysis and extensive experimental results on real-world databases, in offline and online settings, demonstrate the effectiveness of our techniques and their superiority over baseline solutions.

7 Publications Relevant to the Dissertation

1. *Abolfazl Asudeh*, Azade Nazi, Nan Zhang, and Gautam Das. *Efficient Computation of Regret-ratio Minimizing Set: A Compact Maxima Representative*, in **SIGMOD** 2017.
2. *Abolfazl Asudeh*, Nan Zhang, and Gautam Das. *Query Reranking As A Service*, in **PVLDB** 2016, Vol. 9.
3. *Abolfazl Asudeh*, Saravanan Thirumuruganathan, Nan Zhang, and Gautam Das. *Discovering the Skyline of Web Databases*, in **PVLDB** 2016, Vol. 9.
4. *Abolfazl Asudeh*, Gensheng Zhang, Naeemul Hassan, Chengkai Li, and Gergely V. Záruba, *Crowdsourcing Pareto-Optimal Object Finding by Pairwise Comparisons*, in **CIKM** 2015.

References

[1] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001.

[2] R. Fagin, R. Kumar, M. Mahdian, D. Sivakumar, and E. Vee, "Comparing and aggregating rankings with ties," in *PODS*, 2004.

[3] W. Kießling, "Foundations of preferences in database systems," in *VLDB*, 2002, pp. 311–322.

[4] J. Chomicki, "Preference formulas in relational queries," *TODS*, vol. 28, no. 4, 2003.

[5] D. Sacharidis, S. Papadopoulos, and D. Papadias, "Topologically sorted skylines for partially ordered domains," in *ICDE*, 2009.

[6] C.-Y. Chan, P.-K. Eng, and K.-L. Tan, "Stratified computation of skylines with partially-ordered domains," in *SIGMOD*, 2005.

[7] N. Sarkas, G. Das, N. Koudas, and A. K. Tung, "Categorical skylines for streaming data," in *SIGMOD*, 2008, pp. 239–250.

[8] S. Zhang, N. Mamoulis, D. W. Cheung, and B. Kao, "Efficient skyline evaluation over partially ordered domains," in *VLDB*, 2010.

[9] C. Lofi, K. El Maarry, and W.-T. Balke, "Skyline queries in crowd-enabled databases," in *EDBT*, 2013, pp. 465–476.

[10] L. L. Thurstone, "A law of comparative judgment," *Psychological Review*, vol. 34, pp. 273–286, 1927.

[11] P. G. Ipeirotis, F. Provost, and J. Wang, "Quality management on amazon mechanical turk," in *HCOMP*, 2010, pp. 64–67.

[12] "Technical Report. Details omitted for double-blind reviewing. Anonymized version can be made available upon the request of the program committee."

[13] M. E. Rorvig, "The simple scalability of documents," *Journal of the American Society for Information Science*, vol. 41, no. 8, pp. 590–598, 1990.

[14] B. Carterette, P. N. Bennett, D. M. Chickering, and S. T. Dumais, "Here or there: Preference judgments for relevance," in *ECIR*, 2008.

[15] X. Chen, P. N. Bennett, K. Collins-Thompson, and E. Horvitz, "Pairwise ranking aggregation in a crowdsourced setting," in *WSDM*, 2013, pp. 193–202.

[16] V. Polychronopoulos, L. de Alfaro, J. Davis, H. Garcia-Molina, and N. Polyzotis, "Human-powered top-k lists," in *WebDB*, 2013.

[17] S. B. Davidson, S. Khanna, T. Milo, and S. Roy, "Using the crowd for top-k and group-by queries," in *ICDT*, 2013, pp. 225–236.

[18] K. J. Arrow, *Social choice and individual values*. Yale University Press, 1951.

[19] N. N. Liu, M. Zhao, and Q. Yang, "Probabilistic latent preference analysis for collaborative filtering," in *CIKM*, 2009, pp. 759–766.

[20] S. Rendle, C. Freudenthaler, Z. Gantner, and L. Schmidt-Thieme, "BPR: Bayesian personalized ranking from implicit feedback," in *UAI*, 2009, pp. 452–461.

[21] J. Yi, R. Jin, S. Jain, and A. K. Jain, "Inferring users' preferences from crowdsourced pairwise comparisons: A matrix completion approach," in *HCOMP*, 2013.

[22] D. Goldberg, D. Nichols, B. M. Oki, and D. Terry, "Using collaborative filtering to weave an information tapestry," *CACM*, vol. 35, no. 12, pp. 61–70, 1992.

[23] C. Burges, T. Shaked, E. Renshaw, A. Lazier, M. Deeds, N. Hamilton, and G. Hullender, "Learning to rank using gradient descent," in *ICML*, 2005, pp. 89–96.

[24] Y. Cao, J. Xu, T.-Y. Liu, H. Li, Y. Huang, and H.-W. Hon, "Adapting ranking SVM to document retrieval," in *SIGIR*, 2006, pp. 186–193.

[25] C. J. C. Burges, R. Ragno, and Q. V. Le, "Learning to Rank with Nonsmooth Cost Functions," in *NIPS*, 2006, pp. 193–200.

[26] T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval*, vol. 3, no. 3, pp. 225–331, Mar. 2009.

[27] M. Braverman and E. Mossel, "Noisy sorting without resampling," in *SODA*, 2008, pp. 268–276.

[28] K. G. Jamieson and R. D. Nowak, "Active ranking using pairwise comparisons," in *NIPS*, 2011, pp. 2240–2248.

[29] N. Ailon, "Active learning ranking from pairwise preferences with almost optimal query complexity," in *NIPS*, 2011, pp. 810–818.

[30] S. Negahban, S. Oh, and D. Shah, "Iterative ranking from pair-wise comparisons," in *NIPS*, 2012, pp. 2483–2491.

[31] K.-L. Tan, P.-K. Eng, B. C. Ooi, *et al.*, "Efficient progressive skyline computation," in *VLDB*, 2001.

[32] J. Chomicki, P. Godfrey, J. Gryz, and D. Liang, "Skyline with presorting," in *ICDE*, 2003.

[33] D. Kossmann, F. Ramsak, and S. Rost, "Shooting stars in the sky: An online algorithm for skyline queries," in *VLDB*, 2002.

[34] W.-T. Balke, U. Güntzer, and J. X. Zheng, "Efficient distributed skylining for web information systems," in *EDBT*, 2004.

[35] E. Lo, K. Y. Yip, K.-I. Lin, and D. W. Cheung, "Progressive skylining over web-accessible databases," *Data & Knowledge Engineering*, 2006.

[36] C. Sheng, N. Zhang, Y. Tao, and X. Jin, "Optimal algorithms for crawling a hidden database in the web," *VLDB*, 2012.

[37] B. Arai, G. Das, D. Gunopulos, and N. Koudas, "Anytime measures for top-k algorithms," in *VLDB*, 2007.

[38] C. Buchta, "On the average number of maxima in a set of vectors," *Information Processing Letters*, vol. 33, no. 2, 1989.

[39] S. Raghavan and H. Garcia-Molina, "Crawling the hidden web," *VLDB*, 2000.

[40] J. Madhavan, D. Ko, Ł. Kot, V. Ganapathy, A. Rasmussen, and A. Halevy, "Google's deep web crawl," *VLDB*, 2008.

[41] A. Dasgupta, N. Zhang, and G. Das, "Turbo-charging hidden database samplers with overflowing queries and skew reduction," in *EDBT*, 2013.

[42] A. Dasgupta, G. Das, and H. Mannila, "A random walk approach to sampling hidden databases," in *SIGMOD*, 2007.

[43] A. Dasgupta, N. Zhang, and G. Das, "Leveraging count information in sampling hidden databases," in *ICDE*, 2009.

[44] T. Liu, F. Wang, and G. Agrawal, "Stratified sampling for data mining on the deep web," *Frontiers of Computer Science*, vol. 6, no. 2, pp. 179–196, 2012.

[45] F. Wang and G. Agrawal, "Effective and efficient sampling methods for deep web aggregation queries," in *EDBT*, 2011.

[46] D. Papadias, Y. Tao, G. Fu, and B. Seeger, "An optimal and progressive algorithm for skyline queries," in *SIGMOD*, 2003.

[47] X. Lin, Y. Yuan, W. Wang, and H. Lu, "Stabbing the sky: Efficient skyline computation over sliding windows," in *ICDE*, 2005.

[48] A. Asudeh, G. Zhang, N. Hassan, C. Li, and G. Zaruba, "Crowdsourcing pareto-optimal object finding by pairwise comparisons," ser. CIKM, 2015.

[49] J. Pei, B. Jiang, X. Lin, and Y. Yuan, "Probabilistic skylines on uncertain data," in *VLDB*, 2007.

[50] N. Zhang, C. Li, N. Hassan, S. Rajasekaran, and G. Das, "On skyline groups," *TKDE*, vol. 26, no. 4, 2014.

[51] Z. Gong, G.-Z. Sun, J. Yuan, and Y. Zhong, "Efficient top-k query algorithms using k-skyband partition," in *Scalable Information Systems*. Springer, 2009.

[52] I. F. Ilyas, G. Beskales, and M. A. Soliman, "A survey of top-k query processing techniques in relational database systems," *ACM Computing Surveys (CSUR)*, vol. 40, no. 4, 2008.

[53] E. Dellis and B. Seeger, "Efficient computation of reverse skyline queries," in *VLDB*, 2007.

[54] M. L. Yiu and N. Mamoulis, "Efficient processing of top-k dominating queries on multi-dimensional data," in *VLDB*, 2007.

[55] X. Lin, Y. Yuan, Q. Zhang, and Y. Zhang, "Selecting stars: The k most representative skyline operator," in *ICDE*, 2007.

[56] R. Fagin, A. Lotem, and M. Naor, "Optimal aggregation algorithms for middleware," *Journal of Computer and System Sciences*, vol. 66, no. 4, pp. 614–656, 2003.

[57] R. Fagin, R. Kumar, and D. Sivakumar, "Comparing top k lists," *SIAM Journal on Discrete Mathematics*, vol. 17, no. 1, pp. 134–160, 2003.

[58] V. Hristidis and Y. Papakonstantinou, "Algorithms and applications for answering ranked queries using ranked views," *The VLDB Journal*, vol. 13, no. 1, pp. 49–70, 2004.

[59] G. Das, D. Gunopulos, N. Koudas, and D. Tsirogiannis, "Answering top-k queries using views," in *VLDB*, 2006.

[60] Y.-C. Chang, L. Bergman, V. Castelli, C.-S. Li, M.-L. Lo, and J. R. Smith, "The onion technique: indexing for linear optimization queries," in *SIGMOD*, 2000.

[61] D. Xin, C. Chen, and J. Han, "Towards robust indexing for ranked queries," in *VLDB*, 2006.

[62] S. Har-Peled, "On the expected complexity of random convex hulls," *arXiv preprint arXiv:1111.5340*, 2011.

[63] J. L. Bentley, F. P. Preparata, and M. G. Faust, "Approximation algorithms for convex hulls," *Communications of the ACM*, vol. 25, no. 1, pp. 64–68, 1982.

[64] S.-Y. Ihm, K.-E. Lee, A. Nasridinov, J.-S. Heo, and Y.-H. Park, "Approximate convex skyline: a partitioned layer-based index for efficient processing top-k queries," *KBS*, 2014.

[65] C.-Y. Chan, H. Jagadish, K.-L. Tan, A. K. Tung, and Z. Zhang, "Finding k-dominant skylines in high dimensional space," in *SIGMOD*, 2006.

[66] A. Vlachou and M. Vazirgiannis, "Ranking the sky: Discovering the importance of skyline points through subspace dominance relationships," *DKE*, vol. 69, no. 9, 2010.

[67] S. Borzsony, D. Kossmann, and K. Stocker, "The skyline operator," in *ICDE*, 2001.

[68] D. Nanongkai, A. D. Sarma, A. Lall, R. J. Lipton, and J. Xu, "Regret-minimizing representative databases," *VLDB*, vol. 3, 2010.

[69] S. Chester, A. Thomo, S. Venkatesh, and S. Whitesides, "Computing k-regret minimizing sets," *VLDB*, vol. 7, no. 5, 2014.

[70] S. J. Russell, P. Norvig, J. F. Canny, J. M. Malik, and D. D. Edwards, *Artificial intelligence: a modern approach*. Prentice hall Upper Saddle River, 2003, vol. 2.

[71] U. Feige, "A threshold of ln n for approximating set cover," *J. ACM*, 1998.

[72] V. Chvatal, "A greedy heuristic for the set-covering problem," *Mathematics of operations research*, vol. 4, no. 3, pp. 233–235, 1979.

[73] G. B. Dantzig, *Linear programming and extensions*. Princeton university press, 1998.

[74] W. Weil and J. Wieacker, "Stochastic geometry, handbook of convex geometry, vol. a, b, 1391–1438," 1993.

[75] P. Nobili and A. Sassano, "A separation routine for the set covering polytope," in *IPCO*, 1992.

[76] S. G. Kobourov, "Force-directed drawing algorithms," 2004.

[77] W. T. Tutte, "How to draw a graph," *Proc. London Math. Soc*, vol. 13, no. 3, pp. 743–768, 1963.

[78] N. Bruno, S. Chaudhuri, and L. Gravano, "Top-k selection queries over relational databases: Mapping strategies and performance evaluation," *TODS*, 2002.

[79] A. Marian, N. Bruno, and L. Gravano, "Evaluating top-k queries over web-accessible databases," *ACM Trans. Database Syst.*, vol. 29, no. 2, 2004.

[80] T. Kessler Faulkner, W. Brackenbury, and A. Lall, "k-regret queries with nonlinear utilities," *VLDB*, vol. 8, no. 13, 2015.

[81] S. Thirumuruganathan, N. Zhang, and G. Das, "Rank discovery from web databases," *VLDB*, 2013.

[82] P. B. Yale, *Geometry and symmetry*. Courier Corporation, 1968.

[83] K. C.-C. Chang and S.-w. Hwang, "Minimal probing: supporting expensive predicates for top-k queries," in *SIGMOD*. ACM, 2002.

[84] A. Dasgupta, X. Jin, B. Jewell, N. Zhang, and G. Das, "Unbiased estimation of size and other aggregates over hidden web databases," in *SIGMOD*, 2010.

[85] W. Liu, S. Thirumuruganathan, N. Zhang, and G. Das, "Aggregate estimation over dynamic hidden web databases," *VLDB*, 2014.

[86] A. Asudeh, S. Thirumuruganathan, N. Zhang, and G. Das, "Discovering the skyline of web databases," *VLDB*, 2016.

[87] S. Thirumuruganathan, N. Zhang, and G. Das, "Breaking the top-k barrier of hidden web databases," in *ICDE*. IEEE, 2013.