Accelerated Algorithms for α-Happiness Query

Min Xie(✉)

Shenzhen Institute of Computing Sciences, Shenzhen University, Shenzhen, China
xiemin@sics.ac.cn

Abstract. Extracting a good representative subset of tuples that meets the user's needs from a large database is an important problem in multi-criteria decision making. Many queries have been proposed for this purpose, including the top-k query and the skyline query. Unfortunately, these traditional queries either ask the user to specify their needs explicitly or overwhelm users with a large output size. Recently, an α-*happiness* query was proposed, which overcomes the deficiencies of existing queries: users do not need to specify any preference, while they can obtain a small set of tuples such that users are *happy* with the results, i.e., their favorite tuples in the returned subset is guaranteed to be not much worse than their favorite tuples in the whole database. In this paper, we study the α-happiness query. Inspired by the techniques of incremental convex hull computation, we develop two accelerated algorithms, which maintain useful information to avoid redundant computation, in both 2-dimensional and d-dimensional space ($d > 2$). We performed extensive experiments, comparing against the best-known method under various settings on both real and synthetic datasets. Our superiority is demonstrated: we can achieve up to two orders and 7 times of improvements in execution times in 2-dimensional and d-dimensional space, respectively.

Keywords: α-happiness · Incremental convex hull · Decision making

1 Introduction

Nowadays, a database system usually contains millions of tuples and end users may only want to find those tuples that fit their needs. This problem is known as *multi-criteria decision making* [5,18,19], and various queries were proposed to obtain a small representative subset of tuples without asking the user to scan the whole database. An example is the traditional top-k query [18,19], where a user has to provide her preference function, called the *utility function*. Based on the user's utility function, the *utility* of each tuple for this user can be computed, where a higher utility means that the tuple is more preferred. Finally, the best k tuples with the highest utilities are returned. Unfortunately, requiring the user to provide the exact utility function is too restrictive in many scenarios. In this case, the skyline query can be applied [5], which adopts the "dominance" concept. A tuple p is said to *dominate* another tuple q if p is not worse than q on each

B. Li et al. (Eds.): APWeb-WAIM 2022, LNCS 13422, pp. 53–68, 2023.
https://doi.org/10.1007/978-3-031-25198-6_5

attribute and p is better than q on at least one attribute. Intuitively, p will have a higher utility than q w.r.t. *all monotonic* utility functions. Tuples which are not dominated by any other tuples are returned in the skyline query. However, since there is no constraint on the output size of a skyline query, a skyline query can overwhelm the user with many results (at worst the whole database). Motivated by this, a query called α-*happiness* query was studied recently in [27] to overcome the deficiencies of both the top-k query (which requires the users to specify their utility functions) and the skyline query (which might have a large output size).

Informally, an α-happiness query computes a set of tuples, with size as small as possible, that makes the users *happy* where the degree of happiness is quantified as the *happiness ratio* of the user. Specifically, given a set of tuples, a user is $x\%$ happy with the set if the highest utility of tuples in the set is at least $x\%$ of the highest utility of all tuples in the whole database. In this case, we say that the happiness ratio of the user is $x\%$. Clearly, the happiness ratio is a value from 0 to 1. The larger the happiness ratio, the happier the user. The α-happiness query guarantees the happiness ratio of an end user is at least α. In practice, more tuples have to be returned to guarantee a higher happiness level, as expected. However, with more tuples returned, users have to spend more effort to examine the output, which is not desirable as they did in the skyline query. Hence, we want the solution to be as small as possible, to ensure the given happiness level.

Consider a car database application. Assume that Alice wants to buy a car from the car database where each car is described by two attributes, namely horse power (HP) and miles per gallon (MPG). To help Alice for finding her desired car, Alice can specify an α value, which represents the least happiness level she expects. In practice, Alice can set α to be 0.9, indicating that she wants a set of cars whose highest utility is at least 90% of the highest utility of all cars in the database. Then, we execute the α-happiness query, which returns a small set of cars from the database, hoping that Alice will be satisfied (since the happiness ratio of Alice is at least α, as specified). However, if Alice is not satisfied with those cars, she can increase the value of α, and execute the α-happiness query again to obtain more cars with better quality to ensure a higher α.

Although it is NP-hard to solve the α-happiness query [27], various practical algorithms were proposed in the literature. The best-known previous approach for the α-happiness query is Cone-Greedy [27]. However, when we experimentally evaluated Cone-Greedy, its execution time is unnecessarily long. This is because Cone-Greedy did not keep sufficient information and thus, might perform redundant computation, resulting in a long query time. The situation is even worse when the user wants to perform multiple α-happiness queries with different values of α on the same database, which is common in reality since users might adjust the value of α to obtain more/less tuples to fit their needs. Motivated by this, we propose two novel approaches which accelerate Cone-Greedy in both 2-dimensional and d-dimensional space ($d > 2$). Our algorithms are inspired by the incremental convex hull computation in computational geometry, and different from Cone-Greedy, they effectively maintain the information needed during the computation and re-use them when necessary. Our experiments show that our algorithms substantially outperform Cone-Greedy in execution time. Our major contributions are summarized as follows:

- To the best of our knowledge, we are the first one who connect the α-happiness query with the problem of incremental convex hull computation.
- We propose a 2-dimensional algorithm, called 2D-CH, for solving the α-happiness query exactly when each tuple is described by two attributes.
- In d-dimensional space, we propose a novel algorithm for the α-happiness query. In particular, our algorithm effectively maintain useful information, which can be re-used repeatedly, speeding up the overall query.
- We present extensive experiments on both synthetic and real datasets. Our evaluation shows that the proposed algorithms outperform the competitors substantially. Under some practical settings, our 2-dimensional algorithms achieve up to two orders improvement in running time, while our d-dimensional algorithms are around 7 times faster than the exiting ones.

Organization. The rest paper is organized as follows. We discuss the related works in Sect. 2. The α-happiness query and the solution overview are formally introduced in Sect. 3. In Sect. 4, we present the exact algorithm for the α-happiness query in 2-dimensional space and its d-dimensional extension. Finally, experiments are presented in Sect. 5 and Sect. 6 concludes this paper.

2 Related Work

Traditional queries for multi-criteria decision making include top-k queries and skyline queries. In top-k queries [10,13,19,21,28], a concrete utility function is given. Based on the function, the k tuples with the highest utilities are returned. However, it is sometimes difficult to obtain the exact utility function in practice. Alternatively, skyline queries [5] can be applied. However, it is found that the skyline query has a large output size, which is not desirable. Although there are some variants of skyline queries [9,15,20] which alleviate this drawback by introducing an integer k, which controls the output size, it is difficult for these queries to provide theoretical guarantee without knowing the exact utility function.

The α-happiness query was first considered in [2,12], called the *min-size regret query*, and it is later formalized by Xie et al. in [27]. Specifically, given a real number α, an α-happiness query minimizes the output size while keeping the users at least α happy (i.e., the minimum happiness ratio is at least α). The α-happiness query can be considered as a dual version of the well-known k-regret query [15,24,26], which, given an integer k, returns a set S of at most k tuples such that the "utility difference" between the maximum utility of S and the whole dataset D is minimized. See [25] for a recent survey. It has been shown that both the α-happiness query and the k-regret query are NP-hard problems [2,6,7,27].

Algorithms were proposed to get a solution for the α-happiness query, categorized as follows. (1) *ϵ-kernel based.* The first approach formulated it as the well-known *ϵ-kernel* problem [1] and several algorithms [2,6] were proposed to obtain a good approximation. (2) *Space partitioning based.* [2,3] discretized the function space and formulated the α-happiness query as a hitting set problem (or a set cover problem), which provides user-controlled approximations on happiness ratios and output sizes. (3) *Hybrid.* [12] proposed an algorithm which

Table 1. Car database and utilities

Car	HP	MPG	$f_{0.4,0.6}(p)$	$f_{0.2,0.8}(p)$	$f_{0.7,0.3}(p)$	Car	HP	MPG	$f_{0.4,0.6}(p)$	$f_{0.2,0.8}(p)$	$f_{0.7,0.3}(p)$
p_1	0.2	1	0.68	0.84	0.44	p_4	1	0.2	0.52	0.36	0.76
p_2	0.6	0.9	0.78	0.84	0.69	p_5	0.35	0.2	0.26	0.23	0.305
p_3	0.9	0.6	0.72	0.66	0.81	p_6	0.3	0.6	0.48	0.54	0.39

combines the ϵ-kernel and hitting set approaches, improving the efficiency of the existing algorithms. (4) *Geometric-based.* Xie et al. [27] provided a novel geometric interpretation of the α-happiness query, based on which they proposed the state-of-the-art algorithm, denoted by Cone-Greedy for solving the problem. According to the experiments in [27], Cone-Greedy outperforms the existing methods in both output sizes and execution times. We use it as the baseline in our experiments.

Compared with the existing studies [2,3,6,12,27], we utilize the techniques in incremental convex hull construction and propose accelerated algorithms. In particular, we maintain useful information so that intermediate results can be re-used repeatedly to avoid redundant computation. Our algorithms performs particularly well when the users execute multiple α-happiness queries on the same dataset. Our experimental superiority will be demonstrated in Sect. 5.

3 Problem and Overview

The input to our problem is a set D of n tuples (i.e., $|D| = n$) in a d-dimensional space (i.e., each tuple in D is described by d attributes). In this paper, we assume that d is a fixed constant. In the following, we first introduce the terminologies and the background. Then, we give an overview of our solution.

3.1 Preliminary

We use the same terminology as in [27]. We denote the i-th dimensional value of a tuple p in D by $p[i]$ where $i \in [1, d]$. In the rest paper, we also call each tuple as a point in a d-dimensional space. Without loss of generality, we assume that each dimension is normalized to $(0, 1]$, such that there exists a point p in D and $p[i] = 1$ for each $i \in [1, d]$ and a larger value on each dimension is more preferable to all users. Recall that in the car database, each car is associated with 2 attributes, HP and MPG; in the example shown in Table 1, the car database $D = \{p_1, p_2, p_3, p_4, p_5, p_6\}$ consists of 6 cars with normalized attribute values.

Following the assumption in existing studies [14,15,24,26,27], we assume that user's happiness is measured by an unknown *utility function*, which can be regarded as a mapping $f : \mathbb{R}^d_+ \to \mathbb{R}_+$. The *utility* of a point p w.r.t. f is denoted by $f(p)$. A user wants a point which maximizes the utility w.r.t. his/her utility function. Given a utility function f and $S \subseteq D$, we define the *maximum utility of* S *w.r.t.* f, denoted by $U_{max}(S, f)$, to be $\max_{p \in S} f(p)$.

In the following, we introduce two important terms, namely the *function-wise ratio* (*happiness ratio*) and the *minimum happiness ratio*.

Definition 1. *Given a set $S \subseteq D$ and a utility function f, the function-wise ratio of S w.r.t. f, denoted by* $\mathsf{fRatio}(S, f)$*, is defined to be* $\frac{U_{max}(S,f)}{U_{max}(D,f)}$.

Clearly, the value of a function-wise ratio ranges from 0 to 1 since $U_{max}(S, f) \leq U_{max}(D, f)$. Intuitively, when the maximum utility of S is closer to the maximum utility of D, the function-wise ratio of S w.r.t. the user's utility function becomes larger, which indicates that the user feels more satisfied with S. In this sense, the function-wise ratio is also called the *happiness ratio*.

As discussed in Sect. 1, it is difficult to know the user's exact utility function. Thus, we assume that all users' utility functions belong to a function class, denoted by $\mathcal{FC}$. A function class is defined to be a set of functions which share some common characteristics, e.g., the class of *linear utility functions* [15]. Given the function class $\mathcal{FC}$, the *minimum happiness ratio* of a set S can be regarded as the worst-case function-wise ratio w.r.t. a utility function in $\mathcal{FC}$.

Definition 2. *Given a set $S \subseteq D$ and a function class $\mathcal{FC}$, the minimum happiness ratio of S over $\mathcal{FC}$ is defined to be* $\inf_{f \in \mathcal{FC}} \mathsf{fRatio}(S, f)$.

Example 1. To illustrate, assume that $\mathcal{FC}$ has 3 utility functions, namely $f_{0.4,0.6}$, $f_{0.2,0.8}$ and $f_{0.7,0.3}$ where $f_{a,b}(p) = a \times p[1] + b \times p[2]$. Consider p_1 in Table 1. Its utility w.r.t. $f_{0.4,0.6}$ is $f_{0.4,0.6}(p_1) = 0.4 \times 0.2 + 0.6 \times 1 = 0.68$. The utilities of other points in D are computed similarly. Given $S = \{p_1, p_4\}$, the maximum utility of S w.r.t. $f_{0.4,0.6}$ is $U_{\max}(S, f_{0.4,0.6}) = \max_{p \in S} f_{0.4,0.6}(p) = f_{0.4,0.6}(p_1) = 0.68$. Similarly, $U_{\max}(D, f_{0.4,0.6})$ is 0.78. Then, $\mathsf{fRatio}(S, f_{0.4,0.6}) = \frac{U_{\max}(S, f_{0.4,0.6})}{U_{\max}(D, f_{0.4,0.6})} = \frac{0.68}{0.78} = 0.872$. *Similarly,* $\mathsf{fRatio}(S, f_{0.2,0.8}) = 1$ *and* $\mathsf{fRatio}(S, f_{0.7,0.3}) = 0.938$. *The minimum happiness ratio of S over $\mathcal{FC}$ is* $\min\{0.872, 1, 0.938\} = 0.872$. □

Same as [2,12,14,15], we focus on the class of *linear utility functions*, denoted by $\mathcal{L}$, due to its popularity in modeling user preferences and assume each function in $\mathcal{L}$ is equally probable to be used by users. Other classes and distributions of utility functions are considered in [8,17,27] and are not our focus.

Specifically, we assume that each linear utility function f in $\mathcal{L}$ is associated with a d-dimensional non-negative *utility vector* u where $u[i]$ denotes the importance of the i-th dimension in user's preference. Mathematically, we can write: $f(p) = \sum_{i=1}^{d} u[i]p[i] = u \cdot p$. Without loss of generality, we assume $\sum_{i=1}^{d} u[i] = 1$. Thus, $\mathcal{L} = \{f |\ f(p) = u \cdot p$ where $u \in \mathbb{R}_+^d$ and $\sum_{i=1}^{d} u[i] = 1\}$. When it is clear, we refer each f in $\mathcal{L}$ by its utility vector u. Let $\mathsf{minHap}(S)$ be the minimum happiness ratio of S over $\mathcal{L}$. The α-happiness query is stated as follows.

Problem 1. Given a real number $\alpha \in [0, 1]$, the α-happiness query returns a set $S \subseteq D$ with $\mathsf{minHap}(S) \geq \alpha$ such that the size of S, i.e., $|S|$, is minimized.

When there are multiple sets with the minimum size, an α-happiness query simply returns one of them. As stated in Sect. 1, the α-happiness query takes the

advantages of both the top-k query and the skyline query: same as the skyline query, a user does not need to specify any preference and meanwhile, it returns a set with size as small as possible. Recall that $\mathsf{minHap}(S)$ is defined to be the worst-case happiness ratio w.r.t. *any* utility function in $\mathcal{L}$. If $\mathsf{minHap}(S) \geq \alpha$, for *each* user, s/he will be at least α happy with S no matter which function s/he uses from $\mathcal{L}$. The α-happiness query is an NP-hard problem [2,6,7].

3.2 Geometric Interpretation

Note that $\mathcal{L}$ contains an *infinite* number of utility functions. Thus, it is not easy to compute $\mathsf{minHap}(S)$ for a given S directly according to Definition 2. To compute $\mathsf{minHap}(S)$ tractably, Xie et al. [27] interprets the problem geometrically.

We first introduce some geometric concepts. For each point $p \in D$, we define the *orthotope set* of p, denoted by $\mathsf{Orth}(p)$, to be a set of 2^d d-dimensional points constructed by $\{0, p[1]\} \times ... \times \{0, p[d]\}$. That is, for each $i \in [1, d]$, the i-dimensional value of a point in $\mathsf{Orth}(p)$ is equal to either 0 or $p[i]$. Given a set $S \subseteq D$, we define the orthotope set of S, denoted by $\mathsf{Orth}(S)$, to be $\bigcup_{p \in S} \mathsf{Orth}(p)$. Given a set $S \subseteq D$, let $\mathsf{Conv}(S)$ be the *convex hull* (the smallest convex set) of the orthotope set of S [16]. Moreover, a point $p \in \mathsf{Conv}(S)$ is said to be a *vertex* of $\mathsf{Conv}(S)$ if p is not in the convex hull of the other points in $\mathsf{Orth}(S)$. A *facet* of a convex hull is a bounded flat surface that forms a part of the boundary of the convex hull. We denote a facet by the set of points forming it.

Example 2. To illustrate, consider Table 1 where $D = \{p_1, p_2, p_3, p_4, p_5, p_6\}$. *For the ease of presentation, we first visualize* D *in Fig. 1 where the* X_1 *and* X_2 *coordinate represent HP and MPG, respectively. The points in* $\mathsf{Orth}(p_2) = \{p_2, p_2', p_2'', O\}$ *are shown in Fig. 1 where* $p_2' = (0, p_2[2])$, $p_2'' = (p_2[1], 0)$ *and* O *is the origin. Similarly,* $\mathsf{Orth}(p_3)$ *is shown in the same figure.*

Given $S = \{p_2, p_3\}$, *we define* $\mathsf{Orth}(S)$ *to be* $\mathsf{Orth}(p_2) \cup \mathsf{Orth}(p_3)$. *Then, the convex hull* $\mathsf{Conv}(S)$ *is shown in Fig. 2. There are 5 vertices in* $\mathsf{Conv}(S)$, *namely* O, p_2', p_2, p_3 *and* p_3'' *(labeled in Fig. 1), each of which is not in the convex hull of the other points in* $\mathsf{Orth}(S)$. $\{p_2, p_3\}$ *is a facet of* $\mathsf{Conv}(S)$. □

Given a real value $\alpha \in [0, 1]$, we define the *α-shrunk set* of D, denoted by D'_α, to be $\{p'_\alpha | p'_\alpha = \alpha p, \forall p \in D\}$ where p'_α is a proportionally shrunk point of p. When α is clear, we denote D'_α by D' and denote a point in D' by p'. Unless stated explicitly, we stick to the above notations in the rest of this paper.

Given two point sets, say S and T, if for each $p \in S$, p is inside $\mathsf{Conv}(T)$, we say $\mathsf{Conv}(T)$ *covers* $\mathsf{Conv}(S)$ since $\mathsf{Conv}(S)$ is totally contained inside $\mathsf{Conv}(T)$.

Example 3. Let $\alpha = 0.9$. *The α-shrunk set* D' *(shown in white dots) of* D *(shown in black dots) is drawn in Fig. 2 where each point in* D' *is a proportionally scaled point in* D. *Given* $S = \{p_2, p_3\}$, *it is easy to observe from the figure that* $\mathsf{Conv}(S)$ *covers* $\mathsf{Conv}(D')$ *since* $\mathsf{Conv}(D')$ *is totally contained inside* $\mathsf{Conv}(S)$. □

Xie et al. [27] shows that the α-happiness from the geometric perspective.

Lemma 1 ([27]). *Given* $S \subseteq D$ *and* $\alpha \in [0, 1]$, S *is a feasible set of the α-happiness query if* $\mathsf{Conv}(S)$ *covers* $\mathsf{Conv}(D')$, *where* D' *is the α-shrunk set of* D.

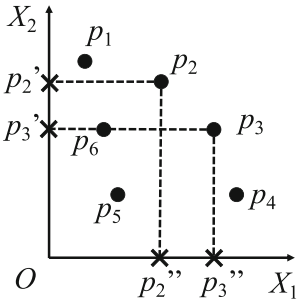

Fig. 1. Orthotope set

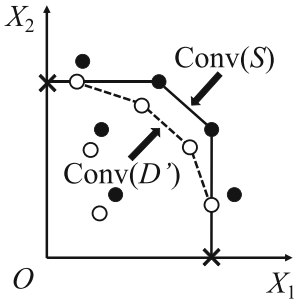

Fig. 2. Convex hull

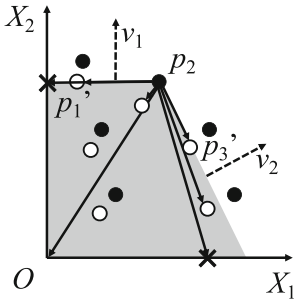

Fig. 3. Conical hull

3.3 Solution Overview

According to Lemma 1, we can solve the α-happiness query by finding a minimum size set S such that $\mathsf{Conv}(S)$ covers $\mathsf{Conv}(D')$. To find such S, the Cone-Greedy algorithm in [27] has the following two major steps (note that the correctness of the procedure below is proven in [27] and we omit it here for lack of space):

1. (Step 1) For each p in D, it computes a function set $\mathcal{F}_p$, whose utilities are maximized by p over points in D', i.e., $\mathcal{F}_p = \{f \in \mathcal{L} \mid f(p) \geq f(p')\ \forall p' \in D'\}$.
2. (Step 2) If finds a set S of tuples from D such that $\bigcup_{p \in S} \mathcal{F}_p = \mathcal{L}$.

Step 2 of Cone-Greedy is reduced to the well-known set-cover problem in [27], where the greedy algorithm is adopted and it gives theoretical guarantees on the output size. We adopt the same approach for Step 2 in this paper. Interested readers can find more details in [27], and we focus on Step 1 next.

Note that when performing Step 1 in Cone-Greedy, redundant operations might be done when computing $\mathcal{F}_p$ for distinct points in D. This is inefficient. In this paper, we adopt a novel strategy for computing $\mathcal{F}_p$, which maintains useful information for all points in D, so that we can re-use those information as much as possible. In the following, we briefly review the procedure in Cone-Greedy and explain why it is inefficient. In Sect. 4, we give our advanced procedures.

Computing $\mathcal{F}_p$ in Cone-Greedy. We first define "conical hull". Given a point p in D, let $V_p = \{t - p|$ for each vertex t of $\mathsf{Conv}(D')\}$. Then we define a *conical hull* of p, denoted by $\mathsf{Cone}(p)$, to be $\mathsf{Cone}(p) = \{q \in \mathbb{R}^d | \ q = p + \sum_{v \in V_p} wv$ where $w \geq 0\}$. Intuitively, $\mathsf{Cone}(p)$ can be regarded as a *convex cone* with apex at p. The boundaries of $\mathsf{Cone}(p)$ are *unbounded facets*, each of which is enclosed by some vectors in V_p and is a flat surface that forms the boundary of $\mathsf{Cone}(p)$.

In geometry, each facet of a conical hull is contained by a *unique* hyperplane (i.e., a subspace of dimensionality $d-1$). Then, for each facet F of $\mathsf{Cone}(p)$, we define an *extreme vector* to be the unit vector (pointing out) perpendicular to the hyperplane containing F. Denote the set of *extreme vectors* of p by $\mathsf{Ext}(p)$.

Example 4. Consider the point p_2 in Fig. 3 as an example. We draw the vectors in $V_{p_2} = \{t - p_2 |$ for each vertex t of $\mathsf{Conv}(D')\}$ in solid arrows. It is constructed by creating a vector for each vertex of $\mathsf{Conv}(D')$. The conical hull $\mathsf{Cone}(p_2)$ is showed in the shaded region in the figure, which is the set of all vectors with the form $p_2 + \sum_{v \in V_{p_2}} wv$ where $w \geq 0$. In this 2-dimensional example, the boundaries of $\mathsf{Cone}(p_2)$ are two unbounded facets, i.e., the rays shooting from p_2 to p'_1 and from p_2 to p'_3. The extreme vectors of p_2 are dashed arrows $\mathsf{Ext}(p_2) = \{v_1, v_2\}$, each of which is perpendicular to a boundary facet of $\mathsf{Cone}(p_2)$. □

Based on the above concepts, Xie et al. [27] define the function set $\mathcal{F}_p$, which is a set of utility functions whose utilities are maximized by p, as follows.

Definition 3. *Given p in D and its $\mathsf{Ext}(p)$, the function set of p, denoted by $\mathcal{F}_p$, is defined to be $\{f \in \mathcal{FC} |\ f(p) = u \cdot p$ and $u = \sum_{v \in \mathsf{Ext}(p)} wv$ where $w \geq 0\}$.*

According to [27], $\mathcal{F}_p$ is uniquely defined by the extreme vectors in $\mathsf{Ext}(p)$. Thus, Cone-Greedy obtain $\mathcal{F}_p$ by computing $\mathsf{Ext}(p)$ as follows:

1. It first computes the vertices in $\mathsf{Conv}(D')$;
2. For each p in D, it computes the set $V_p = \{t - p |$ for each vertex t of $\mathsf{Conv}(D')\}$ and the corresponding conical hull $\mathsf{Cone}(p)$; and
3. It obtains the extreme vectors $\mathsf{Ext}(p)$ based on boundary facets of $\mathsf{Cone}(p)$.

Note that in Cone-Greedy, although the vertices in $\mathsf{Conv}(D')$ are used for all points in D, the vector set V_p is different for each distinct p in D. Therefore, the conical hull $\mathsf{Cone}(p)$ will be computed *independently* for distinct p in D, which might incur redundant computation, since the common information $\mathsf{Conv}(D')$ is not well utilized. In Sect. 4, we show our alternative ways for computing $\mathsf{Ext}(p)$, by maintaining useful information to avoid such redundant computation. Our algorithms are especially efficient when the user wants to execute multiple α-happiness queries on the same D with different values of α. Our experiments show that we are more efficient than the counterpart in Cone-Greedy.

4 Algorithm

4.1 Conceptual Idea

Our algorithm is inspired by the incremental approach of convex hull computation [11]. Specifically, in incremental convex hull computation, a convex hull is built by inserting points iteratively. At the i-th iteration, we have the convex hull of the first i points, and we need to modify this convex hull to include the i-th point. For example in Fig. 4, if we are inserting p_2 to $\mathsf{Conv}(D')$ (shown in solid lines), the convex hull is updated and the vertices become $\{b_1, p_2, p'_3, p'_4, b_2, O\}$. To update $\mathsf{Conv}(D')$, new facets (e.g., $\{p_2, p'_3\}$, shown in dashed lines) are created, and old facets are removed (e.g., $\{p'_1, p'_2\}$ and $\{p'_2, p'_3\}$). It is not hard to observe that the newly created facets indeed give us the desired extreme vectors $\mathsf{Ext}(p)$, since each extreme vector is perpendicular to exactly one newly created

facet (i.e., it is perpendicular to the unique hyperplane containing that facet). For example in Fig. 4, v_2, an extreme vector of p_2, is perpendicular to the newly created facet $\{p_2, p_3'\}$. Motivated by this, we can compute the desired $\mathsf{Ext}(p)$ for each p in D, by adapting the techniques of incremental convex hull computation, pretending that we are inserting p to the convex hull $\mathsf{Conv}(D')$.

4.2 Two-Dimensional Case: 2D-CH

In 2-dimensional space, the vertices (excluding the origin O) of the convex hull $\mathsf{Conv}(D')$ can be organized in a clockwise manner, say $t_1, t_2, \ldots, t_k$, where $\{t_i, t_{i+1}\}$ ($i \in [1, k-1]$) is a facet. For example, in Fig. 4, vertices of $\mathsf{Conv}(D')$ can be organized in order: b_1, p_1', p_2', p_3', p_4', b_2, where b_1 and b_2 are two orthotope points in $\mathsf{Orth}(D')$. $\{p_2', p_3'\}$ is facet of $\mathsf{Conv}(D')$. We store vertices of $\mathsf{Conv}(D')$ clockwise in a doubly-linked list so that we can create new facets efficiently.

Specifically, our 2-dimensional algorithm, called 2D-CH, is proposed by adopting the following strategy for computing the extreme vectors $\mathsf{Ext}(p)$ for p:

1. We first compute the convex hull $\mathsf{Conv}(D')$ and maintain its vertices in a doubly-linked list for efficient facet traversal for all points in D;
2. For each p in D that is not contained inside $\mathsf{Conv}(D')$, we compute the new facets, by pretending that we are inserting p to $\mathsf{Conv}(D')$ (see details below);
3. For each newly created facet, we obtain a desired extreme vector in $\mathsf{Ext}(p)$, which is the unique vector perpendicular to the new facet.

To insert a point p to $\mathsf{Conv}(D')$, we need to determine the correct positions for constructing the new facets. For example, in Fig. 4, p_3' is the desired position, and a new facet is created by connecting p_2 and p_3'. To determine such positions, we need the notion of "visibility". Formally, given a point p and a facet $\{t_i, t_{i+1}\}$ of $\mathsf{Conv}(D')$, $\{t_i, t_{i+1}\}$ is visible to p if p is *above* the unique hyperplane containing $\{t_i, t_{i+1}\}$. The following lemma (proof is intuitive and is omitted) tells us how to determine the correct positions with the notion of "visibility".

Lemma 2. *Given point p and two adjacent facets of $\mathsf{Conv}(D')$, say $F_1 = \{t_{i-1}, t_i\}$ and $F_2 = \{t_i, t_{i+1}\}$, when inserting p to $\mathsf{Conv}(D')$, we create a new facet by connecting p and t_i iff one facet in $\{F_1, F_2\}$ is visible to p and the other is not.*

For example in Fig. 4, $\{p_2', p_3'\}$ is visible to p_2, while $\{p_3', p_4'\}$ is not. To insert p_2 to $\mathsf{Conv}(D')$, we create a new facet by connecting p_2 and p_3' by Lemma 2. Since we maintain vertices of $\mathsf{Conv}(D')$ in a doubly-linked list, the correct position for creating facets can be found efficiently by binary search in the list.

After obtaining the new facets, the extreme vector set construction is straightforward. Note that in 2-dimensional space, there are exactly two extreme vectors for each p. Therefore, the corresponding function set $\mathcal{F}_p$ can be concisely represented by an *angle interval*. Specifically, we define the *angle of a vector v* in 2-dimensional spaces as the angle between the vector Ov and the y-axis, denoted by $\mathsf{Ang}(v)$. Given $\mathsf{Ext}(p) = \{v_1, v_2\}$ of a point p, we define the *angle interval* of p

to be $[\mathsf{Ang}(v_1), \mathsf{Ang}(v_2)]$. Then, it is easy to show that finding a set S such that $\bigcup_{p \in S} \mathcal{F}_p = \mathcal{L}$ is equivalent to finding a set S whose angle intervals covers $[0, \frac{\pi}{2}]$, where the latter problem is the *interval cover problem* [4]. We then employ the popular greedy strategy to solve the interval cover problem optimally.

Example 5. Consider p_2 in Fig. 4 where $\mathsf{Ext}(p_2) = \{v_1, v_2\}$. Since $\mathsf{Ang}(v_1) = 0$ and $\mathsf{Ang}(v_2) = 1.04$, we represent the function set $\mathcal{F}_{p_2}$ as an angle interval $[0, 1.04]$ (labeled in the figure). Similarly, we can compute the angle intervals for other points in D. By the greedy strategy, we find that the angle intervals of p_2 and p_3 covers the entire $[0, \frac{\pi}{2}]$, which gives us the desired set $S = \{p_2, p_3\}$. □

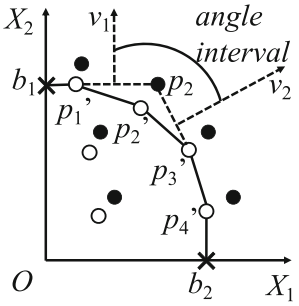

Fig. 4. 2D case

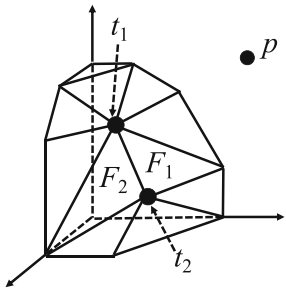

Fig. 5. 3D case

4.3 High-Dimensional Case: HD-CH

The problem is more complicated in the higher-dimensional case, since there is no natural order in the facets of a convex hull and each facet can have multiple adjacent facets (unlike exactly two adjacent facets in the 2-dimensional case).

To extend our algorithm to the high-dimensional case, we define the following notions in a high-dimensional convex hull. The boundaries of a facet are called *ridges*. Intuitively, the ridge signifies the adjacency of two neighbouring facets. For example, the ridges in a 2-dimensional space are points and the ridges in a 3-dimensional space are edges (i.e., the line segment jointed by two points). Given a point p, a ridge is called *horizon ridge* of p if it signifies the adjacency of a visible facet and an invisible facet of p. Intuitively, a horizon ridge indicates the maximum visible region from p to the convex hull. For example in Fig. 5, if F_1 is visible to p and F_2 is not visible to p, the ridge (i.e., edge in this case) $\{t_1, t_2\}$, which signifies the adjacency of F_1 and F_2, is a horizon ridge of p. For each horizon edge, we define an extreme vector of p to be the unit vector perpendicular to the unique hyperplane containing p and the horizon ridge.

With the above definitions, our high-dimensional algorithm, denoted as HD-CH, computes the extreme vector set $\mathsf{Ext}(p)$ as follows:

1. It first computes the convex hull $\mathsf{Conv}(D')$;
2. For each p in D, we maintain its visible facets in a queue $\mathcal{Q}$ and horizon ridges in a set $\mathcal{H}$. Initially, $\mathcal{H}$ is empty and we obtain the first facet F in $\mathcal{Q}$ by facet traversal on $\mathsf{Conv}(D')$. Neighboring facets of F is marked as "unchecked" ;
3. When there is a facet F in $\mathcal{Q}$ with unchecked neighboring facets, we pop F from $\mathcal{Q}$ and check its neighboring facets. Specifically, for each visible neighboring facet, we add it to $\mathcal{Q}$ for later processing; and for each invisible neighboring facet, we obtain a horizon ridge for p and it is inserted to $\mathcal{H}$;
4. Finally, for each horizon ridge in $\mathcal{H}$, we get an extreme vector (i.e., the unit vector perpendicular to the hyperplane containing p and the horizon ridge).

After obtaining the extreme vector set $\mathsf{Ext}(p)$, we adopt the same strategy as Cone-Greedy for constructing the solution S. Note that HD-CH enjoys the same theoretical guarantee on the output size as Cone-Greedy by similar analysis. Interested readers can find more details in [27] and we omit them here.

4.4 Discussion

Compared with the best-known previous approach Cone-Greedy, our 2D-CH and HD-CH algorithms mainly differ in the procedure of constructing the extreme vector set $\mathsf{Ext}(p)$, by employing incremental computation on the convex hull $\mathsf{Conv}(D')$. Note that $\mathsf{Conv}(D')$ is a α-shrunk convex hull of $\mathsf{Conv}(D)$. Therefore, we can compute $\mathsf{Conv}(D)$ once and use it for α-happiness queries with different values of α, by properly scaling $\mathsf{Conv}(D)$. Moreover, given the convex hull $\mathsf{Conv}(D')$, we can use it for all points in D, for computing the desired function set $\mathcal{F}_p$ via facet traversal. In contrast, although Cone-Greedy also computes the vertices $\mathsf{Conv}(D')$ for all points in D, it constructs the conical hull $\mathsf{Cone}(p)$ independently for each p in D, resulting in a large overall execution time. Even worse, when the user wants to execute an α-happiness query with a different value of α on the same dataset, the conical hull $\mathsf{Cone}(p)$ has to be re-computed from scratch for all points in D, since the vector set $V_p = \{t - p |$ for each vertex t of $\mathsf{Conv}(D')\}$ is radically different under different values of α.

5 Experimental Evaluation

We conducted experiments on a machine with 3.20 GHz CPU and 8 GB RAM. All programs were implemented in C/C++. Most experimental settings follow those in [2,12,27]. Both *synthetic* and *real datasets* were used in our experiments.

We generated the widely used anti-correlated datasets by a dataset generator [5]. Unless stated explicitly, for each synthetic dataset, the number of tuples is set to be 100,000 (i.e., $n = 100{,}000$), the dimensionality is set to be 3 (i.e., $d = 3$) and α is set to be 0.99. Following existing studies, we used three real datasets in our experiments: the *Island* dataset [15,27], the *Household* dataset [26] and the *El Nino* dataset [2,7,27]. Island is 2-dimensional, containing 63,383 points, which characterize geographic positions. Household consists of 1,048,576

family tuples in US in 2012 where each family is described by three economic attributes. El Nino contains 178,080 tuples with four oceanographic attributes taken at the Pacific Ocean. For all datasets, each attribute is normalized to (0, 1].

We implemented our algorithms, 2D-CH and HD-CH, and two variants 2D-CH$_{reuse}$ and HD-CH$_{reuse}$, which pre-compute the vertices and convex hulls and re-use them under different values of α. Our algorithms are compared against the state-of-the-art algorithm, Cone-Greedy [27], for the α-happiness query. Note that although there are other algorithms proposed in the literature, [2,6, 12,15], they are shown to be worse than Cone-Greedy in [27] and thus, we only compared Cone-Greedy in the experiments for the ease of presentation. We used the same parameters reported in [27]. Unless specified explicitly, the performance of each algorithm is measured in terms of the *execution time*. Since 2D-CH and HD-CH only differ from Cone-Greedy in the way of computing the function sets, their outputs are the same and we omit them for lack of space.

In the following, we show the experiments on the synthetic and real datasets in Sect. 5.1 and Sect. 5.2. We summarize our findings in Sect. 5.3.

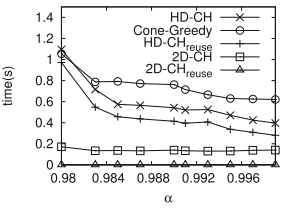

Fig. 6. 2D synthetic

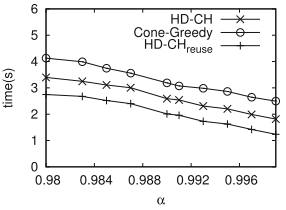

Fig. 7. 3D synthetic

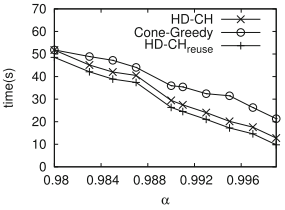

Fig. 8. 4D synthetic

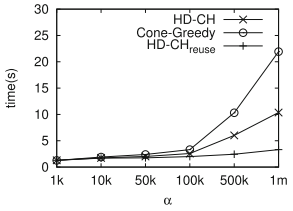

Fig. 9. Vary n

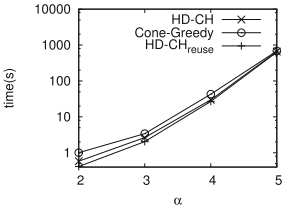

Fig. 10. Vary d

5.1 Results on Synthetic Datasets

In Fig. 6, we evaluated our 2-dimensional algorithms, 2D-CH and 2D-CH$_{reuse}$ on a $2d$ anti-correlated dataset. For completeness, we also include the d-dimensional algorithm, HD-CH and HD-CH$_{reuse}$, in the figure (however, their

performance will be analyzed in later experiments). As shown there, 2D-CH runs much faster than the other algorithms. In particular, it takes less than 0.2 s for all α and its running time is not sensitive to the value of α. This is because in a 2-dimensional dataset, there is an ordering on the vertices, and thus, constructing the functions sets on the α-shrunk convex hull $\mathsf{Conv}(D')$ can be done efficiently via binary search, which is not sensitive to α compared with the other methods. The performance of 2D-CH is further improved by 2D-CH$_{reuse}$, by pre-computing the vertices and re-using them for all points in D under different values of α. Note that Cone-Greedy is the slowest in most cases, e.g., 2D-CH (resp. 2D-CH$_{reuse}$) achieves 5 times (resp. two orders) of improvements in execution times compared with Cone-Greedy when $\alpha = 0.99$.

We proceed with the performance evaluation of our d-dimensional algorithms, HD-CH and HD-CH$_{reuse}$, on $3d$ and $4d$ anti-correlated datasets. The results are presented in Figs. 7 and 8. With the increasing value of α, all algorithms take less time to execute, in the cost of larger output sizes (not shown). This is because when α is large, the convex hull $\mathsf{Conv}(D')$ is "close" to $\mathsf{Conv}(D)$ and thus, each point in D can only "see" a small portion of $\mathsf{Conv}(D')$. Hence, it takes each point a shorter amount of time to construct the function set, which dominates the computational cost, but we need more points to cover the entire $\mathsf{Conv}(D')$. Nevertheless, Cone-Greedy still has the largest execution time, e.g., it takes Cone-Greedy 21 s on the 4-dimensional dataset when $\alpha = 0.999$, as opposed to 12 s by HD-CH. HD-CH$_{reuse}$ further improves the execution time of HD-CH by around 30%, This confirms our claim that our algorithms are especially efficient when the user wants to execute the α-happiness query on the same D with different values of α, since the convex hull can be efficiently pre-computed and used for different α-happiness queries.

We next evaluated the scalability of HD-CH and HD-CH$_{reuse}$, by varying the dimensionality d and the dataset size n in Figs. 9 and 10, where other parameters are fixed to the default setting stated at the beginning of this section. According to the results, we can see that our algorithm scales well w.r.t. both d and n. For example, on a large dataset with 1 million points, HD-CH$_{reuse}$ only takes 3 s to execute, 3 times and 7 times faster than HD-CH and Cone-Greedy, respectively. When the dimensionality is 4, the execution time of Cone-Greedy, HD-CH and HD-CH$_{reuse}$ is 42 s, 29 s and 26 s, respectively. In other words, HD-

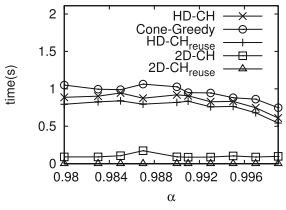

Fig. 11. Island

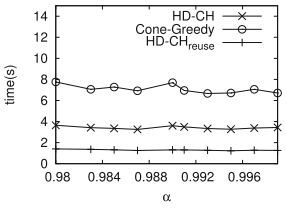

Fig. 12. Household

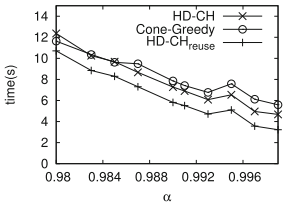

Fig. 13. El Nino

CH and HD-CH$_{reuse}$ outperform the state-of-the-art approach, w.r.t. both n and d, by accelerating the querying time.

5.2 Results on Real Datasets

In this section, we conducted experiments on three commonly used real datasets. The results are shown in Figs. 11, 12 and 13, respectively.

On the 2-dimensional Island dataset (Fig. 11), we plot the performance of both 2-dimensional and d-dimensional algorithms. Consistent to the performance on the synthetic datasets, the running times of our algorithms are much faster than the existing algorithms. For our d-dimensional algorithms HD-CH and HD-CH$_{reuse}$, they achieves 30% speedup against the state-of-the-art Cone-Greedy algorithm. When considering our 2D-CH and 2D-CH$_{reuse}$ algorithms, which are designed for the 2-dimensional case, the improvement of execution time is significant, e.g., one order and two orders of improvement when $\alpha = 0.999$.

The result on the Household dataset are similar and it is shown in Fig. 12. Note that due to the small skyline size on Household, the execution times of all algorithms are not sensitive to the value of α. In this scenario, HD-CH still outperforms Cone-Greedy, e.g., by reducing the average execution time from 7.2 s to 3.5 s. HD-CH$_{reuse}$ further improves the average execution time of HD-CH to 1.3 s, which clearly demonstrates that pre-computing the auxiliary structures is a promising way to speedup the query process. By maintaining intermediate information, we efficiently support the α-happiness queries for different values of α. Note that similar speedup cannot be achieved by the Cone-Greedy algorithm. Although it also computes the vertices of $\mathsf{Conv}(D')$ for all points in D, it has to construct the conical hull independently for each point in D, resulting in a large overall execution time.

Finally, consider the experiments on the El Nino dataset in Fig. 13. Similar to the previous experiments, the performance of Cone-Greedy is worse than that of HD-CH and HD-CH$_{reuse}$. When $\alpha = 0.999$, HD-CH$_{reuse}$ only spends half of the time compared with Cone-Greedy to obtain the desired solution.

5.3 Summary

The experiments on both real and synthetic datasets demonstrated our superiority over the best-known previous approach. We observe the following. (1) On the 2-dimensional datasets, 2D-CH and 2D-CH$_{reuse}$ are the best algorithms, by achieving up to two orders of improvement in execution time, compared with the state-of-the-art algorithm. (2) On the d-dimensional datasets, HD-CH and HD-CH$_{reuse}$ runs much faster than the competitor, e.g., on the Household dataset, the average execution times of HD-CH, HD-CH$_{reuse}$ and Cone-Greedy are 3.5 s, 1.3 s and 7.2 s, respectively. (3) Pre-computing the vertices and convex hulls is a promising way for reducing the query time, especially when the users want to execute multiple α-happiness queries on the same datasets. For example, when $n = 1,000,000$, it only takes HD-CH$_{reuse}$ 3 s to execute, 3 times faster than the HD-CH algorithm. (4) The scalability of our solutions is demonstrated, e.g.,

when varying the dimensionality or the dataset size, our algorithms are consistently faster than Cone-Greedy.

6 Conclusions

This paper proposed two accelerated algorithms for the α-happiness query. Compared with the existing methods, we maintain useful information to avoid redundant computation, accelerating the query process. Our algorithms are particularly good at executing the α-happiness queries with different values of α on the same dataset D. We conducted comprehensive experiments to verify the speedup of our algorithms, which achieve up to two orders of improvement in execution time, compared with the best-known approach. As for future research, we consider introducing user interaction [22–24] in α-happiness queries, so that we can further reduce the solution set size while guaranteeing the happiness ratio.

Acknowledgements. This work was supported by Longhua Science and Technology Innovation Bureau LHKJCXJCYJ202003 and Guangdong Basic and Applied Basic Research Foundation 2022A1515010120.

References

1. Agarwal, P., Har-Peled, S., Varadarajan, K.: Approximating extent measures of points. JACM **51**, 606–635 (2004)
2. Agarwal, P.K., Kumar, N., Sintos, S., Suri, S.: Efficient algorithms for k-regret minimizing sets. In: SEA (2017)
3. Asudeh, A., Nazi, A., Zhang, N., Das, G.: Efficient computation of regret-ratio minimizing set: a compact maxima representative. In: SIGMOD (2017)
4. Bernhard, K., Vygen, J.: Combinatorial Optimization: Theory and Algorithms, 3rd edn. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-662-56039-6
5. Borzsony, S., Kossmann, D., Stocker, K.: The skyline operator. In: ICDE (2001)
6. Cao, W., et al.: k-regret minimizing: efficient algorithms and hardness. In: ICDT (2017)
7. Chester, S., Thomo, A., Venkatesh, S., Whitesides, S.: Computing k-regret minimizing sets. In: VLDB (2014)
8. Faulkner, T.K., Brackenbury, W., Lall, A.: k-regret queries with nonlinear utilities. In: VLDB (2015)
9. He, J., Han, X.: Efficient skyline computation on massive incomplete data. Data Sci. Eng. **7**(2), 102–119 (2022)
10. He, Z., Lo, E.: Answering why-not questions on top-k queries. In: TKDE (2014)
11. Kallay, M.: The complexity of incremental convex hull algorithms in RD. Inf. Process. Lett. **19**(4), 197 (1984)
12. Kumar, N., Sintos, S.: Faster approximation algorithm for the k-regret minimizing set and related problems. In: ALENEX (2018)
13. Liu, P., Wang, M., Cui, J., Li, H.: Top-k competitive location selection over moving objects. Data Sci. Eng. **6**(4), 392–401 (2021)
14. Nanongkai, D., Lall, A., Sarma, A., Makino, K.: Interactive regret minimization. In: SIGMOD (2012)

15. Nanongkai, D., Sarma, A., Lall, A., Lipton, R., Xu, J.: Regret-minimizing representative databases. In: VLDB (2010)
16. Peng, P., Wong, R.: Geometry approach for k-regret query. In: ICDE (2014)
17. Qi, J., Zuo, F., Yao, J.: K-regret queries: from additive to multiplicative utilities. CoRR (2016)
18. Qin, L., Yu, J., Chang, L.: Diversifying top-k results. In: VLDB (2012)
19. Soliman, M., Ilyas, I., Chang, C.: Top-k query processing in uncertain databases. In: ICDE (2007)
20. Tao, Y., Ding, L., Lin, X., Pei, J.: Distance-based representative skyline. In: ICDE (2009)
21. Tao, Y., Xiao, X., Pei, J.: Efficient skyline and top-k retrieval in subspaces. In: TKDE (2007)
22. Wang, W., Wong, R., Xie, M.: Interactive search for one of the top-k. In: SIGMOD (2021)
23. Xie, M., Chen, T., Wong, R.: Find your favorite: an interactive system for finding the user's favorite tuple in the database. In: SIGMOD (2019)
24. Xie, M., Wong, R., Lall, A.: Strongly truthful interactive regret minimization. In: SIGMOD (2019)
25. Xie, M., Wong, R., Lall, A.: An experimental survey of regret minimization query and variants: bridging the best worlds between top-k query and skyline query. VLDB J. **29**, 147–175 (2020)
26. Xie, M., Wong, R., Li, J., Long, C., Lall, A.: Efficient k-regret query algorithm with restriction-free bound for any dimensionality. In: SIGMOD (2018)
27. Xie, M., Wong, R., Peng, P., Tsotras, V.: Being happy with the least: achieving α-happiness with minimum number of tuples. In: ICDE (2020)
28. Xin, D., Han, J., Cheng, H., Li, X.: Answering top-k queries with multi-dimensional selections: the ranking cube approach. In: VLDB (2006)